EXPAND YOUR MACHINE LEARNING KNOWLEDGE
WITH 30 QUESTIONS AND ANSWERS

MACHINE LEARNING Q and AI

Sebastian Raschka, PhD

Machine Learning Q and AI

Expand Your Machine Learning & AI Knowledge With 30 In-Depth Questions and Answers

Sebastian Raschka, PhD

This book is for sale at
http://leanpub.com/machine-learning-q-and-ai

This version was published on 2023-05-21

This book is dedicated to those who tirelessly contribute to advancing the field of machine learning through research and development. Your passion for discovery and innovation and your commitment to sharing knowledge and resources through the open-source community is an inspiration to us all.

Contents

Preface

Over the years, I shared countless educational nuggets about machine learning and deep learning with hundreds of thousands of people. The positive feedback has been overwhelming, and I continue to receive requests for more. So, in this book, I want to indulge both your desire to learn and my passion for writing about machine learning[1].

[1] I will use *machine learning* as an umbrella term for machine learning, deep learning, and artificial intelligence.

Who Is This Book For?

This book is for people with a beginner or intermediate background in machine learning who want to learn something new. This book will expose you to new concepts and ideas if you are already familiar with machine learning. However, it is not a math or coding book. You won't need to solve any proofs or run any code while reading. In other words, this book is a perfect travel companion or something you can read on your favorite reading chair with your morning coffee.

What Will You Get Out of This Book?

Machine learning and AI are moving at a rapid pace. Researchers and practitioners are constantly struggling to keep up with the breadth of concepts and techniques. This book provides bite-sized nuggets for your journey from machine learning beginner to expert, covering topics from various machine learning areas. Even experienced machine learning researchers and practitioners will encounter something new that they can add to their arsenal of techniques.

How To Read This Book

The questions in this book are mainly independent, and you can read them in any desired order. You can also skip individual questions for the most part. However, I organized the questions to bring more structure to the book.

For instance, the first question deals with embeddings, which we refer to in later questions on self-supervised learning and few-shot learning. Therefore, I recommend reading the questions in sequence.

The book is structured into five main chapters to provide additional structure. However, many questions could appear in different chapters without affecting the flow.

Chapter 1, Deep Learning and Neural Networks covers questions about deep neural networks and deep learning that are not specific to a particular subdomain. For example, we discuss alternatives to supervised learning and techniques for reducing overfitting.

Chapter 2, Computer Vision focuses on topics mainly related to deep learning but are specific to computer vision, many of which cover convolutional neural networks and vision transformers.

Chapter 3, Natural Language Processing covers topics around working with text, many of which are related to transformer architectures and self-attention.

Chapter 4, Production, Real-World, And Deployment Scenarios contains questions pertaining to practical scenarios, such as increasing inference speeds and various types of distribution shifts.

Chapter 5, Predictive Performance and Model Evaluation dives a bit deeper into various aspects of squeezing out predictive performance, for example, changing the loss function, setting up k-fold cross-validation, and dealing with limited labeled data.

If you are not reading this book for entertainment but for machine

learning interview preparation, you may prefer a spoiler-free look at the questions to quiz yourself before reading the answers. In this case, you can find a list of all questions, without answers, in the appendix.

Discussion Forum

The best way to ask questions about the book is the discussion forum at https://community.leanpub.com/c/machine-learning-q-a[2]. Please feel free to ask anything about the book, share your thoughts, or just introduce yourself!

[2]https://community.leanpub.com/c/machine-learning-q-a

Sharing Feedback and Supporting This Book

I enjoy writing, and it is my pleasure to share this knowledge with you.

If you obtained a free copy and like this book, you can support me by buying a digital copy on Leanpub at https://leanpub.com/machine-learning-q-and-ai/[3].

For an author, there is nothing more valuable than your honest feedback. I would really appreciate hearing from you and appreciate any reviews! And, of course, I would be more than happy if you recommend this book to your friends and colleagues or share some nice words on your social channels.

[3] https://leanpub.com/machine-learning-q-and-ai/

Acknowledgements

Writing a book is an enormous undertaking. This project would not have been possible without the help of the open source and machine learning communities who collectively created the technologies that this book is about. Moreover, I want to thank everyone who encouraged me to share my flashcard decks, as this book is an improved and polished version of these.

I also want to thank the following readers for helpful feedback on the manuscript:

- Anton Reshetnikov for suggesting a cleaner layout for the supervised learning flowchart in Q30.

About the Author

Sebastian Raschka is a machine learning and AI researcher with a strong passion for education. As Lead AI Educator at Lightning AI, he is excited about making AI and deep learning more accessible and teaching people how to utilize these technologies at scale.

Before dedicating his time fully to Lightning AI, Sebastian held a position as Assistant Professor of Statistics at the University of Wisconsin-Madison, where he specialized in researching deep learning and machine learning. You can find out more about his research on his website[4].

Moreover, Sebastian loves open-source software and has been a passionate contributor for over a decade. Next to coding, he also loves writing and authored the bestselling *Python Machine Learning* book and *Machine Learning with PyTorch and Scikit-Learn.*

If you like to find out more about Sebastian and what he is currently up to, please visit his personal website at https://sebastianraschka.com. You can also find Sebastian on Twitter (@rasbt)[5] and LinkedIn (sebastianraschka)[6].

[4] https://sebastianraschka.com/publications/
[5] https://twitter.com/rasbt
[6] https://www.linkedin.com/in/sebastianraschka

Copyright and Disclaimer

Credits

Cover image by ECrafts / stock.adobe.com.

Introduction

Thanks to rapid advancements in deep learning, we have seen a significant expansion of machine learning and AI in recent years.

On the one hand, this rapid progress is exciting if we expect these advancements to create new industries, transform existing ones, and improve the quality of life for people around the world.

On the other hand, the rapid emergence of new techniques can make it challenging to keep up, and keeping up can be a very time-consuming process. Nonetheless, staying current with the latest developments in AI and deep learning is essential for professionals and organizations that use these technologies.

With this in mind, I began writing this book in the summer of 2022 as a resource for readers and machine learning practitioners who want to advance their understanding and learn about useful techniques that I consider significant and relevant but often overlooked in traditional and introductory textbooks and classes.

I hope readers will find this book a valuable resource for obtaining new insights and discovering new techniques they can implement in their work.

Happy learning,
Sebastian

Chapter 1. Neural Networks and Deep Learning

Q1. Embeddings, Representations, and Latent Space

> Q:

In deep learning, we often use the terms *embedding vectors*, *representations*, and *latent space*. What do these concepts have in common, and how do they differ?

> A:

While all three concepts, embedding vectors, vectors in latent space, and representations, are often used synonymously, we can make slight distinctions:

- representations are encoded versions of the original input;
- latent vectors are intermediate representations;
- embedding vectors are representations where similar items are close to each other.

Embeddings

Embedding vectors, or *embeddings* for short, encode relatively high-dimensional data into relatively low-dimensional vectors.

We can apply embedding methods to create a continuous dense (non-sparse) vector from a one-hot encoding. However, we can also use embedding methods for dense data such as images. For example, the last layers of a convolutional neural network may yield embedding vectors, as illustrated in the figure below[7].

[7] Technically, all intermediate layer outputs of a neural network could yield embedding vectors. Depending on the training objective, the output layer may also produce useful embedding vectors. For simplicity, the convolutional neural network figure above only associates the second-last layer with embeddings.

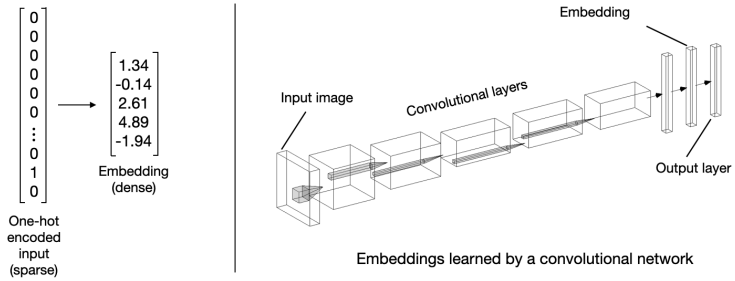


Figure 1.1. An input embedding (left) and an embedding from a neural network (right).

Taking it to the extreme, embedding methods can be used to encode data into two-dimensional dense and continuous representations for visualization purposes and clustering analysis, as illustrated in the figure below.

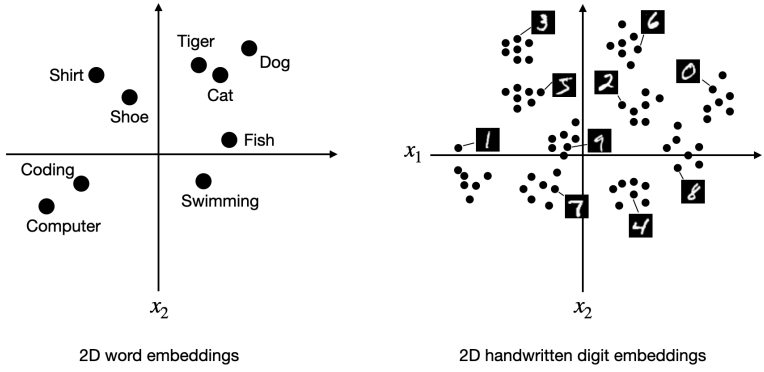


Figure 1.2. Mapping words (left) and images (right) to a two-dimensional feature space.

A fundamental property of embeddings is that they encode *distance* or *similarity*. This means that embeddings capture the semantics of the data such that similar inputs are close in the embeddings space.

Latent space

Latent space is typically used synonymously with *embedding space* – the space into which embedding vectors are mapped.

Similar items can appear close in the latent space; however, this is not a strict requirement. More loosely, we can think of the latent space as any feature space that contains the features, often a compressed version of the original input features. These latent space features can be learned by a neural network, for example, an autoencoder that reconstructs input images, as shown in the figure below.

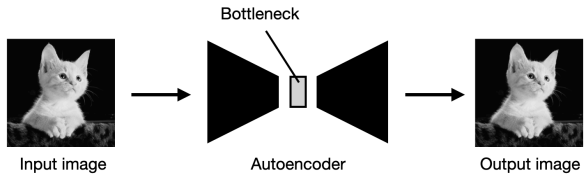


Figure 1.3. An autoencoder that reconstructs the input image after passing through a bottleneck layer.

The *bottleneck* in the figure above represents a small, intermediate neural network layer that encodes or maps the input image into a lower-dimensional representation. We can think of this mapping as a latent space. The training objective of the autoencoder is to reconstruct the input image, that is, minimizing the distance between the input and output images. In order to optimize the training objective, the autoencoder may learn to place the encoded features of similar inputs (for example, cats) close to each other in the latent space, thus creating useful embedding vectors where similar inputs are close in the embedding (latent) space.

Representation

We used the term representation above. A representation is an encoded, typically intermediate form of an input. For instance, an embedding vector or vector in the latent space is a *representation* of the input. However, representations can also be produced by simpler procedures. For example, one-hot encoded vectors are

considered representations of an input.

> Reader quiz:[8]

1-A

Suppose you are training a convolutional network with five convolutional layers followed by three fully connected (FC) layers similar to AlexNet[9] as illustrated in the figure below. You can think of these fully connected layers as two hidden layers and an output layer in a multilayer perceptron.

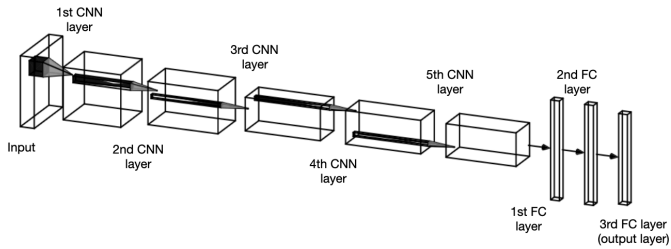


Figure 1.4. An illustration of AlexNet.

Which of the neural network layers can be utilized to produce useful embeddings?

1-B

Name at least one type of input representation that is not an embedding.

[8] Solutions to the reader quizzes are located in the appendix at the end of this book.

[9] Wikipedia summary: https://en.wikipedia.org/wiki/AlexNet. Original publication: Krizhevsky, Sutskever, Hinton (2012). *ImageNet Classification with Deep Convolutional Neural Networks*, http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

Q2. Self-Supervised Learning

> Q:

What is self-supervised learning, when is it useful, and what are the main categories of approaches?

> A:

Self-supervised learning is a pretraining procedure that lets neural networks leverage large unlabeled datasets in a supervised fashion.

Self-supervised is related to transfer learning. Suppose we are interested in training an image classifier to classify bird species. In transfer learning we would pretrain a convolutional neural network on ImageNet[10]. After pretraining on the general ImageNet dataset, we would take the pretrained model[11] and train it on the smaller, more specific target dataset that contains the bird species of interest.

[10] ImageNet is a large, labeled image dataset with many different categories, including various objects and animals. You can find more information at https://en.wikipedia.org/wiki/ImageNet.

[11] Often, we just have to change the class-specific output layer but can otherwise adopt the pretrained network as is.

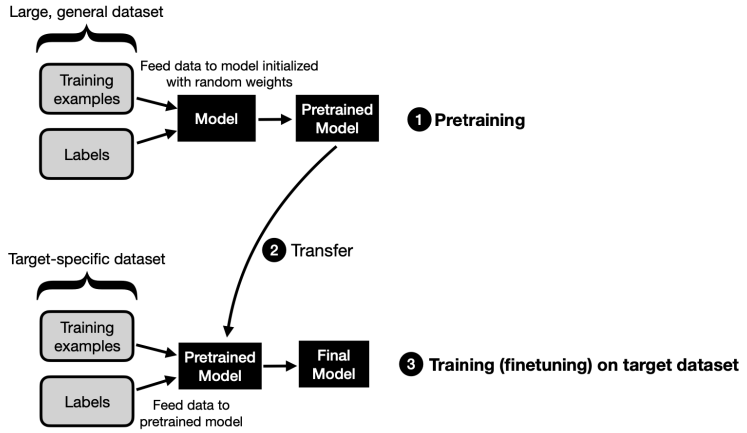


Figure 2.1. An illustration of conventional transfer learning.

Self-supervised learning is an alternative approach to transfer learning where we don't pretrain the model on labeled but *unlabeled* data. We consider an unlabeled dataset for which we do not have label information. We then find a way to obtain labels from the dataset's structure to formulate a prediction task for the neural network. These self-supervised training tasks are also called *pretext tasks*.

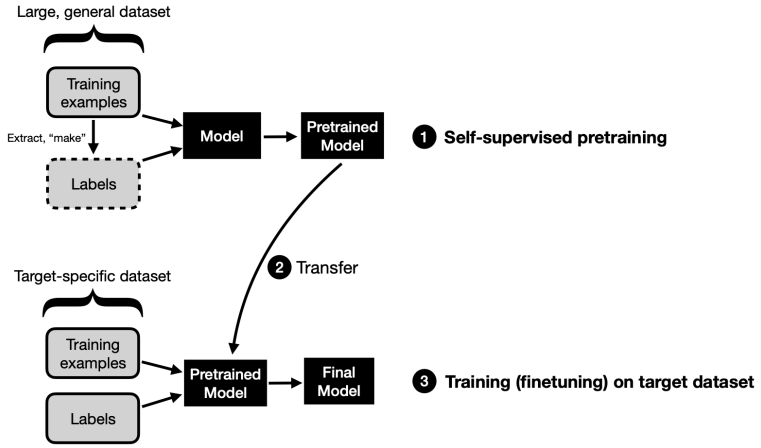


Figure 2.2. Overview of the self-supervised learning pipeline for predictive modeling. The only difference compared to the previous figure is how we obtain the labels during step 1.

Such a self-supervised learning task could be a "missing-word" prediction in a natural language processing context. For example, given the sentence "It is beautiful and sunny outside," we can mask out the word "sunny," feed the network the input It is beautiful and [MASK] outside," and have the network predict the missing word in the "[MASK]" location. Similarly, we could remove image patches in a computer vision context and have the neural network fill in the blanks. Note that these are just two examples of self-supervised learning tasks. Many more methods and paradigms for self-supervised learning exist.

In sum, we can think of self-supervised learning on the pretext task as representation learning. We can then take the pretrained model to finetune it on the target task (also known as the *downstream* task).

When is self-supervised learning useful?

Large neural network architectures require large amounts of labeled data to perform and generalize well. However, for many problem areas, we do not have access to large labeled datasets. With

self-supervised learning, we can leverage unlabeled data. Hence, self-supervised learning is likely useful if we work with large neural networks and we only have a limited amount of labeled training data.

Transformer-based architectures that form the basis of large language models and vision transformers are known to require self-supervised learning for pretraining to perform well.

For small neural network models, for example, multilayer perceptrons with two or three layers, self-supervised learning is typically considered neither useful nor necessary. However, examples of self-supervised learning for multilayer perceptrons and tabular datasets do exist[12][13].

Other contexts where self-supervised learning is not useful are traditional machine learning with nonparametric models such as tree-based random forests or gradient boosting. Conventional tree-based methods do not have a fixed parameter structure (in contrast to the weight matrices, for example). Thus, conventional tree-based methods are not capable of transfer learning and incompatible with self-supervised learning.

What are the main categories of self-supervised learning?

There are two main categories of self-supervised learning: (1) self-prediction and (2) contrastive self-supervised learning.

In self-prediction, we typically change or hide parts of the input and train the model to reconstruct the original inputs.

[12] Bahri, Jiang, Tay, and Metzler (2021). *SCARF: Self-Supervised Contrastive Learning Using Random Feature Corruption*, https://arxiv.org/abs/2106.15147.

[13] Levin, Cherepanova, Schwarzschild, Bansal, Bruss, Goldstein, Wilson, and Goldblum (2022). *Transfer Learning with Deep Tabular Models*, https://arxiv.org/abs/2206.15306.

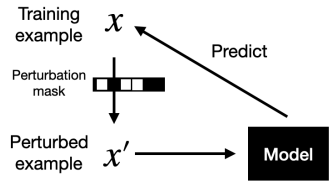


Figure 2.3. A self-prediction task predicting the original training example after applying a perturbation mask.

A classic example is a denoising autoencoder that learns to remove noise from an input image. Or, consider a masked autoencoder that reconstructs the missing parts of an image.

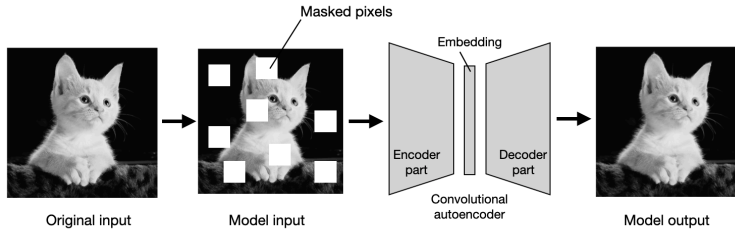


Figure 2.4. A masked autoencoder that learns to reconstruct the original image from a masked image.

Missing (masked) input self-prediction are also commonly used in natural language processing contexts as well. Many generative large language models, such as GPT are trained on a next-word prediction pretext task. Here, we feed the network's text fragments where it has to predict the next word in the sequence. (This is covered in more detail in Q18.)

In contrastive self-supervised learning, we train the neural network to learn an embedding space where similar inputs are close to each other, and dissimilar inputs are far apart. In other words, we train the network to produce embeddings such that the distance between two similar training inputs is minimized. And at the same time, the distance between two different training examples is maximized.

Let's discuss contrastive learning with concrete example inputs. Suppose we have a dataset consisting of random animal images. First, we draw a random image of a cat (the network does not know the label because we assume that the dataset is unlabeled). Then, we augment, corrupt, or perturb this cat image, for example, by adding a random noise layer and cropping it differently, as shown in the figure below.

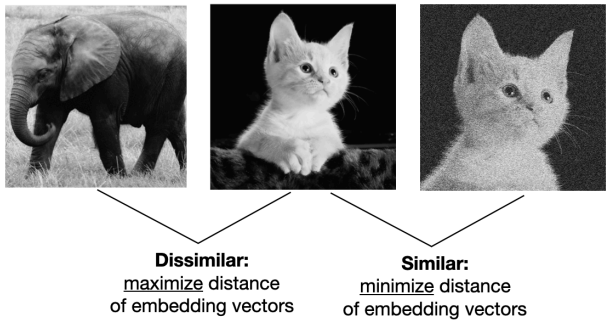


Figure 2.5. Image pairs encountered in contrastive learning.

The perturbed cat image still shows the same cat, so we want to network to produce a similar embedding vector. Then, we also consider a random image drawn from the training set (for example, an elephant, but again, the network doesn't know the label). For the cat-elephant pair, we want the network to produce dissimilar embeddings[14]. This way, we implicitly force the network to capture the image's core content while being somewhat agnostic to small differences and noise.

The following figure summarizes the central concept behind contrastive learning for the perturbed image scenario. Note that the model is shown twice – this is referred to as a *siamese network* setup. Essentially, the same model is utilized in two instances: first, to generate the embedding for the original training example, and

[14]The simplest form of a contrastive loss is the L_2-norm (Euclidean distance) between the embeddings produced by model $M(\cdot)$. For instance, we update the model weights to decrease the distance $||M(cat) - M(cat')||_2$ and increase the distance $||M(cat) - M(elephant)||_2$.

second, to produce the embedding for the perturbed version of the sample.

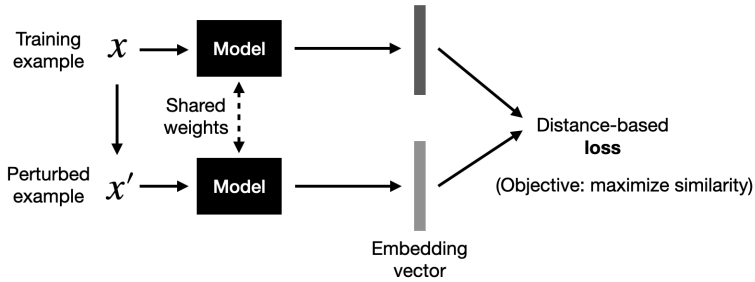


Figure 2.6. Contrastive learning overview.

The paragraphs above outlined the main idea behind contrastive learning, but many different subvariants exist. Broadly, we can categorize these into *sample* contrastive and *dimension* contrastive methods. Above, we described *sample* contrastive methods[15], where we focus on learning embeddings to minimize/maximize distances between training pairs. In *dimension*-contrastive approaches[16], which focus on making only certain variables in the embedding representations of similar training pairs similar while maximizing the distance of others.

> Reader quiz:

2-A

How could we apply self-supervised learning to video data?

2-B

Can self-supervised learning be used for tabular data represented as rows and columns? If so, how can we approach this?

[15] A popular sample constrastive method is, *A Simple Framework for Contrastive Learning of Visual Representations*, https://arxiv.org/abs/2002.05709 by Chen, Kornblith, Norouzi, and Hinton (2020).

[16] *Barlow Twins: Self-Supervised Learning via Redundancy Reduction*, https://arxiv.org/abs/2103.03230, by Zbontar, Jing, Misra, LeCun, and Deny (2021).

Q3. Few-Shot Learning

> Q:

What is few-shot learning? And how does it differ from the conventional training procedure for supervised learning?

> A:

Few-shot learning is a flavor of supervised learning for small training sets with a very small example-to-class ratio. In regular supervised learning, we train models by iterating over a training set where the model always sees a fixed set of classes. In few-shot learning, we are working on a support set from which we create multiple training tasks to assemble training episodes where each training task consists of different classes.

In supervised learning, we fit a model on a training dataset and evaluate it on a test dataset. Typically, the training set contains a relatively large number of examples per class. For example, in supervised learning context, a tiny dataset is the the Iris dataset with 50 examples per class. For deep learning model, even a dataset like MNIST with 5k training examples per class is considered as very small.

In few-shot learning, the number of examples per class is much smaller. We typically use the term *N*-way *K*-shot where *N* stands for the number of classes, and K stands for the number of examples per class. The most common values are *K=1* or *K=5*. For instance, in a 5-way 1-shot problem, we have 5 classes with only 1 example each. The figure below depicts a 3-way 1-shot setting for illustration purposes.

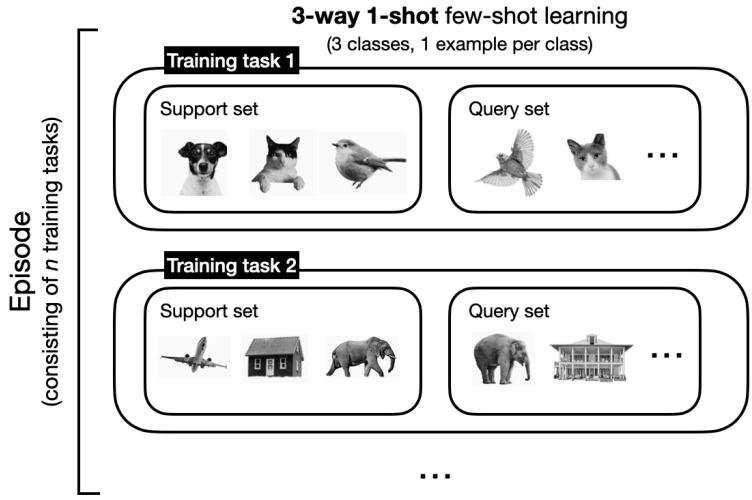


Figure 3.1. Illustration of training tasks in few-shot learning.

Rather than fitting the model to the training dataset, we can think of few-shot learning as "learning to learn." In contrast to supervised learning, we don't have a training dataset but a so-called support set. From the support set, we sample training tasks that mimic the use-case scenario during prediction. For example, for 3-way 1-shot learning, a training task consists of 3 classes with 1 example each. With each training task comes a query image that is to be classified. The model is trained on several training tasks from the support set; this is called an episode.

Then, during testing, the model receives a new task with classes that are different from those seen during training[17]. Again, the task is to classify the query images. Test tasks are similar to training tasks, except that none of the classes during testing overlap with those encountered during training.

[17]The classes encountered during training are also called *base* classes, and the support set during training is also often called *base set*.

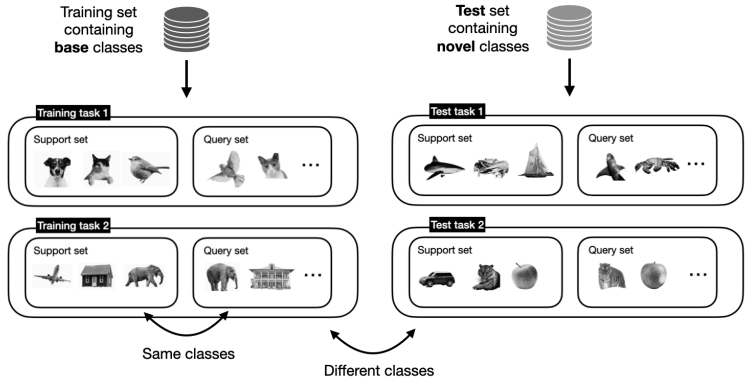


Figure 3.2. Illustration of the classes encountered during training and those encountered during testing.

There are many different types and categories of few-shot learning. In the most common one, meta-learning, the training is essentially about updating the model parameters such that it can *adapt* well to a new task. On a high level, one few-shot learning strategy is to learn a model that produces embeddings where we can find the target class via a nearest-neighbor search among the images in the support set.

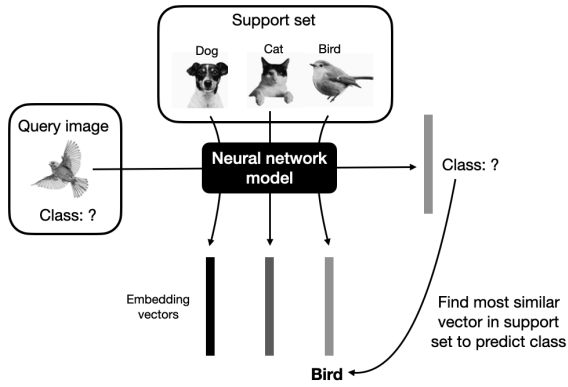


Figure 3.3. Illustration of a few-shot learning approach where the model learns how to produce good embeddings from the support set to classify the query image based on finding the most similar embedding vector.

> Reader quiz:

3-A

How could we partition the MNIST dataset, consisting of 50,000 handwritten digits from 10 classes (0-9), for a one-shot classification context?

3-B

Can you think of some real-world applications or use cases for few-shot learning?

Q4. The Lottery Ticket Hypothesis

> Q:

What is the *lottery ticket hypothesis*, and if it holds true, how can it be useful in practice?

> A:

According to the lottery ticket hypothesis[18], a randomly initialized neural network can contain a subnetwork that, when trained on its own, can achieve the same accuracy on a test set as the original network after being trained for the same number of steps.

The figure below illustrates the training procedure for the lottery ticket hypothesis in a more visual way. We will go through the steps one by one to help clarify the concept.

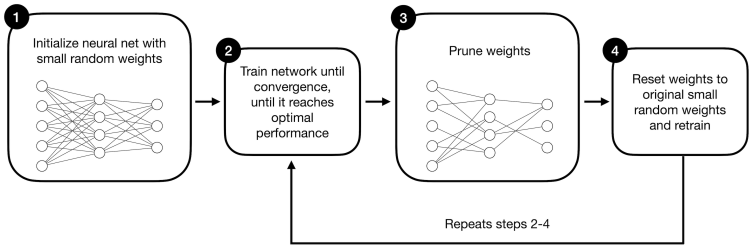


Figure 4.1. Outline of the lottery hypothesis training procedure.

We start with a large neural network (1) that we train until convergence (2), which means that we put in our best efforts to make it perform as best as possible on a target dataset – for example, minimizing training loss and maximizing classification accuracy. This large neural network is initialized as usual using small random weights.

Next, we prune the neural network's weight parameters (3), removing them from the network. We can do this by setting the weights

[18]Original reference: Frankle and Carbin (2018). *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks.* https://arxiv.org/abs/1803.03635.

to zero to create sparse weight matrices[19][20]. Which weights do we prune? The original lottery hypothesis approach follows a concept known as *iterative magnitude pruning*, where the weights with the lowest magnitudes are removed in an iterative fashion.

After the pruning step, we reset the weights to the original small random values used in step 1. It's worth emphasizing that we do not reinitialize the pruned network with any small random weights (as it is typical for iterative magnitude pruning) but reuse the weights from step 1.

The pruning steps 2-4 are then repeated until the desired network size is reached. For example, in the original lottery ticket hypothesis paper, the authors successfully reduced the network to 10% of its original size without sacrificing classification accuracy. As a nice bonus, the pruned (sparse) network, referred to as the winning ticket, even demonstrated improved generalization performance compared to the original (large and dense) network.

Practical implications and limitations

Suppose it is possible to identify smaller subnetworks that have the same predictive performance as their up to 10x larger counterparts. In that case, this can have significant implications for both neural training and inference. Given the ever-growing size of modern neural network architectures, this can help with cutting training costs and the infrastructure required for training. Additionally, smaller networks are more cost-effective and have lower latency when used for inference, making them a valuable option for many applications.

Sounds too good to be true? Maybe. If the *winning tickets* can be identified efficiently, this would be very useful in practice. However, as of this writing, there is no way to find the *winning*

[19]We can either prune individual weights, which is known as *unstructured* pruning. However, we can also prune larger "chunks" from the network, for example, entire convolutional filter channels. This is known as *structured* pruning.

[20]Li, Kadav, Durdanovic, Samet, and Graf (2016). *Pruning Filters For Efficient ConvNets*. https://arxiv.org/abs/1608.08710.

tickets without training the original network. If we include the pruning steps, this is even more expensive than a regular training procedure. Moreover, it was later found that the original weight initialization may not work for finding *winning tickets* for larger-scale networks, and additional experimentation with the initial weights of the pruned networks is required[21].

However, the good news is that *winning tickets* do exist, and even if it's currently not possible to identify these without training their larger neural network counterparts, they can be used for more efficient inference after training.

> **Reader quiz**:

4-A

Suppose you are trying out the lottery ticket hypothesis approach and find that the performance of the subnetworks is not very good (compared to the original network). What are some next steps to try?

4-B

How is the lottery ticket hypothesis related to training a neural network with ReLU activation functions (a ReLU activation function is defined as max(0, x)?

[21]Frankle, Dziugaite, Roy, and Carbin (2019). *Linear Mode Connectivity And The Lottery Ticket Hypothesis*, https://arxiv.org/abs/1912.05671.

Q5. Reducing Overfitting with Data

> Q:

Suppose we train a neural network classifier in a supervised fashion and notice that it suffers from overfitting. What are some of the common ways to reduce[22] overfitting in neural networks through the use of altered or additional data?[23]

> A:

In short, the most successful techniques for reducing overfitting revolve around collecting more high-quality labeled data. However, if collecting more labeled data is not feasible, we can augment the existing data or leverage unlabeled data for pretraining.

Summarized below are the most prominent examples of dataset-related techniques that more or less stood the test of time. We can group these techniques into multiple categories discussed below.

Collecting more data

One of the best ways to reduce overfitting is to collect more (good-quality) data. How do we know that more data is beneficial for minimizing overfitting? We can plot learning curves to find out. To construct a learning curve, we train the model to different sizes of the training set (10%, 20%, etc.) and evaluate the trained model on the same fixed-size validation or test set.

As shown in the figure below, we may observe that the validation accuracy increases with the increasing training set sizes. This indicates that we can improve the model's performance by collecting more data.

[22] While it is ideal to prevent overfitting, it is often not possible to completely eliminate it. Instead, we aim to *reduce* or *minimize* overfitting as much as possible.

[23] Suppose we are not changing the supervised learning procedure. Q30 lists other alternative training techniques for dealing with limited labeled data such as active learning and few-shot learning.

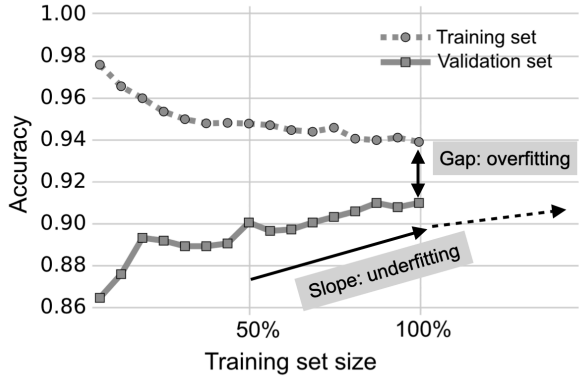


Figure 5.1. Learning curve plot of a model fit to different sizes of the training set.

The gap between training and validation performance indicates the degree of overfitting – the more extensive the gap, the more overfitting occurs. Conversely, the slope indicating an improvement in the validation performance suggests the model is underfitting and can benefit from more data.
Typically, additional data can decrease both underfitting and overfitting.

Data augmentation

Data augmentation refers to generating new data records or features based on existing data. It allows for the expansion of a dataset without additional data collection.

Data augmentation allows us to create different versions of the original input data, which can improve the model's generalization performance. Why? Augmented data can help the model to generalize better since it makes it harder to memorize spurious information via training examples or features (or exact pixel values for specific pixel locations in the case of image data).

Data augmentation is usually standard for image data (see figure below) and text data (see Q16), but data augmentation methods for

tabular data exist, too[24].

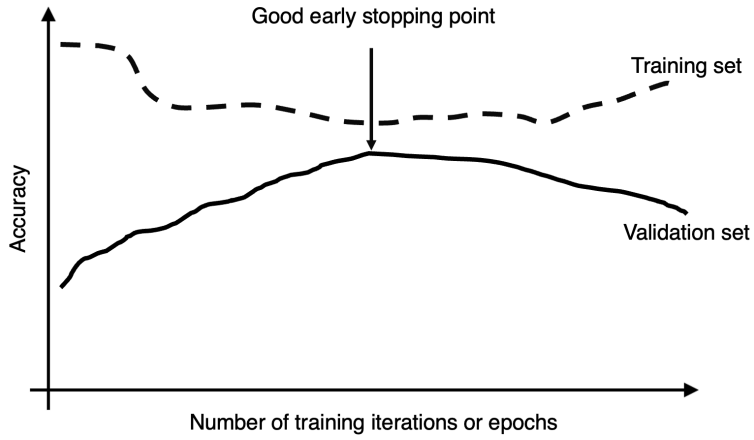


Figure 5.2. Illustration of early stopping.

Another option is to generate new, synthetic data. This falls in-between generating more data and data augmentation. While more common for image data and text, generating synthetic data is also possible for tabular datasets[25][26].

Pretraining

As discussed in the self-supervised section earlier in Q2, self-supervised learning lets us leverage large, unlabeled datasets to pretrain neural networks. This can help reduce overfitting on the smaller target datasets.

As an alternative to self-supervised learning, traditional transfer learning on large labeled datasets are also an option. Transfer

[24]Snow (2020). *DeltaPy: A Framework for Tabular Data Augmentation in Python*, https://github.com/firmai/deltapy.

[25]The GReaT method generates synthetic tabular data using an auto-regressive generative large language model. Reference: Borisov, Seßler, Leemann, Pawelczyk, and Kasneci (2022). *Language Models Are Realistic Tabular Data Generators*, https://arxiv.org/abs/2210.06280.

