Introductory Lecture Notes on Deep Learning

Stephen Gould[*]
stephen.gould@anu.edu.au

February 20, 2026

Abstract

These lecture notes are for a one-semester (12-week) introductory course on deep learning, covering both theory and practice. Students will learn the foundational mathematics behind deep learning and explore topics such as multi-layer perceptrons (MLPs), back-propagation and automatic differentiation, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers. These techniques play a crucial role in modern artificial intelligence (AI) systems, including image and video understanding, natural language processing, generative AI, robotics, medicine and scientific discovery. The course includes various practical assessments to enhance student's understanding and intuition of deep learning and its diverse applications. Students are expected to have strong programming skills and previous exposure to linear algebra, differential calculus, and probability theory. Assessment details do not form part of these notes.

Appetizer

Consider the two-dimensional shape depicted in Figure 1. Nobody would have trouble recognising the shape as a triangle despite the lines not being completely straight and them not meeting precisely at each corner. And when asked why the shape is a triangle, most people would answer that it's because the shape has three sides. Deep learning, however, takes a different perspective. Deep learning says that the figure is a triangle because it *looks* like a triangle. In the same way, a cat is a cat because it looks like a cat, and a dog is a dog because it looks like a dog. No further justification is needed.[1] This is because deep learning methods learn from data, i.e., from training examples, rather than from prescribed rules. In this course we will study deep learning and discover how this is at all possible.

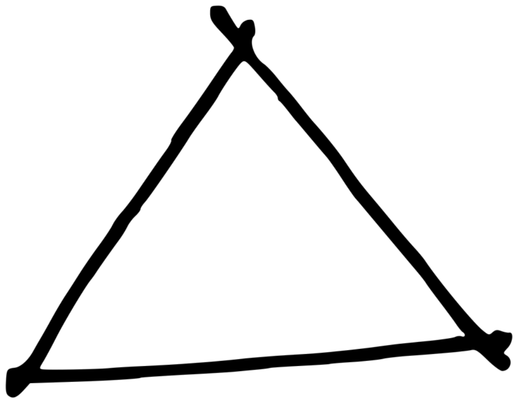


Figure 1: A triangular shape. But why? (Image credit: https://www.kisscc0.com/)

[*]We are grateful to the many students and colleagues who have provided feedback and corrections on earlier versions of these notes.

[1]This is not completely true—there is an entire research field devoted to the important topic of explainable AI, which requires that a human understandable justification be given for the outputs, answers, predictions, and decisions produced by an AI algorithm.

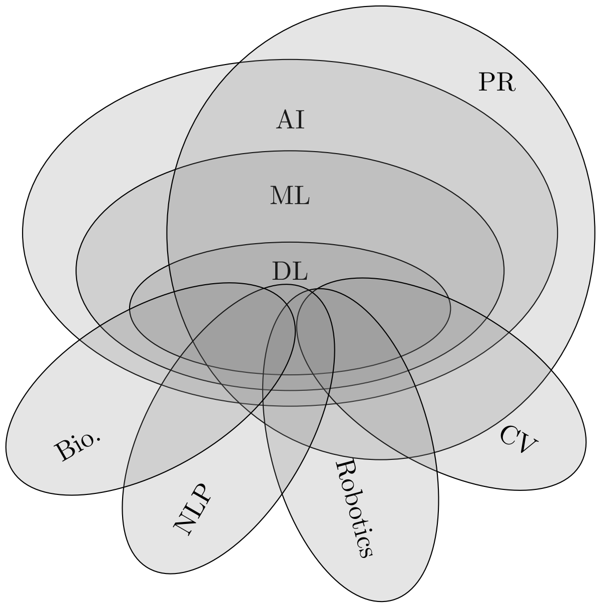


Figure 2: A Venn diagram showing where deep learning fits within the broader field of artificial intelligence. Not shown are the foundational disciplines of mathematics and computer science.

1 Introduction

This lecture begins by contrasting deep learning against classical machine learning and related fields. It then gives a very quick tour of problems in certain influential application areas. The lecture ends by discussing how data is represented and processed within deep learning systems, namely, via the tensor data structure and review of basic linear algebra operations. It is assumed the material in this lecture is mostly review and that most students would have seen it before.

Let us begin by positioning the field of deep learning within the broader scientific discipline of artificial intelligence (AI). Figure 2 provides a good illustration. We can loosely define AI as being about building machines that behave in an intelligent way. Machine learning (ML) is a sub-field of AI where the machines that we build improve their performance on some task by learning from data or with gained experience. Deep learning (DL) then is a sub-field of machine learning where the models that make up these machines—often called **neural networks**—are learned in an end-to-end fashion (i.e., with limited human engineering of features or model architectures).[2] A large part of AI involves pattern recognition (PR), which is an older term that relates mostly to perception and prediction versus action and planning, another large part of AI. Not shown in the diagram are the mathematical and computer science foundations upon which deep learning is based—primarily, linear algebra, probability, calculus, optimisation, programming, algorithms, and data science. This course assumes that you have a basic understanding of these foundation topics, some of which will be reviewed briefly at the end of the lecture.

This course studies deep learning with a focus on certain application areas such as computer vision and natural language processing that have heavily influenced the models and methods in deep learning. In turn, these application areas have gained enormously from the results that deep learning has delivered. Computer vision (CV) is about developing algorithms for machines to understand the world behind images and videos. Some people think of this as inverse graphics. Natural language processing (NLP) studies the structure of language and develops algorithms for extracting information from language—both written and spoken. It enables machines to engage with humans in a natural way. We will discuss various problems studied in both CV and NLP later in the lecture.

It is important to note that there are many areas in computer vision and natural language processing that are not related to deep learning, for example, how cameras capture images of the world or the semantic and syntactic structure of language. Computer vision and natural language processing are still very much their own independent research fields. Likewise, there are many other application areas that have contributed to and benefited from deep learning. The most notable of these are robotics, bioinformatics, scientific discovery, and medicine.

Carl Sagan, the famous astronomer and science educator, once said, “You have to know the past to understand the present.” We can pinpoint a precise moment in time when deep learning became a viable and popular research field. Although research into neural networks can be traced back to the 1960s [32, 39], it wasn't until 2012 where ideas from

[2]We will see a slightly different characterization shortly.

the past were put into a system that was truly competitive with other techniques for solving problems in computer vision—and shortly afterwards natural language processing—and the field that we call deep learning was born. Roughly speaking, research and methods in machine learning and its applications prior to 2012 were characterized by theory and provable algorithms. Systems involved a lot of feature engineering to get to work. Post 2012, deep learning methods have been characterized by data and compute, and building systems with end-to-end composeability. It is still not easy to get systems to work reliably all the time. But, as we will see, the knobs that we get to tune are different to the feature engineering of the past.

While deep learning is the dominant approach to solving problems in computer vision and natural language processing nowadays, it is important to mention that many classical ideas are still very relevant, either in their influence on deep learning techniques, in their adoption as components within larger deep learning systems, or as standalone algorithms. Moreover, developing theories and principles to better understand the behaviour and reliability of deep learning systems is becoming more and more urgent as deep learning algorithms are deployed in real-world engineering, economic, environmental, educational, societal, and health applications.

So what happened in 2012? A few years prior to 2012 a group of researchers from Stanford University led by Fei-Fei Li collected a massive dataset of images from the web (over one million). Each image in the dataset was annotated with a label that described its content employing the help of crowd-sourcing. There were images of dogs and cats of various breeds, cars, airplanes, foods, etc. One thousand different categories in total. The dataset was called ImageNet [9]. It was and remains one of the largest datasets of it's kind. To go along with the dataset, the Stanford group started the ImageNet large-scale visual recognition challenge (ILSVRC), where researchers from around the world could compete on who had the best image recognition algorithm, that is, whose algorithm would correctly recognise the most number of images in a hold-out test set. In 2012, Alex Krizhevsky, a graduate student of Geoffrey Hinton from the University of Toronto, entered the challenge with a deep learning based model [19]. It significantly beat all other models. This was a watershed moment for deep learning. The following year the top five performing entries were based on deep learning. The year after that, and every year since, all entries used deep learning and the error rate steadily dropped. Today deep learning outperforms humans on the ILSVRC challenge.