[26]TabDDPM is a method for generating synthetic tabular data using a diffusion model. Kotelnikov, Baranchuk, Rubachev, and Babenko (2022). *TabDDPM: Modelling Tabular Data with Diffusion Models*, https://arxiv.org/abs/2209.15421.

learning is most effective if the labeled dataset is closely related to the target domain. For instance, if we train a model to classify bird species, we can pretrain a network on a large, general animal classification dataset. However, if such a large animal classification dataset is unavailable, we can also pretrain the model on the relatively broad ImageNet dataset.

The dataset may be extremely small and unsuitable for supervised learning, for example, if there are only a handful of labeled examples per class. If our classifier needs to operate in a context where the collection of additional labeled data is not feasible, we may also consider few-shot learning.

Other methods

The list above covers the main approaches of using and modifying the dataset to reduce overfitting. However, the list above is not meant to be exhaustive. Other common techniques include

- feature engineering and normalization;
- the inclusion of adversarial examples and label or feature noise;
- label smoothing;
- smaller batch sizes;
- data augmentation techniques such as Mix-Up, Cut-Out, and Cut-Mix.

Q6 below covers additional techniques to reduce overfitting from a model perspective and concludes with a discussion of which regularization techniques we should consider in practice.

> Reader quiz:

5-A

Suppose we train an XGBoost model to classify images based on manually extracted features we obtained from our collaborators. The dataset of labeled training examples is relatively small, but

fortunately, your collaborators also have a labeled training set from an older project on a related domain.
We are considering implementing a transfer learning approach to train the XGBoost model. Is this a feasible option, and if yes, how could we do it? (Assume we are only allowed to use XGBoost, no other classification algorithm or model.)

5-B

Suppose we are working on an image classification problem (for this example, consider MNIST-based handwritten digit recognition) and added a decent amount of data augmentation to reduce overfitting in an image classification context. Unfortunately, we observe that the classification accuracy became much worse than before. What are some potential reasons?

Q6. Reducing Overfitting with Model Modifications

> Q:

Suppose we train a neural network classifier in a supervised fashion and we already employ various dataset-related techniques to mitigate overfitting[27]. How can we change the model or make modifications to the training loop to further reduce the effect of overfitting?

> A:

The most successful approaches against overfitting include regularization techniques such as dropout and weight decay. As a rule of thumb, models with a larger number of parameters require more training data to generalize well. Hence, decreasing the model size and capacity can sometimes also help reduce overfitting. Lastly, building model ensembles are among the most effective ways to combat overfitting, but it comes with increased computational expenses.

The various model- and training-related techniques to reduce overfitting can be grouped into three broad categories: adding regularization, choosing smaller models, and building ensemble models. The following paragraphs outline the key ideas and techniques from each category.

Regularization

We can interpret regularization as a penalty against complexity. Classic regularization techniques for neural networks include L_2 regularization and the related weight decay method. We implement L_2 regularization by adding a penalty term to the loss function that is minimized during training. This added term represents the size

[27] Q5 discusses various dataset-related techniques to reduce overfitting.

of the weights, for instance, the squared sum of the weights. The formula below shows an L_2 regularized loss:

$$\text{RegularizedLoss} = \text{Loss} + \frac{\lambda}{n} \sum_j w_j^2,$$

where λ is a hyperparameter that controls the regularization strength.

During backpropagation, the optimizer minimizes the modified loss, now including the additional penalty term, which leads to smaller model weights and can improve generalization to unseen data.

Weight decay is similar to L_2 regularization, but it is applied to the optimizer directly rather than modifying the loss function. Since weight decay has the same effect as L_2 regularization, the two methods are often used synonymously, but there can be subtle depending on the implementation details and optimizer[28].

Dropout and early stopping. Note that many other techniques have regularizing effects. For brevity, we keep this list focused on the most widely used methods, including dropout and early stopping.

Dropout reduces overfitting by randomly setting some of the activations of the hidden units to zero during training. Consequently, the neural network cannot rely on particular neurons to be activated and learns to use a larger number of neurons and learn multiple independent representations of the same data, which helps to reduce overfitting.

In early stopping, we monitor the model's performance on a validation set during training. And we stop the training process when the performance on the validation set begins to decline.

[28]The subtle difference between L2 regularization and weight decay is explained in *Three Mechanisms of Weight Decay Regularization* by Zhang, Wang, Xu, and Grosse (2018), https://arxiv.org/abs/1810.12281.

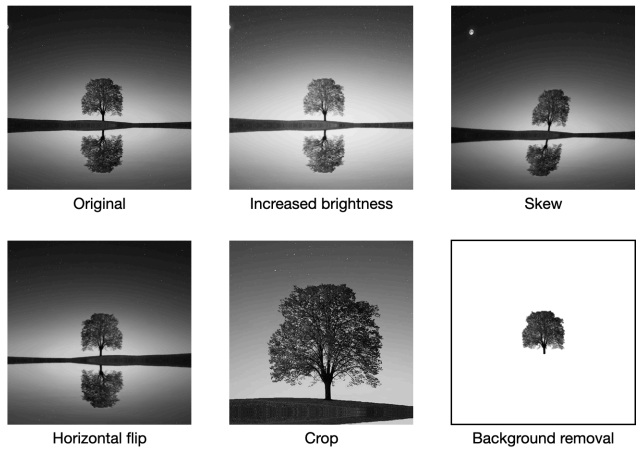


Figure 6.1. A selection of different image data augmentation techniques.

Smaller models

Classic bias-variance theory suggests that reducing model size can reduce overfitting[29]. The intuition behind it is that, as a general rule of thumb, the smaller the smaller the number of model parameters, the smaller its capacity to memorize or overfit to noise in the data.

Pruning. Besides reducing the number of layers and shrinking the layers' widths as a hyperparameter tuning procedure, one approach to obtaining smaller models is iterative pruning. In iterative pruning, we train a large model to achieve the good performance on the original dataset. Then, we iteratively remove parameters of the model, retraining it on the dataset such that it maintains the same predictive performance as the original model. Iterative pruning is used in the *lottery ticket hypothesis* discussed in [#q4.

Knowledge distillation. Another common approach to obtaining smaller models is knowledge distillation. The general idea behind knowledge distillation is that we transfer knowledge from a large, more complex model (called *teacher*) to a smaller model (called

[29]Hastie, Tibshirani, Friedman (2009). *The Elements of Statistical Learning*. Chapter 2.9, Model Selection and Bias-Variance Tradeoff.

student). Ideally, the student achieves the same predictive accuracy as the teacher while being more efficient due to the smaller size. And, as a nice side-effect, the smaller student may overfit less than the larger teacher model.

The original knowledge distillation process is summarized in the figure below. Here, the teacher is first trained in a regular supervised fashion to classify the examples in the dataset well using a conventional cross entropy loss between the predicted scores and ground truth class labels.

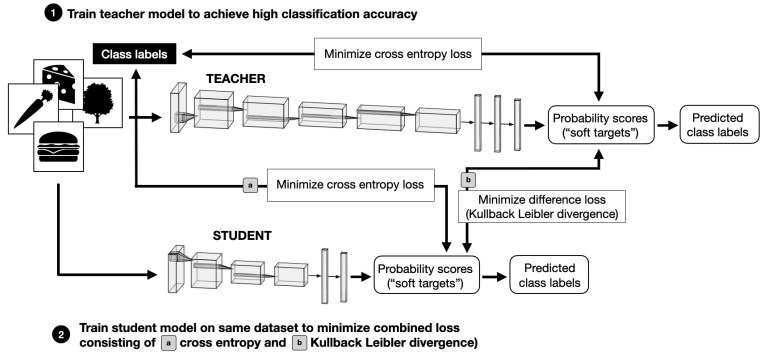


Figure 6.2. Outline of the original knowledge distillation process.

While the smaller student network is trained on the same dataset, the training objective is to minimize both (a) the cross entropy between the outputs and the class labels and (b) the difference between its outputs and the teacher outputs (measured using Kullback Leibler divergence). By minimizing the Kullback Leibler divergence, the difference between the teacher and student score distributions, the student learns to mimic the teacher while being smaller and more efficient. As mentioned above, an additional benefit is that the smaller student models often generalize better than the larger teacher models.

Caveats with smaller models

The original research results indicate that pruning[30] and knowledge distillation[31] can improve the generalization performance, presumably due to smaller model sizes.

However, counterintuitively, recent research studying phenomena like *double decent*[32] and *grokking*[33] also showed that larger, overparameterized models have improved generalization performance if they are trained beyond the point of overfitting.

How can we reconcile the observation that pruned models can exhibit better generalization performance with contradictory observations from studies of double-decent and grokking? Researchers recently showed that the improved training process partly explains the reduction of overfitting due to pruning[34]. Pruning involves more extended training periods and a replay of learning rate schedules that may be partly responsible for the improved generalization performance.

Pruning and knowledge distillation remain excellent ways to improve the computational efficiency of a model. However, while pruning and knowledge distillation can also enhance a model's generalization performance, these techniques are not primary or effective ways of reducing overfitting.

Ensemble methods

Ensemble methods combine predictions from multiple models to

[30]Frankle and Carbin (2018). *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*, https://arxiv.org/abs/1803.03635.

[31]Hinton, Vinyals, and Dean (2015). *Distilling the Knowledge in a Neural Network*, https://arxiv.org/abs/1503.02531.

[32]Double descent refers to the phenomenon where models with either a small or an extremely large number of parameters have good generalization performance, while models with a number of parameters equal to the number of training data points have poor generalization performance. For more additional explanations and references, see https://en.wikipedia.org/wiki/Double_descent.

[33]Grokking reveals that as the size of a dataset decreases, the need for optimization increases, and generalization performance can improve well past the point of overfitting. Power, Burda, Edwards, Babuschkin, and Misra (2022). *Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets*. https://arxiv.org/abs/2201.02177.

[34]Jin, Carbin, Roy, Frankle, and Dziugaite (2022). *Pruning's Effect on Generalization Through the Lens of Training and Regularization*. https://arxiv.org/abs/2210.13738.

improve the overall prediction performance. However, the downside of using multiple models is an increased computational cost.

We can think of ensemble methods as asking a committee of experts: members in a committee often have different backgrounds and experiences. While they tend to agree on basic decisions, they can overrule bad decisions by majority rule. This doesn't mean that the majority of experts is always right, but there is a good chance that the majority of the committee is more often right, on average, than every single member.

Ensemble methods are more prevalent in classical machine learning than deep learning because it is more computationally expensive to employ multiple models than relying on a single one. Or in other words, deep neural networks require significant computational resources, making them less suitable for ensemble methods. (It's worth noting that while we previously discussed Dropout as a regularization technique, it can also be considered an ensemble method that approximates a weighted geometric mean of multiple networks[35].)

Popular examples of ensemble methods are random forests and gradient boosting. However, using majority voting or stacking, for example, we can combine any group of models: an ensemble may consist of a support vector machine, a multilayer perceptron, and a nearest neighbor classifier.

A popular technique that is often used in industry is to build models from k-fold cross-validation. K-fold cross-validation is a model evaluation technique where we train and evaluate a model on k training folds. We then compute the average performance metric across all k iterations to estimate the overall performance measure of the model. After evaluation, we can train the model on the entire training dataset, or the individual models can be combined as an

[35] Baldi and Sadowski (2013). Understanding Dropout. *Advances In Neural Information Processing Systems*, https://proceedings.neurips.cc/paper/2013/hash/71f6278d140af599e06ad9bf1ba03cb0-Abstract.html

ensemble, as shown in the figure below.

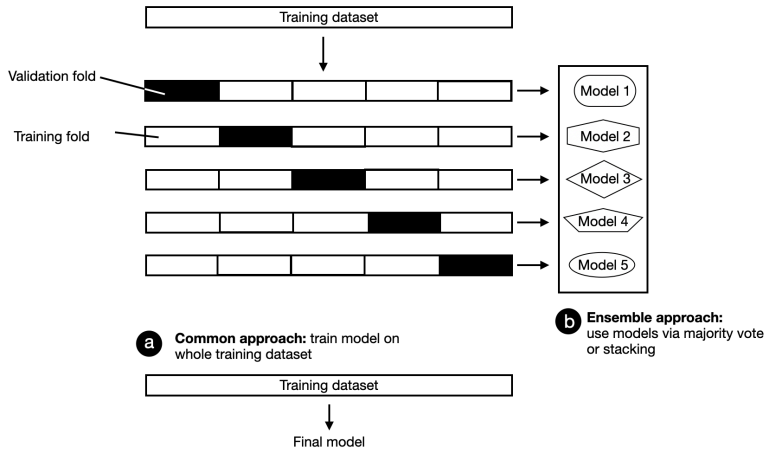


Figure 6.3. Illustration of k-fold cross-validation for creating model ensembles.

Other methods

The non-exhaustive list above includes the most prominent examples of techniques to reduce overfitting. Additional techniques that can reduce overfitting are skip-connections (for example, found in residual networks), look-ahead optimizers, stochastic weight averaging, multitask learning, and snapshot ensembles.

Techniques that aim to reduce overfitting from a data perspective are discussed in Q5.

Choosing regularization techniques

Improving data quality is an essential first step in reducing overfitting. However, for recent deep neural networks with large numbers of parameters, more than improving data quality is required to achieve an acceptable level of overfitting. Therefore, data augmentation and pretraining, along with established techniques such as dropout and weight decay, remain crucial methods to reduce overfitting.

In practice, we need to employ more than one technique to reduce

overfitting. Instead, we can and should combine the abovementioned techniques for an additive effect. To achieve the best results, selecting these techniques should be treated as a hyperparameter optimization problem[36].

> Reader quiz:

6-A

Suppose we are using early-stopping as a mechanism to reduce overfitting. In particular, we are using a modern variant that creates checkpoints of the best model (for instance, the model with highest validation accuracy) during training so that we can load it after the training has completed – this is a mechanism that can be enabled in most modern deep learning frameworks.

Now, a colleague recommends tuning the number of training epochs instead. What are some of the advantages and disadvantages of each approach?

6-B

Ensemble models have been established as a reliable and successful method for decreasing overfitting and enhancing the reliability of predictive modeling efforts. However, there is always a trade-off. What are some of the drawbacks associated with ensemble techniques?

[36] In the paper *Well-tuned Simple Nets Excel on Tabular Datasets*, https://arxiv.org/abs/2106.11189), researchers showed that regularization cocktails need to be tuned on a per-dataset basis.

Q7. Multi-GPU Training Paradigms

> Q:

What are the different multi-GPU training paradigms, and what are their respective advantages and disadvantages?

> A:

The multi-GPU training paradigms can be categorized into two groups: (1) dividing data for parallel processing with multiple GPUs and (2) dividing the model among multiple GPUs to handle memory constraints when the model size surpasses that of a single GPU.

Data parallelism falls into the first category, model parallelism and tensor parallelism fall into the second category, and techniques like pipeline parallelism borrow ideas from both categories. In addition, current software implementations such as DeepSpeed[37], Colossal AI[38], and others blend multiple approaches into a hybrid technique.

Model parallelism

Model parallelism (also referred to as *inter op parallelism*) is perhaps the most intuitive form of parallelization across devices. For example, suppose you have a simple neural network that only consists of 2 layers: a hidden layer and an output layer. Here, we keep one layer on one GPU and the other layer on another GPU. Of course, this can scale to an arbitrary number of layers and GPUs.

This is a good strategy for dealing with limited GPU memory where the complete network does not fit into one GPU. However, there are more efficient ways of using multiple GPUs because of the chain-like structure (layer 1 on GPU 1 $\rightarrow$ layer 2 on GPU 2 $\rightarrow$ ...).

A major disadvantage of model parallelism is that the GPUs have to wait for each other – they cannot efficiently work parallel as they depend on each other's outputs.

[37] https://github.com/microsoft/DeepSpeed
[38] https://github.com/hpcaitech/ColossalAI

Data parallelism

Data parallelism has been the default mode for multi-GPU training for several years. Here, we divide a minibatch into smaller microbatches. Then, each GPU processes a microbatch separately to compute the loss and loss gradients for the model weights. After the individual devices process the microbatches, the gradients are combined to compute the weight update for the next round.

An advantage of data parallelism over model parallelism is that the GPUs can run in parallel – each GPU processes a portion of the training minibatch, a microbatch. However, a caveat is that each GPU requires a full copy of the model. This is obviously not feasible if we have large models that don't fit into the GPU's VRAM.

Tensor parallelism

Tensor parallelism (also referred to as *intra op parallelism*) is a more efficient form of model parallelism (*inter op parallelism*). Here, the weight and activation matrices are spread across the devices instead of distributing whole layers across devices. Specifically, the individual matrices are split, so we split an individual matrix multiplication across GPUs.

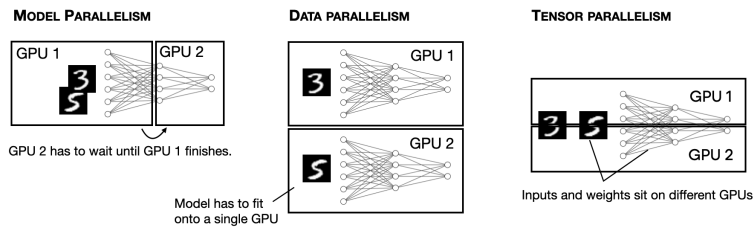


Figure 7.1. Comparison of model parallelism (left), data parallelism (center), and tensor parallelism (right). In model parallelism, we put different layers onto different GPUs to work around GPU memory limitations. In data parallelism, we split batch across GPUs to train copies of the model in parallel, averaging gradients for the weight update afterwards. In tensor parallelism, we split matrices (inputs and weights) across different GPU for parallel processing when models are too large to fit into GPU memory.

There are several ways we can implement tensor parallelism. For example, using basic principles of linear algebra, we can split a matrix multiplication across two GPUs in a row- or column-wise fashion, as illustrated in the figure below. (Note that this concept can be extended to an arbitrary number of GPUs.)

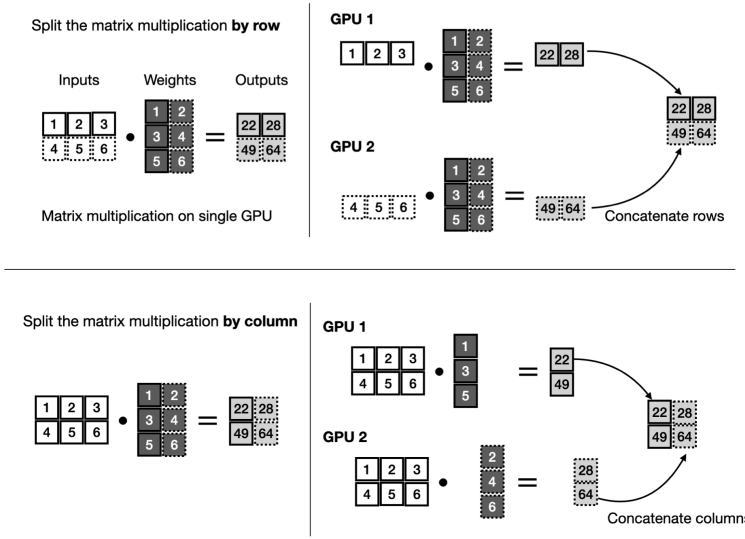


Figure 7.2. Illustration how we can distribute matrix multiplication across different devices. For simplicity, the figure depicts two GPUs, but the concepts extends to an arbitrary number of devices.

An advantage of tensor parallelism is that we can work around memory limitations similar to model parallelism. And at the same time, we can also execute operations in parallel, similar to data parallelism.

A small weakness of tensor parallelism is that it can result in high communication overhead between the multiple GPUs across which the matrices are split or sharded. For instance, tensor parallelism requires frequent synchronization of the model parameters across the devices, which can slow down the overall training process.

Pipeline parallelism

Pipeline parallelism can be seen as a form of model parallelism that tries to minimize the sequential-computation bottleneck. In that sense, we can think of pipeline parallelism as a form of model parallelism that enhances the parallelism between the individual layers sitting on different devices. However, note that it also borrows ideas from data parallelism, such as splitting minibatches further into microbatches.

How does it work? During the forward pass, the activations are passed similar to model parallelism, however, the twist is that the gradients of the input tensor are passed backwards to prevent the devices from being idle. In a sense, pipeline parallelism is a sophisticated hybrid between data and model parallelism, which is described in more detail in the GPipe paper[39] or DeepSpeed pipeline parallelism tutorial[40].

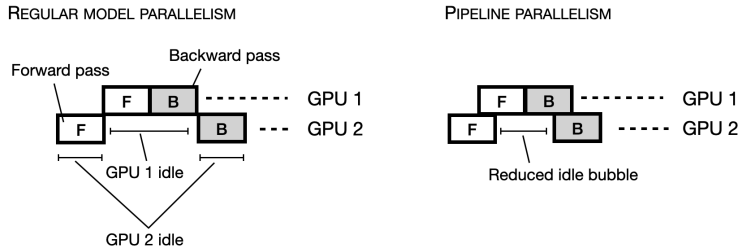


Figure 7.3. A conceptual illustration of pipeline parallelism, which aims to reduce the idle time of GPUs compared to regular model parallelism.

A disadvantage of pipeline parallelism is that it may require significant effort to design and implement the pipeline stages and associated communication patterns. Additionally, the performance gains from pipeline parallelism may not be as substantial as those from other parallelization techniques, such as pure data parallelism, especially for small models or in cases where the communication

[39]Huang, Cheng, Bapna, Firat, Chen, Chen, Lee, Ngiam, Le, Wu, and Chen (2018). *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*, https://arxiv.org/abs/1811.06965.

[40]Pipeline Parallelism: https://www.deepspeed.ai/tutorials/pipeline/

overhead is high.

Pipeline parallelism is definitely an improvement over model parallelism, even though it is not perfect, and there will be idle bubbles. However, for modern architectures that are too large to fit into GPU memory, it is nowadays more common to use a blend of data parallelism and tensor parallelism (as opposed to model parallelism) techniques.

Sequence parallelism

Sequence parallelism is a new concept developed for transformer models[41]. One shortcoming of transformers is that the self-attention mechanism (the original scaled-dot product attention) scales quadratically with the input sequence length. There are, of course, more efficient alternatives to the original attention mechanism that scales linearly[42][43]; however, they are less popular, and most people prefer the original scaled-dot product attention mechanism.

Sequence parallelism splits the input sequence into smaller chunks that can be distributed across GPUs to work around memory limitations as illustrated in the figure below.

[41] Li, Xue, Baranwal, Li, and You (2021). *Sequence Parallelism: Long Sequence Training from [a] System[s] Perspective*, https://arxiv.org/abs/2105.13120.

[42] Tay, Dehghani, Bahri, and Metzler (2020). *Efficient Transformers: A Survey*, https://arxiv.org/abs/2009.06732.

[43] Zhuang, Liu, Pan, He, Weng, and Shen (2023). *A Survey on Efficient Training of Transformers*, https://arxiv.org/abs/2302.01107.

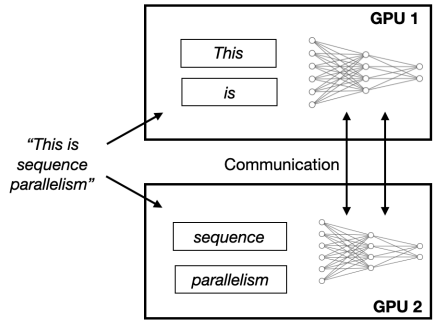


Figure 7.4. A conceptual illustration of sequence parallelism, which aims to reduce computation memory constraints of self-attention mechanisms.

Sequence parallelism is less well-studied than other parallelization techniques. However, conceptually, it would have similar advantages and disadvantages as tensor parallelism. It enables us to train larger models when memory is a constraint due to the sequence length; however, it also introduces additional communication overheads. On the other hand, it also shares shortcoming of data parallelism where we have to duplicate the model and make sure it fits into the device memory.

Another disadvantage of sequence parallelism (depending on the implementation) for multi-GPU training of transformers is that breaking up the input sequence into smaller subsequences can decrease the model's accuracy – mainly when the model is applied to longer sequences.

Which techniques should we use in practice?

Practical recommendations depend on the context. If we train small models that fit onto a single GPU, then data parallelism strategies may be the most efficient. Performance gains from pipeline parallelism may not be as significant as those from other parallelization techniques such as data parallelism, especially for small models or in cases where the communication overhead is high.

If models are too large to fit into the memory of a single GPU, we

need to explore model or tensor parallelism. Tensor parallelism is naturally more efficient since the GPUs can work in parallel – there is no sequential dependency as in model parallelism.

Modern multi-GPU strategies typically combine data parallelism and tensor parallelism (popular examples include DeepSpeed stages 2 and 3[44]).

> Reader quiz:

7-A

Suppose we are implementing our own version of tensor parallelism, which works great when we train our model with an SGD (standard stochastic gradient descent) optimizer.
However, when we try the Adam[45] optimizer, we encounter an out-of-memory device. What could be a potential problem explaining this issue?

7-B

Suppose we don't have access to a GPU and are considering using data parallelism on the CPU. Is this a good idea?

[44] https://www.deepspeed.ai/tutorials/zero/

[45] Kingma and Ba (2014). *Adam: A Method for Stochastic Optimization*, https://arxiv.org/abs/1412.6980.

Q8. The Keys to Success of Transformers

> Q:

What are the main factors that have contributed to the success of transformers?

> A:

In recent years, transformers have emerged as the most successful neural network architecture, particularly for various natural language processing tasks. In fact, transformers are now on the cusp of becoming state-of-the-art for computer vision tasks as well.
The success of transformers can be attributed to several key factors, including their attention mechanisms, ability to be parallelized easily, unsupervised pretraining, and high parameter counts.

Attention mechanism

At the core, the mechanism underlying transformers is the self-attention mechanism introduced with the original transformer architecture in 2017[46]. Although originally, attention mechanisms were developed in the context of image recognition[47] before they were adopted to aid the translation of long sentences in recurrent neural networks, attention mechanisms are the fundamental mechanism of transformers. (The attention mechanisms found in recurrent neural networks and transformers are compared in more detail in Q17.)

What makes attention mechanisms so unique and useful? Suppose we are using an encoder network that is a fixed-length representation of the input sequence or image – this can be a fully connected, convolutional, or attention-based encoder.

[46] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin (2017). *Attention Is All You Need*, https://arxiv.org/abs/1706.03762.

[47] Larochelle and Hinton (2010). *Learning to Combine Foveal Glimpses With A Third-Order Boltzmann Machine*, https://dl.acm.org/doi/10.5555/2997189.2997328.

In a transformer, the encoder uses self-attention mechanisms to compute the importance of each input token relative to other tokens in the sequence, allowing the model to focus on relevant parts of the input sequence. Conceptually, attention mechanisms allow the transformers to attend to different parts of a sequence or image. On the surface, this sounds very similar to a fully connected layer (or special cases of convolutional layers[48]) where each input element is connected via a weight with each other input element in the next layer.

In attention mechanisms, the computation of the attention weights involves comparing each input element to the other. The attention weights obtained by this approach are dynamic and input dependent. In contrast, the weights of a convolutional or fully connected layer are fixed after training.

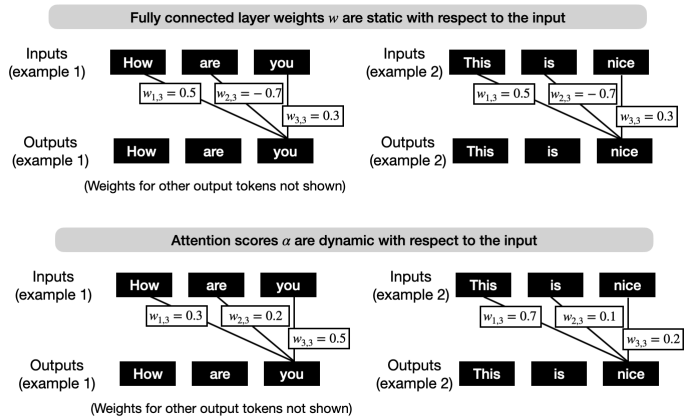


Figure 8.1. Illustration of the conceptual difference between model weights in fully connected layers and attention scores.

Attention mechanisms allow a neural network to selectively weigh the importance of different input features, allowing the model to focus on the most relevant parts of the input for a given task.

Pretraining via self-supervised learning

[48] Q12 discusses contexts where fully connected and convolutional layers are equivalent.

Pretraining transformers via self-supervised learning on large, unlabeled datasets is another key factor in the success of transformers.

During pretraining, the transformer model is trained to predict missing words in a sentence or the next sentence in a document, for example. By learning to predict these missing words or the next sentence, the model is forced to learn general representations of language that can be finetuned for a wide range of downstream tasks.

However, while unsupervised pretraining has been highly effective for natural language processing tasks, its effectiveness for computer vision tasks is still an active area of research. (Please refer to Q2 for a more detailed discussion of self-supervised learning.)

Large numbers of parameters

One noteworthy characteristic of transformers is their large model sizes. For example, the popular 2020 GPT-3 model consists of 175 billion trainable parameters, and other transformers, such as Switch Transformers have trillions of parameters[49].

The scale and number of trainable parameters of transformers are essential factors in their modeling performance, particularly for large-scale natural language processing tasks. For instance, linear scaling laws[50] describe that the training loss improves proportionally to the model size – a two-times size increase in model size can half the training loss, which then translates to a better modeling performance on the downstream target task.

However, it is essential to scale the model size and the number of training tokens equally. This means the number of training tokens should be doubled for every doubling of model size[51]. However,

[49]Fedus, Zoph, and Shazeer (2021). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*, https://arxiv.org/abs/2101.03961.

[50]Kaplan, McCandlish, Henighan, Brown, Chess, Child, Gray, Radford, Wu, and Amodei (2020), *Scaling Laws for Neural Language Models*, https://arxiv.org/abs/2001.08361.

[51]Hoffmann et al. (2022). *Training Compute-Optimal Large Language Models*, https://arxiv.org/abs/2203.15556.

since labeled data is limited, utilizing large amounts of data during unsupervised pretraining is vital.

To summarize, large model sizes and large datasets are critical factors in transformers' success. Additionally, using self-supervised learning, the ability to pretrain transformers is closely tied to using large model sizes and large datasets. This combination has been critical in enabling the success of transformers in a wide range of natural language processing tasks.

Easy parallelization

As mentioned above, transformers require large amounts of parameters and data to achieve their breakthrough modeling performances. However, training such large models on large datasets requires vast computational resources, and it's key that the computations can be parallelized to utilize these resources. Fortunately, transformers are easy to parallelize.

Transformers are easy to parallelize because they take a fixed-length sequence of word or image tokens as input. For instance, the self-attention mechanism used in most transformer architectures involves computing the weighted sum between each input element to each other input element. Furthermore, these pair-wise token comparisons can be computed independently, making the self-attention mechanism relatively easy to parallelize across different GPU cores.

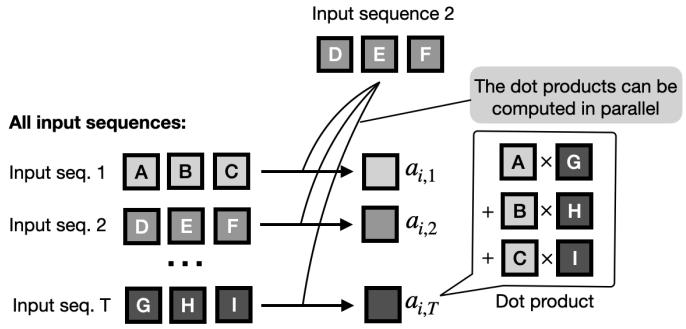


Figure 8.2. A simplified self-attention mechanism without weight parameters.

In addition, the individual weight matrices used in the self-attention mechanism[^q8-attention weights] (not shown) can be distributed across different machines for distributed and parallel computing.

[^q8-attention weights]: If you are interest in learning more about the weights used in self-attention and cross-attention mechanism, I recommend checking out my blog post *Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch* at https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html[52].

Reader quiz:

8-A

Above, we discussed that self-attention is easily parallelizable. And yet, transformers are considered computationally expensive due to self-attention. How can we explain this contradiction?

8-B

Since self-attention scores represent importance weights for the various input elements, can we consider self-attention a form of feature selection?

[52] https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html

Q9. Generative AI Models

> Q:

What are the popular categories of deep generative models in deep learning (also called *generative AI*), and what are their respective downsides?

> A:

There are many different types of deep generative models that have been applied to generating different types of media: images, videos, text, and audio. Going beyond these types of media, models can also be repurposed to generate domain-specific data, for example, organic molecules and protein structures. But first, we have to define the term *generative modeling*. Then we can go over each type of generative model and discuss its strenghts and weaknesses.

Generative versus discriminative modeling

In traditional machine learning, there are two primary approaches to modeling the relationship between input data (x) and output labels (y): generative models and discriminative models. Generative models aim to capture the underlying probability distribution of the input data $p(x)$ or the joint distribution $p(x, y)$ between inputs and labels. In contrast, discriminative models focus on modeling the conditional distribution $p(y \mid x)$ of the labels given the inputs.

A classic example that highlights the differences between these approaches is the comparison of the naive Bayes classifier, a generative model, and the logistic regression classifier, a discriminative model. Both classifiers estimate the class label probabilities $p(y \mid x)$ and can be used for classification tasks.

Logistic regression is considered a discriminative model because it directly models the conditional probability distribution $p(y \mid x)$ of the class labels given the input features, without making assumptions about the underlying joint distribution of inputs and labels.

Naive Bayes, on the other hand, is considered a generative model because it models the joint probability distribution $p(x, y)$ of the input features x and the output labels y. By learning the joint distribution, a generative model like naive Bayes captures the underlying data generation process, which enables it to generate new samples from the distribution if needed.

Nowadays, when we speak of *deep* generative models or sometimes also *deep generative AI*, we often losen this definition to include all types of models capable of producing realistic-looking data (typically, text, images, videos, and sound). The remainder of this section will briefly discuss the different types of deep generative models used to generate such data.

Energy-based Models

EBMs are a class of generative models that learn an energy function, which assigns a scalar value (energy) to each data point. Lower energy values correspond to more likely data points. The model is trained to minimize the energy of real data points while increasing the energy of generated data points. Examples of EBMs include Deep Boltzmann Machines[53].

Somewhat similar naive Bayes and logistic regression, Deep Boltzmann Machines (DBMs) and Multilayer Perceptrons (MLPs) can be considered as generative and discriminative counterparts, with DBMs being more focused on capturing the data generation process and MLPs being more focused on modeling the decision boundary between classes or mapping inputs to outputs.

A Deep Boltzmann Machine consists of multiple layers of hidden nodes (as shown in the figure below). But besides using a different learning algorithm (contrastive divergence instead of backpropagation), DBMs also differ from MLPs in that they consist of binary nodes (neurons) instead of continuous ones.

[53]Salakhutdinov and Hinton (2009). *Deep Boltzmann Machines*, https://proceedings.mlr.press/v5/salakhutdinov09a.html.

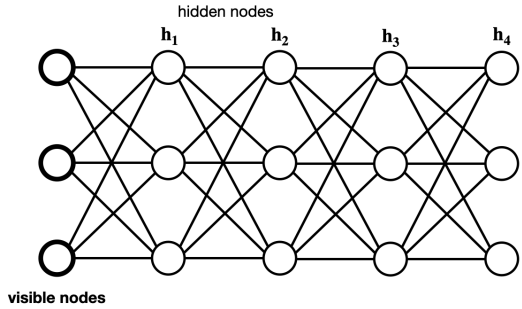


Figure 9.1. A 4-layer Deep Boltzmann Machine with 3 stacks of hidden nodes.

Suppose we are interested in generating images. A DBM can learn the joint probability distribution over the pixel values in a simple image dataset like MNIST. To generate new images, the DBM then samples from this distribution by performing a process called Gibbs sampling. Here, the visible layer of the DBM represents the input image. To generate a new image, start by initializing the visible layer with random values, or alternatively, use an existing image as a seed. Then, after completing several Gibbs sampling iterations, the final state of the visible layer represents the generated image.

Deep Boltzmann machines are among the oldest deep generative models and are mentioned for historical reasons, but they are not very popular for generating data nowadays. The disadvantages of DBMs are that they are expensive and more complicated to train as well as the lower expressivity compared to other models described below, which generally results in lower-quality generated samples.

Variational Autoencoders

Variational Autoencoders (VAE)[54] are built upon the principles of variational inference and autoencoder architectures. Variational inference is a method for approximating complex probability distributions by optimizing a simpler, tractable distribution to be as close as possible to the true distribution. Autoencoders are

[54] Kingma and Welling (2013). *Auto-Encoding Variational Bayes*, https://arxiv.org/abs/1312.6114.

unsupervised neural networks that learn to compress input data into a low-dimensional representation (encoding) and subsequently reconstruct the original data from the compressed representation (decoding) by minimizing the reconstruction error.