Since 2012 progress has continued to accelerate thanks various initiatives and proven effectiveness across a large number of application domains. First, the development and support of high-quality open-source software libraries (supported by industry) such as PyTorch [27], TensorFlow [1] and JAX [3], has made it easy to develop models and reproduce results from other researchers. Second, the availability of large and diverse datasets necessary to train models. Third, the existence of fast and cheap compute such as GPUs for accelerating the core calculations used in deep learning models.[3] Third, a growing community of researchers and engineers that facilitate the rapid sharing of knowledge, ideas, and results. And last, the amazing success of recent large language models (such as ChatGPT, Copilot, Gemini and Claude) and tools for generative AI have attracted significant private and public investment in AI commercialisation and supporting infrastructure.

1.1 Tour of Application Domains

In this section we provide a very quick tour of problems studied in computer vision, natural language processing and other application domains. These fields are huge so this tour are just the tip of the iceberg to give you a flavour of the types of tasks being solved, and the language used to describe them. Many tasks are related and there there are many variants that differ in underlying assumptions, categories to be recognized, availability and quality of training data, additional or different imaging modalities, etc. All have been advanced by deep learning in one way or another.

1.1.1 Computer Vision

Perhaps the most basic task in computer vision requiring machine learning is **image classification**.[4] Here the goal is to assign a single label to the entire image—the label can be binary (e.g., yes/no to a particular question) or multi-class where the system returns a label from a pre-defined set, typically of categories of objects (e.g., dog, cat, etc.). Algorithms often also provide a confidence score in addition to the label. These tasks are somewhat ill-defined and a critical inbuilt assumption is that the image neatly fits into one of the categories and the object (for object label sets) is dominant and well-framed within the image.

Going beyond a pre-defined set of categories, **open vocabulary image classification** allows images to be labeled with any noun, even those not seen during training of the model. A typical task of this type might be *what object is this?* This is a very challenging task and methods are usually augmented with an external knowledge base or pre-trained large vision-and-language model.

Structured prediction is a more refined machine learning approach outputs structured data rather than a single

[3]Though, even with GPU-acceleration training large deep learning models can take several weeks or more. The original AlexNet model took five to six days to train using two NVIDIA GTX 580 GPUs [19].

[4]There are many other tasks in computer vision, such as involving multi-view geometry, that do not require machine learning.

categorical label or regression value. In the context of computer vision the canonical example is **object detection** where the algorithm is expected to output a set of object labels and bounding box pairs corresponding to objects found in the image. Another example is the **(human) pose estimation** task, where we are asking to *locate all body joints* such as hands, elbows, shoulders, etc. Here a skeleton model defines the joints and how they are connected in the model, which constrains the location of one joint with respect to another.

There is a very large and specialized research area on **text recognition**, which aims to *locate and read text* within the image. This has applications from number plate recognition for automating parking lot ticketing, to interpreting receipts for financial systems, to reading text in-the-wild for place recognition or language translation (e.g., of menus at foreign restaurants).

Pixel labeling tasks also go beyond giving a single label to the entire image and instead assign one or more class labels to every pixel in the image. These labels can represent semantics, such as a sky pixel, or indicate membership such as all pixels having the same (integer) label belong to the same object. The most basic example of the latter type is **(unsupervised) image segmentation** or over-segmentation, which *breaks images up into meaningful regions* sometimes called **superpixels**. This is often a pre-processing building block for some more sophisticated downstream algorithm.

Panoptic segmentation, for example, labels each pixel with a semantic category (form a pre-defined set) and an instance identifier. This allows separation of multiple instances of the same object, e.g., multiple people in a crowd. Background categories are often considered to not have different instances. Sometimes researchers will use the terms **things** and **stuff** to distinguish between categories with distinct well-defined boundaries (and hence countable instances) versus those with amorphous shape and extent such as sky and grass.

Pixel labeling tasks are not restricted to semantic categories. **Monocular depth estimation** is a task that labels every pixel with its depth to the camera. If we have two cameras (or two views from the same camera) with known relative pose then the problem becomes **stereo depth estimation**, which can be solved by understanding the geometry of image formation and using traditional computer vision techniques [14].

A large area of computer vision involves searching and sorting operations on which many downstream tasks are built. In **feature matching** we are interested in finding corresponding pixels between two (or more) images, i.e., pixels that represent the same location in 3D space. This helps with applications such as camera calibration, object tracking, and 3D reconstruction. In **visual search** we are interested in finding information based on a query image such as the name of a famous landmark. In **attribute ranking** a set of images is sorted according to some visual attribute, which is useful for applications like online shopping.

There are several tasks in computer vision that require reasoning over temporally evolving scenes. In **tracking** we are asked to detect and track objects as they move through an environment where they may sometimes be occluded by other objects in the scene. The task of **activity recognition** is akin to image classification but for video data where we ask what activity is happening in a (short) video clip. Just like image classification this can be extended to activity detection and activity segmentation for longer video clips.

Several computer vision tasks are concerned with editing or synthesizing images as the output. There has been a long history of work focused on **image denoising**, which aims to remove noise and restore corrupted images, **image colorization**, which aims to convert a black and white image to colour, and **image inpainting**, which aims to remove an object from an image and fill in the background with a plausible reconstruction. In **video stabilization**, post-processing is used to remove jitter caused by a handheld camera and motion blur caused by fast moving objects.

More recently, deep learning has enabled tasks such as **novel view synthesis**, which depicts a scene from a viewpoint not previously seen by the model, i.e., different to where the image (or set of images) was originally taken. Neural radiance fields (NeRFs) are a popular class of deep learning model for novel view synthesis. Another task is **restyling**, which changes the style but not the viewpoint or content of a scene, e.g., *depict an adult as a child.* We can also create completely novel (fake) images and videos using **generative AI**, which involves both vision and language understanding.

1.1.2 Natural Language Processing

One of the core areas of natural language processing (NLP) is understanding the structure and meaning of language, including morphology, syntax and semantics. **Morphology** characterises the formation of words, e.g., “running”, “ran” and “runs” are all related to the dictionary word “run.” **Syntax** is about how words are combined into grammatically valid sentences. It involves identifying parts-of-speech such as nouns and verbs, and parsing sentences to understand the relationship between words. **Semantics** is concerned with the meaning of words, sentences and larger pieces of text and spoken language. For example, recognising names and associating pronouns with a particular person or people (**named entity recognition**) and distinguishing between the meanings of words that have the same spelling or sound (**word sense disambiguation**), e.g., “bank” as a financial institution or the side of a river. The ability to automatically parse text and perform **information extraction** comes from our ability to understand the syntax and semantics of language.

The task of **speech recognition** is a long standing AI problem that involves transcribing a speech signal into written language, and is now largely solved by deep learning methods. One of the key challenges is that different words can sound the same, as alluded to above, that different speakers pronounce words differently, and that background noise and context all play an important role in us understanding speech. The inverse problem of **speech synthesis**—generating a speech signal from text—is much easier although still a challenge for natural sounding speech.

Another well-studied problem is NLP is **machine translation**, e.g., translating from English to French, or from Chinese to German. Here the problem is that different languages have different rules and constructs so we cannot simply build a one-to-one mapping of words. Methods initially invented for solving machine translation are now at the heart of large language models (LLMs) that enable tasks such as **dialogue** (i.e., chatbots), **question answering**, **text generation** and **summarization**. Some claim that these models are capable of **commonsense reasoning**. Indeed, large language models exhibit remarkable fluency when applied to these tasks, even if their accuracy is sometimes lacking.