The VAE model consists of two main submodules, as summarized in the figure below: an encoder network and a decoder network. The encoder network takes, for example, an input image and maps it to a latent space by learning a probability distribution over the latent variables. This distribution is typically modeled as a Gaussian with parameters (mean and variance) that are functions of the input image. The decoder network then takes a sample from the learned latent distribution and reconstructs the input image from this sample. The goal of the VAE is to learn a compact and expressive latent representation that captures the essential structure of the input data while being able to generate new images by sampling from the latent space[55].

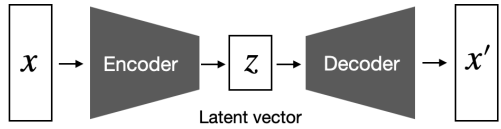


Figure 9.2. Illustration of the encoder and decoder submodules of an autoencoder where *x'* represents the reconstructed input *x*. In a standard variational autoencoder, the latent vector is sampled from a distribution that approximates a standard Gaussian distribution.

Training a VAE involves optimizing the model's parameters to minimize a loss function composed of two terms: a reconstruction loss and a Kullback Leibler (KL)-divergence regularization term. The reconstruction loss ensures that the decoded samples closely resemble the input images, while the KL-divergence term acts as a surrogate loss that encourages the learned latent distribution to be close to a predefined prior distribution (usually a standard Gaussian).

[55] See Q1 for more details on latent representations.

To generate new images, we then sample points from the latent space's prior (standard Gaussian) distribution and pass them through the decoder network, which generates new, diverse images that look similar to the training data.

Disadvantages of VAE are the complicated loss function consisting to separate terms and the often low expressiveness, which can result in blurrier images compared to other models such as generative adversarial networks described below.

Generative Adversarial Networks

While both Generative Adversarial Networks (GANs)[56] and VAEs are latent variable models that generate data by sampling from a learned latent space, their architectures and learning mechanisms are fundamentally different.

GANs consist of two neural networks, a generator, and a discriminator, that are trained simultaneously in an adversarial manner. The generator takes a random noise vector from the latent space as input and generates a synthetic data sample (for example, an image)[57]. The discriminator's task is to distinguish between real samples from the training data and fake samples generated by the generator, as illustrated in the figure below.

[56]Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville, and Bengio (2014). *Generative Adversarial Networks*, https://arxiv.org/abs/1406.2661.

[57]The generator in a GAN somewhat resembles the decoder of a VAE in terms of its functionality. During inference, both GAN generators and VAE decoders take random noise vectors sampled from a known distribution (for example, a standard Gaussian) and transform them it into synthetic data samples, such as an images.

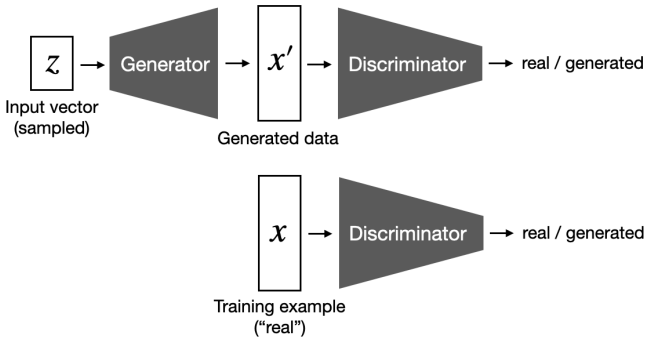


Figure 9.3. The components of a GAN model, where the discriminator predicts whether its inputs are real or generated by the generator.

The generator aims to produce samples that are indistinguishable from real data, while the discriminator aims to accurately identify whether a sample is real or generated. This adversarial process leads to a generator capable of producing high-quality and diverse samples.

However, one of the main disadvantages of GANs is their unstable training due to the adversarial nature of the loss function and learning process. Balancing the learning rates of the generator and discriminator can be difficult, and it can often result in oscillations, mode collapse, or non-convergence. The second main disadvantage of GANs is the low diversity of its generated outputs, often due to mode collapse. Here, the generator is able to fool the discriminator successfully with a small set of samples., which are only representative of a small subset of the original training data.

Flow-based models

The core concept of flow-based models, also known as normalizing flows, is inspired by longstanding methods in statistics. The primary goal is to transform a simple probability distribution (like a Gaussian) into a more complex one using invertible transformations.

Although the concept of normalizing flows has been a part of the

statistics field for a long time, the implementation of early flow-based deep learning models, particularly for image generation, is a relatively recent development. One of the pioneering models in this area was the NICE[58] approach, which emerged in 2014.
Related to VAEs and the generator in GANs, the idea behind NICE is to generate complex data (like images) from simple random noise. However, how it generates the data from the random noise fundamentally differs from VAEs and GANs.

NICE begins with a simple probability distribution, often something straightforward like a normal distribution. You can think of this as a kind of "random noise" — data with no particular shape or structure. NICE then applies a series of transformations to this simple distribution. Each transformation is designed to make the data look more like the final target (for instance, the distribution of real-world images). These transformations are "invertible," meaning we can always reverse them back to the original simple distribution. After several successive transformations, the simple distribution has been morphed into a complex distribution that closely matches the distribution of the target data (like images). We can now generate new data that looks like the target data by picking random points from this complex distribution.

The concept of a flow-based model is illustrated in the figure below. At first glance, the illustration is very similar to the VAE illustration above. However, while VAEs use a neural network encoder, such as a convolutional neural network, the flow-based model uses simpler decoupling layers, such as simple linear transformations. Also, while the decoder in a VAE is independent of the encoder, in the flow-based model, the data-transforming functions are mathematically inverted to obtain the outputs.

[58] Dinh, Krueger, and Bengio (2014). *NICE: Non-linear Independent Components Estimation*, https://arxiv.org/abs/1410.8516.

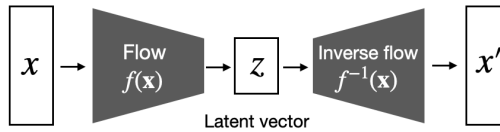


Figure 9.4. Illustration of a flow-based model mapping the complext input distribution to a simpler distribution and back.

Unlike VAEs and GANs, flow-based models provide exact likelihoods, which gives us insights into how well the generated samples fit the training data distribution. This can be useful in anomaly detection or density estimation, for example. However, the quality of flow-based models for generating image data is usually lower than GANs.
Also, flow-based models often require more memory and computational resources than GANs or VAEs due to their need for storing and computing inverses of transformations.

Autoregressive models

Autoregressive models are based on predicting the next value based on the current (and past) values. A popular example includes large language models for text generation, like ChatGPT, which are covered in Q18.

Similar to generating one word at a time, in the context of image generation, autoregressive models like PixelCNN[59] try to predict one pixel at a time, given the pixels it has seen so far. The order in which pixels are predicted can be from top-left to bottom-right, in a raster scan order, or any other defined order.

Suppose we have an image of size $H \times W$ (where H is the height and W is the width), ignoring the color channel for simplicity. This image consists of N pixels, $i = 1, ..., N$. Now, the probability of observing a particular image in the dataset is $P(Image) = P(i_1, i_2, ..., i_N)$. And based on the concepts of statistics, we can

[59] van den Oord, Kalchbrenner, Vinyals, Espeholt, Graves, and Kavukcuoglu (2016). *Conditional Image Generation with PixelCNN Decoders*, https://arxiv.org/abs/1606.05328.

decompose this joint probability into conditional probabilities:

$P(Image) = P(i_1, i_2, ..., i_N) = P(i_1) \cdot P(i_2|i_1) \cdot P(i_3|i_1, i_2) \ldots P(i_N|i_1 to i_{N-1})$.

Here, $P(i_1)$ is the probability of the first pixel, `P(i_2|i_1)` is the probability of the second pixel given the first pixel, $P(i_3|i_1, i_2)$ is the probability of the third pixel given the first and second pixels, and so on.

So, In the context of image generation, an autoregressive model essentially tries to predict one pixel at a time, given the pixels it has seen so far, as illustrated in the figure below.

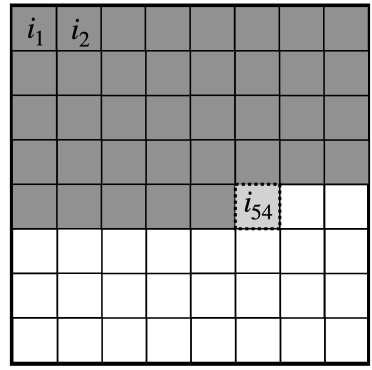


Figure 9.5. Illustration of autoregressive pixel generation, where pixels $i_1 ... i_{53}$ represent the context, and pixel i_54 is the next pixel to be generated.

The advantage of autoregressive models is that the next-pixel (or word) prediction is relatively straightforward and interpretable. In addition, autoregressive models can compute the likelihood of data exactly, similar to flow-based models, which can be useful for tasks like anomaly detection. Furthermore, autoregressive models are easier to train than GANs as they don't suffer from issues like mode collapse and other training instabilities.

The problem with autoregressive models is that they can be slow at generating new samples. This is because they have to generate data one step at a time (for example, pixel-by-pixel for images), which

can be computationally expensive. Then, autoregressive models can also struggle to capture long-range dependencies because each output is only conditioned on previously generated outputs.

So, regarding overall image quality, autoregressive models are usually worse than GANs, which are harder to train but better than the faster flow-based models.

Diffusion models

As discussed in the previous section, flow-based models transform a simple distribution (like a standard normal distribution) into a complex one (the target distribution) by applying a sequence of invertible and differentiable transformations (flows).

Similar to flow-based models, diffusion models also apply a series of transformations. However, the underlying concept is fundamentally different. Diffusion models transform the input data distribution into a simple noise distribution over a series of steps using stochastic differential equations.

The diffusion processes, is a stochastic process where noise is progressively added to the data until it resembles a simpler distribution like Gaussian noise. Then, to generate new samples, the process is reversed, starting from noise and progressively removing it.

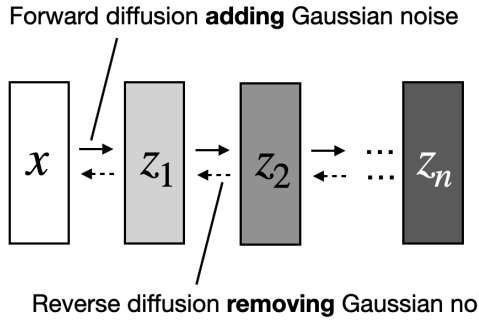


Figure 9.6. Illustration of a diffusion process adding and removing Gaussian noise from an input image x. During inference, the reverse diffusion process is used to generate a new image x starting with the noise tensor z_n sampled from a Gaussian distribution$

While both diffusion models and flow-based models are generative models aiming to learn complex data distributions, they approach the problem from different angles: flow-based models use deterministic invertible transformations, while diffusion models use the above-mentioned stochastic diffusion process.

Recent projects, such as *Stable Diffusion*[60] in 2022, established new state-of-the-art performance in in generating high-quality images with realistic details and textures. Moreover, diffusion models are easier to train than GANs. But their downside is that they are slower to sample from, since they require running a series of steps in sequential steps similar to flow-based models and autoregressive models.

Consistency Model

Consistency models train a neural network to map a noisy image to a clean one. The network is trained on a dataset of pairs of noisy and clean images. The network learns to identify the patterns in the clean images that are modified by the noise. Once the network is

[60]Stable Diffusion (https://github.com/CompVis/stable-diffusion) is an implementation of the paper *High-Resolution Image Synthesis with Latent Diffusion Models* (2021) by Rombach, Blattmann, Lorenz, Esser, and Ommer.

trained, it can be used to generate reconstructed images from noisy images in one step.

The consistency model training employs an ordinary differential equation (ODE) trajectory, a path that a noisy image follows as it is gradually denoised. The ODE trajectory is defined by a set of differential equations that describe how the noise in the image changes over time, illustrated in the figure below.

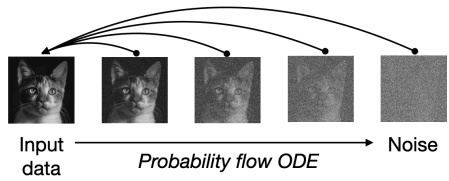


Figure 9.7. Illustration of consistency models that learn to map any point form a probability flow ODE, which smoothly converts data to noise, to the input.

As of this writing, consistency models are the most recent type of generative AI model. Based on the original paper proposing this method[61], consistency models rival diffusion models in terms of image quality. Moreover, consistency models are faster than diffusion models because they do not require an iterative process to generate images but generate images in a single step.

However, while consistency models allow for faster inference, they are still expensive to train because they require a large dataset of pairs of noisy and clean images.

Conclusion

Deep Boltzmann machines are interesting from a historical perspective. Flow-based and autoregressive models are nice for estimating exact likelihoods. However, other models are usually the first choice when it comes to generating high-quality images.

Particularly VAEs and GANs were battling each other hard when

[61] Song, Dhariwal, Chen, and Sutskever (2023). Consistency Models, https://arxiv.org/abs/2303.01469.

generating high-fidelity images over the years. However, in 2022, diffusion models have begun taking over image generation almost entirely. Consistency models are a promising alternative to diffusion models, but it will remain to be seen whether they become more widely adopted to generate state-of-the-art results.

'However, the trade-off here is that sampling from diffusion models is generally slower. This is because it involves a sequence of noise-removal steps that must be run in order, similar to autoregressive models. This can make them less practical for some applications requiring fast sampling.

> Reader quiz:

9-A

How would you evaluate the quality of the images generated by an generative AI model?

9-B

Given the description of consistency models above, how would you use them to generate new images?

Q10. Sources of Randomness

> Q:

What are the common sources of randomness when training deep neural networks that can cause non-reproducible behavior during training and inference?

> A:

When training or using machine learning models such as deep neural networks, several sources of randomness can lead to different results every time we train or run these models, even though we use the same overall settings. Some of these effects are accidental, and some of these are intended. The following sections will categorize and discuss these various sources of randomness[62].

Model weight initilization

All common deep neural network frameworks, including TensorFlow and PyTorch, randomly initialize the weights and bias units at each layer by default. This means that the final model will be different every time we start the training.

In practice, it is thus recommended to run the training (if the computational resources permit) at least a handful of times: Sometimes, unlucky initial weights can cause the model not to converge or converge to a local minimum corresponding to poorer predictive accuracy.

[62]Hands-on examples for most of these categories are provided in the supplementary material available at https://github.com/rasbt/MachineLearning-QandAI-book.

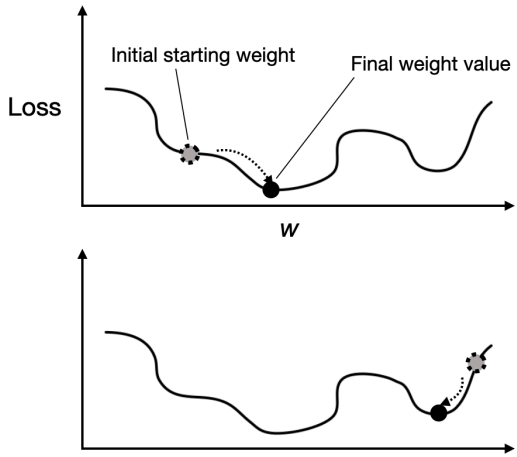


Figure 10.1. Different starting weights can lead to different final weights after the training converges.

However, it is possible to turn the random weight initialization into a deterministic random initialization by seeding the random generator. For example, if we seed the random generator with a seed such as 123, the weights will still be initialized with small random values. Still, the neural network will be initialized with the same small random weights whenever someone tries to reproduce the results.

Dataset sampling and shuffling

When we train and evaluate machine learning models, we usually start by dividing a dataset into a training and a test set. This requires random sampling since we have to decide which examples we put into a training set and which examples we put into a test set.

Furthermore, we may use model evaluation techniques such as k-fold cross-validation or holdout validation, which means that we split the training set into a training, a validation, and a test

dataset[63], which are also sampling procedures that are influenced by randomness.
However, it is possible to turn the random weight initialization into a deterministic random initialization by seeding the random generator. For example, if we seed the random generator with a seed such as 123, the weights will still be initialized with small random values. Still, the neural network will be initialized with the same small random weights whenever someone tries to reproduce the results.

Unless we use a fixed random seed, we get a different model each time we partition the dataset or tune or evaluate the model using k-fold cross-validation since the training partitions will differ.

Nondeterministic algorithms

We may include random components and algorithms depending on the architecture and hyperparameter choices. A popular example is dropout[64]. Dropout works by randomly setting a fraction of a layer's units to zero during training, which helps the model learn more robust and generalized representations. This "dropping out" is typically applied at each training iteration with a probability p, a hyperparameter that controls the fraction of units dropped out. Typical values for p are in the range of 0.2 to 0.8.

If we are interested in reproducible training runs, we need to seed the random generator before training with dropout (analogous to seeding the random generator before dropout). During inference, we need to disable dropout to guarantee deterministic results – each deep learning framework has a specific setting for that.

[63]Interested readers can learn more about different data sampling and model evaluation techniques in Raschka (2018), *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*, https://arxiv.org/abs/1811.12808.

[64]Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, https://jmlr.org/papers/v15/srivastava14a.html.

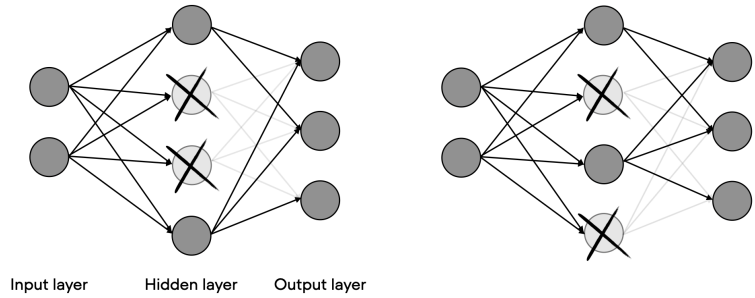


Figure 10.2. Dropout randomly drops a subset of the hidden layer nodes in each forward pass during training

Different runtime algorithms

The most intuitive or simple implementation of an algorithm or method is not always the one that we use in practice. For example, when training deep neural networks, we often use efficient alternatives and approximations to gain speed and resource advantages during training and inference.

A popular example is the convolution operation used in convolutional neural networks. There are several ways we can implement the convolution operation:

1. The classic *direct* convolution: The common implementation of discrete convolution via an elementwise product between the input and the window, followed by summing the result to get a single number.[65]
2. FFT-based convolution: Uses Fast Fourier Transform (FFT) to convert the convolution into an element-wise multiplication in the frequency domain.[66]
3. Winograd-based convolution: An efficient algorithm for small filter sizes, like 3x3, that reduces the number of

[65] See Q12 for a discussion of the convolution operation.

[66] Chi, Jiang, and Mu (2020). *Fast Fourier Convolution*, https://dl.acm.org/doi/abs/10.5555/3495724.3496100.

> multiplications required for the convolution.[67]

These algorithms have different trade-offs in terms of memory usage, computational complexity, and speed.

When running deep neural networks on GPUs, by default, libraries such as cuDNN (CUDA Deep Neural Network library), which are used in PyTorch and TensorFlow, can choose different algorithms for performing convolution operations[68]. While these approximations yield similar results, subtle numerical differences can accumulate during training and cause the training to converge to slightly different local minima.

Hardware and drivers

Training deep neural networks on different hardware can also result in different results due to small numeric differences even though the same algorithms are used, and the same operations are executed. These differences may sometimes be due to different numeric precision for floating point operations. However, small numeric differences may arise due to hardware and software optimization, even at the same precision. For instance, different hardware platforms may have specialized optimizations or libraries that can slightly alter the behavior of deep learning algorithms.

> "Across different architectures, no cuDNN routines guarantee bit-wise reproducibility. For example, there is no guarantee of bit-wise reproducibility when comparing the same routine run on NVIDIA Volta™ and NVIDIA Turing™, NVIDIA Turing, and NVIDIA Ampere architecture."[69]

[67] Alam, Anderson, Barabasz, and Gregg (2023). *Winograd Convolution for Deep Neural Networks: Efficient Point Selection*, https://arxiv.org/abs/2201.10369.

[68] Deterministic algorithm choice has to be explicitly enabled. In PyTorch, for example, this can be done by setting `torch.use_deterministic_algorithms(True)`; see https://pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html for details.

[69] https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#reproducibility

Randomness and Generative AI

Besides the various sources of randomness mentioned above, certain models can also exhibit random behavior during inference that we can describe as *randomness by design*. For instance, generative image and language models[q9-q10] may create different results for identical prompts to create a diverse sample of results. For image models, this is often so that users can select the most accurate and aesthetically pleasing image. For language models, this is often to vary the responses, for example, in chat agents, to avoid repetition.

The intended randomness in generative image models during inference is often due to sampling different noise values at each step of the reverse process – in diffusion models, a noise schedule defines the noise variance added at each step of the diffusion process.

Autoregressive large language models like GPT[70] tend to create different outputs for the same input prompt. The ChatGPT UI even has a "Regenerate response" button for that. The ability to generate different results is due to the sampling strategies these models employ. Techniques such as top-k sampling, nucleus sampling, or temperature scaling influence the model's output by controlling the degree of randomness. This is a feature, not a bug, since it allows for diverse responses and prevents the model from producing overly deterministic or repetitive outputs.

Top-k sampling works by sampling tokens from the top k most probable candidates at each step of the next-word generation process. For instance, given an input prompt, the language model produces a probability distribution over the entire vocabulary (the candidate words) for the next token. Each token in the vocabulary is assigned a probability based on the model's understanding of the context. The selected top-k tokens are then renormalized so that the probabilities sum to 1. Finally, a token is sampled from the renormalized top-k probability distribution, and it is appended to the input prompt. This process is repeated for the desired length

[70] Autoregressive large language models are discussed in more detail in Q18.

of the generated text or until a stop condition is met

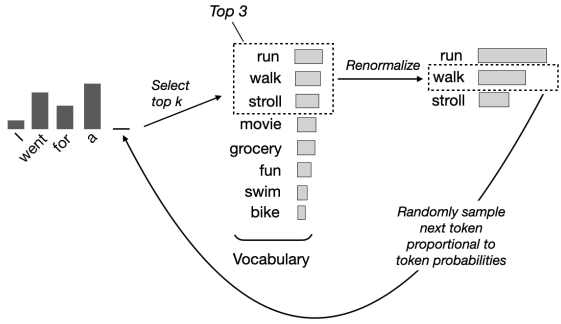


Figure 10.3. Illustration of the top-k sampling process used in generative large language models

Nucleus sampling (also known as top-p sampling) is related to top-k sampling, which also aims to balance diversity and coherence in the output. Nucleus and top-k sampling mainly differ in selecting the candidate tokens for sampling at each step of the generation process. Top-k sampling selects the k most probable tokens from the probability distribution produced by the language model, regardless of their probabilities. The value of k remains fixed throughout the generation process.

Nucleus sampling, on the other hand, selects tokens based on a probability threshold p. Then, it accumulates the most probable tokens in descending order until their cumulative probability meets or exceeds the threshold p. In contrast to top-k sampling, the size of the candidate set (nucleus) can vary at each step.

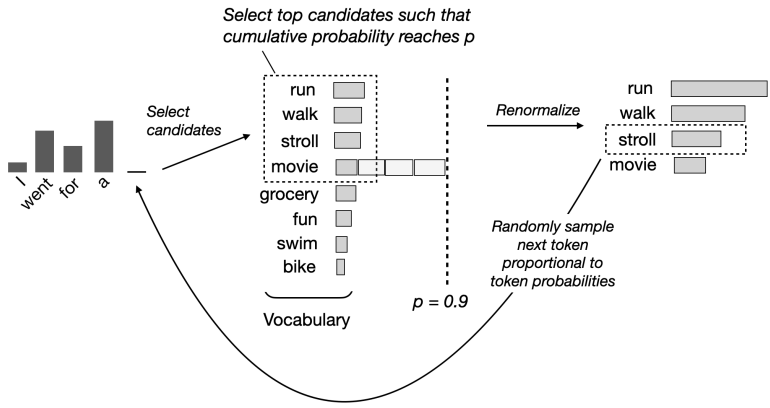


Figure 10.4. Illustration of the nucleus process used in generative large language models

Temperature scaling is a technique used to control the sharpness of the distribution by modifying the logits (values returned by the output layer) based on a temperature parameter (T). A higher temperature (T > 1) flattens the distribution, making the model more likely to explore diverse tokens. In comparison, a lower temperature (0 < T < 1) sharpens the distribution, focusing the model on the most probable tokens. Temperature scaling can be combined with top-k or nucleus sampling to balance diversity and coherence in text generation. For instance, we often first apply temperature scaling to adjust the sharpness of the probability distribution based on our desired level of exploration and then use top-k or nucleus sampling to limit the sampling space to the most probable tokens.

> Reader quiz:

10-A

Suppose we trained a neural network with top-k or nucleus sampling where k and p are hyperparameter choices. Can we make the model behave deterministically during inference without changing the code?

10-B

Can you think of scenarios where deterministic dropout behavior during inference is desired?

Chapter 2. Computer Vision

Q11. Calculating the Number of Parameters

> Q:

How do we compute the number of parameters in a convolutional neural network? Suppose we are working with a convolutional network with 2 convolutional layers with kernel size 5. The first convolutional layer has 3 input channels and 5 output channels. The second convolutional layer has 5 input and 12 output channels. The stride of these convolutional layers is 1.
Furthermore, the network has 2 pooling layers with kernel size 3 and stride 2. Lastly, the network has 2 fully connected hidden layers with 192 and 128 hidden units each, where the output layer is a classification layer for 10 classes. The architecture of this network is illustrated in the figure below. What is the number of trainable parameters in this convolutional network?

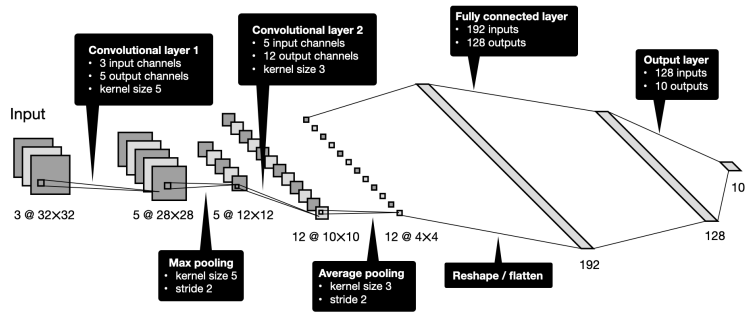


Figure 11.1. Illustration of a convolutional neural network with two convolutional and two fully connected layers.

> A:

We can approach this problem from left to right, computing the number of parameters for each layer and then summing up these counts to obtain the total number of parameters. Each layer's

number of trainable parameters consists of weights and bias units.

In a convolutional layer, the number of weights depends on the kernel's width and height and the number of input and output channels. The number of bias units depends on the number of output channels only. To illustrate the computation step-by-step, suppose we have a kernel width and height of 5, 1 input channel, and 1 output channel, as illustrated in the figure below. In this case, we have 26 parameters since we have $5 \times 5 = 25$ weights via the kernel plus the bias unit. The computation to compute an output value or pixel z is $z = b + \sum_j w_j x_j$, where x_j represents an input pixel, w_j represents a weight parameter of the kernel, and b is the bias unit.

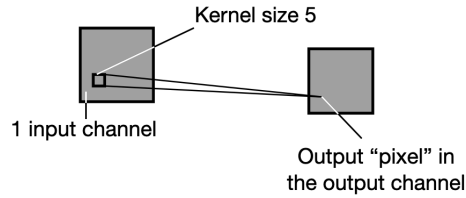


Figure 11.2. A convolutional layer with only one input and one output channel. The parameter count is 25.

Now, suppose we have 3 input channels as illustrated in the figure below. In that case, we compute the output value by performing the above-mentioned operation, $\sum_j w_j x_j$ for each input channel and then add the bias unit. For 3 input channels, this would involve three different kernels with three sets of weights: $z = \sum_j w_j^{(1)} x_j + \sum_j w_j^{(2)} x_j + \sum_j w_j^{(3)} x_j + b$. Since we have 3 sets of weights ($w^{(1)}, w^{(2)}$, and $w^{(3)}$ for $j = [1, \ldots, 25]$) we have $3 \times 25 + 1 = 76$ parameters in this convolutional layer.

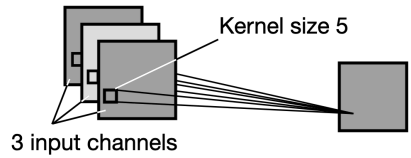


Figure 11.3. A convolutional layer with only three input and one output channel. The parameter count is 76.

We use one kernel for each output channel, where each kernel is unique to a given output channel. So, if we extend the number of output channels from 1 to 5 as shown in the figure below, we extend the number of parameters by a factor of 5. In other words, if the kernel for one output channel had 76 parameters, the 5 kernels required for the 5 output channels would have $5 \times 76 = 380$ parameters.

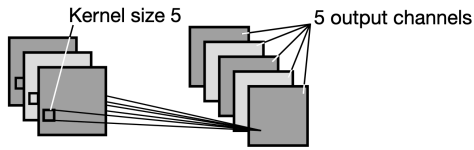


Figure 11.4. A convolutional layer with three input and five output channels. The parameter count is 76.

Now, returning to the neural network architecture illustrated at the beginning of this section, we compute the number of parameters in the convolutional layers as follows. For example, the first convolutional layer has 3 input channels, 5 output channels, and a kernel size of 5. Thus, its number of parameters is $5 \times (5 \times 5 \times 3) + 5 = 380$.

The second convolutional layer, with 5 input channels, 12 output channels, and a kernel size of 3, has $12 \times (3 \times 3 \times 5) + 12 = 552$ parameters. Since the pooling layers do not have any trainable parameters, we can count $380 + 552 = 932$ for the convolutional part of this architecture. Next, let's move on to the fully connected layers.

Counting the number of parameters in a fully connected layer is relatively straightforward. A fully connected node connects each

input node to each output node, so the number of weights is the number of inputs times the number of outputs plus the bias units added to the output. For example, if we have a fully connected layer with 5 inputs and 3 outputs as shown in the figure below, we thus have $5 \times 3 = 15$ weights and 3 bias units, that is, 18 parameters in total.

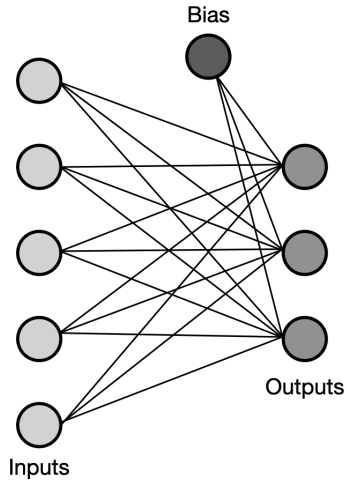


Figure 11.5. A fully connected layer with five inputs and three outputs. The parameter count is 18.

Returning once more to the neural network architecture illustrated at the beginning of this chapter, we can now calculate the parameters in the fully connected layers as follows: $192 \times 128 + 128 = 24,704$ in the first fully connected layer and $128 \times 10 + 10 = 1,290$ in the second fully connected layer, the output layer. Hence, we have $24,704 + 1,290 = 25,994$ in the fully connected part of this network.

Now, adding the 932 parameters from the convolutional layers and the 25,994 parameters from the fully connected layers, we can

conclude that this network's total number of parameters is 26,926[71].

Why do we care about the number of parameters after all? First, the number of parameters lets us estimate a model's complexity. As a rule of thumb, the more parameters we have, the more training data is recommended to train the model well.

Moreover, the number of parameters lets us estimate the size of the neural network, which helps estimate whether it can fit into GPU memory[72].

> **Reader quiz**:

11-A

Suppose we want to optimize the neural network using plain stochastic gradient descent (SGD) or an Adam optimizer. What are the respective numbers of parameters that need to be stored for SGD and Adam?

11-B

Suppose we are adding three batch normalization (BatchNorm) layers: one after the first convolutional layer, one after the second convolutional layer, and another one after the first fully connected layer – we typically do not want to add BatchNorm layers to the output layer. How many additional parameters do these three BatchNorm layers add to the model?

[71] As a bonus, interested readers can find PyTorch code in the supplementary materials of this book to compute the number of parameters programmatically: https://github.com/rasbt/MachineLearning-QandAI-book.

[72] The memory requirement during training often exceeds the model size due to the additional memory required for carrying out matrix multiplications and storing gradients. However, the model size can give us a rough ballpark estimate of whether training the model on a given hardware setup is feasible.

Q12. The Equivalence of Fully Connected and Convolutional Layers

> Q:

Under which circumstances are fully connected and convolutional layers equivalent?

> A:

There are exactly two scenarios: (1) the size of the convolutional filter is equal to the size of the receptive field, and (2) the size of the convolutional filter is one. To illustrate these two scenarios, consider a fully connected layer with two input and four output units, as shown in the following figure.

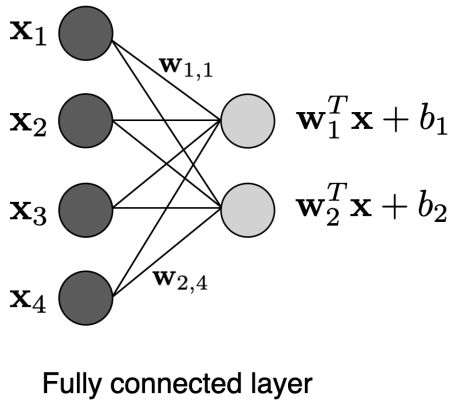


Figure 12.1. A fully connected layers with four inputs and two outputs.

The fully connected layer in the figure above consists of eight weights and two bias units. The first output node can be computed via the following dot product: $w_{1,1} \times x_1 + w_{1,2} \times x_2 + w_{1,3} \times x_3 + w_{1,4} \times x_4 + b_1$. Analagously, we can compute the second output unit via $w_{2,1} \times x_1 + w_{2,2} \times x_2 + w_{2,3} \times x_3 + w_{2,4} \times x_4 + b_2$.

Scenario 1: The kernel size is equal to the input size

Let us start with the first scenario: the size of the convolutional filter is equal to the size of the receptive field. Recall from Q11 how we compute a number of parameters in a convolutional kernel with one input channel and multiple output channels. We have a kernel size of 2×2, one input channel, and two output channels. The input size is also `2\times 2`,$ a reshaped version of the four inputs depicted in the previous figure.

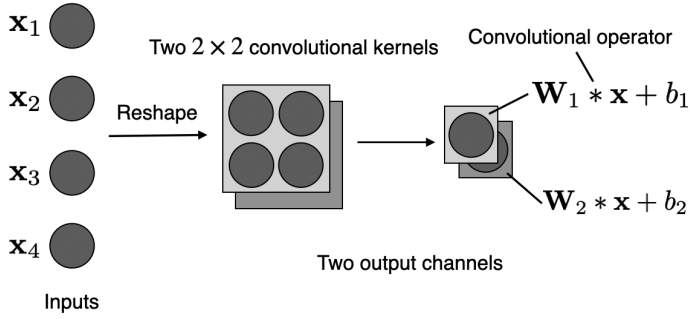


Figure 12.2. A convolutional layer with a $2times2$ kernel that equals the input size and two output channels.

If the convolutional kernel dimensions equal the input size, as depicted in the figure above, there is no sliding-window mechanism in the convolutional layer. For the first output channel, we have the set of weights

$$\mathbf{W}_1 = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{1,3} & w_{1,4} \end{bmatrix}.$$

And for the second output channel, we have the set of weights

$$\mathbf{W}_2 = \begin{bmatrix} w_{2,1} & w_{2,2} \\ w_{2,3} & w_{2,4} \end{bmatrix}$$

If the inputs are organized as

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix},$$

we calculate the first output channel as $o_1 = \sum_i (W_1 * \mathbf{x})_i + b_1$,

where the comvolutional operator $*$ is equal to an element-wise multiplication. In other words, we have an element-wise multiplication between two matrices, W_1 and $\mathbf{x}$, and then compute the output as the sum over these elements. Note that this equals the dot product in the fully connected layer. And lastly, we add the bias unit. The computation for the second output channel works analogously: $o_2 = \sum_i (W_2 * \mathbf{x})_i + b_2$. [73]

Scenario 2: The kernel has size one

The second scenario assumes that we reshape the input into an input "image" with 1×1 dimensions where the number of "color channels" equals the number of input features as depicted in the figure below.

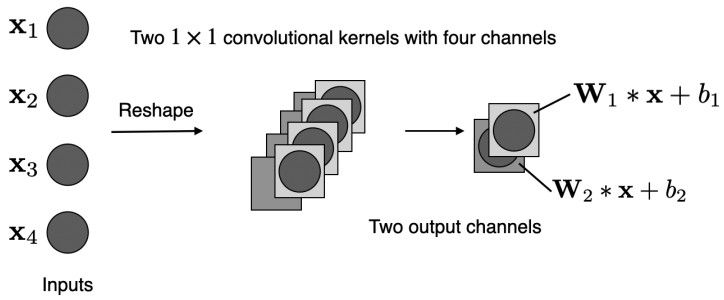


Figure 12.3. A convolutional layer with a $1times1$ kernel with four input and two output channels.

Each kernel consists of a stack of weights equal to the number of input channels. For instance, for the first output layer, the weights are $\mathbf{W}_1 = [w_1^{(1)} w_1^{(2)} w_1^{(3)} w_1^{(4)}]$, and the weights for the second channel are $\mathbf{W}_2 = [w_2^{(1)} w_2^{(2)} w_2^{(3)} w_2^{(4)}]$.

To get a better visual for this computation, it can be useful to check

[73] As a bonus, interested readers can find PyTorch code in the supplementary materials of this book to show this equivalence programmatically: https://github.com/rasbt/MachineLearning-QandAI-book.

out Q11[74].

How is this useful?

The fact that fully connected layers can be implemented as equivalent convolutional layers does not have immediate performance or other advantages. Still, it aids in understanding the mechanics of these layers. However, there may be special cases where hardware is better optimized for carrying out one operation over the other. Furthermore, if one so desires, it lets us implement convolutional neural networks without any use of fully connected layers.

Reader quiz:

12-1

How would increasing the stride affect the equivalence discussed above?

12-2

Does padding affect the equivalence between fully connected layers and convolutional layers?

[74] As a bonus, interested readers can find PyTorch code in the supplementary materials of this book to show this equivalence programmatically: https://github.com/rasbt/MachineLearning-QandAI-book.

Q13. Large Training Sets for Vision Transformers

> Q:

Why do vision transformers (ViTs) generally require larger training sets than convolutional neural networks (CNNs)?

> A:

Each machine learning algorithm and model encodes a particular set of assumptions or prior knowledge in its design. These assumptions are also commonly referred to as inductive biases. Some inductive biases are workarounds to make algorithms computationally more feasible, others are based on domain knowledge, and some are both.

CNNs have more inductive biases that are hard-coded as part of the algorithmic design, so they generally require less training data than ViTs. In a sense, ViTs are given more degrees of freedom and can or must learn certain inductive biases from the data (assuming that these biases are conducive to optimizing the training objective). But everything that needs to be learned requires more training examples.

Inductive biases

To better understand the inductive biases in CNNs, let us list the main ones below.

1. Local connectivity. In CNNs, each unit in a hidden layer is connected to only a subset of neurons in the previous layer. We can justify this restriction by assuming that neighboring pixels are more relevant to each other than pixels that are farther apart. Intuitively, we can think of this assumption in the context of recognizing edges or contours in an image.
2. Weight sharing. Via the convolutional layers, we use the same small set of weights (the kernels or filters) throughout

the whole image. This reflects the assumption that the same filters are useful for detecting the same patterns in different parts of the image.

3. Hierarchical processing. CNNs consist of multiple convolutional to extract features from the input image. As the network progresses from the input to the output layers, low-level features are successively combined to form increasingly complex features, ultimately leading to the recognition of more complex objects and shapes. Furthermore, the convolutional filters in these layers learn to detect specific patterns and features at different levels of abstraction.
4. Spatial invariance. This refers to the property that the output of a model remains consistent even if the input signal is shifted to a different location within the spatial domain. This characteristic arises from the combination of local connectivity, weight sharing, and the hierarchical architecture mentioned earlier.

The combination of local connectivity, weight sharing, and hierchical processing in a CNN leads to spatial invariance, allowing the model to recognize the same pattern or feature regardless of its location in the input image. Let us illustrate this further via translation invariance, which is a special case of spatial invariance where we are only focusing on shifting an object – we do not rotate it.

Translation invariance

Translation invariance is a specific case of spatial invariance where the output remains the same after a shift or translation of the input signal in the spatial domain. Now, convolutional layers and networks are not truly translation invariant but achieve a certain level of translation equivariance.

What is the difference between translation invariance and equivariance? Translation invariance means that the output does not change with an input shift, while translation equivariance implies

that the output shifts with the input in a corresponding manner. In other words, if we shift the input object to the right, the results will be correspondingly shifted to the right, as illustrated in the figure below.

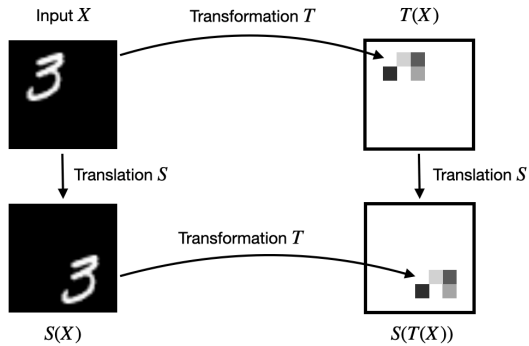


Figure 13.1. An illustration of equivariance demonstrated by depicting how an image transforms under various translations of the relevant object.

As mentioned earlier, CNNs achieve translation equivariance through a combination of their local connectivity, weight sharing, and hierarchical processing properties. The figure below depicts a convolutional operation to illustrate the local connectivity and weight-sharing priors.

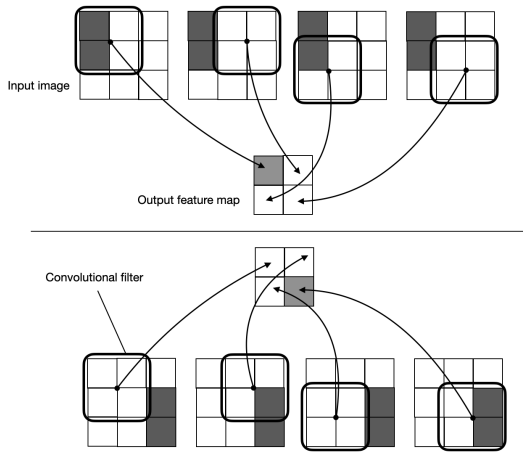


Figure 13.2. Illustration of translation equivariance where a convolutional filter captures the input signal (the two dark blocks) irrespective of where it is located in the input.

For comparison, a fully connected network such as a multilayer perceptron lacks this spatial invariance or equivariance. To illustrate this point, let us picture a multilayer perceptron with one hidden layer. Each pixel in the input image is connected with each value in the resulting output. If we shift the input by one or more pixels, a different set of weights will be activated, as illustrated in the figure below.

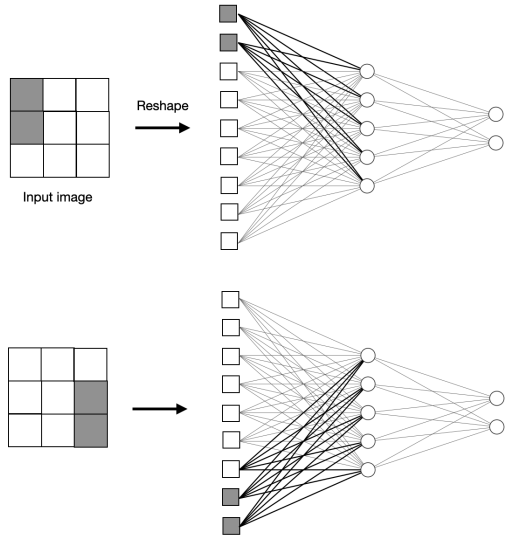


Figure 13.3. Fully connected layers have a unique weight for each input position, and specific weights will not be activated in the same way if the input location changes.

Like fully connected networks, the ViT architecture (and transformer architecture in general) lacks the inductive bias for spatial invariance or equivariance[75]. For instance, the model produces different outputs if we place the same object in two different spatial locations within an image. This is not ideal, as the semantic meaning of an object remains the same based on its location. Consequently, it must learn these invariances directly from the data. To facilitate the learning of useful patterns present in CNNs, pretraining over a larger dataset is required.

ViTs can still outperform CNNs

Now, the hard-coded assumptions via the inductive biases listed above reduce the number of parameters in CNNs substantially

[75]One workaround for adding translation invariance in ViTs is via relative positional embeddings that consider the relative distance between two tokens in the input sequence as proposed in *Self-Attention with Relative Position Representations* (2018) by Shaw, Uszkoreit, and Vaswani, https://arxiv.org/abs/1803.02155.

compared to fully connected layers[76]. Vice versa, ViTs tend to have larger numbers of parameters than CNNs, which require more training data.

ViTs may underperform compared to popular CNN architectures without extensive pretraining, but they can perform very well with a sufficiently large pretraining dataset. In contrast to language transformers, where unsupervised pretraining (i.e., self-supervised learning[77]) is a preferred choice, vision transformers are often pretrained using large labeled datasets such as ImageNet, which provides millions of labeled images for training, and regular supervised learning.

An example is the comparison made in the original ViT paper[78] between ResNet, a convolutional network, and the original ViT architecture for various dataset sizes used in pretraining. The research revealed that the ViT architecture outperformed the convolutional network only after pretraining on at least 100 million images.

Do ViTs learn differently?

Note that ViTs are not free from any inductive biases. For example, vision transformers "patchify" the input image to process each input patch individually. Here, each patch can attend to all other patches so the model learns relationships between far-apart patches in the input image.

[76](Q11) explains how to calculate the number of parameters in fully connected and convolutional layers.

[77]See Q2 for details about self-supervised learning.

[78]Dosovitskiy, Beyer, Kolesnikov, Weissenborn, Zhai, Unterthiner, Dehghani, Minderer, Heigold, Gelly, Uszkoreit, and Houlsby (2020) *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, https://arxiv.org/abs/2010.11929.

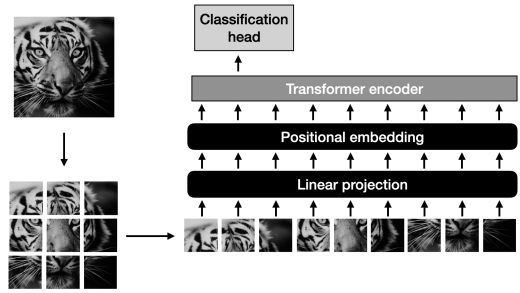


Figure 13.4. An illustration of how a vision transformer operates on image patches.

The *patchify* inductive bias is important because it allows ViTs to scale to larger image sizes without increasing the number of parameters in the model, which can be computationally expensive. By processing smaller patches individually, ViTs can efficiently capture spatial relationships between image regions while benefiting from the global context captured by the self-attention mechanism.

And this raises another question: how and what do ViTs learn from the training data? ViTs learn more uniform feature representations across all layers, with self-attention mechanisms enabling early aggregation of global information. In addition, the residual connections in ViTs strongly propagate features from lower to higher layers, in contrast to the more hierarchical structure of CNNs[79].

ViTs tend to focus more on global than local relationships because the self-attention mechanism in ViTs allows the model to consider long-range dependencies between different parts of the input image. Consequently, the self-attention layers in ViTs are often considered low-pass filters that focus more on shapes and curvature.

In contrast, the convolutional layers in CNNs are often considered high-pass filters that focus more on texture. But we should note that convolutional layers can act as both high-pass and low-pass filters, depending on the learned filters at each layer. High-pass

[79]Raghu, Unterthiner, Kornblith, Zhang, Dosovitskiy (2021). *Do Vision Transformers See Like Convolutional Neural Networks?*, https://arxiv.org/abs/2108.08810.

filters detect an image's edges, fine details, and texture, while low-pass filters capture more global, smooth features and shapes. CNNs achieve this by applying convolutional kernels of varying sizes and learning different filters at each layer.

ViTs or CNNs?

ViTs have recently begun outperforming CNNs if enough data is available for pretraining. However, this doesn't make CNNs obsolete, as methods such as EfficientNetV2 are less memory- and data-hungry.

Moreover, recent ViT architectures don't solely rely on large datasets, parameter numbers, and self-attention. Instead, they have taken inspiration from CNNs and added soft convolutional inductive biases[80] or even complete convolutional layers[81] to get the best of both worlds.

In conclusion, vision transformer architectures without convolutional layers generally have fewer spatial and locality inductive biases than convolutional neural networks. Consequently, vision transformers need to learn data-related concepts such as local relationships among pixels. Thus, vision transformers require more training data to achieve good predictive performance and produce acceptable visual representations in generative modeling contexts.

> Reader quiz:

13-A

Consider the *patchification* of the input images illustrated in the figure above. The size of the resulting patches controls a computational and predictive performance tradeoff. The optimal patch size depends on the specific application and desired trade-off between computational cost and model performance. Do smaller patches typically result in higher or lower computational costs?

[80] d'Ascoli, Touvron, Leavitt, Morcos, Biroli, Sagun (2021). *ConViT: Improving Vision Transformers with Soft Convolutional Inductive Biases*, https://arxiv.org/abs/2103.10697.

[81] Wu, Xiao, Codella, Liu, Dai, Yuan, Zhang (2021), *CvT: Introducing Convolutions to Vision Transformers*, https://arxiv.org/abs/2103.15808.

13-B

Following up on the question above, do smaller patches typically lead to a higher or lower prediction accuracy?

Chapter 3. Natural Language Processing

Q15. The Distributional Hypothesis

> Q:

What is the distributional hypothesis in NLP? Where is it used, and how far does it hold true?

> A:

The distributional hypothesis is a linguistic theory suggesting that words occurring in the same contexts tend to have similar meanings[82]. To put it succinctly, the more similar the meanings of two words are, the more often they appear in similar contexts.

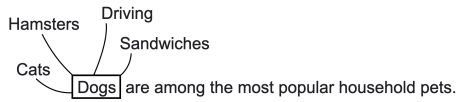


Figure 15.1. According to the distributional hypothesis, words that occur in similar contexts have a more similar meaning. So, for example, the words *Cats* and *Dogs* often occur in similar contexts and are more related (both are mammals and pets) than a *cat* and a *sandwich*.

Looking at large datasets, this may hold more or less, but it is easy to construct individual counter-examples. An intuitive counter-example is the phenomenon of polysemy, which is when a word has multiple meanings that are related but not identical. For example, let's consider the word *bank*. It can refer to a financial institution, the "rising ground bordering a river," the "steep incline of a hill," or a "protective cushioning rim"[83]. However, these different meanings have different distributional properties and may not always occur in similar contexts.

Nonetheless, the distributional hypothesis is quite useful. Word embeddings[84] such as Word2vec[85], and many large language trans-

[82]Original source: *Distributional Structure* (1954) by Harris, https://doi.org/10.1080/00437956.1954.11659520.

[83]See https://www.merriam-webster.com/dictionary/bank.

[84]See Q1 for a more in-depth discussion of embeddings.

[85]Mikolov, Chen, Corrado, Dean (2013). *Efficient Estimation of Word Representations in Vector Space*, https://arxiv.org/abs/1301.3781.

former models rely on this idea (for example, the masked language model in BERT[86] and the next-word pretraining task used in GPT[87]).

The Word2vec approach uses a simple, two-layer neural network to encode words into embedding vectors such that the embedding vectors of similar words are close (semantically and syntactically). There are two ways to train a Word2vec model: the continuous bag-of-words (CBOW) and the skip-gram approach.

Using the CBOW approach, Word2vec learns to predict the current words by using the surrounding context words. In the skip-gram model and vice versa, the Word2vec model predicts the context words from a selected word. In the continuous bag-of-words architecture, the Word2vec model predicts a masked word from the window of surrounding context words. While skip-gram is more effective for infrequent words, CBOW is usually faster to train.

After training, the word embeddings are placed within the vector space so that words with common contexts in the corpus, meaning words with semantic and syntactic similarities, are positioned close to each other. Conversely, dissimilar words are located further apart in the embedding space.

[86]Devlin, Chang, Lee, Toutanova (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, https://arxiv.org/abs/1810.04805.

[87]Radford, Narasimhan (2018). *Improving Language Understanding by Generative Pre-Training*, https://www.semanticscholar.org/paper/Improving-Language-Understanding-by-Generative-Radford-Narasimhan/cd18800a0fe0b668a1cc19f2ec95b5003d0a5035.

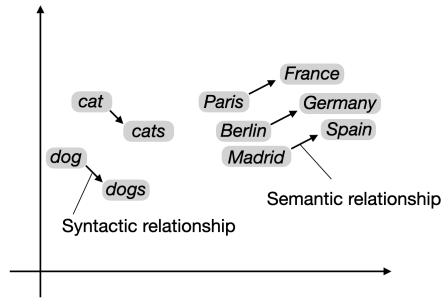


Figure 15.2. Illustration of Word2vec embeddings in a two-dimensional vector space.

BERT uses a masked language modeling approach that involves masking (hiding) some of the words in a sentence, and its task is to predict these masked words based on the other words in the sequence. This approach is a form of self-supervised learning used to pretrain large language models[88]. The pretrained model produces embeddings where similar words (or tokens) are close in the embedding space[89].

Regarding the masked language modeling task, BERT pretraining is conceptually similar to Word2vec's CBOW approach. However, CBOW is a simpler model that does not consider the order of words in a sentence – CBOW only considers the distributional patterns of words in a larger corpus.

Being a decoder instead of an encoder model, GPT takes a slightly different approach than BERT and learns to predict the following words in a sequence based on previous words. Where BERT is a bidirectional language model that considers the whole input sequence, GPT only strictly parses previous sequence elements. (This is why BERT is usually better suited for classification tasks, whereas GPT is more suited for text generation tasks.)
Similar to BERT, GPT produces high-quality contextualized word

[88]See Q2 for more information about self-supervised learning.

[89]Liu, Gardner, Belinkov, Peters, Smith (2019). *Linguistic Knowledge and Transferability of Contextual Representations*, https://arxiv.org/abs/1903.08855.

embeddings that capture semantic similarity[90].

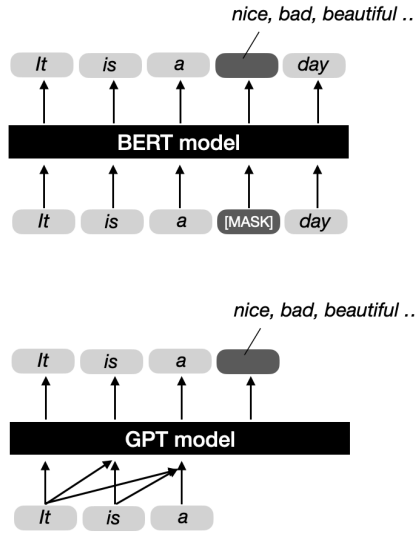


Figure 15.3. BERT's masked language modeling approach and GPT's next-word prediction for pretraining.

To summarize, while there are counter-examples where the distributional hypothesis does not hold, it is a very useful concept that forms the cornerstone of modern language transformer models.

> Reader quiz:

15-A

Consider the case of homophones – words that sound the same but have different meanings. For example, the words *there* and *their* sound the same but have different meanings. Are homophones another example of when the distributional hypothesis does not hold?

15-B

[90]Petroni, Rocktaeschel, Lewis, Bakhtin, Wu, Miller, Riedel (2019). *Language Models as Knowledge Bases?*, https://arxiv.org/abs/1909.01066.

Can you think of other domains where a concept similar to the distributional hypothesis applies?

Q16. Data Augmentation for Text

> Q:

What are common data augmentation techniques for text data?

> A:

Data augmentation is useful for artificially increasing the dataset size to improve model performance, for instance, by reducing the degree of overfitting, as discussed in Q5. Data augmentation is very common when developing computer vision models such as convolutional neural networks and vision transformers, where we use standard techniques such as rotation, scaling, flipping, cropping, and altering the brightness or contrast of an image.

Similarly, there exist several techniques for augmenting text data. The most common ones include synonym replacement, word deletion, word position swapping, sentence shuffling, noise injection, back translation, and text generated by large language models (LLMs). We will cover these different techniques in the sections below[91].

Synonym replacement

In synonym replacement, we randomly choose words in a sentence – often nouns, verbs, adjectives, and adverbs – and replace them via synonyms, as shown in the example below:

- Original sentence: "The cat quickly jumped over the lazy dog."
- Augmented sentence: "The cat rapidly jumped over the idle dog."

This can help the model to learn that different words can have similar meanings and thus improve its ability to understand and

[91]Optional code examples are provided in the supplementary material at https://github.com/rasbt/MachineLearning-QandAI-book.

generate text. However, using this technique carefully is essential, as not all synonyms are interchangeable in all contexts.

In practice, synonym replacement is often done by using a thesaurus such as WordNet[92].

Word deletion

Word deletion randomly removes certain words from the original text to create new variants while still trying to maintain the overall meaning of the sentence, as shown in the example below:

- Original sentence: "The cat quickly jumped over the lazy dog."
- Augmented sentence: "The cat quickly jumped over the lazy dog." (removed "quickly")

However, caution should be taken not to remove critical words that may significantly alter a sentence's meaning. For example, the following word deletion would be suboptimal:

- Original sentence: "The cat quickly jumped over the lazy dog."
- Augmented sentence: "The quickly jumped over the lazy dog." (removed "cat")

Also, the deletion rate should be carefully chosen to ensure the text still makes sense after words have been removed.

Word position swapping

Word position swapping, also known as word shuffling or permutation, is another method of text data augmentation where the positions of words in a sentence are swapped or rearranged to create new versions of the sentence.

Let's use the previous sentence, "The cat quickly jumped over the lazy dog," as an example again. By swapping the positions of some words, we might get the following:

[92]Miller 1995, *WordNet: A Lexical Database For English*, https://dl.acm.org/doi/10.1145/219717.219748.

- Original sentence: "The cat quickly jumped over the lazy dog."
- Augmented sentence: "Quickly the cat jumped the over lazy dog."

While these sentences may sound grammatically incorrect or strange in English, they can still provide valuable training information for data augmentation. This is because the model can still recognize the important words and their associations with each other.

However, this method has its limitations. For example, shuffling words too much or in certain ways can drastically change the meaning of a sentence or make it completely nonsensical.

Sentence shuffling

Sentence shuffling is a data augmentation technique where entire sentences within a paragraph or a document are rearranged to create new versions of the input text. This technique is particularly useful in tasks that deal with document-level analysis or paragraph-level understanding, such as document classification, topic modeling, or text summarization.

In contrast to the abovementioned techniques, such as word position swapping, word deletion, and synonym replacement, sentence shuffling maintains the internal structure of individual sentences. Still, it changes their order within the broader context.

In contrast, the word-based methods above, like word position swapping, can make sentences grammatically incorrect or change their meaning entirely if not applied carefully, as word choice or order is often critical to the meaning of sentences in many languages.

Sentence shuffling is useful when the order of sentences is not crucial to the overall meaning of the text. Still, it may not work well if the sentences are logically or chronologically connected. For example, consider the following paragraph:

- "I went to the supermarket."
- "Then, I bought ingredients to make pizza."
- "Afterwards, I made some delicious pizza."

When reshuffling it as follows, it disrupts the logical and temporal progression of the narrative:

- "Afterwards, I made some delicious pizza."
- "Then, I bought ingredients to make pizza."
- "I went to the supermarket."

Noise injection

Noise injection is an umbrella term that includes other techniques above to create variation in the text, such as synonym, replacement, word shuffling, word deletion, and sentence shuffling. However, it may also be used to refer to character-level techniques such as inserting random letters, characters, or typos. Examples of these techniques are shown below:

- **Random character insertion**: "The cat qzuickly jumped over the lazy dog." (Inserted a "z" in the word "quickly".)
- **Random character deletion**: "The cat quickl jumped over the lazy dog." (Deleted "y" from the word "quickly".)
- **Typo introduction**: "The cat qickuly jumped over the lazy dog." (Introduced a typo in "quickly", changing it to "qickuly".)

These modifications are beneficial for tasks that involve spell-checking and text correction. But it can also help make the model more robust to imperfect inputs.

Back translation

Back translation is one of the most widely used techniques to create variation in texts. Here, the sentence is first translated from the

original language into one or more different languages and then back into the original language. Translating back and forth often results in sentences semantically similar to the original sentence but introduces slight variations in structure, vocabulary, or grammar. This generates additional, diverse examples for training without altering the overall meaning.

Let's take the familiar sentence "The cat quickly jumped over the lazy dog" and look at an example of back translation using German as our intermediary language:

- First, we translate the sentence into German. We might get: "Die Katze sprang schnell über den faulen Hund."
- Then, we translate this German sentence back into English. We could get: "The cat jumped quickly over the lazy dog."

Please note that the degree of change in the sentence through back translation depends on the languages used and the specifics of the machine translation model. In this case, the sentence remained very similar. Still, in other cases or with other languages, you might see more significant changes in wording or sentence structure while still maintaining the same overall meaning.

Also note that this method requires access to reliable machine translation models or services, and care must be taken to ensure that the back-translated sentences retain the essential meaning of the original sentences.

Synthetic data

Synthetic data generation is an umbrella term that includes the abovementioned techniques, such as back translation, synonym replacement, word deletion, word position swapping, sentence shuffling, noise injection, and so forth. All these methods generate new data by making small changes to existing data, maintaining the overall meaning while creating something new.

Modern techniques to generate synthetic data now also include using LLMs, for example, GPT-4. We can use these models to either generate new data from scratch by using "complete the sentence" or "generate example sentences" prompts, among others. Or, we could use it as an alternative to back translation, prompting it to rewrite a sentence, as shown in the figure below.

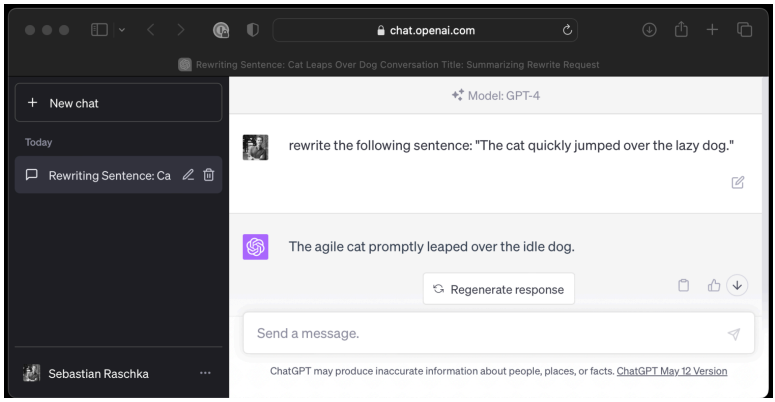


Figure 16.1. Using ChatGPT to rewrite sentences.

When to use data augmentation?

The data augmentation techniques discussed above, like synonym replacement, word deletion, and word position swapping, are commonly used in tasks such as text classification, sentiment analysis, and other NLP tasks where the amount of available labeled data might be limited.

However, when it comes to pretraining large language models (LLMs), they are usually trained on such a vast and diverse dataset that these augmentation techniques might not be used as extensively as in other, more specific NLP tasks. This is because LLMs aim to capture the statistical properties of the language, and the vast amount of data they are trained on often provides a sufficient variety of contexts and expressions.

However, in the finetuning stages of LLMs, where a pretrained

model is adapted to a specific task with a smaller, task-specific dataset, data augmentation techniques might become more relevant again, mainly if the task-specific labeled dataset size is limited.

> Reader quiz:

16-A

Could the use of text data augmentation help with privacy concerns?

16-B

What are some instances where data augmentation may not be beneficial for a specific task?

Q17. "Self"-Attention

> Q:

The scaled-dot product attention mechanism proposed by the original transformer architecture in the influential *Attention Is All You Need*[93] paper is often referred to as self-attention. Why is it called "self"-attention, and how is it different from regular attention?

> A:

It's called self-attention because it's an attention mechanism for all the elements of the same set. In contrast, the original attention developed for recurrent neural networks (RNNs) is applied between two different sequences, the encoder and the decoder embeddings.

Attention in RNNs

It may be easiest to summarize the difference between regular attention and self-attention with an illustration. The following figure depicts the attention mechanism that was originally proposed for RNNs to deal with long sequences[94].

[93] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin (2017). *Attention Is All You Need*, https://arxiv.org/abs/1706.03762

[94] Bahdanau, Cho, and Bengio (2014). *Neural Machine Translation by Jointly Learning to Align and Translate*, https://arxiv.org/abs/1409.0473.

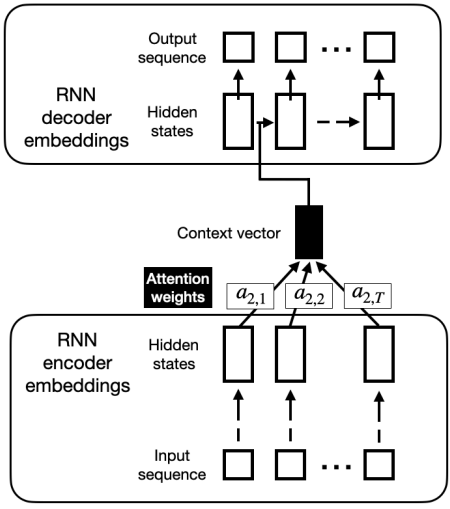


Figure 17.1. Illustration of the attention mechanism, which was originally proposed for RNNs. The values $alpha_{2,T} to\ alpha_{2,T}$ represent the attention weights for the second sequence element and each other element in the sequence from 1 to *T*.

As shown in the figure above, the original attention mechanism developed for RNNs involves two RNNs. The RNN at the bottom, computing the attention weights, represents the encoder part. The RNN at the top, producing the output sequence, is a decoder.

The takeaway is that the original attention mechanism developed for RNNs is applied between two different sequences, the encoder and the decoder embeddings. Note that for each generated output sequence element, the decoder RNN at the top is based on a hidden state plus a context vector generated by the encoder. This context vector involves *all* elements of the input sequence. This context vector is a weighted sum of all input elements where the attention scores ($\alpha's$) represent the weighting coefficients. This allows the decoder to access all input sequence elements (the context) at each step. And the key idea is that the attention weights (and context) may differ and change dynamically at each step.

What motivates this complicated setup? The reason behind this

encoder-decoder design is that we cannot translate sentences word by word. This would result in grammatically incorrect outputs, as illustrated by the RNN architecture A in the figure below.

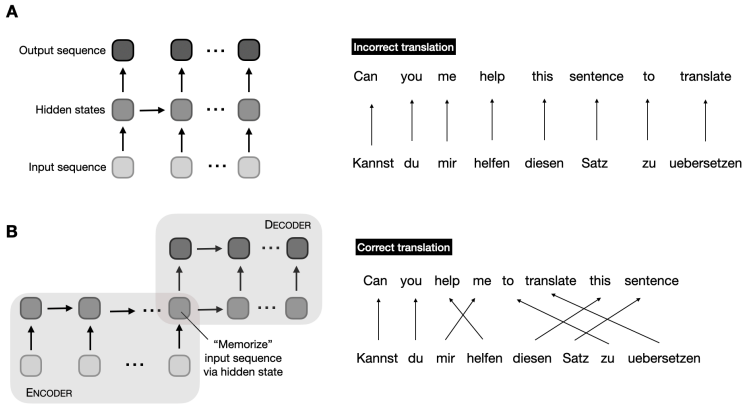


Figure 17.2. Illustration of two different sequence-to-sequence RNN designs for sentence translation. Subfigure A represents a regular sequence-to-sequence RNN that may be used to translate a sentence from German to English word by word. Subfigure B depicts an encoder-decoder RNN that first reads the whole sentence before translating it.

The RNN architecture setup depicted in subfigure A is more suited for time series tasks where we want to make one prediction at a time, for example, predicting a given stock price day by day. For tasks like language translation, we typically opt for an encoder-decoder RNN as depicted in subfigure B above. Here, the RNN encodes the input sentence, stores it in an intermediate hidden representation, and then generates the output sentence. However, this creates a bottleneck where the RNN has to memorize the whole input sentence via a single hidden state, which does not work very well for longer sequences.

The bottleneck depicted in subfigure B above prompted the attention mechanism's original design, allowing the decoder part to access all elements in the input sentence at each time step. And the attention scores give different weights to the different input elements depending on the current word that the decoder

generates. For example, when generating the word "help" in the output sequence, the word "helfen" in the German input sentence may get a large attention weight as it's highly relevant in this context.

Self-attention

Approximately three years after the attention-for-RNN mechanism, which we discussed above, researchers asked whether the RNN backbone is even needed. This led to the design of the original transformer architecture and self-attention mechanism[95].

In self-attention, the attention mechanism is applied between all elements in the same sequence (as opposed to involving two sequences). This is depicted in the simplified attention mechanism in the figure below. Similar to the attention mechanism for RNNs, the context vector is an attention-weighted sum over the input sequence elements.

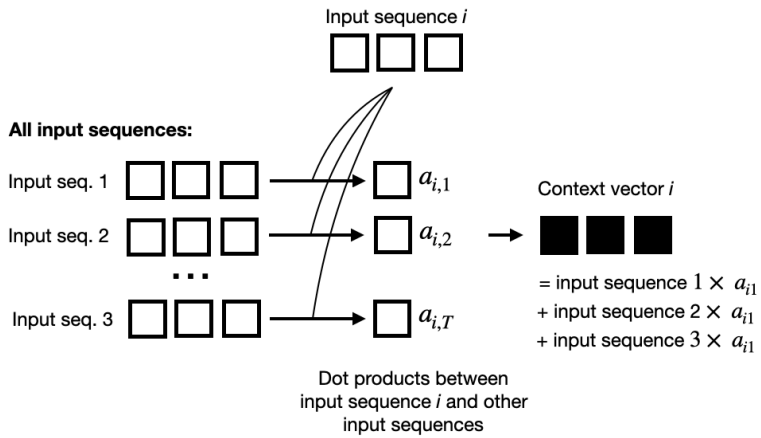


Figure 17.3. A simple self-attention mechanism without weight matrices.

While the figure above depicts a simple attention mechanism without weight matrices, the self-attention mechanism used in trans-

[95] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin (2017). *Attention Is All You Need*, https://arxiv.org/abs/1706.03762

formers typically involves multiple weight matrices to compute the attention weights. Furthermore, cross-attention is a flavor of self-attention that can be applied to different sequence elements[96].

> **Reader quiz**:

17-A

Considering that self-attention compares each sequence element with itself, what is the time and memory complexity of self-attention?

17-B

We discussed self-attention in the context of natural language processing. Could this mechanism be useful for computer vision applications as well?

[96] If you are in learning more about the parameterized self-attention and cross-attention mechanism, I recommed checking out my blog post *Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch* at https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html.

Q18. Encoder- And Decoder-Style Transformers

> Q:

What are the differences between encoder- and decoder-based language transformers?

> A:

Fundamentally, both encoder- and decoder-style architectures use the same self-attention layers to encode word tokens. However, the main difference is that encoders are designed to learn embeddings that can be used for various predictive modeling tasks such as classification. In contrast, decoders are designed to e new texts, for example, answering user queries.

The Original Transformer

The original transformer architecture[97], which was developed for English-to-French and English-to-German language translation, utilized both an encoder and a decoder, as illustrated in the figure below.

[97]Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin (2017). *Attention Is All You Need*, https://arxiv.org/abs/1706.03762

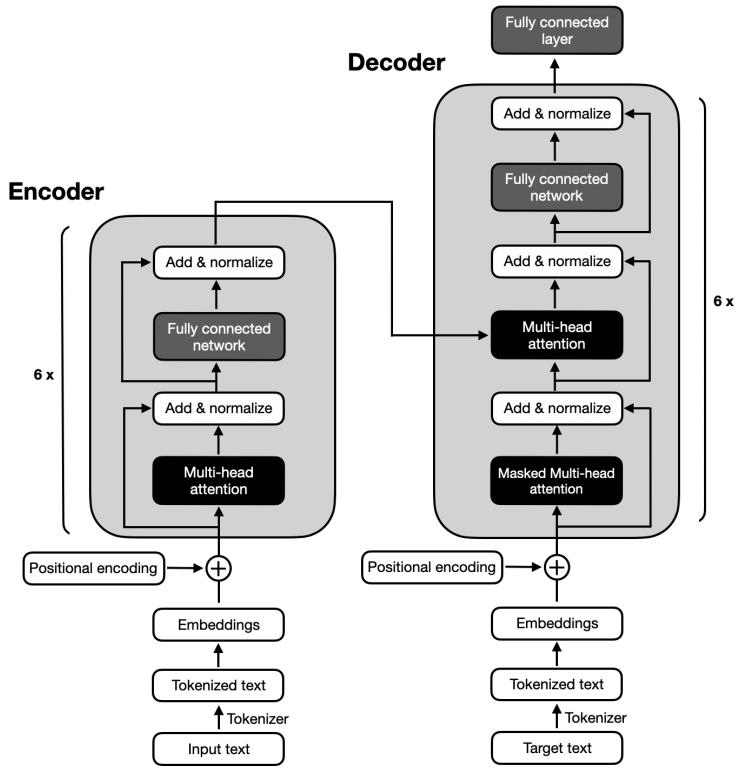


Figure 18.1. Illustration of the original transformer architecture.

In the figure above, the input text (that is, the sentences of the text that is to be translated) is first tokenized into individual word tokens, which are then encoded via an embedding layer[98] before it enters the decoder part. Then, after adding a positional encoding vector to each embedded word, the embeddings go through a multi-head self-attention layer. The multi-head attention layer is followed by an "Add & normalize" step, which performs a layer normalization and adds the original embeddings via a skip connection (also known as a residual or shortcut connection). Finally, after entering a "fully connected layer," which is a small multilayer perceptron

[98] See Q1 for more details about embeddings.

consisting of two fully connected layers with a nonlinear activation function in between, the outputs are again added and normalized before they are passed to a multi-head self-attention layer of the decoder part.

The decoder part in the figure above has a similar overall structure as the encoder part. The key difference is that the inputs and outputs are different. The encoder receives the input text that is to be translated, and the decoder generates the translated text.