1.1.3 Multimodal Problems (Vision and Language)

One of the more impressive feats that deep learning has been able to achieve is unified reasoning over multiple data modalities such as vision and language. This manifests itself in many different tasks. In **image captioning** the goal is to describe an image in words. This is refined in **visual question answering** (VQA) where we ask specific free-form questions (i.e., not necessarily seen during training) of an image. For example, given an appropriate image where such questions make sense, *what color are her eyes? what is the mustache made of?*

Visual and language navigation (VLN) is a relatively new area of research where we ask a robot (or simulated agent) to follow a natural language navigation instruction using visual cues, e.g., *exit the dining room via the door next at the far end of the table, walk to the kitchen and stop in front of the fridge.*

Another popular vision and language task is to **locate a specific object in an image**, such as to *find the light blue truck.* Here the algorithm is expected to return a bounding box or segmentation mask of the object being localized. The task can be combined with VLN to navigate to a particular location and then identify a target object in the scene. Continuing from the example above we might ask the robot to *open the fridge and locate the bottle of milk.*

As mentioned above, in addition to basic image editing tasks, we can combine vision and language to create completely novel (fake) images and videos using **generative AI** from text prompts such as *an image of an astronaut riding a horse* and *a video of a woolly mammoth.*

1.1.4 Robotics and Sequential Decision Making

Game play has been a long-standing test of AI systems. Research in this area includes state representation, search and planning under uncertainty, learning of policies, opponent modeling, and long-horizon credit assignment. Games can be deterministic with perfect information (e.g., chess or Go) or stochastic and partially observable (e.g., poker). **Reinforcement learning** is a classic machine learning approach to develop policies for sequential decision making that occurs in complex games. Deep learning variants [23, 36] have demonstrated amazing results in this area.

Planning tasks generalise beyond simple games and focus choosing a sequence of actions to achieve some goal while respecting constraints and optimizing objectives. Markov decision processes (MDPs) and partially-observable MDPs are classic models that solve the problem of planning under uncertainty and usually involve reinforcement learning to solve. Planning underpins many high-level AI applications, from logistics and scheduling to decision-making in robotics and autonomous systems.

In robotic movement and manipulation, perception (e.g., object detection and pose estimation), planning, and control are combined to navigate and interact with objects in a physical environment. Unlike games or abstract planning, manipulation must contend with noisy sensors, imperfect actuation, deformable objects, and complex physical interactions. Autonomous driving [35] is a special type of robotic task with the goal of developing vehicles that can drive themselves either in controlled situations or in open-ended environments with many other moving vehicles and pedestrians. Research into autonomous driving has brought many safety and driver assist technologies into regular vehicles (such as lane tracking and collision detection).

1.1.5 Scientific Discovery and Medicine

Science and medicine has seen an increasing application of deep learning methods over the past several years. One significant area of research is in understanding biological processes and novel drug discovery. The 2024 Nobel Prize in Chemistry awarded to David Baker, Demis Hassabis and John Jumper, was in acknowledgment of their work in computational protein design and protein structure prediction, enabling the creation of entirely new proteins. Hassabis is the founder of DeepMind where work initially focusing on game play (e.g., AlphaGo) has been applied to numerous

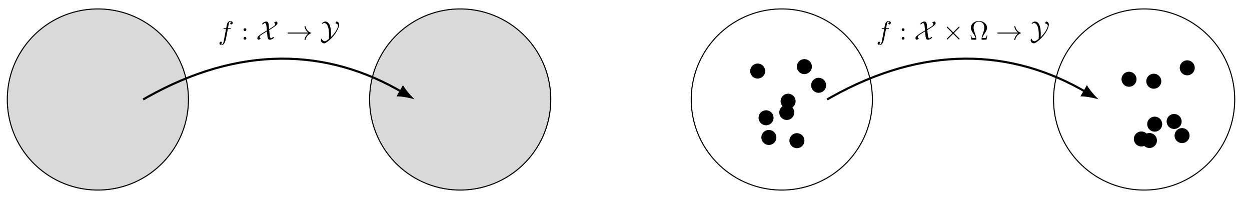

$f : \mathcal{X} \to \mathcal{Y}$
$f : \mathcal{X} \times \Omega \to \mathcal{Y}$

Figure 3: Machine learning from 10,000ft.

scientific endevours including AlphaFold for predicting the structure of proteins.

Other areas of deep learning in science includes weather forecasting, medical image analysis, clinical decision support, and surrogate models for complex physical processes such as fluid dynamics. In the not-too-distant future we may seen brain-computer interfaces driven by deep learning models that interpret brain activity and link your thoughts directly to controlling your phone and other devices.

1.2 Machine Learning from 10,000ft

At its core machine learning is about finding a function f that maps from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$ (see Figure 3). Since we can't practically search over all possible mappings we define a function class parametrized by θ and train a model on a dataset of samples from $\mathcal{X}$ and $\mathcal{Y}$, denoted $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$. For example, the function class may be restricted to only linear functions or only polynomials or neural networks of a given architecture. The goal is for the model perform well over the entire space—it is no good to just memorize the training samples. This is, of course, very difficult since much of the space is unseen (i.e., not covered by samples in $\mathcal{D}$). The mapping f is learned by minimizing (over the parameters θ of the model) a loss function L, that in some sense measures the inaccuracy of the model. Typically, the loss function decomposes over sampled input-output pairs,

$$\text{minimize}_\theta \quad L(\theta; \mathcal{D}) \triangleq \textstyle\sum_{(x,y)\in\mathcal{D}} \ell(f(x;\theta), y) \tag{1}$$

This is called **empirical risk minimization** [37].[5] Here the loss function L tells us **what** to do, and the parametrized function $f(\cdot;\theta)$ tells us **how** to do it.[6] The objective (composed of the loss L and the mapping function f) is, in general, nonconvex in the parameters θ. As such, we can only hope to find a local minima of the learning problem, which we do using **gradient descent** or one of its variants.[7]

It is common to represent the machine learning model diagrammatically as,[8]

$$\begin{tikzcd} & \theta \arrow[d] & \\ x \arrow[r] & \boxed{f} \arrow[r] & y \end{tikzcd}$$

or

$$\begin{tikzcd} x \arrow[r] & \boxed{f_\theta} \arrow[r] & y \end{tikzcd}$$

We can also incorporate the loss function in the diagram when we want to stress how the model is trained,

$$\begin{tikzcd} & \theta \arrow[d] & & \\ x^{(i)} \arrow[r] & \boxed{f} \arrow[r] & \hat{y} \arrow[r] & \boxed{\ell} \\ y^{(i)} \arrow[urrr] & & & \end{tikzcd}$$

To summarise, empirical risk minimization (ERM) [37] is a principled framework for choosing model parameters in machine learning. The theory of empirical risk minimization states that we should choose the parameters of the model

[5]The function L is often scaled by $1/N$. Technically it is then the expected loss called **risk** with the function ℓ being the **loss**. However, most researchers will use the terms interchangeably, or simply refer to any function being minimized in deep learning a loss function.

[6]Often you'll see notation f_θ, indicating that θ indexes f from some function class $\mathcal{F}$. However, in these notes we prefer $f(\cdot;\theta)$ to make it clear that the function takes parameters as an argument and avoid confusion when discussing function compositions in later lectures.

[7]In practical applications suboptimal solutions are often desirable since finding the globally optimal solution can result in poor generalization of the model to unseen test samples. Techniques such as regularization, data-augmentation, and model selection on a hold-out validation set all help in this regard, and will be discussed in later lectures.

[8]In the sequel we will sometimes use symbol y to denote the target output (i.e., groundtruth) and sometimes the estimated output of the network (i.e., as a shorthand for $f(x;\theta)$). At other times we will use $\hat{y}$ for the estimated output. The meaning should be clear from the context.