Encoders

The encoder part in the original transformer, illustrated in the preceding figure, is responsible for understanding and extracting the relevant information from the input text. It then outputs a continuous representation (embedding) of the input text that is passed to the decoder. Finally, the decoder generates the translated text (target language) based on the continuous representation received from the encoder.

Over the years, various encoder-only architectures have been developed based on the encoder module of the original transformer model outlined above. Notable examples include BERT[99] and RoBERTa[100].

BERT (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) is an encoder-only architecture based on the Transformer's encoder module. The BERT model is pretrained on a large text corpus using masked language modeling (illustrated in the figure below) and next-sentence prediction tasks.

[99] *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* (2018). Devlin, Chang, Lee, and Toutanova, https://arxiv.org/abs/1810.04805.

[100] *RoBERTa: A Robustly Optimized BERT Pretraining Approach* (2018). Liu, Ott, Goyal, Du, Joshi, Chen, Levy, Lewis, Zettlemoyer, and Stoyanov https://arxiv.org/abs/1907.11692.

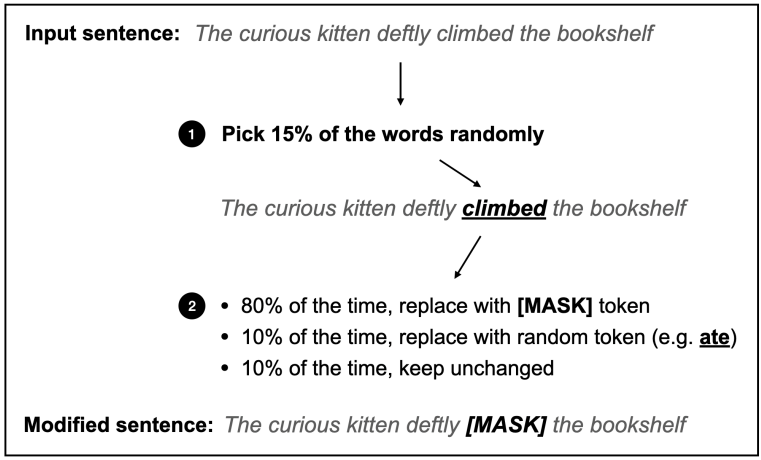


Figure 18.2. Illustration of the masked language modeling pretraining objective used in BERT-style transformers.

The main idea behind masked language modeling is to mask (or replace) random word tokens in the input sequence and then train the model to predict the original masked tokens based on the surrounding context.

Next to the masked language modeling pretraining task illustrated in the figure above, the next-sentence prediction task asks the model to predict whether the original document's sentence order of two randomly shuffled sentences is correct. For example, two sentences, in random order, are separated by the [SEP] token:

- [CLS] Toast is a simple yet delicious food [SEP] It's often served with butter, jam, or honey.
- [CLS] It's often served with butter, jam, or honey. [SEP] Toast is a simple yet delicious food.

The [CLS] token is a placeholder token for the model, prompting the model to return a *True* or *False* label indicating whether the sentences are in the correct order or not.

The masked language and next-sentence pretraining objective[101] allow BERT to learn rich contextual representations of the input texts, which can then be finetuned for various downstream tasks like sentiment analysis, question-answering, and named entity recognition.

RoBERTa (**R**obustly **o**ptimized **BERT** **a**pproach) is an optimized version of BERT. It maintains the same overall architecture as BERT but employs several training and optimization improvements, such as larger batch sizes, more training data, and eliminating the next-sentence prediction task. These changes resulted in RoBERTa achieving better performance on various natural language understanding tasks than BERT.

Decoders

Coming back to the original transformer architecture outlined at the beginning of this section, the multi-head self-attention mechanism in the decoder is similar to the one in the encoder, but it is masked to prevent the model from attending to future positions, ensuring that the predictions for position i can depend only on the known outputs at positions less than i. As illustrated in the figure below, the decoder generates the output word by word.

[101] It shall be noted that this pretraining is a form of self-supervised learning (see Q2 for more details).

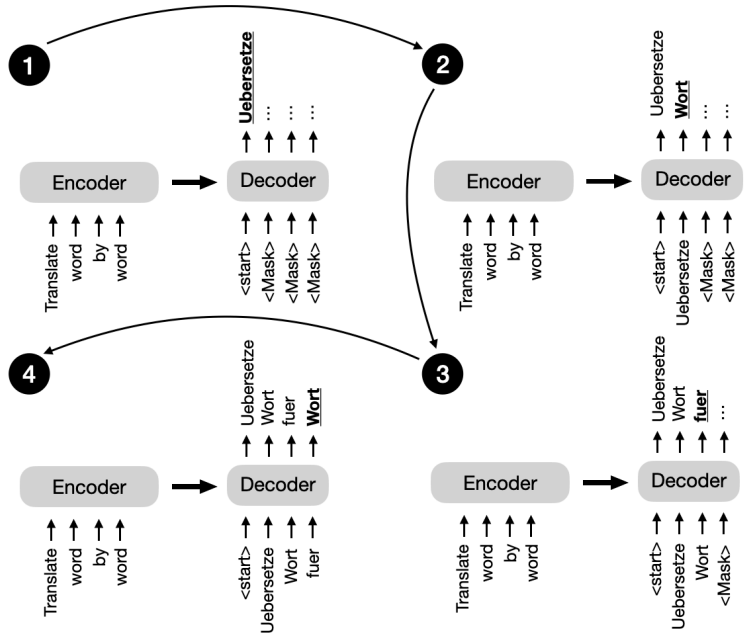


Figure 18.3. Illustration of the next-sentence prediction task used in the original transformer.

This masking (shown explicitly in the figure above, although it happens internally in the decoder's multi-head self-attention mechanism) is essential to maintain the autoregressive property of the transformer model during training and inference. The autoregressive property ensures that the model generates output tokens one at a time and uses previously generated tokens as context for generating the next word token.

Over the years, researchers have built upon the original encoder-decoder transformer architecture and developed several decoder-only models that have proven to be highly effective in various natural language processing tasks. The most notable models include the GPT family.

The GPT (**G**enerative **P**re-trained **T**ransformer) series are decoder-

only models pretrained on large-scale unsupervised text data and finetuned for specific tasks such as text classification, sentiment analysis, question-answering, and summarization. The GPT models, including GPT-2, GPT-3[102], and the more recent GPT-4, have shown remarkable performance in various benchmarks and are currently the most popular architecture for natural language processing.

One of the most notable aspects of GPT models is their emergent properties. Emergent properties refer to the abilities and skills that a model develops due to its next-word prediction pretraining. Even though these models were only taught to predict the next word, the pretrained models are capable of text summarization, translation, summarization, question answering, classification, and more. Furthermore, these models can perform new tasks without updating the model parameters via in-context learning, which is discussed in more detail in Q19.

Encoder-Decoder Hybrids

Next to the traditional encoder and decoder architectures, there have been advancements in the development of new encoder-decoder models that leverage the strengths of both components. These models often incorporate novel techniques, pre-training objectives, or architectural modifications to enhance their performance in various natural language processing tasks. Some notable examples of these new encoder-decoder models include BART[103] and T5[104].

Encoder-decoder models are typically used for natural language processing tasks that involve understanding input sequences and

[102]Brown et al. (2020). *Language Models are Few-Shot Learners*, https://arxiv.org/abs/2005.14165

[103]Lewis, Liu, Goyal, Ghazvininejad, Mohamed, Levy, Stoyanov, and Zettlemoyer (2018). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*, https://arxiv.org/abs/1910.13461.

[104]Raffel, Shazeer, Roberts, Lee, Narang, Matena, Zhou, Li, and Liu (2019). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, https://arxiv.org/abs/1910.10683.

generating output sequences, often with different lengths and structures. They are particularly good at tasks where there is a complex mapping between the input and output sequences and where it is crucial to capture the relationships between the elements in both sequences. Some common use cases for encoder-decoder models include text translation and summarization.

Terminology and Jargon

All of these methods, encoder-only, decoder-only, and encoder-decoder models, are sequence-to-sequence models (often abbreviated as *seq2seq*). Note that while we refer to BERT-style methods as encoder-only, the description *encoder-only* may be misleading since these methods also *decode* the embeddings into output tokens or text during pretraining.

In other words, both encoder-only and decoder-only architectures are "decoding." However, the encoder-only architectures, in contrast to decoder-only and encoder-decoder architectures, are not decoding in an autoregressive fashion. Autoregressive decoding refers to generating output sequences one token at a time, conditioning each token on the previously generated tokens. Encoder-only models do not generate coherent output sequences in this manner. Instead, they focus on understanding the input text and producing task-specific outputs, such as labels or token predictions.

Conclusion

In brief, encoder-style models are popular for learning embeddings used in classification tasks, encoder-decoder-style models are used in generative tasks where the output heavily relies on the input (for example, translation and summarization), and decoder-only models are used for other types of generative tasks including Q&A. Since the first transformer architecture emerged, hundreds of encoder-only, decoder-only, and encoder-decoder hybrids have been developed, as summarized in the figure below.

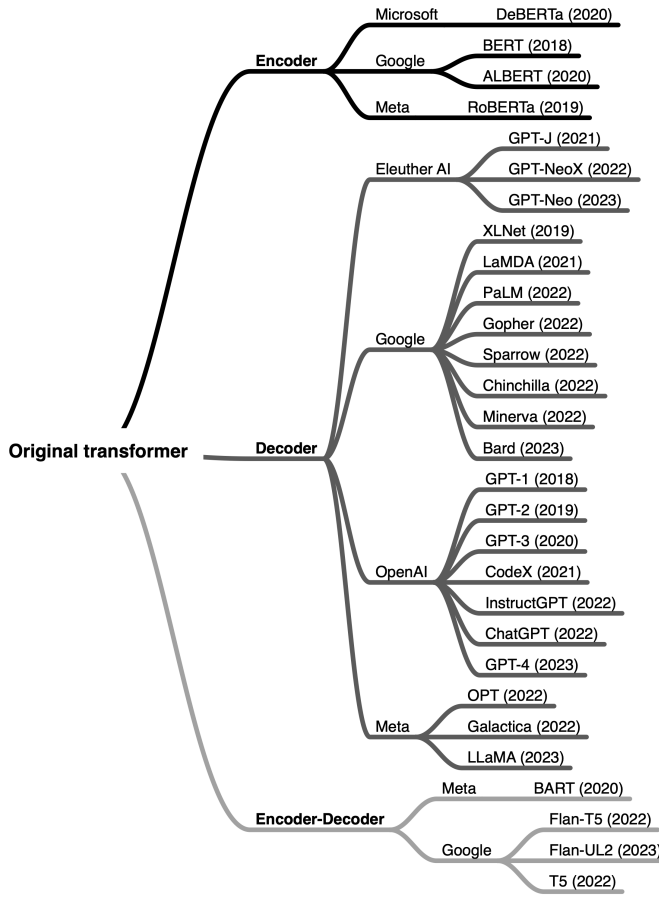


Figure 18.4. An overview of some of the most popular large language transformers organized by architecture type and developers.

While encoder-only models gradually lost in popularity, decoder-only models like GPT exploded in popularity thanks to breakthrough in text generation via GPT-3, ChatGPT, and GPT-4. However, encoder-only models are still very useful for training predictive models based on text embeddings versus generating texts.

> Reader quiz:

18-A

As discussed earlier, BERT-style encoder models are pretrained using masked language modeling and next-sentence prediction pretraining objectives. How could we adopt such a pretrained model for a classification task, for example, predicting whether a text has a positive or negative sentiment?

18-B

Can we finetune or a decoder-only model like GPT for classification?

Q19. Using and Finetuning Pretrained Transformers

> Q:

What are the different ways we can use and finetune pretrained large language models (LLMs)?

> A:

Firstly, most pretrained LLMs or language transformers can be utilized without the need for further finetuning. For instance, we can employ a feature-based method to train a new downstream model (for example, a linear classifier) using embeddings generated by a pretrained transformer.
Secondly, we can showcase examples of a new task within the input itself, which means we can directly exhibit the expected outcomes without requiring any updates or learning from the model.

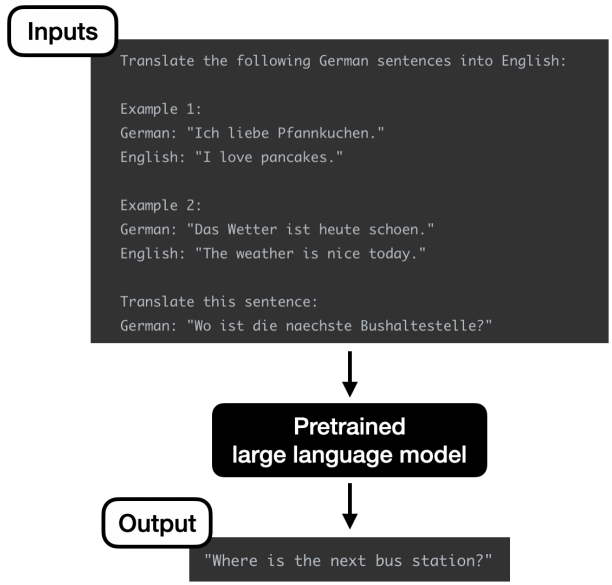


Figure 19.1. An illustration of prompting, which doesn't require model tuning. Here, we provide examples of the target task via the input.

Thirdly, it's possible to finetune all or just a small number of parameters to achieve the desired outcomes. The following sections will outline these different types of approaches.

The Three Conventional Methods

Let us start with the conventional methods for utilizing pretrained transformers: training another model on feature embeddings, finetuning output layers, and finetuning all layers. We will be discussing these in the context of classification.

In the feature-based approach, we load the pretrained model and keep it "frozen" – this means we do not update any parameters of the pretrained model. Instead, we treat the model as a feature extractor that we apply to our new dataset. Then, we train a downstream model on these embeddings. This downstream model can be any model we like (random forests, XGBoost, etc.), but linear classifiers typically perform best. This is likely because pretrained

transformers like BERT and GPT already extract high-quality, informative features from the input data. These feature embeddings often capture complex relationships and patterns, making it easy for a linear classifier to effectively separate the data into different classes. Furthermore, linear classifiers, such as logistic regression or support vector machines, tend to have strong regularization properties. These regularization properties help prevent overfitting when working with high-dimensional feature spaces generated by pretrained transformers. This feature-based approach is the most efficient method since it doesn't require updating the transformer model at all. Furthermore, the embeddings can be pre-computed for a given training dataset (since they don't change) when training a classifier for multiple training epochs.

When we talk about finetuning pretrained LLMs, the conventional methods include updating only the output layers (we will refer to this method as finetuning I) and updating all layers (finetuning II).

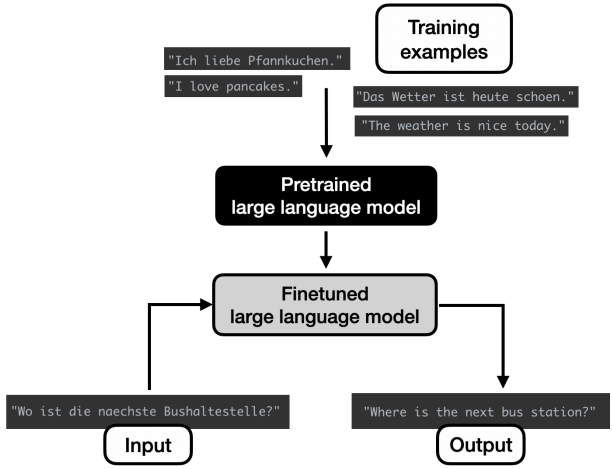


Figure 19.2. An illustration of the general finetuning worfklow of large language models.

Finetuning I is related to the feature-based approach above but adds one or more output layers to the LLM itself. The backbone of the

LLM remains frozen, and we only update the model parameters in these new layers. We don't need to backpropagate through the whole network, so this approach is relatively efficient regarding throughput and memory requirements.

In finetuning II, we load the model and add one or more output layers, similar to finetuning I. But instead of only backpropagating through the last layers, we update *all* layers via backpropagation, making this the most expensive approach. However, while this method is computationally more expensive than the feature-based approach and finetuning I, it typically leads to better modeling or predictive performance. This is especially true for more specialized domain-specific datasets. The following figure below summarizes the three approaches described above.

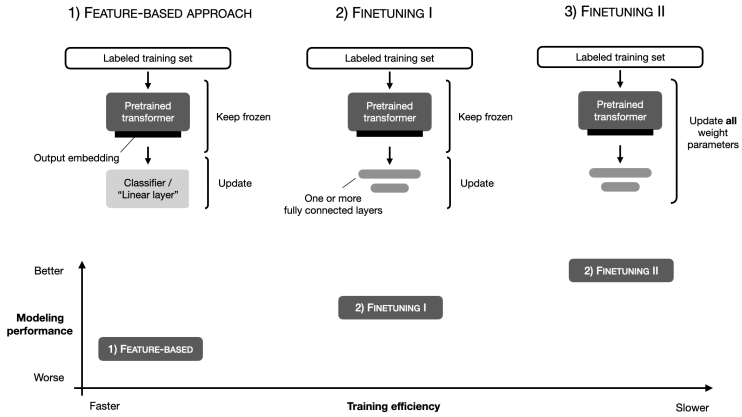


Figure 19.3. The three conventional approaches for utilizing pretrained LLMs.

In-Context Learning, Indexing, and Prompt Tuning

LLMs like GPT-2 and GPT-3[105] popularized the concept of in-context learning, often called zero or few-shot learning in this

[105] *Language Models are Unsupervised Multitask Learners* (2018), https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe. and *Language Models are Few-Shot Learners* (2020), https://arxiv.org/abs/2005.14165.

context. However, the in-context learning-based definition of few-shot learning differs from the conventional approach to few-shot learning[106].

The idea behind in-context learning is to provide context or examples of the task within the input or prompt, allowing the model to infer the desired behavior and generate appropriate responses. This approach takes advantage of the model's ability to learn from vast amounts of data during pretraining, which includes diverse tasks and contexts.

For example, suppose we want to use in-context learning for few-shot German-English translation using a large-scale pretrained language model like GPT-3. To do so, we provide a *few* examples of German-English translations to help the model understand the desired task, as shown below:

```
Translate the following German sentences into English:

Example 1:
German: "Ich liebe Pfannkuchen."
English: "I love pancakes."

Example 2:
German: "Das Wetter ist heute schoen."
English: "The weather is nice today."

Translate this sentence:
German: "Wo ist die naechste Bushaltestelle?"
```

Generally, in-context learning does not perform as well as finetuning for certain tasks or specific datasets. This is because in-context learning relies on the pretrained model's ability to generalize from its training data without further adapting its parameters for the particular task at hand.

[106] See Q3 for more details about few-shot learning.

However, it's worth noting that in-context learning has its advantages. It can be particularly useful when labeled data for finetuning is limited or unavailable. It also enables rapid experimentation with different tasks without finetuning the model parameters in cases where we don't have direct access to the model or only interact with the model through a UI or API (for example, ChatGPT).

Related to in-context learning is the concept of "hard" prompt tuning. Where the previously described finetuning methods update the model parameters to perform better at a task at hand, hard prompt tuning aims to optimize the prompt itself to achieve better performance. Prompt tuning does not modify the model parameters but may involve using a smaller labeled dataset to identify the best prompt formulation for the specific task. For example, to improve the prompts for the German-English translation task above, we may try the following three prompting variations:

```
1) "Translate the German sentence '{german_sentence}' int\
o English: {english_translation}"

2) "German: '{german_sentence}' | English: {english_trans\
lation}"

3) "From German to English: '{german_sentence}' -> {engli\
sh_translation}"
```

Prompt tuning is a resource-efficient alternative to parameter finetuning. However, the performance of prompt tuning is usually not as good as full model finetuning, as it does not update the model's parameters for a specific task, potentially limiting its ability to adapt to task-specific nuances. Furthermore, prompt tuning can still be labor intensive since it requires either human involvement comparing the quality of the different prompts[107] – this is often known

[107]The recent Auto Prompt Engineer method by Zhou et al. (2022) proposes to use another LLM for automatic prompt generation and evaluation: *Large Language Models Are Human-Level Prompt Engineers*, https://arxiv.org/abs/2211.01910.

as *hard* prompting since the input tokens are not differentiable. In contrast, *soft* prompting strategies such as *prefix tuning* optimize embedded versions of the prompts, which we will discuss in the next section on *parameter-efficient finetuning*.

Yet another way to leverage a purely in-context learning-based approach is indexing[108]. In the context of LLMs, we can think of indexing as a workaround based on in-context learning that allows us to turn LLMs into information retrieval systems to extract information from external resources and websites. Here, an indexing module parses a document or website into smaller chunks, embedded into vectors that can be stored in a vector database. Then, when a user submits a query, the indexing module computes the vector similarity between the embedded query and each vector stored in the database. Finally, the indexing module retrieves the top k most similar embeddings to synthesize the response.

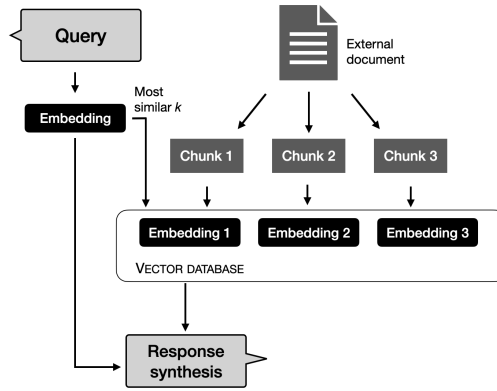


Figure 19.4. LLM-indexing to retrieve information from external documents.

Parameter-Efficient Finetuning

In recent years, many methods have been developed to adapt pretrained transformers more efficiently for new target tasks. These methods are commonly referred to as parameter-efficient finetun-

[108] https://github.com/jerryjliu/llama_index

ing (PEFT), with the currently most popular methods[109] summarized in the figure below.

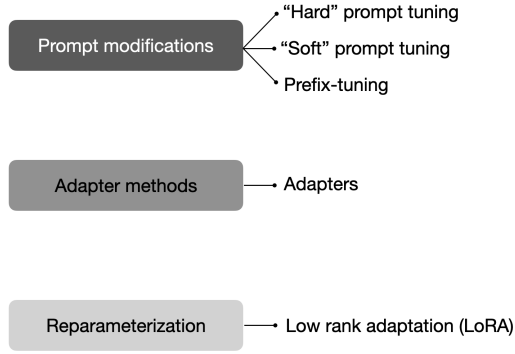


Figure 19.5. The main categories of parameter-efficient finetuning techniques with the most popular examples.

The prompt tuning approach mentioned in the previous section (*In-Context Learning and Prompt Tuning*) is often also referred to as "hard" prompt tuning to distinguish it from other "soft" prompting methods[110]. As described earlier, we modify the discrete input tokens in hard prompt tuning. In soft prompt tuning, we utilize trainable parameter tensors instead.

The main idea behind soft prompt tuning is to prepend a trainable parameter tensor (the "soft prompt") to the embedded query tokens. The prepended tensor is then tuned to improve the modeling performance on a target dataset using gradient descent. In pseudo-code, soft prompt tuning can be described as follows,

[109] Interested readers can find a survey of more than 40 research papers covering various PEFT methods in Lialin, Deshpande, and Rumshisky (2023), *Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning*, https://arxiv.org/abs/2303.15647.

[110] One of the first instances of soft prompting is described in Lester, Al-Rfou, and Constant (2021), *The Power of Scale for Parameter-Efficient Prompt Tuning*, https://arxiv.org/abs/2104.08691.

```
x = EmbeddingLayer(input_ids)
x = concatenate([soft_prompt_tensor, x],
                dim=seq_len)
output = model(x)
```

where the `soft_prompt_tensor` has the same feature dimension as the embedded inputs produced by the embedding layer.

Prefix tuning[111] is a popular prompt tuning method related to the soft prompt tuning approach mentioned above. The main difference between soft prompt tuning and prefix tuning is that in prefix tuning, we prepend trainable tensors (soft prompts) to each transformer block instead of only the embedded inputs, as illustrated in the next figure and pseudocode below:

```
def transformer_block_with_prefix(x):
    soft_prompt = FullyConnectedLayers(
      soft_prompt)  # Prefix
    x = concatenate([soft_prompt, x],  # Prefix
                    dim=seq_len)       # Prefix
    residual = x
    x = SelfAttention(x)
    x = LayerNorm(x + residual)
    residual = x
    x = FullyConnectedLayers(x)
    x = LayerNorm(x + residual)
    return x
```

The advantage of prefix tuning is that it can stabilize the training.

[111] Li and Liang (2021). *Prefix-Tuning: Optimizing Continuous Prompts for Generation*, https://arxiv.org/abs/2101.00190.

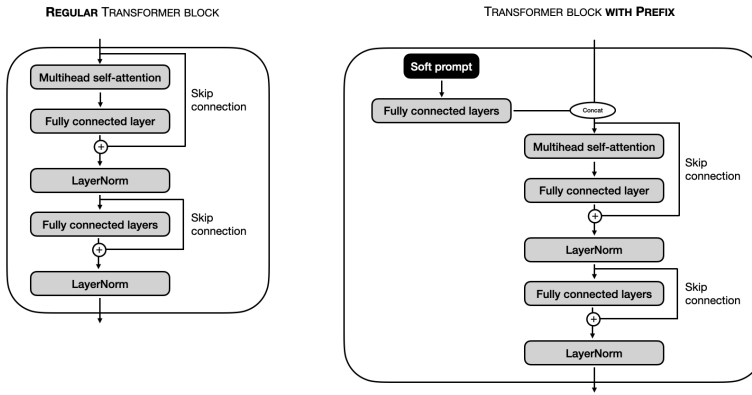


Figure 19.6. Outline of prefix tuning.

Both soft prompt tuning and prefix tuning are considered parameter-efficient since they only require training the prepended parameter tensors, not the LLM parameters themselves.

Adapter methods are related to prefix tuning in that they add additional parameters to the transformer layers. In the original adapter method[112], additional fully connected layers were added after the multihead self-attention and existing fully connected layers in each transformer block, as illustrated in the figure below. Only the new adapter layers are updated when training the LLM using the adapter method, while the remaining transformer layers remain frozen. Since the adapter layers are usually small – the first fully connected layer in an adapter block projects its input into a low-dimensional representation and the second layer projects it back into the original input dimension – the adapter method is usually considered parameter-efficient.

[112]Houlsby, Giurgiu, Jastrzebski, Morrone, Laroussilhe, Gesmundo, Attariyan, and Gelly (2019). Parameter-Efficient Transfer Learning for NLP, https://arxiv.org/abs/1902.00751.

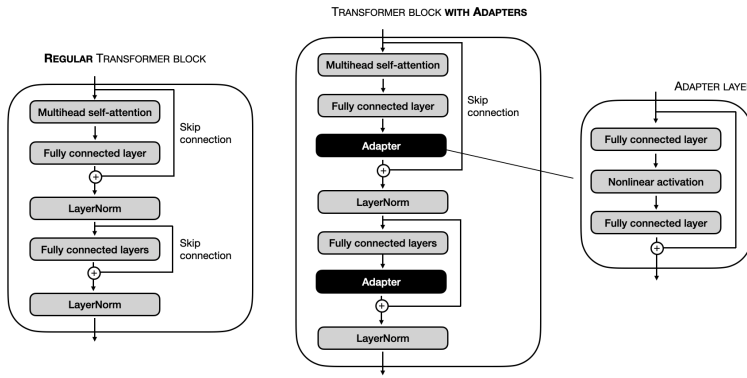


Figure 19.7. A comparison between a regular transformer block (left) and a transformer block with adapter layers.

In pseudo-code, the adapter method can be written as follows:

```
def transformer_block_with_adapter(x):
    residual = x
    x = SelfAttention(x)
    x = FullyConnectedLayers(x)  # adapter
    x = LayerNorm(x + residual)
    residual = x
    x = FullyConnectedLayers(x)
    x = FullyConnectedLayers(x)  # adapter
    x = LayerNorm(x + residual)
    return x
```

Low-rank adaptation (LoRA)[113] is another popular PEFT method worth considering. *Low-rank* adaptation refers to reparameterizing pretrained LLM weights using low-rank transformations.

Low-rank transformation is a technique to approximate a high-dimensional matrix or dataset using a lower-dimensional representation. The lower-dimensional representation (or low-rank approximation) is achieved by finding a combination of fewer dimensions

[113]Hu, Shen, Wallis, Allen-Zhu, Li, Wang, Wang, and Chen (2021). *LoRA: Low-Rank Adaptation of Large Language Models*, https://arxiv.org/abs/2106.09685.

that can effectively capture most of the information in the original data[114].

For example, suppose ΔW represents the parameter update for a weight matrix of the LLM with dimension $\mathbb{R}^{A \times B}$.. We can decompose the weight update matrix into two smaller matrices: $\Delta W = W_A W_B$, where `W_A \in \mathbb{R}^{A \times h} $` and $W_A \in \mathbb{R}^{h \times B}$. Here, we keep the original weight frozen and only train the new matrices W_A and W_B. How is this parameter efficient if we introduce new weight matrices? The new matrices can be very small. For example, suppose $A = 25$ and $B = 50$, then the size of ΔW is $25 \times 50 = 6,250$. If $h = 5$, then W_A has 125 parameters, and W_B has 250 parameters, and the two matrices combined only have $125 + 250 = 375$ parameters in total.

After learning the weight update matrix, we can then write the matrix multiplication of a fully connected layer as follows, using pseudo-code:

```
def lora_forward_matmul(x):
    h = x . W  # regular matrix multiplication
    h += x . (W_A . W_B) * scalar
    return h
```

In the pseudo-code above, `scalar` is a scaling factor that adjusts the magnitude of the combined result (original model output plus low-rank adaptation). This balances the pretrained model's knowledge and the new task-specific adaptation.

According to the LoRA paper, the modeling performance of models using LoRA performs slightly better than models using Adapters across several task-specific benchmarks. Often, LoRA performs even better than models finetuned using the Finetuning II method described at the beginning of Q19.

[114]Popular low-rank transformation techniques include principal component analysis (PCA) and singular vector decomposition (SVD).

Reinforcement Learning with Human Feedback

The previous section focused on making finetuning more efficient. Switching gears, how can we improve the modeling performance of LLMs via finetuning? The conventional way to adapt or finetune an LLM for a new target domain or task is using a supervised approach with labeled target data. For instance, using the finetuning II approach discussed earlier allows us to adapt a pretrained LLM and finetune it on a target task, for example, sentiment classification, using a dataset that contains texts with sentiment labels such as "positive," "neutral," and "negative."

An alternative approach to supervised finetuning is an approach referred to as reinforcement learning with human feedback (RLHF). For example, ChatGPT and its predecessor InstructGPT[115] are two popular examples of pretrained LLMs (GPT-3) finetuned using RLHF.

In RLHF, a pretrained model is finetuned using a combination of supervised learning and reinforcement learning – the approach was popularized by the original ChatGPT model, which was in turn based on InstructGPT[116]. Human feedback is collected by having humans rank or rate different model outputs, providing a reward signal. The collected reward labels can then be used to train a reward model that is then used to guide the LLMs adaptation to human preferences. The reward model is learned via supervised learning (typically using a pretrained LLM as base model). Next, the reward model is then used to update the pretrained LLM that is to be adapted to human preferences – the training uses a flavor of reinforcement learning called proximal policy optimization[117]. Why use a reward model instead of training the pretained model on the human feedback directly? That's because involving humans

[115]Ouyang and colleagues (2022). *Training Language Models To Follow Instructions With Human Feedback*, https://arxiv.org/abs/2203.02155.

[116]Ouyang et al. (2022). *Training language models to follow instructions with human feedback*, https://arxiv.org/abs/2203.02155

[117]Schulman, Wolski, Dhariwal, Radford, and Klimov (2017) *Proximal Policy Optimization Algorithms*, https://arxiv.org/abs/1707.06347.

in the learning process would create a bottleneck since we cannot obtain feedback in real-time.

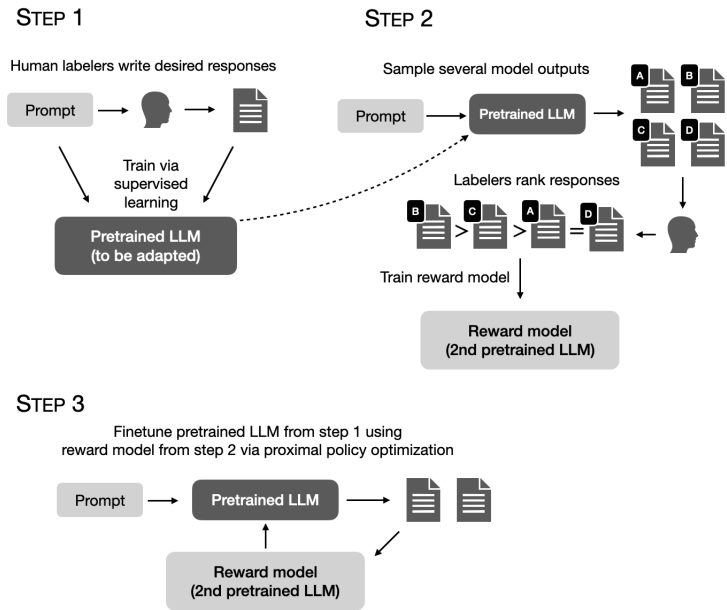


Figure 19.8. The *reinforcement learning with human feedback* (RLHF) process based on InstructGPT

Conclusion

While finetuning all layers of a pretrained LLM remains the gold standard for adaption to new target tasks, several efficient alternatives exist for leveraging pretrained transformers. For instance, we can effectively apply LLMs to new tasks while minimizing computational costs and resources by utilizing feature-based methods, in-context learning, or parameter-efficient finetuning techniques.

The three conventional methods – feature-based approach, finetuning I, and finetuning II – provide different computational efficiency and performance trade-offs. Parameter-efficient finetuning methods like soft prompt tuning, prefix tuning, and adapter methods

further optimize the adaptation process, reducing the number of parameters that need to be updated. Reinforcement learning with human feedback (RLHF) presents an alternative approach to supervised finetuning, potentially improving modeling performance.

In summary, the versatility and efficiency of pretrained LLMs continue to advance, offering new opportunities and strategies for effectively adapting these models to a wide array of tasks and domains. As research in this area progresses, we can expect further improvements and innovations in using pretrained language models.

> Reader quiz:

19-A

When does it make more sense to use in-context learning over finetuning and vice versa?

19-B

In prefix tuning, adapters, and LoRA, how can we ensure that the model preserves (and not forgets) the original knowledge?

Q20. Evaluating Generative Language Models

> Q:

What are the standard metrics for evaluating the quality of text generated by large language models?

> A:

Perplexity, BLEU, and ROUGE scores are some of the most common evaluation metrics used in natural language processing to assess the performance of Large Language Models (LLMs) across various tasks.

Why do we care about metrics? Ultimately, there is no way around human quality judgments, but human evaluations are tedious, expensive, hard-to-automate, and subjective. Hence, we develop metrics to provide objective summary scores to measure progress and compare different approaches.

Overview

Perplexity is directly related to the loss function used for pre-training LLMs and is commonly used to evaluate text generation and text completion models. Essentially, it quantifies the average uncertainty of the model in predicting the next word in a given context – the lower the perplexity, the better.

The BLEU (Bilingual Evaluation Understudy) score is a widely used metric for evaluating the quality of machine-generated translations. It measures the overlap of n-grams between the machine-generated translation and a set of human-generated reference translations. A higher BLEU score indicates better performance, ranging from 0 (worst) to 1 (best).

The ROUGE (Recall-Oriented Understudy for Gisting Evaluation) score is a metric primarily used for evaluating automatic summarization (and sometimes machine translation) models. It measures

the overlap between the generated summary and reference summaries.

We can think of perplexity as an intrinsic metric and BLEU and ROUGE as extrinsic metrics. To illustrate the difference between intrinsic and extrinsic metrics, think of optimizing the cross entropy to train an image classifier. The cross entropy is a loss function we optimize during training, but our end goal is to maximize the classification accuracy. Since classification accuracy cannot be optimized directly during training – it's not differentiable – we minimize the surrogate loss function like the cross entropy. Minimizing the cross entropy loss more or less correlates with maximizing the classification accuracy.

Perplexity is often used as an evaluation metric to compare the performance of different language models, but it is not the optimization target during training. BLEU and ROUGE are more related to classification accuracy, or rather precision and recall. In fact, BLEU is a precision-like score to evaluate the quality of a translated text. ROUGE is a recall-like score to evaluate summarized texts.

The following paragraphs will discuss the mechanics of these metrics in more detail.

Perplexity

Perplexity is closely related to the cross entropy that is directly minimized during training, which is why we refer to perplexity as an intrinsic metric.

Perplexity is defined as $2^{H(p,q)/n}$, where $H(p, q)$ is the cross entropy[118] between the true distribution of words p and the predicted distribution of words q. As cross entropy decreases, perplexity decreases as well – the lower the perplexity, the better. And n is the sentence length (the number of words or tokens) to normalize the score.