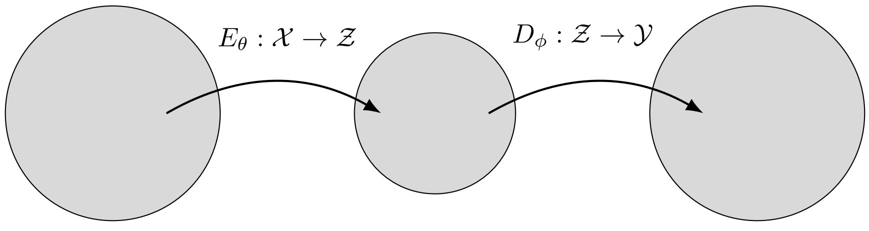

$E_\theta : \mathcal{X} \to \mathcal{Z}$
$D_\phi : \mathcal{Z} \to \mathcal{Y}$

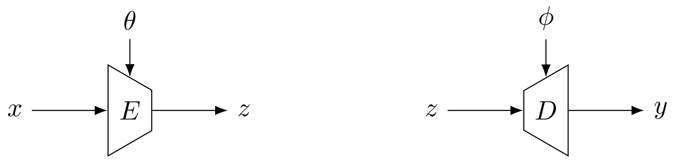
Figure 4: A popular paradigm in deep learning involves an encoder E_θ that maps the input into some latent space followed by a decoder D_ϕ that maps from the latent space to the output space. This has numerous applications from language translation to image generation.

θ that minimize L over all possible parameters (i.e., the "hypothesis" class). To prevent overfitting on the training dataset or to incorporate prior information into the model we often include a **regularization term** on θ,

$$L(\theta; \mathcal{D}) = \sum_{i=1}^{N} \ell(f(x^{(i)}; \theta), y^{(i)}) + \lambda R(\theta) \tag{2}$$

where λ is a hyperparameter that controls the regularization strength. This is known as regularized empirical risk minimization. The most common regularization function is the squared-norm of the parameters, $R(\theta) = \frac{1}{2}\|\theta\|_2^2$, which has a nice theoretical interpretation and also results in a simple modification to gradient descent based update rules.

A very common pattern in deep learning is the **encoder-decoder** paradigm. Here one model, the encoder, maps from the input space $\mathcal{X}$ to some latent space $\mathcal{Z}$. A second model then maps from the latent space to the output space $\mathcal{Y}$, as shown in Figure 4. The encoder and decoder models each have their own parameters and can be trained together or separately. Usually the latent space is smaller than the input or output spaces, and hence acts to compress the representation of the data. As a graphical illustration of this, we often draw encoder functions (E) and decoder functions (D) as follows,



Parameters are often omitted from these diagrams for brevity. Owing to the shape when the diagrams are combined, encoder-decoder models are sometimes called **hourglass** models.

Encoder-decoder models can be divided into **auto-encoders**, where the input and output spaces are the same and **cross-encoders**, where the input and output spaces are different. One use of an auto-encoder is to learn a low-dimensional (and sometimes sparse) representation of the data. We do not need explicit supervision for this task since the aim is to reconstruct the input x from its encoded version $z = E(z; \theta)$, i.e., we want $y = D(E(x; \theta); \phi)$ to be close to x. The latent variable z is called a **bottleneck** between x and y.

Cross-encoders have numerous applications. Indeed, many of the recent advances in deep learning and generative AI can be considered as a cross-encoder. With convolutional neural network (CNN) encoders and decoders we can implement semantic segmentation and style transfer algorithms; with recurrent neural network (RNN) encoders and decoders we can implement language translation models; and with a mixed CNN encoder and RNN decoder we can implement image captioning. Multiple different encoders can also be used to learn **joint embedding** spaces for images and language (e.g., CLIP [30] and BLIP [20]), allowing us to map from one semantic space to another.

1.2.1 Ingredients of a Machine Learning System

There are several ingredients that make up a machine learning system. They should be considered carefully whenever designing, debugging or deploying a system for a specific task. Perhaps the most obvious ingredient is the **data**, which includes the input signals (i.e., set of images) and the target labels for supervised tasks. The data is used for training as well as evaluating the system, and we need to be sure there is sufficient quantity and quality for both of these roles. How these are **represented**, that is what datastructures are used for storing and processing, plays an important part of the system design. For example, labels can be represented as text strings, integers (each indexing a list of categories), or one-hot vectors; videos can be stored in a compressed encoded format or as a sequence of extracted frames.

Next, the **model architectures** (also known as networks) and **training objectives** (i.e., losses and priors), which we touched on above, are central to the workings of the system. Equally important is the **learning algorithm** used

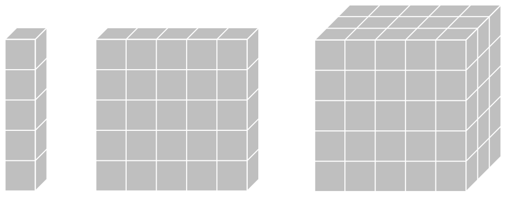


Figure 5: The basic datastructures in deep learning are vectors, matrices and tensors.

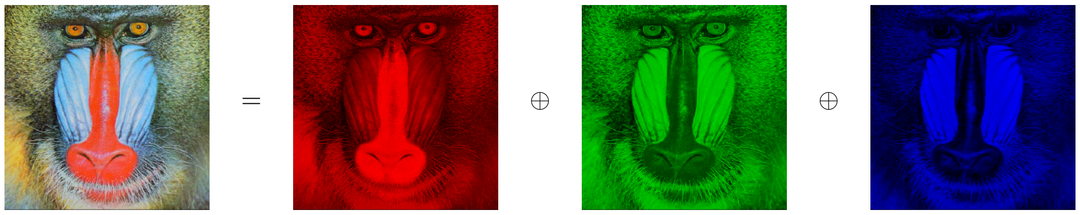


Figure 6: A colour image is represented using a third-order tensor composed of red, green and blue channels. (Mandrill image from USC Image Database, `https://sipi.usc.edu/database/`)

to tune the model's parameters using the training data and guided by the training objective. A crucial element of the learning algorithm for deep learning systems is how the parameters are initialised as this can have a big affect on the resulting performance and convergence rate of the algorithm.

Last, is the **evaluation metrics**. These map our judgment of how a system should perform into quantifiable scores. Using a single measure of performance can often be misleading. For example, reporting detection rates for a positive class (e.g., detecting cancer) and ignoring false positives can be misled by a system that always returns positive. It is important to think critically of the evaluation metrics, what they mean and where they may fail.

Each one of these ingredients—data, representations, architectures, objectives, and metrics—encode **(inductive) biases** that affect system behaviour and provide levers to change behaviour. We will study each of them in this course in the context of computer vision and deep learning.

1.3 Data Representation and Linear Algebra Basics

Deep learning builds heavily on linear algebra (and a bit of differential calculus) for much of its numerical processing. Linear algebra is an incredibly rich area of mathematics and we will only review a few key topics here. We recommend the following online resources for students interested in delving more deeply into the field:

- Gilbert Strang's MIT lectures, `https://www.youtube.com/playlist?list=PL49CF3715CB9EF31D`
- 3Blue1Brown animated math videos, `https://www.3blue1brown.com/topics/linear-algebra`

The first thing we need to consider is how data is represented and stored. For the purposes of this course (mostly), a (column) **vector** is a one-dimensional array of numbers, a **matrix** is a two-dimensional array of numbers, and a **tensor** is an arbitrary-dimensional array of numbers (see Figure 5). The number of dimensions is called the **order** of the tensor, so an first-order tensor is just a vector.

The term **dimension** can also be used to specify the number of entries a vector, e.g., a 3-dimensional vector, but the meaning should be clear from the context. In deep learning we tend to describe vectors, matrices and tensors by their **size** or **shape**, e.g., a matrix with two rows and three columns, that is, a 2-by-3 matrix, has shape `(2, 3)`. A matrix with the same number of rows as columns is said to be **square**.

An **image** is an array of red, green and blue (RGB) colour pixels. As such it can be represented by a third-order tensor with integer values in the range 0–255 or floating-point values in the range 0.0–1.0 (see Figure 6). The order in which data is stored is not consistent across software frameworks: In OpenCV and `numpy` images have shape $(H, W, 3)$, whereas in PyTorch they have shape $(3, H, W)$, where H and W and the height and width of the image, respectively.

We will often refer to a colour image as a 3-**channel** or 3-**plane** image. Instead of colour, which can be thought of as a 3-dimensional vector, we can generalize to C-dimensional feature vectors for each pixel assembled into a **feature map** of size (C, H, W), also called a C-channel feature map. A video then can be represented using a fourth-order tensor with shape $(3, H, W, T)$, where T denotes the number of frames in the video.

1.3.1 Transpose

Given an $m \times n$ matrix, A, we can define its transpose, A^T, as the $n \times m$ matrix where the rows and columns of A have been swapped. For example,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 3 & 6 \end{bmatrix} \qquad A^T = \begin{bmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 5 & 6 \end{bmatrix} \tag{3}$$

It should be clear from this definition that the transpose of a column vector is a row vector and vice versa, e.g.,

$$a = \begin{bmatrix} 8 \\ 5 \\ 3 \\ 6 \end{bmatrix} \qquad a^T = \begin{bmatrix} 8 & 5 & 3 & 6 \end{bmatrix} \tag{4}$$

A square matrix that is equal to its transpose is called **symmetric**. The following properties of transpose are true:

- $\left(A^T\right)^T = A$
- $(AB)^T = B^T A^T$
- $(A+B)^T = A^T + B^T$

The idea of transpose extends to tensors where we then also need to specify exactly which channels are being swapped.

1.3.2 Special Vectors and Matrices

Some special vectors up often and can be used to simplify notation, e.g., the all-zeros vector $\mathbf{0}_n$ and all-ones vector $\mathbf{1}_n$. The canonical vector e_i is a vector with all zeros except for the i-th location, which contains a one. This sometimes also called an indicator vector or one-hot vector and can be used to denote a category or class label in classification problems.

The same ideas extend to matrices, e.g., $\mathbf{0}_{m\times n}$, $\mathbf{1}_{m\times n}$ and E_{ij} for the zero matrix, all-ones matrix, and indicator matrix with a one for the (i,j)-th component and zeros elsewhere, respectively. Another special matrix that you should know about is the **diagonal matrix** which is a square matrix where the off-diagonal entries are all zeros. The diagonal entries may or may not be zero. When the diagonal entries are all ones the matrix is called the identity matrix and denoted by $I_{n\times n}$. The identity matrix serves the role of one in scalar arithmetic—multiplying by the identity matrix copies the multiplicand to the output.

Subscripts denoting the size of the (zero, ones and identity) vectors and matrices are often omitted when it is clear from the context.

1.3.3 Addition and Multiplication

Two matrices of the *same size* can be added together in a componentwise fashion so that $C = A + B$ means

$$C_{ij} = A_{ij} + B_{ij} \quad \forall i = 1, \ldots m \text{ and } j = 1, \ldots n \tag{5}$$

The product of a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $x \in \mathbb{R}^n$, written $y = Ax$, is an m-dimensional vector with elements

$$y_i = \sum_{j=1}^{n} A_{ij} x_j \tag{6}$$

for $i = 1, \ldots, m$. The cost of matrix-vector multiplication is $O(mn)$.

The product of two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ is an $m \times p$ matrix with elements

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{7}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

Note that the order of the matrices matters and for the product to exist we need the number of columns of A to equal the number of rows of B. However, matrix multiplication is associative, $(AB)C = A(BC)$, and distributive, $A(B + C) = AB + BC$. Matrix multiplication is not, in general, commutative, $AB \neq BA$.

The cost of matrix multiplication in $O(mnp)$.

Consider a matrix $A \in \mathbb{R}^{m\times n}$ and vector $x \in \mathbb{R}^n$. We can interpret the product $y = Ax \in \mathbb{R}^m$ in the following ways:

- the i-th entry of y is the inner product of the i-th row of A (denoted by $a_i^T \in \mathbb{R}^{1\times n}$ here) and x

$$y = \begin{bmatrix} — a_1^T — \\ — a_2^T — \\ \vdots \\ — a_m^T — \end{bmatrix} x = \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_m^T x \end{bmatrix}$$

- y is a linear combination of the columns of A (denoted by $a_i \in \mathbb{R}^m$ here for the i-th column)

$$y = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix} x = \sum_{i=1}^{n} x_i a_i$$

1.3.4 Inner Product and Euclidean Norm

The **inner product** between two vectors (of equal length) is defined by

$$\begin{aligned} \langle x, y\rangle &= x_1 y_1 + \cdots + x_n y_n && (8) \\ &= x^T y && (9) \end{aligned}$$

Some important properties of inner product are:

- $\langle \alpha x, y\rangle = \alpha \langle x, y\rangle$
- $\langle x + y, z\rangle = \langle x, z\rangle + \langle y, z\rangle$
- $\langle x, y\rangle = \langle y, x\rangle$
- $\langle x, x\rangle \geq 0$
- $\langle x, x\rangle = 0$ if and only if $x = 0$

The row vector x^T represents a linear function $\mathbb{R}^n \to \mathbb{R}$.

For $x \in \mathbb{R}^n$, we define the (Euclidean) **norm** as

$$\begin{aligned} \|x\|_2 &= \sqrt{x_1^2 + \cdots + x_n^2} && (10) \\ &= \sqrt{x^T x} && (11) \end{aligned}$$

The quantity $\|x\|_2$ measures the length of the vector (from the origin).

Some important properties of norm are:

- $\|\alpha x\| = |\alpha| \|x\|$
- $\|x + y\| \leq \|x\| + \|y\|$ (triangle inequality)
- $\|x\| \geq 0$
- $\|x\| = 0$ if and only if $x = 0$

Other important (vector) norms are $\|x\|_1 = |x_1| + \cdots + |x_n|$ and $\|x\|_\infty = \max\{x_1, \cdots, x_n\}$.

1.3.5 Batch Processing and Broadcasting

Deep learning frameworks process data in batches, passed as tensors. The first dimension of the tensor is the batch dimension. So, for example, a batch of n-dimensional vectors has shape (B, N), and a batch of colour images has shape $(B, 3, H, W)$.

Example. For the operation $y = Ax + b$ we might have

$$\begin{aligned} X &= \{x^{(1)}, \ldots, x^{(B)}\} && \text{(input)} && (12) \\ Y &= \{Ax^{(1)} + b, \ldots, Ax^{(B)} + b\} && \text{(output)} && (13) \end{aligned}$$

Many PyTorch functions are batch-aware and support broadcasting where data along singleton dimensions is automatically replicated to match arguments within an operation, e.g., for $(M \times N)$-matrix A, M-vector b, and batch of N-vectors x,

```
    y = torch.matmul(A.view(1, M, N), x.view(B, N, 1)).view(B, M) + b.view(1, M)
    # could also be done with: y = torch.einsum("ij,kj->ki", A, x) + b
```

computes $y^{(k)} = Ax^{(k)} + b$ on each element $k = 1, \ldots, B$ of the batch.

1.3.6 Tensors in PyTorch

In addition to their shape (`size`), tensors in PyTorch have several other properties:

- `dtype` to specify numerical precision, e.g., 32-bit float
- `device`, e.g., on CPU or GPU
- `requires_grad` will be discussed later
- `grad` holds gradient for training (will be discussed later)

There are many ways to create a tensor. For example,

```
    A = torch.tensor([[8, 5], [3, 6]])
```

creates the matrix

$$A = \begin{bmatrix} 8 & 5 \\ 3 & 6 \end{bmatrix} \tag{14}$$

A `Parameter` is a type of learnable `Tensor` used within PyTorch modules that you will encounter later in the course.

It will often prove useful to be able to **reshape** a tensor when implementing deep learning models. Reshaping is always possible so long as the total number of elements does not change, e.g.,

```
    A = torch.tensor([[8, 5, 1], [3, 6, 2]])
    B = A.reshape([3, 2])
```

creates matrices

$$A = \begin{bmatrix} 8 & 5 & 1 \\ 3 & 6 & 2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 8 & 5 \\ 1 & 3 \\ 6 & 2 \end{bmatrix} \tag{15}$$

Note that this is different to transpose, which would create

$$A^T = \begin{bmatrix} 8 & 3 \\ 5 & 6 \\ 1 & 2 \end{bmatrix} \tag{16}$$

from PyTorch code

```
    A.transpose(0, 1)
```

The method **view** can also be used whenever the memory layout of the underlying data does not need to change, e.g., `B = A.view([3, 5])`.