[118]While we typically compute the cross entropy using a natural logarithm, we calculate the cross entropy and perplexity with a base-2 logarithm for the intuitive relationship to hold. However, whether we use a base-2 or natural logarithm is a minor implementation detail.

In practice, since the probability for each word in the target sentence is always 1, we compute the cross entropy as the logarithm of the probability scores returned by the model we want to evaluate. In other words, if we have the predicted probability score for each word in a sentence s, we can compute the perplexity directly as follows:

$$Perplexity(s) = 2^{-\frac{1}{n}\log_2(p(s))}.$$

In the formula above

- s is the sentence or text we want to evaluate, for example, "The quick brown fox jumps over the lazy dog";
- $p(s)$ are the probability scores returned by the model;
- n is the number of words or tokens.

For example, if the model returns the following probability scores,

$$[0.99, 0.85, 0.89, 0.99, 0.99, 0.99, 0.99, 0.99],$$

the perplexity is

$$2^{-\frac{1}{8}\sum_i \log_2 p(w_i)}$$
$$= 2^{-1/8 \cdot \sum \log_2(0.99*0.85*0.89*0.99*0.99*0.99*0.99*0.99))}$$
$$= 1.043$$

If the sentence was "The fast black cat jumps over the lazy dog" with probabilities $[0.99, 0.65, 0.13, 0.05, 0.21, 0.99, 0.99, 0.99]$, the corresponding perplexity would be 2.419.

(Interested readers can find a code implementation and example of this calculation in the supplementary materials[119].)

A high perplexity indicates that a language model is less effective at predicting the next word in a sequence, implying greater uncertainty in its predictions. We aim for a low perplexity in language models, as it indicates better performance in predicting the next

[119] https://github.com/rasbt/MachineLearning-QandAI-book

word in a sequence, reflecting a more accurate understanding of language patterns and context.

BLEU score

BLEU is the most popular and widely used metric for evaluating translated texts. It's used in almost all LLMs capable of translation, including popular tools such as OpenAI's Whisper and GPT models, so it's worth discussing how it works.

BLEU was originally developed to capture or automate the essence of human evaluation; the original BLEU paper found a high correlation with human evaluations[120] (although this was later disproven[121]).

BLEU is a reference-based metric that compares the model output to human-generated references. In short, BLEU measures the lexical overlap between the model output and the human-generated references based on a precision score.

So, how does it work exactly? As a precision-based metric, BLEU counts how many words in the generated text (candidate text) occur in the reference text divided by the reference text length, where the reference text is a sample translation provided by a human, for example. (Instead of using individual words, this is commonly done for n-grams, but for simplicity, we will stick to words or 1-grams.)

The BLEU score calculation is summarized in the illustration below, and interested readers can also find a code implementation in the supplementary materials[122].

[120]Papineni, Roukos, Ward, and Zhu (2002). *BLEU: a Method for Automatic Evaluation of Machine Translation*, https://aclanthology.org/P02-1040/.

[121]Callison-Burch, Osborne, and Koehn (2006). *Re-evaluating the role of BLEU in machine translation research*, https://aclanthology.org/E06-1032/.

[122]https://github.com/rasbt/MachineLearning-QandAI-book

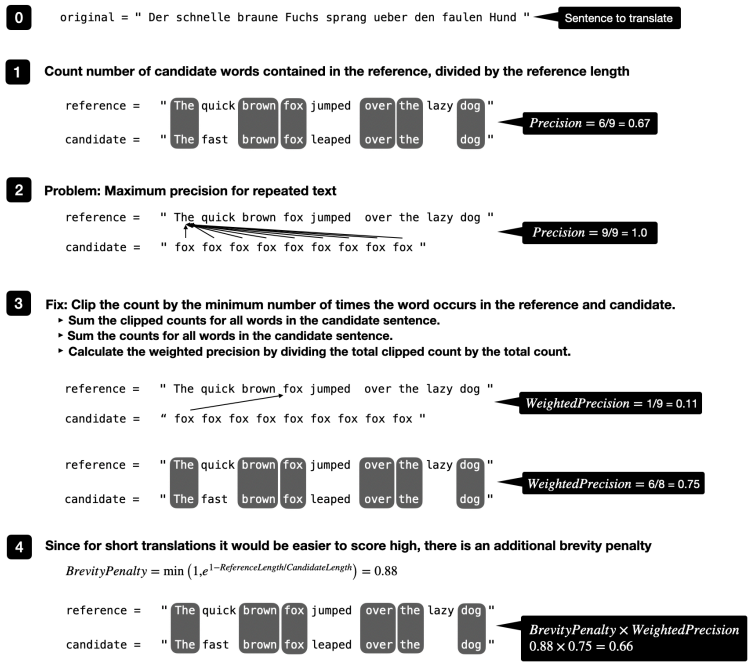


Figure 20.1. Calculation of the 1-gram BLEU score for illustration purposes. (In practice, BLEU is often computed for 4-grams.)

BLEU has several shortcomings[123] that are mostly owed to the fact that BLEU measures string similarity and similarity alone is not sufficient for capturing quality. Furthermore, since BLEU relies on exact string matches, it is sensitive to lexical variations and incapable of identifying semantically similar translations that use synonyms or paraphrases. In other words, BLEU may assign lower scores to translations that are, in fact, accurate and meaningful.

Is BLEU flawed? Yes. Is it still useful? Also yes. BLEU is a helpful tool to measure or assess whether a model improves during training – it's a proxy for fluency. However, it is not very well suited to give

[123] I can highly recommend Benjamin Marie's insightful article, "12 Critical Flaws of BLEU," based on 37 studies published over 20 years: https://medium.com/@bnjmn_marie/12-critical-flaws-of-bleu-1d790ccbe1b1.

a correct assessment of the quality of the generated translations and is not well suited for detecting issues. In other words, use it as a model selection tool, not a model evaluation tool[124].

ROUGE score

Where BLEU is commonly used for translation tasks, ROUGE[125] is a popular metric for scoring text summaries.

There are quite some similarities between BLEU and ROUGE. The precision-based BLEU score checks how many words in the candidate translation occur in the reference translation. The ROUGE score also takes a flipped approach, checking how many words in the reference text appear in the generated text (here typically a summarization instead of translation) – this can be interpreted as a recall-based score.

Modern implementations compute the ROUGE as an F1 score that is the harmonic mean of recall (how many words in the reference occur in the candidate text) and precision (how many words in the candidate text occur in the reference text) as illustrated in the figure below.

[124]The currently most popular alternatives to BLEU are METEOR and COMET.

[125]Lin (2004). *ROUGE: A Package for Automatic Evaluation of Summaries*, https://aclanthology.org/W04-1013/

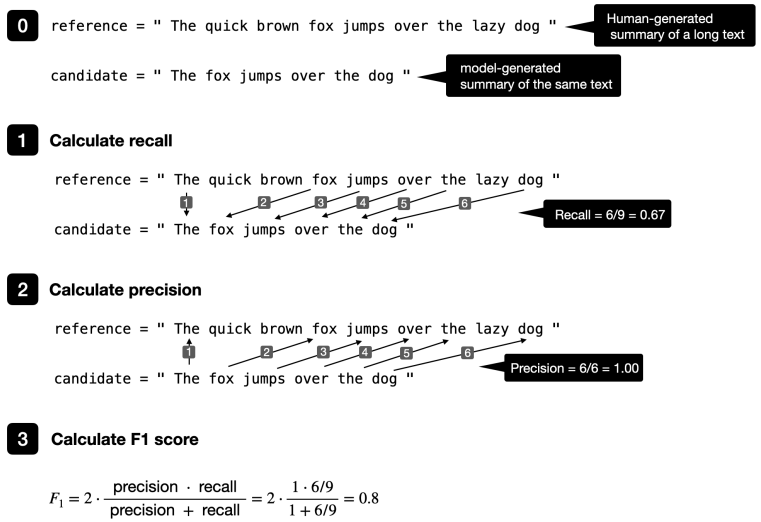


Figure 20.2. Illustration of the 1-gram ROUGE score computation. (In practice, ROUGE is often computed for bigrams, that is, 2-grams.)

(Interested readers can find a computational example in the supplementary materials[126].)

Besides this ROUGE-1 (the F1-score based ROUGE score for 1-grams), there are more variants:

1. ROUGE-N: This measures the overlap of N-grams between the candidate and reference summaries. For example, ROUGE-1 would look at the overlap of individual words (1-grams), while ROUGE-2 would consider the overlap of 2-grams (bigrams).
2. ROUGE-L: This metric measures the longest common subsequence (LCS) between the candidate and reference summaries. It captures the longest co-occurring in-order subsequence of words, which may have gaps in between.

[126] https://github.com/rasbt/MachineLearning-QandAI-book

3. ROUGE-S: This metric measures the overlap of skip-bigrams, which are word pairs with a flexible number of words in between. It can be useful to capture the similarity between sentences with different word orderings.

ROUGE shares similar weaknesses as BLEU. Like BLEU, ROUGE does not account for synonyms or paraphrases. It measures the n-gram overlap between the candidate and reference summaries, which can lead to lower scores for semantically similar but lexically different sentences. It is still worth knowing about ROUGE since *all* papers introducing new summarization models at computational linguistic conferences in 2021 use ROUGE – 69% of these papers use *only* ROUGE according to *Repairing the Cracked Foundation: A Survey of Obstacles in Evaluation Practices for Generated Text*[127].

BERTScore

BERTScore[128] can be used for translations and summaries, and it captures the semantic similarity better than traditional metrics like BLEU and ROUGE. In particular, it's more robust to paraphrasing.

For readers familiar with the inception score for generative vision models, BERTScore takes a similar approach of using embeddings[129] from a pretrained model. Here, BERTScore measures the similarity between a candidate text and a reference text by leveraging the contextual embeddings produced by the BERT model[130].

We can summarize the BERT score computation via the following steps:

1. Obtain the candidate text via the LLM you want to evaluate (PaLM, LLaMA, GPT, BLOOM, etc.)

[127] https://arxiv.org/abs/2202.06935

[128] Zhang, Kishore, Wu, Weinberger, and Artzi (2019). *BERTScore: Evaluating Text Generation with BERT*, https://arxiv.org/abs/1904.09675.

[129] See Q1 for more information about embeddings.

[130] BERT is an encoder-style transformer, further discussed in Q18.

2. Tokenize the candidate and reference texts into subwords (preferably using the same tokenizer used for training BERT)
3. Use a pretrained BERT model to create the embeddings for all tokens in the candidate and reference texts
4. Then, compare each token embedding in the candidate text to all token embeddings in the reference text, computing their cosine similarity
5. Align each token in the candidate text with the token in the reference text that has the highest cosine similarity
6. Compute the final BERTScore by taking the average similarity scores of all tokens in the candidate text.

(Interested readers can find a computational example in the supplementary materials[131].)

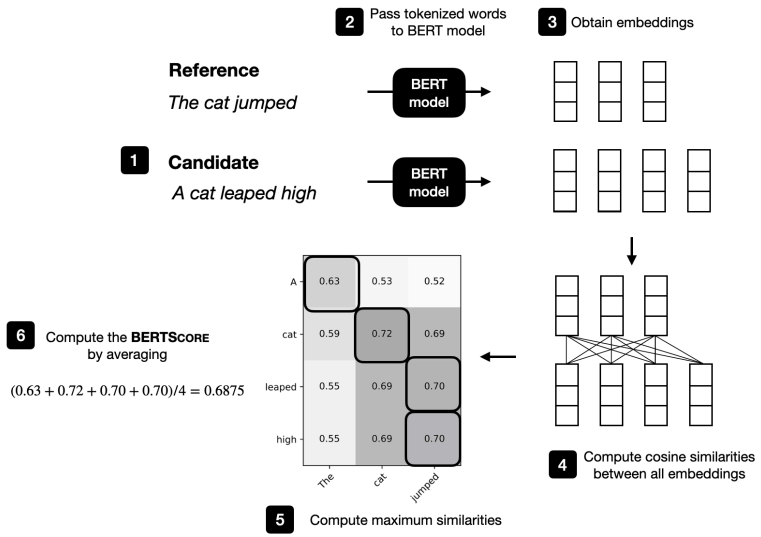


Figure 20.3. Illustration of how to compute the BERTScore.

BERTScore is more robust to paraphrasing and captures semantic similarity better due to its contextual embeddings. However, it

[131] https://github.com/rasbt/MachineLearning-QandAI-book

may be computationally more expensive as it requires using a pre-trained BERT model for the evaluation.

It's important to note that while BERTScore provides a useful automatic evaluation metric, it is also not perfect and should be used alongside other evaluation techniques, including human judgment.

Conclusion

All metrics covered above are surrogates or proxies to evaluate how useful the model is in terms of measuring how well the model compares to human performance for accomplishing a goal[132]. As mentioned earlier, the best way to evaluate LLMs is to assign human raters who judge the results. However, since this is often expensive and not easy to scale, we use the aforementioned metrics to estimate model performance. To quote from the InstructGPT[133] paper:

> "Public NLP datasets are not reflective of how our language models are used. . . . [They] are designed to capture tasks that are easy to evaluate with automatic metrics."

Moreover, it is also important to incorporate ethical considerations when assessing LLMs to ensure they are used responsibly and fairly.

> Reader quiz:

Q20-A

Looking at step 5 in the previous figure illustrating the BERTScore computation, we see that the cosine similarity between the two

[132]Besides BERTScore, ROUGE, BLEU, and perplexity, several other popular evaluation metrics are used to assess the predictive performance of LLMs. Interested readers can learn more in the *Evaluation of Text Generation* survey by Celikyilmaz, Clark, and Gao https://arxiv.org/abs/2006.14799.

[133]Ouyang and colleagues (2022). *Training Language Models To Follow Instructions With Human Feedback*, https://arxiv.org/abs/2203.02155.

embeddings of "cat" are not 1.0, where 1.0 indicates a maximum cosine similarity. Why is that?

Q20-B

In practice, we might find that the BERTScore is not symmetric. This means that switching the candidate and reference sentence could result in different BERTScores for specific texts. How could we address this?

Chapter 4. Production, Real-World, And Deployment Scenarios

Q21. Stateless And Stateful Training

> Q:

What is the difference between stateless and stateful training workflows in the context of production and deployment systems?

> A:

Both stateless training and stateful training refer to different ways of training a production model.

Stateless (re)training

Stateless training is a conventional, traditional approach where we first train an initial model on the original training set and then retrain it as new data arrives. Hence, stateless training is also commonly referred to as stateless *retraining*[134].

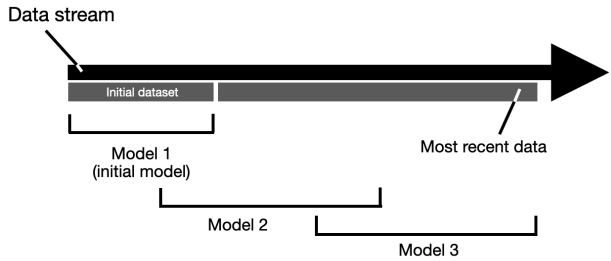


Figure 21.1. Illustration of stateless retraining.

As illustrated in the figure above, stateless retraining is like a sliding window approach where we retrain the initial model on different parts of the data from a given data stream. For example, to update the initial model (*Model 1*) to a newer model (*Model 2*), we train the model on 30% of the initial data and 70% of the most recent data at a given point in time.

Stateless retraining is a straightforward approach where we can adapt the model to the most recent changes in the data and

[134] Huyen, C. (2022). *Designing Machine Learning Systems*. O'Reilly Media, Inc.

feature-target relationships via retraining the model from scratch in user-defined checkpoint intervals. This approach is prevalent with conventional machine learning systems that cannot be finetuned as part of a transfer or self-supervised learning workflow (see Q2). For example, standard implementations of tree-based models such as random forests and gradient boosting (XGBoost, CatBoost, and LightGBM) fall into this category.

Stateful training

In stateful training, we train the model on an initial batch of data and then update it periodically (as opposed to retraining it) when new data arrives.

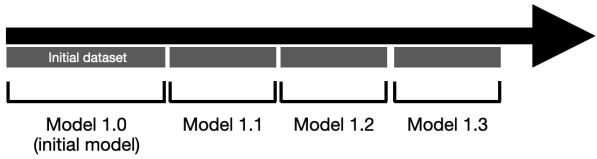


Figure 21.2. Illustration of stateful training.

In the illustration of stateful training above, we do not retrain the initial model (*Model 1.0*) from scratch but update or finetune it as new data arrives. This approach is particularly attractive for models compatible with transfer learning or self-supervised learning.

The stateful approach mimics a transfer or self-supervised learning workflow where we adopt a pretrained model for finetuning. However, stateful training differs fundamentally from transfer and self-supervised learning as it updates the model to accommodate concept, feature, and label drifts. In contrast, transfer and self-supervised learning aim to adopt the model for a different classification task. For instance, in transfer learning, the target labels often differ. In self-supervised learning, we obtain the target labels from the dataset features.

One significant advantage of stateful training is that we do not need to store data for retraining – we can use it to update the model as

soon as it arrives. This is particularly attractive when data storage is a concern due to privacy or resource limitations.

> Reader quiz:

21-A

Suppose you train a classifier for stock trading recommendation using a random forest model, including the moving average of the stock price as a feature. Now, since new stock market data arrives daily, you are thinking about how to update the classifier daily to keep it up to date. Should you take a stateless training or stateless retraining approach to update the classifier?

21-B

Suppose you deployed a large language model (transformer) such as ChatGPT that can answer user queries. The dialogue interface includes a "thumbs up" and "thumbs down" button so that users can give direct feedback based on the generated queries. While collecting the user feedback, you are not updating the model immediately as new feedback arrives. However, you are planning to release a new or updated model at least once per month.
Is this a good candidate for stateless or stateful retraining?

Q22. Data-Centric AI

> Q:

What is data-centric AI, how does it compare to the conventional modeling paradigm, and how do we decide it's the right fit for a project?

> A:

Data-centric AI is a paradigm or workflow where we keep the model training procedure fixed and iterate over the dataset to improve the predictive performance of a model.

In the context of data-centric AI, we can think of the conventional workflow[135], which is often part of academic publishing, as model-centric AI. However, in an academic research setting, we are typically interested in developing new methods (for example, neural network architectures or loss functions). Here, we consider existing benchmark datasets to compare the new method to previous approaches to determine whether it is an improvement over the status quo.

[135]While data-centric AI is a relatively new term, the idea behind it is not new. Many readers want to point out that they had used a data-centric approach in their projects before the term was coined. In my opinion, data-centric AI was created to make "caring about data quality" attractive again – data collection and curation is often considered tedious or thankless. This is analogous to how the term *deep learning* made neural networks attractive again.

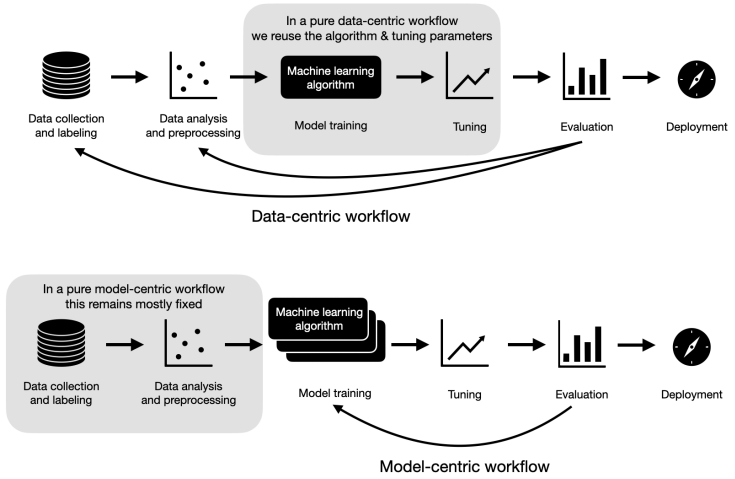


Figure 22.1. Illustrating a data-centric versus model-centric machine learning workflow.

Why can't we have both?

In short, data-centric AI focuses on changing the data to improve performance. Model-centric approaches focus on modifying the model to improve performance. Ideally, we can do both in an applied setting where we want to get the best possible predictive performance. However, if we are in a research setting or an exploratory stage of an applied project, varying too many variables simultaneously is messy. If we change both model and data simultaneously, it is hard to pinpoint which change is responsible for the improvement.

What are the different methods for data-centric AI?

It is important to emphasize that data-centric AI is a paradigm and workflow, not a particular technique. So, data-centric AI implicitly includes

- analyses and modifications of training data, from outlier removal to missing data imputation;

- data synthesis and data augmentation techniques;
- data labeling and label cleaning methods;
- the classic active learning setting where a model suggests which data points to label.

We consider an approach *data-centric* if we only change the data (using the methods above), not the other aspects of the modeling pipeline.

What are the advantages of data-centric AI?

In machine learning and AI, we often use the phrase "garbage in, garbage out." Here, it means that poor-quality data will result in a poor predictive model. Or in other words, we cannot expect a well-performing model from a low-quality dataset.

There is one pattern I often observe in applied academic projects where researchers want to use machine learning to replace an existing methodology. Often, researchers only have a small dataset of examples (let's say 100s of training examples). Labeling data is often expensive or considered boring and thus best avoided. In these cases, the researchers would spend unreasonable time trying out different machine learning algorithms and weeks on model tuning. In these cases, investing additional time or resources in labeling additional data would be worthwhile.

The main advantage of data-centric AI is that it puts the data first. So, if we invest resources to create a higher-quality dataset, all modeling approaches will benefit from it downstream.

How do we decide if data-centric AI is the right fit?

Taking a data-centric approach is often a good idea in an applied project where we want to improve the predictive performance to solve a particular problem. It makes sense to start with a modeling baseline and improve the dataset since it can often be more worth-

while than trying out bigger, more expensive models[136].

If our task is to develop a new or better methodology, such as a new neural network architecture or loss function, then a model-centric approach might be a better choice. Using an established benchmark dataset, and not changing it, will make it easier to compare the new modeling approach to previous work.

In a real-world project, alternating between data-centric and model-centric modes makes a lot of sense. Early on, investing more in data quality makes sense because it will benefit all models. Then, once a good dataset is available, it makes sense to hone in on the model tuning part, to improve performance.

> Reader quiz:

22-A

A recent trend is the increased use of predictive analytics in healthcare. For example, suppose your healthcare provider developed an AI system that analyzes patients' electronic health records and provides recommendations for lifestyle changes or preventive measures. For this, the provider requires you to monitor and share your health data (such as pulse and blood pressure) daily. Is this an example of data-centric AI?

22-B

Suppose you train a convolutional neural network, specifically a ResNet-34, to classify images in the CIFAR-10 and ImageNet datasets. To reduce overfitting and improve classification accuracy, you are experimenting with different data augmentation techniques, such as image rotation and cropping. Is this approach data-centric?

[136]In the paper "A Few More Examples May Be Worth Billions of Parameters" (https://arxiv.org/abs/2110.04374), the researchers found that increasing the model size usually improves performance, but so does the addition of training examples. Assuming small training sets (<2k) for classification, extractive question answering, and multiple choice tasks, adding 100 examples can result in the same performance gain as adding billions of parameters.

Q23. Speeding Up Inference

> Q:

What are techniques to speed up model inference through optimization without changing the model architecture and without sacrificing accuracy?

> A:

In machine learning and AI, model inference refers to making predictions or generating outputs using a trained model.

The main general techniques for improving model performance during inference include parallelization, vectorization, loop tiling, operator fusion, and quantization, which are discussed in detail in the sections below.

Other techniques to improve inference speeds include knowledge distillation and pruning, which are discussed in Q6. However, knowledge distillation and pruning affect the model architecture, resulting in smaller models, which is why they are out of scope for this question.

Parallelization

One common way to achieve better parallelization during inference is to run the model on a batch rather than a single sample at a time. This is sometimes also referred to as *batch inference* and assumes that were are receiving multiple inputs samples or user inputs simultaneously or within a short time window, as illustrated in the figure below.

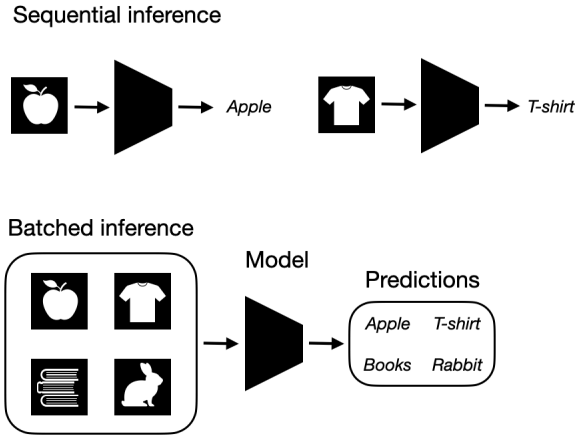


Figure 23.1. Comparison of sequential inference and batch inference.

Note that setting a reasonable timeout to run the inference is essential even if the desired batch size is not reached. This is to ensure that users don't have to wait too long.

Furthermore, if we exceed a given batch size per time unit, we can divide the input samples or batch into several smaller subsets or "chunks." These chunks can then be processed simultaneously rather than sequentially if we have multiple compute cores. This is similar to the data parallelism multi-GPU strategy described in Q7.

Vectorization

Vectorization refers to performing operations on entire data structures, such as arrays (tensors) or matrices, in a single step rather than using iterative constructs like for-loops. Ysing vectorization, multiple operations from the loop are performed simultaneously using SIMD (Single Instruction, Multiple Data) instructions that are available on most modern CPUs. This approach takes advantage of the low-level optimizations in many computing systems (such as

BLAS[137]) and often results in signficant speed-ups.

For example, suppose we wanted to compute the dot product between two vectors. The non-vectorized way of doing this would be to use a for-loop, iterating over each element of the arrays one by one. However, this can be quite slow, especially for large arrays. With vectorization, you can perform the dot product operation on the entire arrays at once as shown in the figure below.

Classic for-loop

```
x = [1.2, 2.2, 3.3, 4.4]
w = [5.5, 6.6, 7.7, 8.8]

output = 0.

for x_j, w_j in zip(x, w):
    output += x_j * w_j

print(output)
```

```
85.25
```

Vectorized implementation

```
import torch

x = torch.tensor([1.2, 2.2, 3.3, 4.4])
w = torch.tensor([5.5, 6.6, 7.7, 8.8])

x.dot(w)
```

```
tensor(85.2500)
```

Illustration of a classic for-loop versus a vectorized dot product computation in Python.

In the context of linear algebra or deep learning frameworks like TensorFlow or PyTorch, vectorization is typically done automatically. This is because these frameworks are designed to work with multi-dimensional arrays (also known as tensors), and their operations are inherently vectorized. This means that when you perform functions using these frameworks, you automatically leverage the power of vectorization, resulting in faster and more

[137]BLAS stands for Basic Linear Algebra Subprograms (https://en.wikipedia.org/wiki/Basic_-Linear_Algebra_Subprograms) and is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, matrix multiplication, and others. Many array and deep learning libraries like NumPy and PyTorch use BLAS under the hood.

efficient computations.

Loop tiling

Loop tiling (also often referred to as *loop nest optimization*[138]) is an advanced optimization technique to enhance data locality by breaking down a loop's iteration space into smaller chunks or "tiles." This ensures that once data is loaded into cache, all possible computations are performed on it before it is cleared.

The concept of loop tiling for accessing elements in a 2-dimensional array is illustrated in the figure below. In a regular for-loop, we iterate over columns and rows one element at a time, whereas in loop tiling, we subdivide the array into smaller tiles.

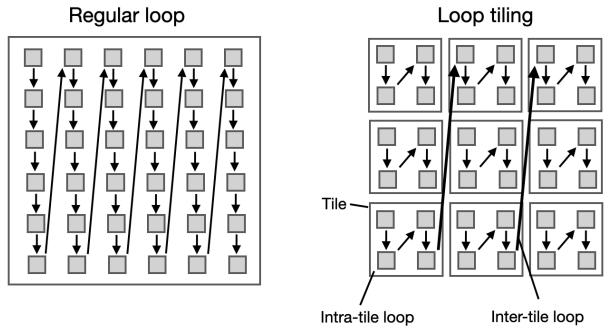


Illustration of loop tiling for a 2-dimensional array.

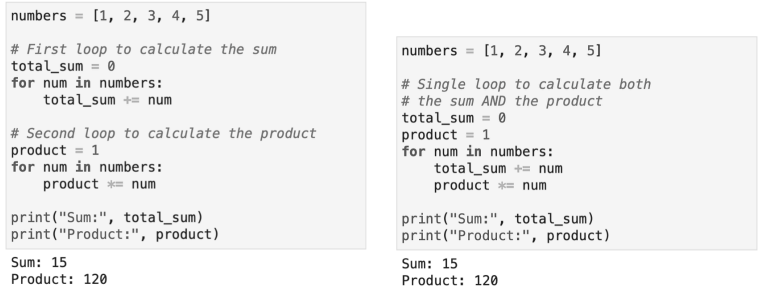
Note that in high-level languages, such as Python, we don't usually perform loop tiling because Python does not allow control over the cache memory like lower level languages such as C or C++. These kinds of optimizations are often handled by underlying libraries like NumPy or PyTorch when performing operations on large arrays.

Operator fusion

Operator fusion, sometimes called loop fusion, is an optimization technique that combines multiple loops into a single loop. This is

[138] https://en.wikipedia.org/wiki/Loop_nest_optimization

illustrated in the figure below, where two separate loops to calculate the sum and the product of an array of numbers are fused into a single loop.



```
numbers = [1, 2, 3, 4, 5]

# First loop to calculate the sum
total_sum = 0
for num in numbers:
    total_sum += num

# Second loop to calculate the product
product = 1
for num in numbers:
    product *= num

print("Sum:", total_sum)
print("Product:", product)
```

```
Sum: 15
Product: 120
```

```
numbers = [1, 2, 3, 4, 5]

# Single loop to calculate both
# the sum AND the product
total_sum = 0
product = 1
for num in numbers:
    total_sum += num
    product *= num

print("Sum:", total_sum)
print("Product:", product)
```

```
Sum: 15
Product: 120
```

Illustration of operator fusion, fusing two for loops (left) into one (right).

This can improve the performance of a model by reducing the overhead of loop control, decreasing memory access times by improving cache performance, and possibly enabling further optimizations through vectorization.

Doesn't this contradict the concept of loop tiling earlier, where we break down a for-loop into multiple loops? However, they are complementary techniques used for different optimizations, and they are applicable in different situations. Operator fusion is about reducing the total number of loop iterations and improving data locality when the entire data fits into cache. Loop tiling is about improving cache utilization when dealing with larger multi-dimensional arrays that do not fit into cache.

Related to operator fusion is the concept of reparameterization, which can often also be used to simplify multiple operations into one. Popular examples include training a network with multi-branch architectures that are reparameterized into single-stream architectures during inference[139]. This reparameterization approach

[139] In RepVGG, for example, each branch during training consists of a series of convolutions. Once training is complete, the model is reparameterized into a single sequence of convolutions. For more details, see Ding, Zhang, Ma, Han, Ding, and Sun (2021). *RepVGG: Making VGG-style ConvNets Great Again* https://arxiv.org/abs/2101.03697.

differs from traditional operator fusion as it does not merge multiple operations into a single operation. Instead, it rearranges the operations in the network to create a more efficient architecture for inference.

Quantization

Quantization reduces the computational and storage requirements of machine learning models, particularly deep neural networks. This technique involves converting the continuous, floating-point numbers representing weights and biases in a trained neural network to discrete, lower-precision representations such as integers.Using less precision reduces the model size and makes it quicker to execute, which can lead to significant improvements in speed and hardware efficiency during inference.

In the realm of deep learning, it has become increasingly common to quantize trained models down to 8-bit[140] and 4-bit[141] integers. This technique is especially prevalent in the deployment of large language models, for instance.

There are two main categories of quantization. For instance, in post-training quantization, the model is first trained normally with full-precision weights, and then the weights are quantized after training. Quantization-aware training, on the other hand, introduces the quantization step during the training process. This allows the model to learn to compensate for the effects of quantization, which can help maintain the model's accuracy.

However, it's important to note that quantization can occasionally lead to a reduction in model accuracy, which is why it is not as good a fit to answer the original question as the other categories above.

> Reader quiz:

[140]Dettmers, Lewis, Belkada, and Zettlemoyer (2022). *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*, https://arxiv.org/abs/2208.07339.

[141]Frantar, Ashkboos, Hoefler, and Alistarh (2022). *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*, https://arxiv.org/abs/2210.17323.

23-A

Q7 covered several different multi-GPU training paradigms to speed up the model training. Using multiple GPUs can, in theory, also speed up model inference. However, in practice, there are several reasons why this approach is often not the most efficient or practical one. Can you think of what these reasons are?

23-B

Vectorization and loop tiling (or loop blocking) are two different strategies for optimizing operations that involve accessing array elements. When would you use which?

Chapter 5. Predictive Performance and Model Evaluation

Q25. Poisson and Ordinal Regression

> Q:

Consider *Poisson regression* and *ordinal regression*; when do we use which over the other?

> A:

Usually, we use Poisson regression is for when the target variable represents count data[142] (positive integers). As an example of count data, consider the number of colds contracted on an airplane or the number of guests visiting a restaurant on a given day.

Figure 25.1. An illustration of ordinal labels

Ordinal data is a subcategory of categorical data where the categories have a natural order, for example, 3 > 2 > 1. Ordinal data is often represented as positive integers and may look similar to count data. For example, consider the star rating on Amazon (1 star, 2 stars, 3 stars, etc.). However, ordinal regression does not make any assumptions about the distance between the ordered categories. As another example of ordinal data, consider disease severity, *severe* > *moderate* > *mild* > *none*. While we typically map the disease severity variable to an integer representation (4 > 3 > 2 > 1), there is no assumption that the distance between 4 and 3 (severe and moderate) is the same as the distance between 2 and 1 (mild and none).

In short, we use Poisson regression for count data. We use ordinal

[142] Besides the target variable representing counts, the data should also be Poisson-distributed, which means that the mean and variance are roughly the same. (For large means, we can use a normal distribution to approximate a Poisson distribution.)

regression when we know that certain outcomes are "higher" or "lower" than others, but we are not sure how much or if it even matters.

> Reader quiz:

25-A

Suppose we want to predict the number of goals a soccer player will score in a particular season. Is this a problem that can be solved using ordinal regression or Poisson regression?

25-B

Suppose you asked someone to sort the last three movies they have watched based on their order of preference. Ignoring the fact that this dataset is a tad to small for machine learning, what approach would be best suited for this kind of data?

Q27. Proper Metrics

> Q:

What are the three properties of a distance function that make it a *proper* metric?

List the three properties of a metric space. Then discuss whether commonly used loss functions such as mean squared error and the cross-entropy loss are proper metrics.

> A:

Consider two vectors or points $\mathbf{v}$ and $\mathbf{w}$ and their distance $d(\mathbf{v}, \mathbf{w})$.

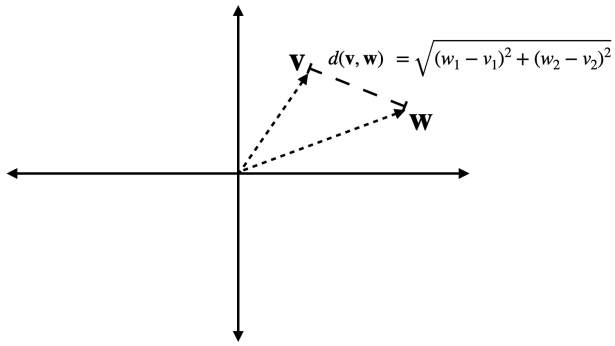


Figure 27.1. Euclidean distance between two two-dimensional vectors.

The criteria of a proper metric are the following:

1. The distance between two points is always non-negative $d(\mathbf{v}, \mathbf{w}) \geq 0$. Also, the distance can only be zero if the two points are identical, that is, $\mathbf{v} = \mathbf{w}$.
2. The distance is symmetric, i.e., $d(\mathbf{v}, \mathbf{w}) = d(\mathbf{w}, \mathbf{v})$.
3. The distance function satisifies the *triangle inequality* for any three points: $\mathbf{v}, \mathbf{w}, \mathbf{x}$, which means that

$d(\mathbf{v}, \mathbf{w}) \leq d(\mathbf{v}, \mathbf{x}) + d(\mathbf{x}, \mathbf{w})$.

To get a better intuition for the triangle inequality, think of the points as vertices of a triangle. And if we consider any triangle, the sum of two sides is always larger than the third.

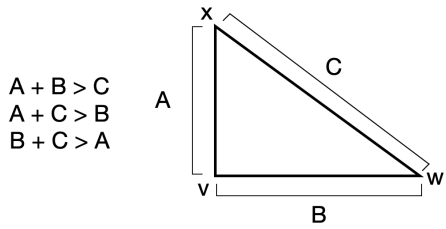


Figure 27.2. Illustration of the triangle inequality.

Is the mean squared error a proper metric?

The mean squared error loss (MSE) computes the squared Euclidean distance between a target variable y and a predicted target value $\hat{y}$:

$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$.

The index i denotes the i-th datapoint in the dataset or sample. For simplicity, we will consider the squared error (SE) loss between two data points (however, the insights below also hold for the MSE):

$\text{SE}(y, \hat{y}) = (y - \hat{y})^2$.