A very common reshaping is to **flatten** the tensor (sometimes called vectorizing), e.g., `C = A.flatten()` produces

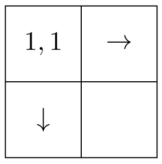
$$C = \begin{bmatrix} 8 & 5 & 1 & 3 & 6 & 2 \end{bmatrix} \tag{17}$$

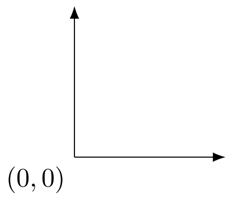
Note that elements are rearranged rowwise (called row-major ordering).

1.3.7 Coordinate Systems

A final comment on indexing and coordinate systems. In mathematics we generally index from 1, whereas in code we generally index from 0. This is a source of endless confusion and bugs. Now you know you wont fall into this trap.

To add further confusion, in linear algebra we index matrices and tensors from top-left to bottom-right, i.e., the top-left element of an $m \times n$ matrix is the $(1, 1)$-th entry and the bottom-right element is the (m, n)-th entry. Since we represent images as tensors, this will also be the default indexing for pixels. So the top-left pixel is the first pixel in the image. However, in geometry, coordinate systems increase upwards from the origin. In this coordinate system it is the bottom-left pixel that is the first pixel in the image. Be aware that different tools and different authors may use different coordinate systems. If you view an image and everything is upside down then flip the coordinate system.





1.4 Further Reading and Resources

The following is a list of recent textbooks that augments that material covered in these lecture notes:

- Goodfellow et al., *Deep Learning*, 2016 (`http://www.deeplearningbook.org/`)
- Zhang et al., *Dive into Deep Learning*, 2023 (`http://https://d2l.ai/`)
- Prince, *Understanding Deep Learning*, 2023 (`https://udlbook.github.io/udlbook/`)
- Scardapane, *Alice's Adventures in a Differentiable Wonderland*, 2024 (`https://www.sscardapane.it/alice-book`)

You should also familiarize yourself with the PyTorch deep learning library (`https://pytorch.org/`), which has excellent documentation and tutorials to work through.

Finally, there are some very high quality lectures and courses online from various institutions, including:

- Stanford (Fei-Fei, Karpathy, et al.), `https://cs231n.stanford.edu/`
- Michigan (Johnson), `https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/`
- York University (Derpanis), `https://www.eecs.yorku.ca/~kosta/Courses/EECS6322/`
- University of Amsterdam (Lippe), `https://uvadlc-notebooks.readthedocs.io/en/latest/`

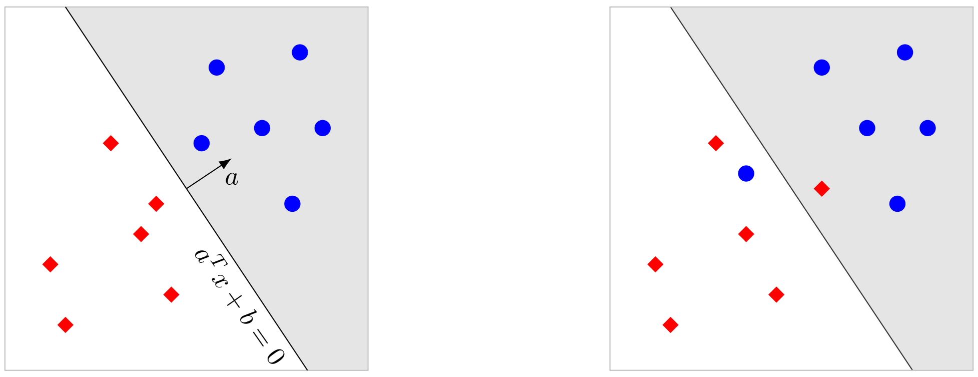


Figure 7: Example of linearly separable (left) and non-separable (right) data for binary classification in 2D.

2 Linear Classification and Multilayer Perceptrons

In this lecture we introduce the notion of logistic regression in the context of binary classification. We discuss the limitations of logistic regression for non-linearly separated data. This motivates the multi-layer perceptron (MLP), which can be shown (under mild assumptions) to be a universal function approximator. Building on the basic framework of MLPs, we discuss multi-class classification and gradient descent for fitting model parameters.

Binary classification involves data instances that can be divided into two distinct categories. For example, images of faces and images not of faces. Typically we are given a set of training examples $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$ composed of n-dimensional input features $x^{(i)} \in \mathbb{R}^n$ representing each data instance, and target labels $y^{(i)} \in \{0,1\}$. Instances with label `1` are called **positive** (e.g., faces), while those with label `0` are called **negative** (e.g., not faces).

In this setting we can think of two different tasks. First, the **regression task** is to learn a density function $P(y \mid x)$ from the training examples, which estimates the probability of `0` or `1` on a new input x. Second is the **classifier task**, which learns a classifier function $f : \mathbb{R}^n \to \mathbb{R}$ from the training examples that predicts `1` on an input x if $f(x) \geq 0$ and `0` otherwise. Often the classifier function is a **linear**, or more correctly, affine, function of the features, $f(x) = a^T x + b$. These tasks are related as we will later see.

Figure 7 depicts two examples of binary classification in a two-dimensional features space. In the first example (left), a linear function can be found to correctly separate the positive examples from the negative examples. In the second example (right), no such linear function exists, and the data is said to be **inseparable**. Note that the affine function $a^T x + b$ measures (signed) distance from the hyperplane that divides the space.

2.1 Logistic Regression

Roughly speaking, logistic regression turns an affine function of the input into a probability distribution over outputs. Let $y \in \{0,1\}$ be a random variable with distribution defined by

$$P(y = 1 \mid x) = \frac{\exp(a^T x + b)}{1 + \exp(a^T x + b)} \tag{18}$$

where a and b are parameters and $x \in \mathbb{R}^n$ is the observed feature vector. The **maximum likelihood principle** states that we should choose parameters a and b that maximizes the probability of the observed data under the model,

$$P(\mathcal{D}) = \prod_{i=1}^{N} P\left(y^{(i)} \mid x^{(i)}\right) \tag{19}$$

In practice, we (equivalently) minimize the negative log-likelihood function instead,

$$\begin{aligned} \ell(a, b; \mathcal{D}) &= -\log\left(\prod_{i:y^{(i)}=1} \frac{\exp(a^T x^{(i)} + b)}{1 + \exp(a^T x^{(i)} + b)} \prod_{i:y^{(i)}=0} \frac{1}{1 + \exp(a^T x^{(i)} + b)}\right) & (20) \\ &= -\sum_{i:y^{(i)}=1} (a^T x^{(i)} + b) + \sum_{i=1}^{N} \log\Big(1 + \exp(a^T x^{(i)} + b)\Big) & (21) \end{aligned}$$

which is convex in variables $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

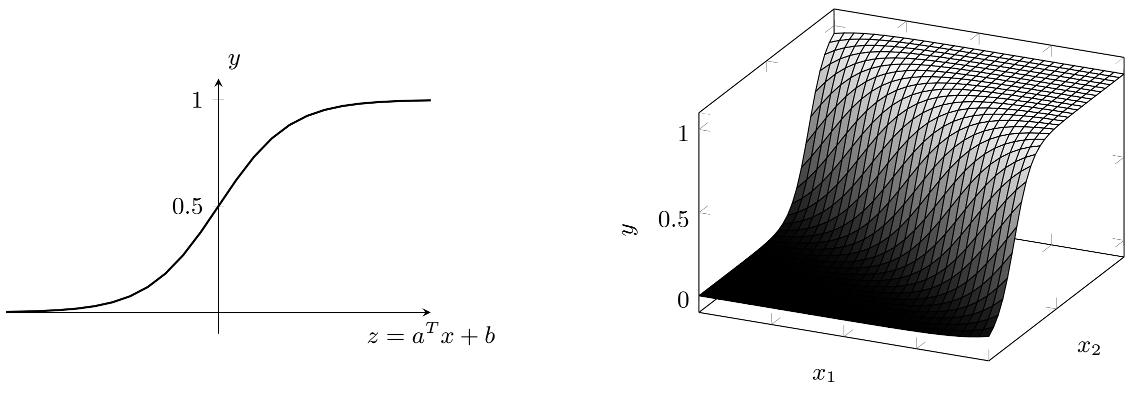


Figure 8: The logistic function, $y = (1 + \exp(-a^T x - b))^{-1}$, for 1D and 2D features.

The **logistic function** is an s-shaped curve (sigmoid) that maps from real numbers to the interval $[0, 1]$. It is therefore useful for modeling binary probability distributions. An extension of the logistic function is the so-called softmax function, which we will see later. The definition of the logistic function for scalar variable $z \in \mathbb{R}$ is,

$$\sigma(z) = \frac{\exp(z)}{1 + \exp(z)} \tag{22}$$

$$= \frac{1}{1 + \exp(-z)} \tag{23}$$

Replacing z with $a^T x + b$ allows us to learn a probability distribution conditioned on vector-valued features $x \in \mathbb{R}^n$,

$$P(y = 1 \mid x; a, b) = \sigma(a^T x + b) \tag{24}$$

as we have done above. A property of the logistic function is that $\sigma(z) \geq 0.5$ if, and only if, $z \geq 0$. So a classification rule $a^T x + b \geq 0$ corresponds to $P(y \mid x) \geq 0.5$, i.e., predict positive on an input x if the probability is above 50%.

An illustration of the logistic function for scalar $z = a^T x + b$ and two-dimensional features (x_1, x_2) is shown in Figure 8. Observe that the function is bound between zero and one. Note also that for the two-dimensional case (and higher-dimensions) when we look along the a direction we get a one-dimensional logistic curve.

Revisiting our example of binary classification over two-dimensional data, we can now see what a logistic function fitted to the data might look like as illustrated in Figure 9. Notice that in the separable case the logsitic curve is much sharper than in the non-separable case. In fact, in the separable case the function can be made arbitrarily sharp as we continue to train the model. In the non-separable case, the value of the logistic function gives us an estimate of the probability of an example being labeled positive.

Having our classifier too confident around the decision boundary, even in the separable case, is undesirable. Remember we are fitting to a set of training data, and what we really want is for our model to generalise to unseen test data. One way to reduce the confidence of the classifier on positive and negative training examples is through **regularization**. Here we add a penalty on the magnitude of the model's parameters, and minimize the resulting combination of loss function and regularizer,

$$\text{minimize}_{a,b} \quad \ell(a, b; \mathcal{D}) + \frac{\lambda}{2}\|a\|^2 \tag{25}$$

Note that we have not put any regularization on the offset, or bias, parameter b.

This type of regularization is also called **weight decay** because of it resulting in pushing the parameter values towards zero as evident when considering the gradient of the regularized loss,

$$\nabla_a \left(\ell(a, b; \mathcal{D}) + \frac{\lambda}{2}\|a\|^2 \right) = \nabla_a \ell + \lambda a \tag{26}$$

Ignoring the $\nabla_a \ell$ term for the moment, then taking a step in the negative gradient direction will change parameter a to $(1 - \eta\lambda)a$, where η is the gradient step size (or learning rate), to be discussed later in the lecture. That is, regularization tends to push a to zero. From the definition of the logistic function in Equation 23 it should be clear that as parameter a approaches zero, the value of the logistic $\sigma(a^T x + b)$ approaches a constant, i.e., $\frac{1}{1+e^{-b}}$. Hence, regularization tends to flatten the fitted logistic curve as show in Figure 10.

There are several other techniques for preventing over fitting that we'll see throughout the course, e.g., data augmentation. One such technique is **temperature scaling** that can be applied post-hoc. Here we introduce a temperature

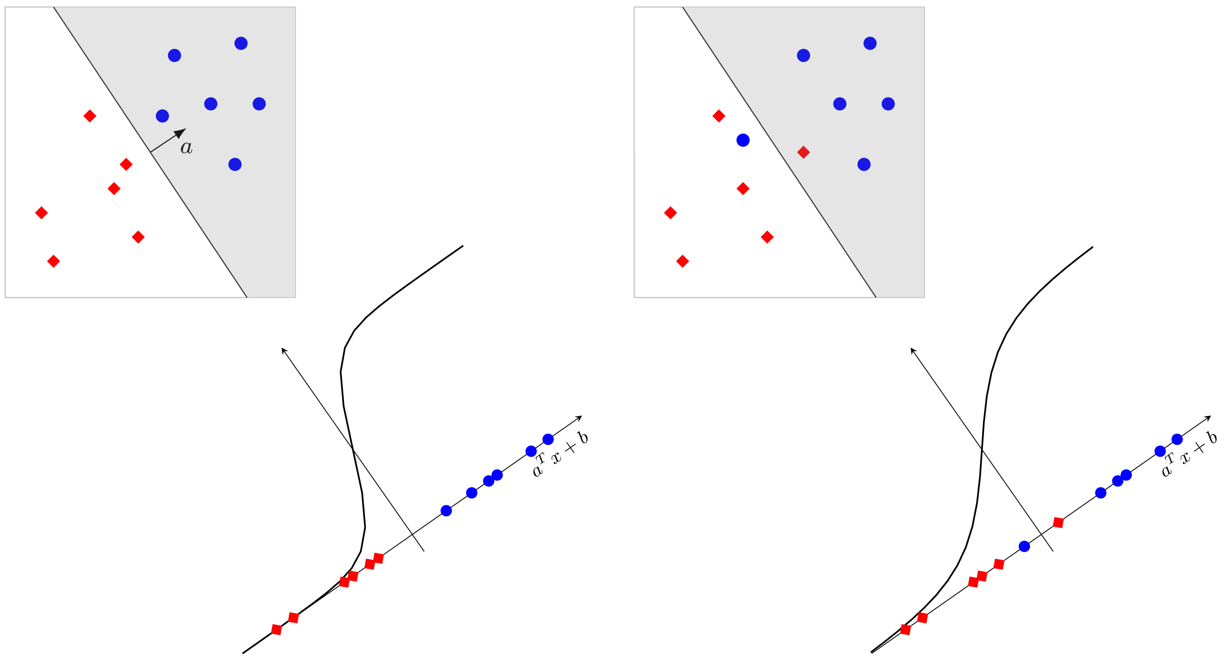


Figure 9: Illustration of logistic functions fitted to two-dimensional separable and non-separable data. The logistic curves are drawn aligned to the direction of a.

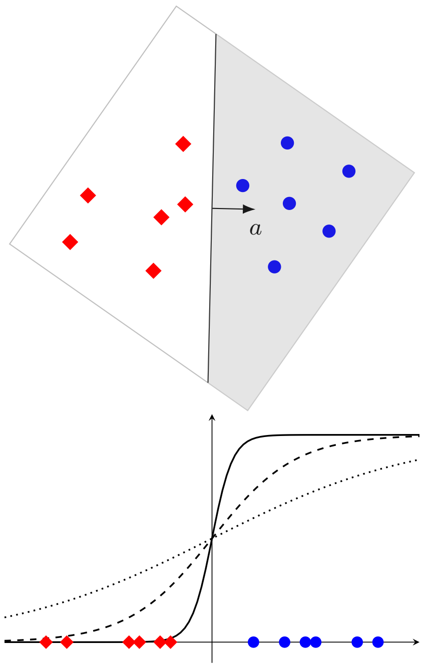


Figure 10: Regularization can be used to flatten a logistic function to reduce confidence around the decision boundary. We have rotated the feature space compared to Figure 9 for easier comparison of the different logistic curves.