Criterion 1. The SE satisfies the first part of the first criterion: *The distance between two points is always non-negative.* Since we are raising the difference to the power of 2, it cannot be negative.

Criterion 2. How about the second criterion, *the distance can only be zero if the two points are identical*? Due to the subtraction in the SE, it is intuitive to see that it can only be 0 if the prediction matches the target variable, $y = \hat{y}$.

We saw tha the SE satisfies the first criterion of a proper metric, and we can again use the *square* to confirm that it also satisfies the second criterion, *the distance is symmetric*. Due to the square, we have $(y - \hat{y})^2 = (\hat{y} - y)^2$.

Criterion 3. At first glance, it seems that the squared error loss also satisfies the triangle inequality. Intuitively, you can check this by choosing three arbitrary numbers (here: 1, 2, 3):

1. $(1-2)^2 \leq (1-3)^2 + (2-3)^2$
2. $(1-3)^2 \leq (1-2)^2 + (2-3)^2$,
3. $(2-3)^2 \leq (1-2)^2 + (1-3)^2$.

However, there are values where this is not true, for example $d(a,c) = 4,\ d(a,b) = 2,$ and $d(b,c) = 3$, where $4^2 \lneq= 2^2 + 3^2$. [143] the triangle inequality does not hold.

In contrast, the root-mean squared error does satisfy the triangle inequality, and the example above works out: $4 \lneq 2 + 3$.

The root-squared error $\sqrt{(y-\hat{y})^2}$ is essentially the same as the L_2 or Euclidean distance between two points, which is known to satisfy the triangle inequality[144].

Since it does not satisfy the triangle inequality via the example above, we conclude that the (mean) squared error loss is not a proper metric while the root-mean squared error (or Euclidean distance) is a proper metric.

Is the cross-entropy loss a proper metric?

Cross-entropy is used to measure the distance between two probability distributions. In machine learning contexts, we use the discrete cross-entropy loss (CE) between class label y and the predicted probability p when we train logistic regression or neural network classifiers on a dataset consisting of n training examples:

$$\text{CE}(\mathbf{y}, \mathbf{p}) = -\frac{1}{n} \sum_{i=1}^{n} y^{(i)} \cdot \log\left(p^{(i)}\right).$$

[143] Example taken from Chai, Draxler (2014). *Root mean square error (RMSE) or mean absolute error (MAE)? Arguments against avoiding RMSE in the literature*, https://gmd.copernicus.org/articles/7/1247/2014/. However, I could not find any three scalar values a, b, and c to reproduce this.

[144] For example, see references in https://en.wikipedia.org/wiki/Euclidean_distance.

Again, for simplicity, we will look at the cross-entropy function (H) between only two data points:

$H(y, p) = -y \cdot \log(p)$.

Criterion 1. The cross-entropy loss satisfies one part of the first criterion. The distance is always non-negative because the probability score is a number in the range [0, 1]. Hence, $\log(p)$ ranges between $-\infty$ and 0. The important part is that the H function (see above) includes a negative sign. Hence, the cross-entropy ranges between ∞ and 0 and thus satisfies one aspect of criterion 1.

However, the cross entropy loss is not zero for two identical points. For example $H(0.9, 0.9) = -0.9 \log(0.9) = 0.095$.

Criterion 2. The second criterion is violated by the cross-entropy loss because it's not symmetric: $-y \cdot \log(p) \neq -p \cdot \log(y)$.

Let's illustrate this with a concrete, numeric example:

$-1 \cdot \log(0.5) = 0.693$

$-0.5 \cdot \log(1) = 0$.

Criterion 3. Does the cross-entropy loss satisfy the triangle inequality,
$H(r, p) \geq H(r, q) + H(q, p)$?

It does not. We can illustrate this with an example. Suppose we choose $r = 0.9, p = 0.5, q = 0.4$. We have

$H(0.9, 0.5) = 0.624$

$H(0.9, 0.4) = 0.825$

$H(0.4, 0.5) = 0.277$.

We can see that $0.624 \geq 0.825 + 0.277$ does not hold.

We can conclude that while the cross-entropy loss is a useful loss function for training neural networks via (stochastic) gradient decent, it is not a proper distance metric as it does not satisfy any of the three criteria above.

> Reader quiz:

27-A

Suppose we consider using the mean absolute error (MAE) as an alternative to the root mean square error (RMSE) for measuring the performance of a machine learning model, where

$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} \left| y^{(i)} - \hat{y}^{(i)} \right|$

and

$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)} \right)^2}$.

However, a colleague argues that the MAE is not a proper distance metric in metric space because it involves an absolute value, so we should prefer using the RMSE. Is this argument correct?

27-B

Based on your answer to **27-A**, would you say that the MAE is better than the RMSE, or vice versa?

Q28. The *k* in *k*-fold cross-validation

> Q:

K-fold cross-validation is a common choice for evaluating machine learning classifiers because it lets us use all training data to simulate how well a machine learning algorithm might perform on new data. What are the advantages and disadvantages of choosing a large *k*?

> A:

We can think of *k*-fold cross-validation as a workaround for model evaluation when we have limited data. In machine learning model evaluation, we care about the generalization performance of our model – how well it performs on new data.

In *k*-fold cross-validation, we use the training data for model selection and evaluation by partitioning it into *k* validation rounds and folds. If we have *k* folds, we have *k* iterations, leading to *k* different models. So, using *k*-fold cross-validation, we usually evaluate the performance of a particular hyperparameter configuration by computing the average performance over the *k* models. This performance reflects or approximates the performance of a model trained on the complete training dataset after evaluation[145].

[145]You can find a longer and more detailed explanation in my 2018 article *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*, https://arxiv.org/abs/1811.12808.

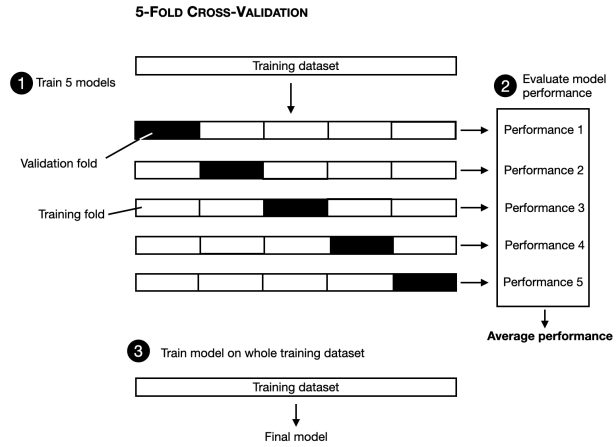


Figure 28.1. Illustration of k-fold cross-validation for model evaluation where *k=5*. The average performance over the five validation folds and models can be used to approximate the performance of the final model.

Training folds become too similar

If k is too large, the training sets are too similar between the different rounds of cross-validation. The k models are thus very similar to the model we obtain by training on the whole training set. In this case, we can still leverage the advantage of k-fold cross-validation: evaluating the performance for the entire training set[146] via the held-out validation fold in each round. However, a disadvantage of a large k is that analyzing how the machine learning algorithm with the particular choice of hyperparameter setting behaves on different training datasets is more challenging.

Increases computational budget

Besides the issue mentioned above with datasets that are too similar, running k-fold cross-validation with a large value of k is also computationally more demanding. A larger k is more expensive since it increases (1) the number of iterations and (2) the training set size at each iteration. This is especially an issue if we work with relatively large models that are expensive to train, for example,

[146] Here, we obtain the training set by concatenating all *k-1* training folds in a given iteration.

contemporary deep neural networks. Common choices for k are typically 5 or 10 for practical and historical reasons [147].

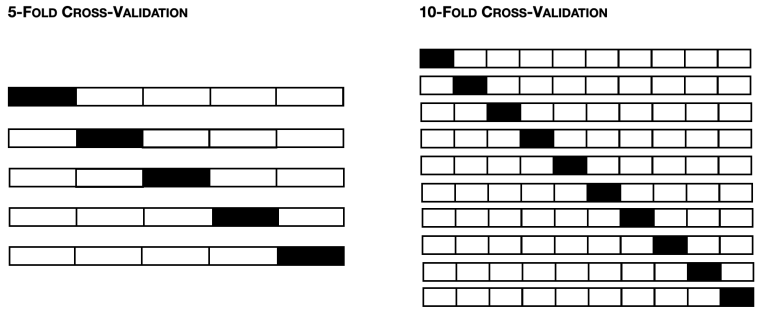


Figure 28.2. Comparison of 5-fold and 10-fold cross-validation. In 10-fold cross-validation, we use 9/10 (90%) of the data for training in each round, whereas in 5-fold cross-validation, we only use 4/5 (80%) of the data.

However, this does not mean large training sets are bad. Besides reducing the variance of the performance estimates (which can be good or bad), they can reduce the pessimistic bias of the performance estimate (which is mostly a good thing) if we assume that the model training can benefit from more training data [148].

Reminding ourselves of the purpose of k-fold cross-validation

When deciding upon an appropriate value of k, we are often guided by computational performance and conventions. However, it's worthwhile to define the purpose and context when we are using k-fold cross-validation. For example, if we care primarily about approximating the predictive performance of the final model, using a large k makes sense. This way, the training folds are very similar to the combined training dataset, yet, we still get to evaluate the model on all data points via the validation folds.

[147] The common recommendation goes back to Kohavi's work in 1995, *A Study of Cross-Validation And Bootstrap For Accuracy Estimation And Model selection*, https://dl.acm.org/doi/10.5555/1643031.1643047. This study found that *k=10* offers a good bias and variance trade-off for classical machine learning algorithms (decision trees and naive Bayes classifiers) on a handful of small datasets.

[148] See figure 15 in Q5 for an example of a learning curve.

On the other hand, if we care to evaluate how sensitive a given hyperparameter configuration and training pipeline is to different training datasets, then choosing a smaller number for k makes more sense.

Since most practical scenarios consist of two steps: (1) tuning hyperparameters and (2) evaluating the performance of a model, we can also consider a two-step procedure. For instance, we can use a smaller k during hyperparameter tuning. This will help speed up the hyperparameter search and probe the hyperparameter configurations for robustness (in addition to the average performance, we can also consider the variance as a selection criterion). Then, after hyperparameter tuning and selection, we can increase the value of k to evaluate the model[149].

> Reader quiz:

28-A

To provide the model with as much training data, we consider using leave-one-out cross-validation (LOOCV). LOOCV is a special case of k-fold cross-validation where k is equal to the number of training examples such that the validation folds only contain a single data point. A colleague mentions that LOOCV is *defect* for discontinuous loss function and performance measures such as classification accuracy. For instance, for a validation fold consisting of only one example, the accuracy is always either 0 (0%) or 1 (99%). Is this really a problem?

28-B

We discussed model selection and model evaluation as two use cases of k-fold cross-validation. Can you think of other use cases?

[149]However, reusing the same dataset for model selection and evaluation introduces biases, and it is usually better to use a separate test set for model evaluation. Also, nested cross-validation may be preferred as an alternative to k-fold cross-validation.

Q29. Training and Test Set Discordance

> Q:
You trained a model that performs much better on the test dataset than on the training dataset. Since a similar model configuration worked well on a similar dataset before, you suspect something might be unusual with the data. What are some approaches and mitigation issues for looking into training and test set discrepancies?

> A:

Before investigating the datasets in more detail, we should check for technical issues in the data loading and evaluation code. For instance, a simple sanity check is to temporarily replace the test set with the training set and to reevaluate the model. In this case, we should see identical training and test set performances (since these datasets are now identical). If we notice a discrepancy, we likely have a bug in the code – in my experience, such bugs are often related to incorrect shuffling or inconsistent (often missing) data normalization.

Since the test set performance is much better than the training set performance, we can rule out overfitting. More likely, there are substantial differences in the training and test data distributions. These distributional differences may affect both the features and the targets. Here, plotting the target or label distributions of training and test data is a good idea. For example, a common issue is that the test set is missing certain class labels if the dataset was not shuffled properly before splitting it into training and test data. For small tabular datasets, it is also feasible to compare feature distributions in the training and test set using histograms.

Looking at feature distributions is a good approach for tabular data, but this is trickier for image and text data. A relatively easy and

more general approach to check for training-test set discrepancies is *adversarial validation*.

Adversarial validation is a technique to identify the degree of similarity between the training and test data. In adversarial validation, we first merge the training and test set into a single dataset. Then we create a binary target variable that distinguishes between training and test data. For instance, can we use a new "*Is test?*" label where we assign the label 0 for training data and the label 1 for test data. We then use *k*-fold cross-validation or repartition the dataset into a training and test set and train a machine learning model as usual. Ideally, we want the model to perform poorly, indicating that the training and test data distribution is similar. On the other hand, if the model performs well in predicting the *Is test* label, it suggests a discrepancy between training and test data that we need to investigate further.

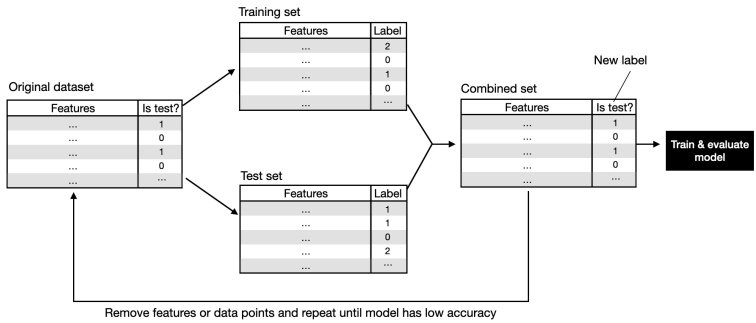


Figure 29.1. Outline of the adversarial validation workflow for detecting training and test set discrepancies.

What are some mitigation issues if we detect a training-test set discrepancy using adversarial validation? If we work with a tabular dataset, we can remove features one at a time to see if it can help address the issue – this is because spurious features can sometimes be highly correlated with the target variable. For this, we can use

sequential feature selection algorithms[150] with an updated objective: for example, instead of maximizing classification accuracy, we minimize classification accuracy. For cases where removing features is not so trivial (for example, image and text data), we can also investigate whether removing individual training instances that are different from the test set can address the discrepancy issue.

Reader quiz:

29-A

What is a good performance baseline for the adversarial prediction task?

29-B

Since training datasets are often bigger than test datasets, adversarial validation often results in an imbalanced prediction problem (we have more examples with "*Is test?*" equal to false. Is this an issue, and if so, how can we mitigate that?

[150] SequentialFeatureSelector: The popular forward and backward feature selection approaches (including floating variants), https://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/)

Q30. Limited Labeled Data

> Q:
Suppose we plotted a learning curve[151] and found that the machine learning model overfits and could benefit from more training data. Name different approaches for dealing with limited labeled data in supervised machine learning settings.

> A:

Next to collecting more data, there are several methods more or less related to regular supervised learning that we can use in limited-labeled data regimes.

1) Label more data

Collecting additional training examples is often the best way to improve the performance of a model[152]. However, this is often not feasible in practice. Listed below are various alternative approaches.

2) Bootstrapping the data

Similar to the topics discussed in *Q5, Reducing Overfitting with Data*, it can be helpful to "bootstrap" the data by generating modified (augmented) or artificial (synthetic) training examples to boost the performance of the predictive model.

Of course, improving the quality of data can also lead to improved predictive performance of a model, as discussed in *Q22, Data-Centric AI*.

3) Transfer learning

Transfer learning describes training a model on a general dataset (e.g., ImageNet) and then finetuning the pretrained target dataset (e.g., a specific dataset consisting of different bird species). Transfer

[151] See Q5, Figure 14 for a refresher of what a learning curve looks like.
[152] A learning curve is a good diagnostic to find out whether the model can benefit from more data. See Figure 15 in Q5 for details.

learning is usually done in the context of deep learning, where model weights can be updated. This is in contrast to tree-based methods since most decision tree algorithms are nonparametric models that do not support iterative training or parameter updates[153].

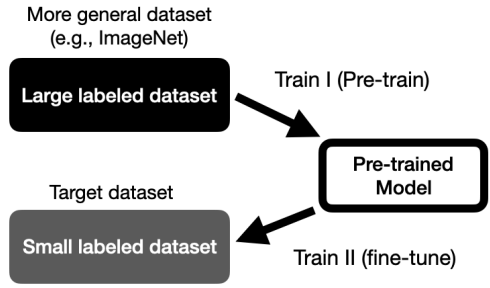


Figure 30.1. Illustration of transfer learning.

4) Self-supervised learning

Similar to transfer learning, self-supervised learning, the model is pretrained on a different task before it is finetuned to a target task for which only limited data exists. However, in contrast to transfer learning, self-supervised learning usually relies on label information that can be directly and automatically extracted from unlabeled data. Hence, self-supervised learning is also often called unsupervised pretraining. Common examples include "next word" (e.g., used in GPT) or "masked word" (e.g., used in BERT) prediction in language modeling. Or, an intuitive example from computer vision includes inpainting: predicting the missing part of an image that was randomly removed. (For more details about self-supervised learning, also see Q2.)

[153]While decision trees for incremental learning are not commonly implemented, algorithms for training decision trees in an iterative fashion do exist (https://en.wikipedia.org/wiki/Incremental_decision_tree).

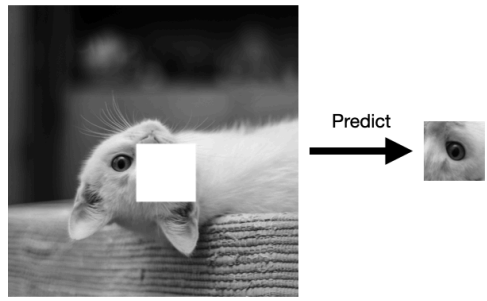


Figure 30.2. Illustration of inpainting for self-supervised learning.

5) Active learning

In active learning, we typically involve manual labelers or users for feedback during the learning process. However, instead of labeling the entire dataset upfront, active learning includes a prioritization scheme for suggesting unlabeled data points for labeling that maximize the machine learning model's performance.

The name *active learning* refers to the fact that the model is *actively* selecting data for labeling in this process. For example, the simplest form of active learning selects data points with high prediction uncertainty for labeling by a human annotator (also referred to as an *oracle*).

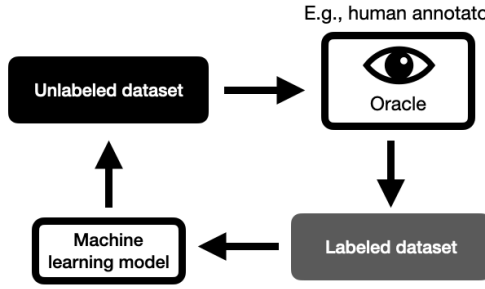


Figure 30.3. Illustration active learning.

6) Few-shot learning

In a few-shot learning scenario, we often deal with extremely small datasets where we usually only have a handful of examples per class. In research contexts, 1-shot (1 example per class) and 5-shot (5 examples per class) are very common.
An extreme case of few-shot learning is zero-shot learning, where no labels are provided. A recently popular example of zero-shot learning is GPT-3 and related language models. Here, the user has to provide all the necessary information via the input prompt, as illustrated in the figure below.

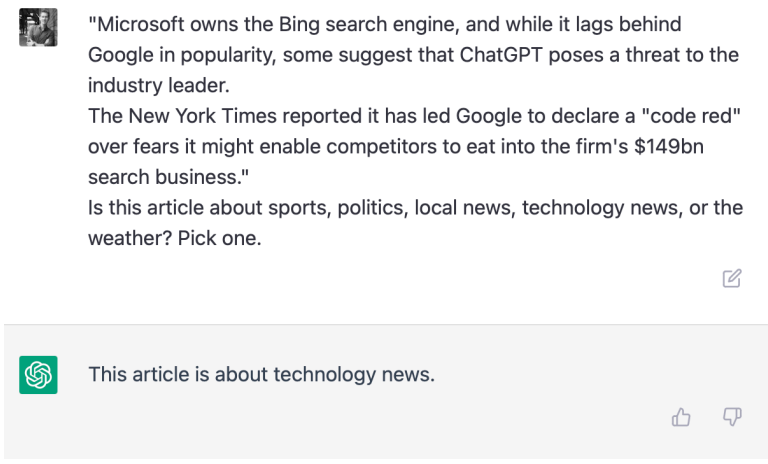


Figure 30.4. Example of zero-shot classification with ChatGPT.

(For more details about self-supervised learning, also see Q3.)

7) Meta-learning

We can think of meta-learning as "learning to learn" – we develop methods that learn how machine learning algorithms can best learn from data.
Over the years, the machine learning community developed several approaches for meta-learning. To further complicate matters, meta-learning can refer to different processes.

Meta-learning is one of the main subcategories of few-shot learning

(mentioned above). Here, the focus is on learning a *good* feature extraction module. The feature extraction module converts support and query images into vector representations. These vector representations are optimized for determining the predicted class of the query example via comparisons with the training examples in the support set. (This form of meta-learning is illustrated in Q3, Figure 13.)

Another branch of meta-learning, unrelated to the few-shot learning approach above, is focused on extracting meta-data (also called meta-features) from datasets for supervised learning tasks. The meta-features are descriptions of the dataset itself. For example, these can include the number of features and statistics of the different features (kurtosis, range, mean, etc.).

The extracted meta-features provide information for selecting a machine learning algorithm for the given dataset at hand. Using this approach, we can narrow down the algorithm and hyperparameter search spaces, which helps reduce overfitting when the dataset is small.

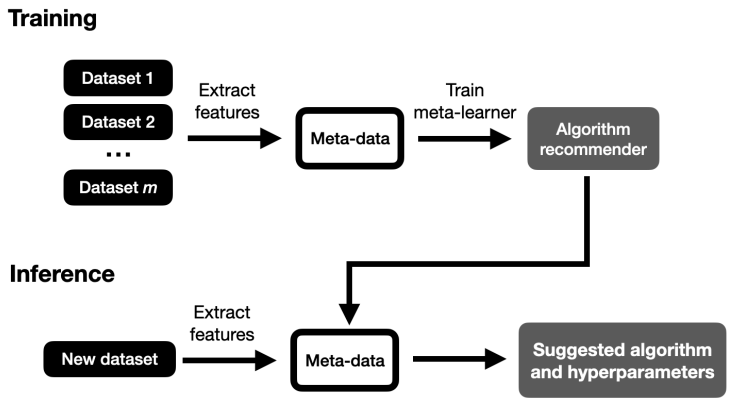


Figure 30.5. Illustration of the meta-learning process involving the extraction of meta-data.

8) Weakly supervised learning

Weakly supervised learning is a procedure where we use an external label source to generate labels for an unlabeled dataset. Often, the labels created by a weakly supervised labeling function are more noisy or inaccurate than those produced by a human or domain expert; hence, the term *weakly* supervised.

Often, we can develop or adopt a rule-based classifier to create the labels in weakly supervised learning – these rules usually only cover a subset of the unlabeled dataset.

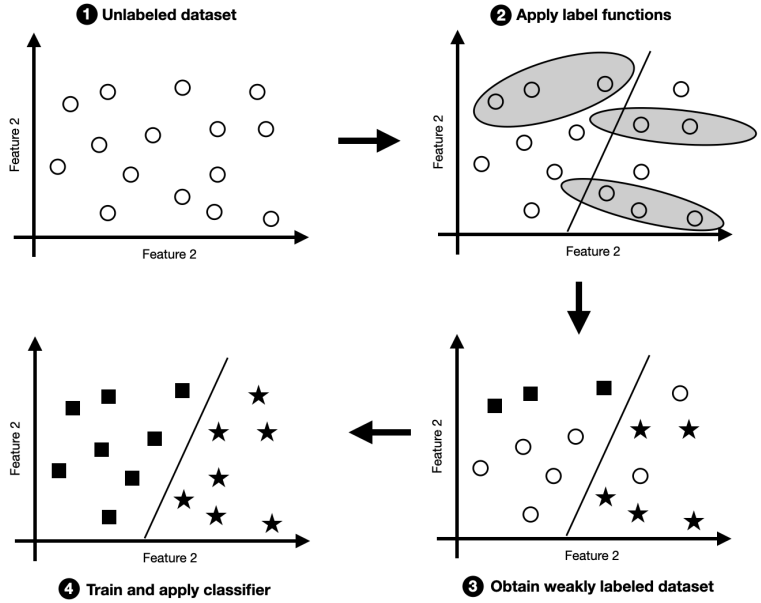


Figure 30.6. Illustration of weakly supervised learning.

Imagine the context of email spam classification as an example of a rule-based approach for data labeling. In weak supervision, we could design a rule-based classifier based on the keyword "SALE" in the email subject header line to identify a subset of spam emails. Note that while we may use this rule to label certain emails as spam-positive, we should not apply this rule to label emails without SALE as non-spam but leave those either unlabeled or apply a different

rule to these[154].

In short, weakly supervised learning is an approach for increasing the number of labeled instances in the training set. Hence, other techniques, such as semi-supervised, transfer, active, and zero-shot learning, are fully compatible with weakly supervised learning.

9) Semi-supervised learning

Semi-supervised learning is closely related to weakly supervised learning described above: we create labels for unlabeled instances in the dataset. The main difference between weakly supervised and semi-supervised learning is *how* we create the labels[155].

In weak supervision, we create labels using an external labeling function that is often noisy, inaccurate or only covers a subset of the data. In semi-supervision, we do not use an external label function but leverage the structure of the data itself.

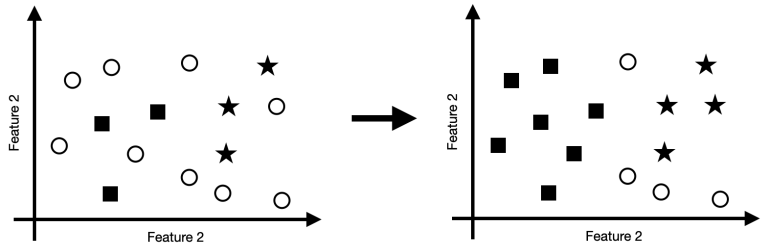


Figure 30.7. Illustration of semi-supervised learning.

In semi-supervised learning, we can, for example, label additional data points based on the density of neighboring labeled data points, as illustrated in the figure below.

While we can apply weak supervision to an entirely unlabeled dataset, semi-supervised learning requires at least a portion of

[154]There is a subcategory of weakly supervised learning referred to as PU-learning. In PU-learnining, which is short for positive-unlabeled learning, we only label and learn from positive examples.

[155]Semi-supervised learning is sometimes referred to as a subcategory of weakly supervised learning and vice versa.

the data to be labeled. In practice, it is possible first to apply weak supervision to label a subset of the data and then use semi-supervised learning to label instances that were not captured by the labeling functions.

10) Self-training

Self-training is a category that falls somewhere between semi-supervised learning and weakly supervised learning. In self-training, we train a model or adopt an existing model to label the dataset. This model is also referred to as a *pseudo-labeler*.

Since the model used in self-training does not guarantee accurate labels, self-training is related to weakly supervised learning. Moreover, while we use or adopt a machine learning model for this pseudo-labeling, self-training is also related to semi-supervised learning.

An example of self-training is knowledge distillation, discussed in Q6.

11) Multi-task learning

Multi-task learning trains neural networks on multiple, ideally related tasks. For example, suppose we are training a classifier to detect spam emails; here, spam classification is the main task. In multi-task learning, we can add one or more related tasks the model has to solve. These additional tasks are also referred to as *auxiliary tasks*. If the main task is email spam classification, an auxiliary task could be classifying the email's topic or language.

Typically, multi-task learning is implemented via multiple loss functions that have to be optimized simultaneously – one loss function for each task. The auxiliary tasks serve as an inductive bias, guiding the model to prioritize hypotheses that can explain multiple tasks. This approach often results in models that perform better on unseen data[156].

[156]Caruana (1997). *Multi-task learning.* Machine Learning. 28: 41–75. https://doi.org/10.1023%2FA%3A1007379606734

There are two subcategories of multi-task learning: (1) multi-task learning with *hard* parameter sharing and (2) multi-task learning with *soft* parameter sharing[157].

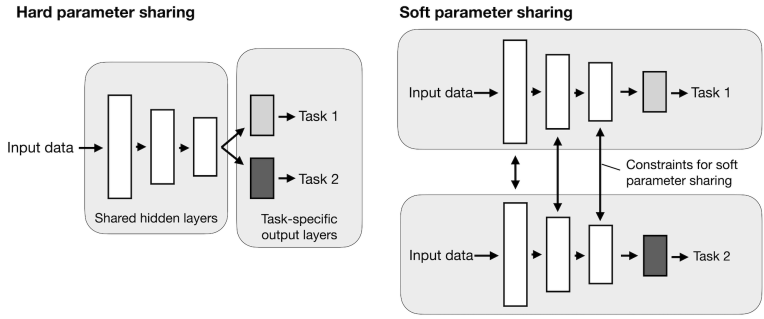


Figure 30.8. Illustration of the two main types of multi-task learning. For simplicity, the figure depicts only two tasks, but multitask learning can be used for any number of tasks.

The figure above illustrates the difference between hard and soft parameter sharing. In hard parameter sharing, only the output layers are task-specific, while all tasks share the same hidden layers and neural network backbone architecture. In contrast, soft parameter sharing uses separate neural networks for each task, but regularization techniques such as distance minimization between parameter layers are applied to encourage similarity among the networks.

12) Multi-modal learning

While multi-task learning involves training a model with multiple tasks and loss functions, multi-modal learning focuses on incorporating multiple types of input data.

Common examples of multi-modal learning are architectures that take both image and text data as input[158]. Depending on the task,

[157]Ruder (2017). *An Overview of Multi-Task Learning in Deep Neural Networks.* https://www.ruder.io/multi-task/.

[158]Multi-modal learning is not restricted two only two modalities and can be used for any number of inputs modalities.

we may employ a matching loss that forces the embedding vectors (Q1) between related images and text to be similar, as shown in the figure below.

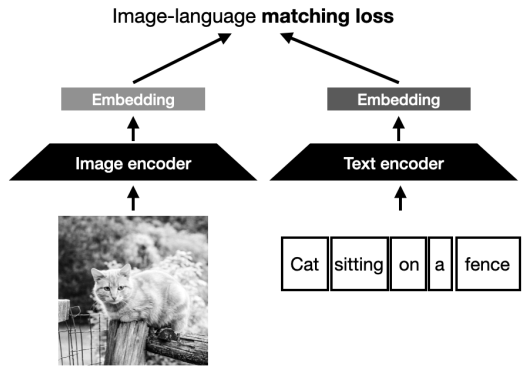


Figure 30.9. Illustration of multi-modal learning with a matching loss that forces embeddings from different types of inputs to be similar.

The figure above shows image and text encoders as separate components. The image encoder can be a convolutional backbone or a vision transformer, and the language encoder can be a recurrent neural network or language transformer. However, it's common nowadays to use a single transformer-based module that can simultaneously process image and text data[159].

Optimizing a matching loss, as shown in the previous figure, can be useful for learning embeddings that can be applied to various tasks, such as image classification or summarization. However, it is also possible to directly optimize the target loss, like classification or regression, as the figure below illustrates.

[159]For example, VideoBERT is a model that with a joint module that processes both video and text for action classification and video captioning. Reference: Sun, Myers, Vondrick, Murphy, Schmid (2019). *VideoBERT: A Joint Model for Video and Language Representation Learning*, https://arxiv.org/abs/1904.01766.

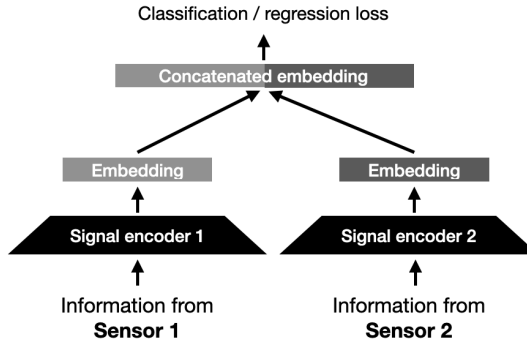


Figure 30.10. Illustration of multi-modal learning for optimizing a supervised learning objective.

Intuitively, models that combine data from different modalities generally perform better than uni-modal models because they can leverage more information. Moreover, recent research suggests that the key to the success of multi-modal learning is the improved quality of the latent space representation[160].

13) Inductive biases

Choosing models with stronger inductive biases can help to lower data requirements by making assumptions about the structure of the data. For example, due to their inductive biases, convolutional networks require less data than vision transformers as discussed in Q13.

Which techniques should we use?

Now that we covered several techniques for lowering the data requirements, which ones should we use?

Collecting more data and techniques such as data augmentation and feature engineering are compatible with all the methods discussed above. Also, multi-task learning and multi-modal inputs can be used with the other learning strategies outlined above. If the

[160]Huang, Du, Xue, Chen, Zhao, Huang, (2021). *What Makes Multi-Modal Learning Better Than Single (Provably)*, https://arxiv.org/abs/2106.04538.

model suffers from overfitting, techniques from Q5 and Q6 should also be included.

How about active learning, few-shot learning, transfer learning, self-supervised learning, semi-supervised learning, and weakly supervised learning? Which technique(s) to try highly depends on the context, and the figure below provides an overview that can be used for guidance.

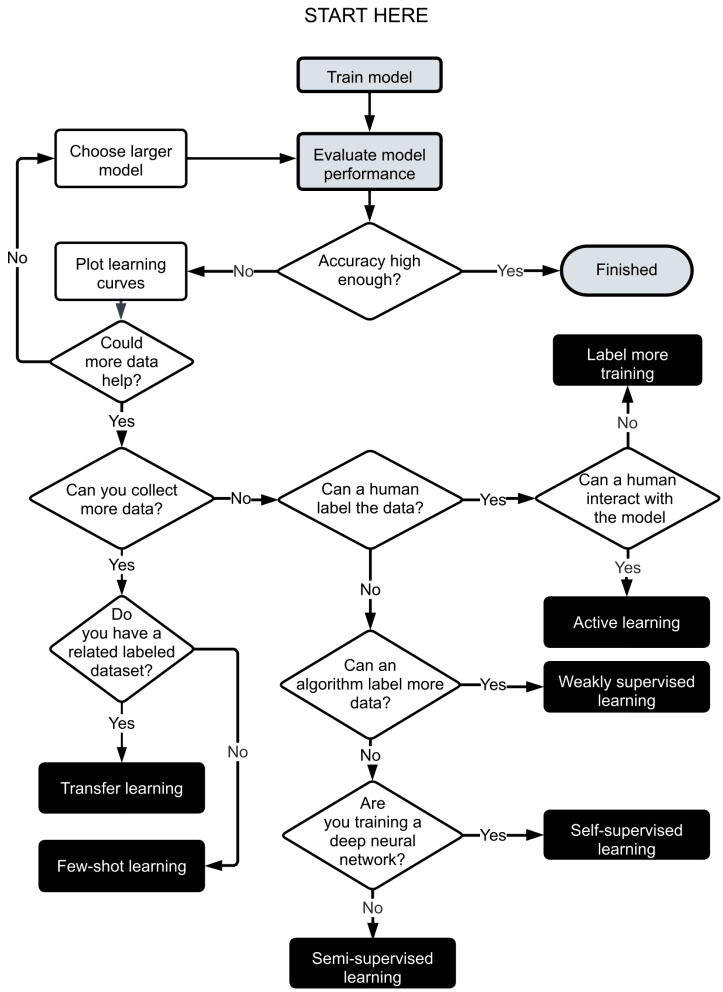


Figure 30.11. Recommendations for choosing a supervised learning techniques. The black boxes are not terminal nodes but arch back to Evaluate model performance (the arrows were omitted to avoid visual clutter).

> Reader quiz:

30-A

Given the task of constructing a machine learning model that utilizes images to detect manufacturing defects on the outer shells of tablet devices similar to iPads, we have access to the following data:

- Millions of images of various computing devices, including smartphones, tablets, and computers, which are not labeled.
- Thousands of labeled pictures of smartphones depicting various types of damage.
- Hundreds of labeled images specifically related to the target task of detecting manufacturing defects on tablet devices.

How could we approach this problem using self-supervised learning or transfer learning?

30-B

In active learning, selecting difficult examples for human inspection and labeling is often based on confidence scores. Neural networks can provide such scores by using the logistic sigmoid or softmax function in the output layer to calculate class-membership probabilities. However, it is widely recognized that deep neural networks exhibit overconfidence on out-of-distribution data[161], rendering their use in active learning ineffective. So, what are some other methods to obtain confidence scores using deep neural networks for active learning?

[161] Nguyen, Yosinski, Clune, (2015). *Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Image*, https://arxiv.org/abs/1412.1897.

Afterword

I hope you found this book to be a valuable and informative resource in the pursuit of knowledge in machine learning and AI.

Whether the book has provided helpful explanations and introductions for future studies or actionable insights that you can apply in your work, it would make me very happy to know that it has benefited you.

If you have found this book valuable, I would be grateful if you could share your experience and spread the word to others who may also find it helpful as well. (Who knows, if there is enough interest, I may start a second volume with a selection of brand-new Q & A's that I stashed away.)

Thank you for reading, and I wish you the best of luck in all your endeavors in the fascinating world of machine learning and AI.

Appendix A: Reader Quiz Solutions

Chapter 1. Neural Networks and Deep Learning

Answer 1-A

Suppose you are training a convolutional network with five convolutional layers followed by three fully connected (FC) layers similar to AlexNet. You can think of these fully connected layers as two hidden layers and an output layer in a multilayer perceptron. Which of the neural network layers can be utilized to produce useful embeddings?

The final layer before the output layer – the second fully connected layer in this case – may be most useful for embeddings.
However, we could also use all other intermediate layers to create embeddings. Since the later layers tend to learn higher-level features, these later layers are typically more semantically meaningful and better suited for different types of tasks, including related classification tasks.

Answer 1-B

Name at least one type of input representation that is not an embedding.

- One-hot encoding (covered in Q1)
- Histogram (for example, an image histogram[162] created from an image)
- A bag-of-word[163] representation of an input sentence

[162] https://en.wikipedia.org/wiki/Image_histogram
[163] https://en.wikipedia.org/wiki/Bag-of-words_model

Answer 2-A

How could we apply self-supervised learning to video data?

- Predict the next frame in the video (analogous to next-word prediction in large language models such as GPT)
- Predict missing or masked frames (analogous to large language models such as BERT)
- Inpainting: Predict masked or missing parts (pixel areas) in a video
- Coloring: convert the video to grayscale and then train the model to predict the color

Answer 2-B

Can self-supervised learning be used for tabular data represented as rows and columns? If so, how can we approach this?

We can remove (mask) feature values and train a model to predict these (analogous to classic data imputation). An example of a method that uses this approach is TabNet[164].

It is also possible to use contrastive learning by generating augmented versions of the training examples in the original raw feature space or the embedding space. For example, the SAINT[165] method employs this approach.

Answer 3-A

How could we partition the MNIST dataset, consisting of 50,000 handwritten digits from 10 classes (0-9), for a one-shot classification context?

Similar to a supervised learning approach, we first divide the dataset into a training set and a test set. We then further divide

[164]Arik andPfister (2019). *TabNet: Attentive Interpretable Tabular Learning*, https://arxiv.org/abs/1908.07442.

[165]Somepalli, Goldblum, Schwarzschild, Bruss, and Goldstein (2021). *SAINT: Improved Neural Networks for Tabular Data via Row Attention and Contrastive Pre-Training*, https://arxiv.org/abs/2106.01342.

the training and test sets into subsets, one image from each class. To design the training task, we only consider a subset of classes, for example, the classes (digits) 0, 1, 2, 5, 6, 8, 9. Then, for testing, we use the remaining classes 3, 4, 7. For each classification task, the neural network receives only one example per image.

Answer 3-B

Can you think of some real-world applications or use cases for few-shot learning?

- Consider a medical imaging scenario for a rare disease. A few-shot system may only have one or a handful of cases for this disease and is asked to identify it based on this limited number of examples.
- Another example of a few-shot system is a recommender that only has a limited number of items a user rated. Based on this limited number of examples, the model has to predict future products the user may like.
- Imagine a warehouse robot that has to learn to recognize new objects as a company increases its inventory. The robot has to learn to recognize and adapt to these new objects based on only a few examples.

Answer 4-A

Suppose you are trying out the lottery ticket hypothesis approach and find that the performance of the subnetworks is not very good (compared to the original network). What are some next steps to try?

- Increase the size of the initial neural network. It might be possible that the chosen network is too small to contain a suitable subnetwork.
- Try a different random initialization (e.g., by changing the random seed). The lottery hypothesis assumes that *some*

randomly initialized networks contain highly-accurate sub-networks that can be obtained by pruning, but not all networks may have such sub-networks.

Answer 4-B

How is the lottery ticket hypothesis related to training a neural network with ReLU activation functions (a ReLU activation function is defined as max(0, x)?

When training a neural network with ReLU activation functions, specific activations will be set to zero if the function input is less than zero. This will cause certain nodes in the hidden layers to not contribute to the computations – this is sometimes referred to as "dead neurons." While ReLU activations do not directly cause sparse weights, the zero activation outputs sometimes lead to zero weights that are not recoverable.
This observation supports the lottery hypothesis, which suggests that well-trained networks may contain sub-networks with sparse, trainable weights that can be pruned without loss of accuracy.

Answer 5-A

Suppose we train an XGBoost model to classify images based on manually extracted features we obtained from our collaborators. The dataset of labeled training examples is relatively small, but fortunately, your collaborators also have a labeled training set from an older project on a related domain.
We are considering implementing a transfer learning approach to train the XGBoost model. Is this a feasible option, and if yes, how could we do it? (Assume we are only allowed to use XGBoost, no other classification algorithm or model.)

XGBoost is a tree-based gradient-boosting implementation that does not support transfer learning (as of this writing). In contrast to artificial neural networks, XGBoost is a non-parametric model that we cannot readily update as new data arrives; hence, regular transfer learning would not work here.

However, it is possible to use the results of an XGBoost model trained on one task as features for another XGBoost model. So, consider an overlapping set of features for both datasets. For example, we could design a classification task in a self-supervised fashion for the combined dataset. Then, we train a second XGBoost model on the target dataset that takes the original feature set as input, along with the output of the first XGBoost model.

Answer 5-B

Suppose we are working on an image classification problem (for this example, consider MNIST-based handwritten digit recognition) and added a decent amount of data augmentation to reduce overfitting in an image classification context. Unfortunately, we observe that the classification accuracy became much worse than before. What are some potential reasons?

When applying data augmentations, we usually have to increase the training time as well; it is possible that we needed to train the model longer.

Or, we may have applied too much data augmentation. Augmenting the data too much can result in excessive variations that do not reflect the natural variations in the data, leading to overfitting or poor generalization to new data. In the case of MNIST, this can also include translating or cropping the image too strongly such that the digits become unrecognizable due to missing parts.

Another possibility is that we applied naive, domain-inconsistent augmentation. For example, suppose we are mirroring or flipping images vertically or horizontally. For MNIST does not make sense for handwritten digits since this would create numbers that don't exist in the real world.

Answer 6-A

Suppose we are using early-stopping as a mechanism to reduce overfitting. In particular, we are using a modern variant that creates checkpoints of the best model (for instance, the model with highest

validation accuracy) during training so that we can load it after the training has completed – this is a mechanism that can be enabled in most modern deep learning frameworks.

Now, a colleague recommends tuning the number of training epochs instead. What are some of the advantages and disadvantages of each approach?

Tuning the number of training epochs is a simpler and more universal approach – especially older frameworks don't support model checkpointing, so changing the number of training epochs may be a easier solution, which is particularly attractive for small datasets and models where each hyperparameter configuration is cheap to run and evaluate. This approach also eliminates the need for monitoring the performance on a validation set during training, making it a straightforward and easy-to-use method.

The early-stopping and checkpointing approach is particularly attractive if we work with models that are expensive to train. It's generally also a more flexible and robust method for preventing overfitting. However, a downside of this approach can be that in noisy training regimes, we may end up prioritizing an early epoch even though the validation set accuracy is not a good estimate of the generalization accuracy.

Answer 6-B

Ensemble models have been established as a reliable and successful method for decreasing overfitting and enhancing the reliability of predictive modeling efforts. However, there is always a trade-off. What are some of the drawbacks associated with ensemble techniques?

One obvious downside of ensemble methods is the increased computational cost. For example, if we build a neural network ensemble of 5 neural networks, this ensemble can be five times as expensive as every single model.

While we often consider the inferencing costs mentioned above, the

increased storage cost is another significant limitation. Nowadays, most computer vision and language models have millions or even billions of parameters that have to be stored in a distributed setting. Model ensembling complicates this further.

Reduced interpretability is yet another cost that we incur when using model ensembles. Understanding and analyzing the predictions of a single model can already be challenging. Depending on the ensembling approach, we add yet another layer of complexity that reduces interpretability.

Answer 7-A

Suppose we are implementing our own version of tensor parallelism, which works great when we train our model with an SGD (standard stochastic gradient descent) optimizer.
However, when we try the Adam optimizer, we encounter an out-of-memory device. What could be a potential problem explaining this issue?

The Adam optimizer implements an adaptive method that comes with internal weight parameters. Adam has 2 optimizer parameters (mean and variance) per model parameter. So, instead of splitting the weight tensors of the model, we also have to split the optimizer states to work around memory limitations. (Note that this is already implemented in most DeepSpeed parallelization techniques.)

Answer 7-B

Suppose we don't have access to a GPU and are considering using data parallelism on the CPU. Is this a good idea?

Data parallelism could conceptually work on a CPU, but the benefits would be limited. For example, instead of duplicating the model in CPU memory to train multiple models on different batches of the dataset in parallel, it could make more sense to increase the data throughput.

Answer 8-A

Above, we discussed that self-attention is easily parallelizable. And yet, transformers are considered computationally expensive due to self-attention. How can we explain this contradiction?

Self-attention has quadratic compute and memory complexity due to the n-to-n comparisons (where n is the input sequence length), which makes transformers computationally costly compared to other neural network architectures. Moreover, decoder-style transformers such as GPT generate outputs one token at a time, which cannot be parallelized during inference – although generating each token is still highly parallelizable as discussed above.

Answer 8-B

Since self-attention scores represent importance weights for the various input elements, can we consider self-attention a form of feature selection?

Yes, we can think of self-attention as a form of feature selection, although there are differences between this and other types of feature selection. It is important to differentiate between hard and soft attention in this context. Soft attention computes importance weights for all inputs, whereas hard attention selects a subset of the inputs. Hard attention is more like masking, where certain inputs are set to 0 or 1, while soft attention allows for a continuous range of importance scores.
However, the main difference between attention and feature selection is that feature selection is typically a fixed operation, while attention weights are computed dynamically based on the input. With feature selection algorithms, the selected features are always the same, whereas, with attention, the weights can change based on the input.

Answer 9-A

How would you evaluate the quality of the images generated by an generative AI model?

Automating this evaluation is inherently difficult, and the gold

standard is currently human evaluation and judgment. However, a few metrics exist as quantitative measures.

To evaluate the diversity of the generated images, one can compare the distribution between the conditional class distribution and the marginal class distribution of generated samples, for example, using a Kullback-Leibler divergence, which is a measure that is also used in the VAE to make the latent space vectors similar to a standard Gaussian. The higher the KL, the more diverse the generated images.

One can also compare the statistics of generated images to real images in the feature space of a pre-trained model. A low similarity indicates that the two distributions are close to each other, which is generally a sign of better image quality. This approach is also often known as Fréchet Inception Distance.

Answer 9-B

Given the description of consistency models above, how would you use them to generate new images?

Like the generators of GANs, VAEs, or diffusion models, a consistency model takes a noise tensor sampled from a simple distribution (such as a standard Gaussian) as input and generates a new image.

Answer 10-A

Suppose we trained a neural network with top-k or nucleus sampling where k and p are hyperparameter choices. Can we make the model behave deterministically during inference without changing the code?

Yes, we can make top-k sampling deterministic by setting *k=1* so that the model will always select the word with the highest probability score as the next word when generating the output text.

We can also make nucleus sampling deterministic, for instance, by setting the probability mass threshold p to 1. By setting p to 1, you include the entire probability distribution of the tokens in the

nucleus. This would make the model always choose the token with the highest probability.

Answer 10-B

Can you think of scenarios where deterministic dropout behavior during inference is desired?

In some cases, the random behavior of dropout during inference can be desirable for building model ensembles.

Answer 11-A

Suppose we want to optimize the neural network using plain stochastic gradient descent (SGD) or an Adam optimizer. What are the respective numbers of parameters that need to be stored for SGD and Adam?

SGD only has the learning rate as a hyperparameter, but it does not have any parameters. So, it does not add any additional parameters to be stored besides the gradients calculated for each weight parameter during backpropagation.

The Adam optimizer is more complex and requires more storage. Specifically, Adam keeps an exponentially decaying average of past gradients (first moment) and an exponentially decaying average of past squared gradients (second raw moment) for each parameter. Therefore, for each parameter in the network, Adam needs to store two additional values. So, if we have n parameters in the network, Adam requires storage for $2n$ additional parameters.

If the network has n trainable parameters, Adam adds $2n$ parameters to be tracked. For example, in the case of AlexNet, which consists of 26,926 as calculated earlier, Adam requires 53,852 additional values in total (two times 26,926).

Answer 11-B

Suppose we are adding three batch normalization (BatchNorm) layers: one after the first convolutional layer, one after the second convolutional layer, and another one after the first fully connected

layer – we typically do not want to add BatchNorm layers to the output layer. How many additional parameters do these three BatchNorm layers add to the model?

Each BatchNorm layer learns two sets of parameters during training: a set of scaling coefficients (gamma) and a set of shifting coefficients (beta). These are learned so that the model can undo the normalization when it is found to be detrimental to learning.

Each of these sets of parameters (gamma and beta) has the same size as the number of channels (or neurons) in the layer they normalize because these parameters are learned separately for each channel (or neuron).

So, for the first BatchNorm layer following the first convolutional layer with 5 output channels, this adds 10 additional parameters. For the second BatchNorm layer, following the second convolutional layer with 12 output channels, this adds 24 additional parameters.

The first fully connected layer has 128 output channels, which means 256 additional BatchNorm parameters. The second fully connected layer is not accompanied by a BatchNorm layer since it's the output layer.

So, BatchNorm adds $10 + 24 + 256 = 290$ additional parameters to the network.

Answer 12-A

How would increasing the stride affect the equivalence discussed above?

Just increasing the stride from one to two (or larger values) should not affect the equivalence since the kernel size is equal to the input size in both scenarios, so there is no sliding-window mechanism at play here.

Answer 12-B

Does padding affect the equivalence between fully connected layers and convolutional layers?

Increasing the padding to values larger than zero will affect the results. Due to the padded inputs, we will have the sliding-window convolutional operation where the equivalence with fully connected layers no longer holds. In other words, the padding would alter the input's spatial dimensions, which would no longer match the kernel size, and result in more than one output value per feature map.

Answer 13-A

Consider the patchification of the input images illustrated in the figure above. The size of the resulting patches controls a computational and predictive performance tradeoff. The optimal patch size depends on the specific application and desired trade-off between computational cost and model performance. Do smaller patches typically result in higher or lower computational costs?

Using smaller patches increases the number of patches for a given input image, leading to a higher number of tokens being fed into the transformer. This results in increased computational complexity, as the self-attention mechanism in transformers has quadratic complexity with respect to the number of input tokens. Consequently, smaller input patches make the model computationally more expensive.

Answer 13-B

Following up on the question above, do smaller patches typically lead to a higher or lower prediction accuracy?

Using larger input patches may result in the loss of finer details and local structures in the input image, which can potentially negatively affect the model's predictive performance. Interested readers might enjoy the *FlexiViT*[166] paper that studies the computational and

[166]Beyer, Izmailov, Kolesnikov, Caron, Kornblith, Zhai, Minderer, Tschannen, Alabdulmohsin, Pavetic. *FlexiViT: One Model for All Patch Sizes*, https://arxiv.org/abs/2212.08013.

predictive performance trade-offs as a consequence of the patch size and number.

Chapter 3. Natural Language Processing

Answer 15-A

Consider the case of homophones – words that sound the same but have different meanings. For example, the words there and their sound the same but have different meanings. Are homophones another example of when the distributional hypothesis does not hold?

Homophones are words that are spelled differently and sound the same but have different meanings. Due to the different meanings, we expect two homophones to appear in other contexts. For example, "I can see you over there" and "Their paper is very nice."

Since the distributional hypothesis says that words with similar meanings should appear in similar contexts, homophones do not contradict the distribution

Answer 15-B

Can you think of other domains where a concept similar to the distributional hypothesis applies?

The underlying idea of the distributional hypothesis can be applied to other domains, for example, computer vision. In the case of images, objects that appear in similar visual contexts are likely to be semantically related. Or, on a lower level, neighboring pixels are likely semantically related as they are part of the same object – this idea is used in masked autoencoding for self-supervised learning on image data[167].

Another example is protein modeling. For example, researchers showed[168] that language transformers that are trained on protein

[167]We covered masked autoencoders in Q2.

[168]Rives, Meier, Sercu, Fergus (2020). *Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences*, https://www.biorxiv.org/content/10.1101/622803v1.full.

sequences (a string representation where each letter represents an amino acid, for example, "MNGTEGPNFYVPFSNKTGVV...") learn embeddings where similar amino acids cluster together. For example, the hydrophobic amino acids such as V, I, L, and M appear in one cluster, and aromatic amino acids such as F, W, and Y appear in another cluster. In this context, we can think of an amino acid as an equivalent to a word in a sentence.

Answer 16-A

Could the use of text data augmentation help with privacy concerns?

Assuming that the existing data does not suffer from privacy concerns, data augmentation helps generate variations of the existing data without the need to collect additional data, which can help with privacy concerns.

However, if the original data includes personally identifiable information, even augmented or synthetic data could potentially be linked back to individuals, especially if the augmentation process doesn't sufficiently obscure or alter the original data.

Answer 16-B

What are some instances where data augmentation may not be beneficial for a specific task?

Data augmentation might be less beneficial if the original dataset is already large and diverse enough that the model isn't overfitting or underperforming due to a lack of data. This is, for example, often the case when pretraining LLMs.

Also, the performance of highly domain-specific models (for example, in the medical, law, and financial domains) could be adversely affected by techniques such as synonym replacement and back translation due to replacing domain-specific terms with a certain meaning.

In general, in contexts of tasks highly sensitive to wording choices, data augmentation must be applied with particular care.

Answer 17-A

Considering that self-attention compares each sequence element with itself, what is the time and memory complexity of self-attention?

The self-attention mechanism has quadratic time and memory complexity.

More precisely, we can express the time and memory complexity of self-attention as $O(N^2 \times d)$ where N is the length of the sequence, and `$d` is the dimensionality of the embedding of each element in the sequence.

This is because self-attention involves computing a similarity score between each pair of elements in the sequence, resulting in an $N \times N$ similarity matrix. This matrix is then used to compute weighted averages of the sequence elements, resulting in an $N \times d$ output representation.

This can make self-attention computationally expensive and memory-intensive, particularly for long sequences or large values of d.

Answer 17-B

We discussed self-attention in the context of natural language processing. Could this mechanism be useful for computer vision applications as well?

Yes. Interestingly, self-attention may partly be inspired by the spatial attention mechanisms[169] used in convolutional neural networks for image processing. Spatial attention is a mechanism that allows a neural network to focus on specific regions of an image that are relevant to a given task. It works by selectively weighting the importance of different spatial locations in the image, which allows the network to "pay more attention" to certain areas and ignore others.

[169] Xu, Ba, Kiros, Cho, Courville, Salakhutdinov, Zemel, Bengio (2015). *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, https://arxiv.org/abs/1502.03044.

Answer 18-A

As discussed earlier, BERT-style encoder models are pretrained using masked language modeling and next-sentence prediction pretraining objectives. How could we adopt such a pretrained model for a classification task, for example, predicting whether a text has a positive or negative sentiment?

If we want to adapt a pretrained BERT model for classification, you need to add an output layer for classification – this is often referred to as classification head.

As discussed, BERT uses a [CLS] token for the next-sentence prediction task during pretraining. Instead of training it for next-sentence prediction, we can finetune a new output layer for our target prediction task, for example, sentiment classification.

The embedded [CLS] output vector serves as a summary of the entire input sequence. We can think of it as a feature vector and train a small neural network on top of it, typially a fully connected layer followed by a softmax activation function to predict the class probabilities. The fully connected layer's output size should match the number of classes in our classification task. Then, we can train it using backpropagation as usual. Different finetuning strategies (updating all layers versus only th

Answer 18-B

Can we finetune or a decoder-only model like GPT for classification?

Yes, we can finetune a decoder-only model like GPT for classification tasks, although it might not be as effective as using encoder-based models like BERT.

In contrast to BERT, we do not need to use a special [CLS], but the fundamental concept is similar to finetuning an encoder-style model for classification. We add a classification head (a fully connected layer and a softmax activation) and train it on the

embedding (the final hidden state) of the first output token. (This is analogous to using the [CLS] token embedding.)

Answer 19-A

When does it make more sense to use in-context learning over finetuning and vice versa?

In-context learning is useful if we don't have access to the model. Besides, in-context learning is useful if we want to adapt the model to similar tasks that the model wasn't trained to do.

In contrast, finetuning is useful for adapting the model to a new target domain. For example, suppose the model was pretrained on a general corpus, and we want to apply it to financial data or documents. Here, it would make sense to finetune the model on data from that target domain.

Note that in-context learning can be used with a finetuned model as well. For example, when a pretrained language model is finetuned on a specific task or domain, in-context learning then leverages the model's ability to generate responses based on the context provided within the input that may be more accurate given the target domain compared to in-context learning without finetuning.

Answer 19-B

In prefix tuning, adapters, and LoRA, how can we ensure that the model preserves (and not forgets) the original knowledge?

This is done implicitly. In prefix tuning, adapters, and LoRA, the original knowledge of the pretrained language model is preserved by keeping the core model parameters frozen while introducing additional learnable parameters that adapt to the new task.

Answer 20-A

Looking at step 5 in the previous figure illustrating the BERTScore computation, we see that the cosine similarity between the two embeddings of "cat" are not 1.0, where 1.0 indicates a maximum cosine similarity. Why is that?

If we used an embedding technique that processes each word independently, for example, Word2Vec, we would expect the cosine similarity between the "cat" embeddings to be 1.0. However, we are using a transformer model to produce the embeddings in this case. Transformers use self-attention mechanisms[170] that consider the whole context (i.e., input text) when producing the embedding vectors. Since the word "cat" is used in two different sentences, the BERT model produces a different embedding for these two instances of "cat."

Answer 20-B

In practice, we might find that the BERTScore is not symmetric. This means that switching the candidate and reference sentence could result in different BERTScores for specific texts. How could we address this?

Switching the candidate and reference texts has the same effect as calculating the maximum cosine similarity scores across columns (as shown in step 5 of the previous figure) versus rows, which can result in different BERTScores for specific texts. That's why the BERTScore is often computed as an F1 score similar to ROUGE in practice. I.e., we calculate the BERTScore one way ("recall") and the other ("precision") and then compute the harmonic mean (F1 score).

Chapter 4. Production, Real-World, And Deployment Scenarios

Answer 21-A

Suppose you train a classifier for stock trading recommendation using a random forest model, including the moving average of the stock price as a feature. Now, since new stock market data arrives daily, you are thinking about how to update the classifier daily to keep it up to date. Should you take a stateless training or stateless retraining approach to update the classifier?

Random forests, typically based on CART decision trees, cannot

[170] See Q17 for more information about self-attention.

be readily updated as new data arrives. Hence, a stateless training approach would be the only viable option. On the other hand, suppose we switched to using neural network models such as recurrent neural networks. In that case, a stateful approach could make more sense since the neural network could be readily updated on new data. (However, in the beginning, comparing stateful and stateless systems side by side is always a good idea before deciding which method works best.)

Answer 21-B

Suppose you deployed a large language model (transformer) such as ChatGPT that can answer user queries. The dialogue interface includes a "thumbs up" and "thumbs down" button so that users can give direct feedback based on the generated queries. While collecting the user feedback, you are not updating the model immediately as new feedback arrives. However, you are planning to release a new or updated model at least once per month. Is this a good candidate for stateless or stateful retraining?

A stateful retraining approach makes the most sense here. Instead of training a new model on a combination of existing data, including user feedback, it makes more sense to update the model based on user feedback. Large language models are usually pretrained in a self-supervised fashion and then finetuned via supervised learning. Training large language models is very expensive, so updating the model via stateful retraining makes sense rather than training it from scratch again.

Answer 22-A

A recent trend is the increased use of predictive analytics in healthcare. For example, suppose your healthcare provider developed an AI system that analyzes patients' electronic health records and provides recommendations for lifestyle changes or preventive measures. For this, the provider requires you to monitor and share your health data (such as pulse and blood pressure) daily. Is this an example of data-centric AI?

From the information provided, it is unclear whether this is a data-centric approach. The AI system relies heavily on data inputs to make predictions and recommendations, but that's true for any machine learning approach for AI. To determine whether this approach is an example of data-centric AI, we need to know how the AI system was developed. If the AI system was developed by using a fixed model and refining the training data, this could qualify as a data-centric approach; otherwise, it's just regular machine learning and predictive modeling.

Answer 22-B

Suppose you train a convolutional neural network, specifically a ResNet-34, to classify images in the CIFAR-10 and ImageNet datasets. To reduce overfitting and improve classification accuracy, you are experimenting with different data augmentation techniques, such as image rotation and cropping. Is this approach data-centric?

If we are keeping the model fixed – that means reusing the same ResNet-34 architecture – and only changing the data augmentation approach to investigate its influence on the model performance, we could consider this a data-centric approach. However, data augmentation is also routinely done as part of any modern machine learning pipeline, and the use of data augmentation alone does not imply whether it's a data-centric approach. A data-centric approach, under the modern definition, suggests that we actively study the difference between various dataset-enhancing techniques while keeping the remaining modeling and training pipeline fixed.

Answer 23-A

Q7 covered several different multi-GPU training paradigms to speed up the model training. Using multiple GPUs can, in theory, also speed up model inference. However, in practice, there are several reasons why this approach is often not the most efficient or practical one. Can you think of what these reasons are?

For instance, one downside of using multi-GPU strategies for infer-

ence is the additional communication overhead between the GPUs. However, for inference tasks, which are relatively small compared to training since they don't require gradient computations and updates, the time it takes to communicate between GPUs could outweigh the time saved by parallelization.

Also, managing multiple GPUs means higher equipment and energy costs. In practice, optimizing models for single-GPU or CPU performance is usually more worthwhile. If multiple GPUs are available, processing multiple samples in parallel on separate GPUs often makes more sense than processing the same sample via multiple GPUs.

Answer 23-B

Vectorization and loop tiling (or loop blocking) are two different strategies for optimizing operations that involve accessing array elements. When would you use which?

Loop tiling is often combined with vectorization. For example, after applying loop tiling, each tile can be processed using vectorized operations. This allows us to use SIMD instructions on data that is already in the cache, increasing the effectiveness of both techniques.

Chapter 5. Predictive Performance and Model Evaluation

Answer 25-A

Suppose we want to predict the number of goals a soccer player will score in a particular season. Is this a problem that can be solved using ordinal regression or Poisson regression?

If we try to predict the number of goals a player scores (based on data from past seasons, for example), it's a Poisson regression problem. On the other hand, we could also apply an ordinal regression model to the different players to rank them by the number of goals they will score. However, since the goal difference is constant and can be quantified (for example, the difference between 3 and 4 goals

is the same as 15 and 16 goals), it's not an ideal problem for an ordinal regression model.

Answer 25-B

Suppose you asked someone to sort the last three movies they have watched based on their order of preference. Ignoring the fact that this dataset is a tad to small for machine learning, what approach would be best suited for this kind of data?

This is a ranking issue that resembles an ordinal regression issue, but there are some differences. Since we are only aware of the relative order of the movies, a pairwise ranking algorithm might be a more appropriate solution than an ordinal regression model.

However, if the person is asked to assign numerical labels to each movie on a scale, such as 1 to 5 (similar to the star rating system on Amazon), then it would be possible to train and use an ordinal regression model on this type of data.

Answer 27-A

Suppose we consider using the mean absolute error (MAE) as an alternative to the root mean square error (RMSE) for measuring the performance of a machine learning model, where

$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$

and

$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}$.

However, a colleague argues that the MAE is not a proper distance metric in metric space because it involves an absolute value, so we should prefer using the RMSE. Is this argument correct?

Since the MAE is based on an absolute value around the distance, it naturally satisfies the first criterion: it can't be negative. Also, the MAE is the same if we swap the values y and $\hat{y}$; hence, it satisfies the second criterion. How about the triangle inequality? Similar to how the RMSE is the same as the Euclidean distance or L2 norm, the

MAE is similar to the L1 norm between two vectors. Since all vector norms satisfy the triangle inequality[171], our colleague is incorrect.

Furthermore, even if the MAE were not a proper metric, it can still be a useful model evaluation metric; for example, consider the classification accuracy.

Answer 27-B

*Based on your answer to **27-A**, would you say that the MAE is better than the RMSE, or vice versa?*

The MAE assigns equal weight to all errors, while the RMSE places more emphasis on errors with larger absolute values due to the quadratic exponent. As a result, the RMSE is always larger than the MAE. However, no metric is universally better than the other, and they have both been used to assess model performance in countless studies over the years[172].

Answer 28-A

To provide the model with as much training data, we consider using leave-one-out cross-validation (LOOCV). LOOCV is a special case of k-fold cross-validation where k is equal to the number of training examples such that the validation folds only contain a single data point. A colleague mentions that LOOCV is defect for discontinuous loss function and performance measures such as classification accuracy. For instance, for a validation fold consisting of only one example, the accuracy is always either 0 (0%) or 1 (99%). Is this really a problem?

This is not a problem if we only care about the average performance. For example, If we have a dataset of 100 training examples, and the model predicts 70 out of the 100 validation folds correctly, we estimate the model accuracy as 70%. However, suppose we

[171]Horn and Johnson: Matrix Analysis, Cambridge University Press, 1990.

[172]If you are interested in additional comparisons between MAE and RMSE, you may like the following article by Willmott and Matsuura: *Advantages of the Mean Absolute Error (MAE) over the Root Mean Square Error (RMSE) in assessing average model performance*, https://www.int-res.com/abstracts/cr/v30/n1/p79-82.

are interested in analyzing the variance of the estimates from the different folds. In that case, LOOCV is not very useful since each fold only consists of a single training example, so we cannot compute the variance of each fold and compare it to other folds.

Answer 28-B

We discussed model selection and model evaluation as two use cases of k-fold cross-validation. Can you think of other use cases?

Another use case of *k*-fold cross-validation is model ensembling. For example, in 5-fold cross-validation, we train five different models since we have five slightly different training sets. However, instead of training a final model on the whole training set, we can combine the five models into a model ensemble (this is particularly popular on Kaggle). See Figure 18 in Q6 for an illustration of this process.

Answer 29-A

What is a good performance baseline for the adversarial prediction task?

As a performance baseline, it's a good idea to implement a zero-rule classifier, i.e., a majority class classifier. Since we typically have more training than test data, we can compute the performance of a model that always predicts "Is test? False", which should result in 70% accuracy if we have partitioned the original dataset into 70% training data and 30% test data. If the of the model trained on the adversarial validation dataset exceeds this baseline noticeably (say 80%), then we may have a serious discrepancy issue to investigate further.

Answer 29-B

*Since training datasets are often bigger than test datasets, adversarial validation often results in an imbalanced prediction problem (we have more examples with "*Is test?*" equal to false. Is this an issue, and if so, how can we mitigate that?*

Overall, this is not a big issue since we are mainly interesting if there is a strong deviation from a random baseline. I.e., if we compare the accuracy of the adversarial validation model against the baseline (rather than 50% accuracy) then there should be no issue. However, it may be even better to consider evaluation metrics like Matthew's correlation coefficient or ROC or precision-recall area-under-the-curve values instead of classification accuracy.

Answer 30-A

Given the task of constructing a machine learning model that utilizes images to detect manufacturing defects on the outer shells of tablet devices similar to iPads, we have access to the following data:

- *Millions of images of various computing devices, including smartphones, tablets, and computers, which are not labeled.*
- *Thousands of labeled pictures of smartphones depicting various types of damage.*
- *Hundreds of labeled images specifically related to the target task of detecting manufacturing defects on tablet devices.*

How could we approach this problem using self-supervised learning or transfer learning?

- While we often like to think of self-supervised learning and transfer learning as separate approaches, they don't have to be exclusive. For instance, we could pretrain a model on a labeled or larger unlabeled image data using self-supervised learning (*here:* the million of unlabeled images corresponding to the various computing devices).
- Instead of starting with random weights, we can use the neural network weights from self-supervised learning to follow up with transfer learning via the thousands of labeled smartphone pictures. Since smartphones are related to tablets, transfer learning is a very promising approach here.

- Finally, after the self-supervised pretraining and transfer learning, we can finetune the model on the hundreds of labeled images of the target task, the tablets.

Answer 30-B

In active learning, selecting difficult examples for human inspection and labeling is often based on confidence scores. Neural networks can provide such scores by using the logistic sigmoid or softmax function in the output layer to calculate class-membership probabilities. However, it is widely recognized that deep neural networks exhibit overconfidence on out-of-distribution data[173]*, rendering their use in active learning ineffective. So, what are some other methods to obtain confidence scores using deep neural networks for active learning?*

Besides mitigation techniques for the overconfident scores from a neural network's output layer, we can also consider various ways of ensembling to obtain confidence scores. For instance, instead of disabling dropout during inference, we can leverage dropout to obtain multiple different predictions for a single example to compute the predicted label uncertainty.

Another option is to construct model ensembles from different segments of the training set using k-fold cross-validation, as discussed in the ensemble section of Q6.

[173] Nguyen, Yosinski, Clune, (2015). *Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Image*, https://arxiv.org/abs/1412.1897.

Appendix B: List of Questions

Below is a list of all questions for readers who prefer a quick, spoiler-free overview before reading the book.

Chapter 1. Neural Networks and Deep Learning

- **Q1**. In deep learning, we often use the terms *embedding vectors*, *representations*, and *latent space*. What do these concepts have in common, and how do they differ?
- **Q2**. What is self-supervised learning, when is it useful, and what are the main categories of approaches?
- **Q3**. What is few-shot learning? And how does it differ from the conventional training procedure for supervised learning?
- **Q4**. What is the *lottery ticket hypothesis*, and if it holds true, how can it be useful in practice?
- **Q5**. Suppose we train a neural network classifier in a supervised fashion and notice that it suffers from overfitting. What are some of the common ways to reduce overfitting in neural networks through the use of altered or additional data?
- **Q6**. Suppose we train a neural network classifier in a supervised fashion and we already employ various dataset-related techniques to mitigate overfitting. How can we change the model or make modifications to the training loop to further reduce the effect of overfitting?
- **Q7**. What are the different multi-GPU training paradigms, and what are their respective advantages and disadvantages?
- **Q8**. What are the main factors that have contributed to the success of transformers?

- **Q9**. What are the popular categories of deep generative models in deep learning (also called *generative AI*), and what are their respective downsides?
- **Q10**. What are the common sources of randomness when training deep neural networks that can cause non-reproducible behavior during training and inference?

Chapter 2. Computer Vision

- **Q11**. How do we compute the number of parameters in a convolutional neural network? Suppose we are working with a convolutional network with 2 convolutional layers with kernel size 5. The first convolutional layer has 3 input channels and 5 output channels. The second convolutional layer has 5 input and 12 output channels. The stride of these convolutional layers is 1.
  Furthermore, the network has 2 pooling layers with kernel size 3 and stride 2. Lastly, the network has 2 fully connected hidden layers with 192 and 128 hidden units each, where the output layer is a classification layer for 10 classes. The architecture of this network is illustrated in the figure below. What is the number of trainable parameters in this convolutional network?
- **Q12. **Under which circumstances are fully connected and convolutional layers equivalent?
- **Q13**. Why do vision transformers (ViTs) generally require larger training sets than convolutional neural networks (CNNs)?

Chapter 3. Natural Language Processing

- **Q13**. What are the main factors that have contributed to the success of transformers?
- **Q15**. What is the distributional hypothesis in NLP? Where is it used, and how far does it hold true?

- **Q16**. What are common data augmentation techniques for text data?
- **Q17**. The scaled-dot product attention mechanism proposed by the original transformer architecture in the influential *Attention Is All You Need* paper is often referred to as self-attention. Why is it called "self"-attention, and how is it different from regular attention?
- **Q18**. What are the differences between encoder- and decoder-based language transformers?
- **Q19**. What are the different ways we can use and finetune pretrained large language models (LLMs)?
- **Q20**. What are the standard metrics for evaluating the quality of text generated by large language models?

Chapter 4. Production, Real-World, And Deployment Scenarios

- **Q21**. What is the difference between stateless and stateful training workflows in the context of production and deployment systems?
- **Q22**. What is data-centric AI, how does it compare to the conventional modeling paradigm, and how do we decide it's the right fit for a project?
- **Q23**. What are techniques to speed up model inference through optimization without changing the model architecture and without sacrificing accuracy?

Chapter 5. Predictive Performance and Model Evaluation

- **Q25**. Consider *Poisson regression* and *ordinal regression*; when do we use which over the other?
- **Q27**. What are the three properties of a distance function that make it a *proper* metric?

- **Q28**. K-fold cross-validation is a common choice for evaluating machine learning classifiers because it lets us use all training data to simulate how well a machine learning algorithm might perform on new data. What are the advantages and disadvantages of choosing a large k?
- **Q29**. You trained a model that performs much better on the test dataset than on the training dataset. Since a similar model configuration worked well on a similar dataset before, you suspect something might be unusual with the data. What are some approaches and mitigation issues for looking into training and test set discrepancies?
- **Q30**. Suppose we plotted a learning curve and found that the machine learning model overfits and could benefit from more training data. Name different approaches for dealing with limited labeled data in supervised machine learning settings.