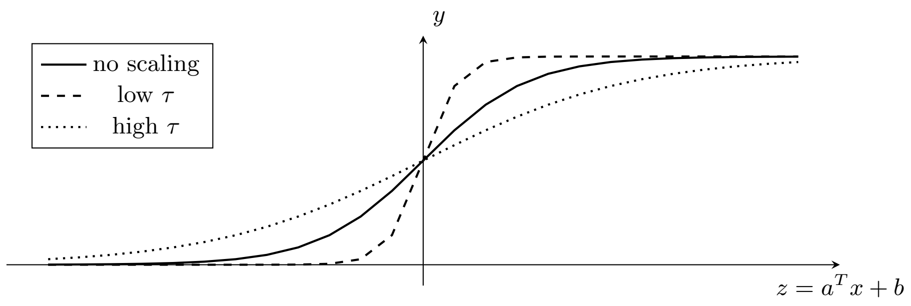


Figure 11: Temperature scaling. The parameter $\tau > 0$ controls the sharpness of the logistic curve.

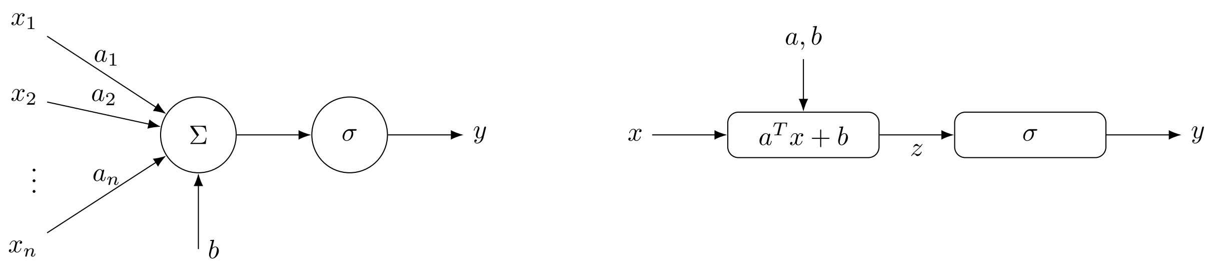


Figure 12: Classic view (left) and modern view (right) of a single neuron as either the explicit expression $y = \sigma\left(\sum_{i=1}^{n} a_i x_i + b\right)$ or more compact expression $y = \sigma(a^T x + b)$.

parameter $\tau > 0$ to the logistic function,

$$y = \sigma\left(\frac{z}{\tau}\right) = \frac{1}{1 + \exp\left(-z/\tau\right)} \tag{27}$$

The effect of τ is shown in Figure 11. A high value of τ has a similar affect to regularization, tending to flatten the curve. A value of τ less than one will sharpen the curve.

2.1.1 Logistic Regression as a Single Neuron

A very naive model of a biological neuron in the brain is a cell that fires its output if its accumulated input exceeds some threshold. We can very roughly view logistic regression as a single artificial neuron, as shown in Figure 12(left). The output y will be high if the weighted sum of the inputs $z = a^T x + b$ is greater than zero, and low otherwise. The arrows represent scalar signals. This is the classical depiction of a neuron in an artificial neural network, and is somewhat biologically inspired. The more modern view in deep learning is as a computation graph, illustrated more compactly in Figure 12(right). Here, the arrows can represent vector (or higher-order tensor) signals.

2.1.2 The XOR Problem and Multi-layer Perceptrons

As we saw earlier, a logistic regression classifier is only able to correctly classify linearly separable data. A famous example of a problem where the data cannot be separated by a single linear decision boundary is the XOR problem. See Figure 13(left). Here examples from the positive class (blue circles) are distributed in two clusters, one close to the point $(0, 1)$ and the other close to the point $(1, 0)$. Likewise, examples from the negative class (red diamonds) are distributed in two clusters, one close to the point $(1, 1)$ and the other close to the point $(0, 0)$. No straight line separates the two positive clusters from the two negative clusters.

The positive and negative examples can, however, be separated if we allow an additional layer of processing, as illustrated in Figure 13(right). In the first layer we construct a (linear) decision boundary $a_1^T x + b_1$ to separate the negative cluster close to $(1, 1)$ from the remaining three clusters. This can be thought of as implementing a NAND logic gate, $\xi_1 = \sigma(a_1^T x + b_1)$, where values close to 0 map to `False` and values close to 1 map to `True`. Still in the first layer, we construct another (linear) decision boundary $a_2^T x + b_2$ to separate the negative cluster close to $(0, 0)$ from the remaining three clusters. Similar to the first decision boundary, this can be thought of as implementing an OR logic gate, $\xi_2 = \sigma(a_2^T x + b_2)$. If we plotted ξ_1 versus ξ_2 we would find that the positive examples cluster around $(\xi_1, \xi_2) = (1, 1)$, whereas the negative examples cluster around either $(\xi_1, \xi_2) = (0, 1)$ or $(\xi_1, \xi_2) = (1, 0)$. As such, we can construct a second layer (linear) decision boundary $c^T z + d$, which acts as an AND logic gate on the first layer's output $z = (\xi_1, \xi_2)$ to solve the XOR problem.

In the preceding example the first layer of processing defines new features for the second layer. This motivates the **multi-layer perceptron** [39, 32]. More formally, a multi-layer perceptron is defined by composing multiple layers of

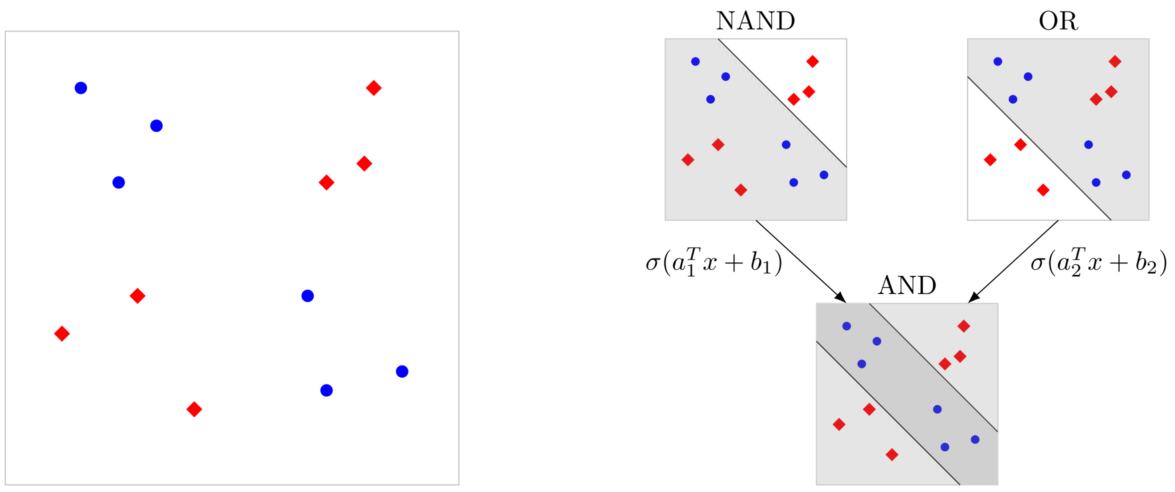


Figure 13: The XOR problem (left) can be solved with a two-layer network (right).

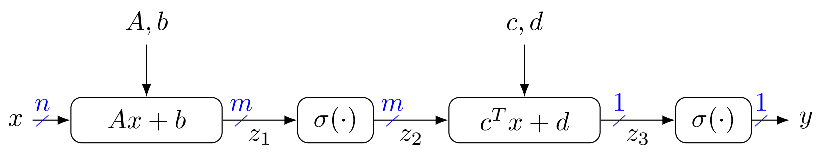


Figure 14: A two-layer percepton, implementing $y = \sigma(c^T \sigma(Ax + b) + d)$. Signal shape is indicated on the arrows.

logistic functions. For example, a two-layer perceptron is the composed function

$$y = \sigma(c^T \sigma(Ax + b) + d) \tag{28}$$

with parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, and $d \in \mathbb{R}$, and where the logistic function σ is applied elementwise on $z_1 = Ax + b \in \mathbb{R}^m$. Parameters of multi-layer perceptrons (and other linear layers in deep learning) are often called **weights**. Figure 14 shows a graphical depiction of the two-layer perceptron.

To see concretely how the two-dimensional XOR problem fits the multi-layer perceptron formulation, observe that we can stack the parameters from the two neurons from the first layer as

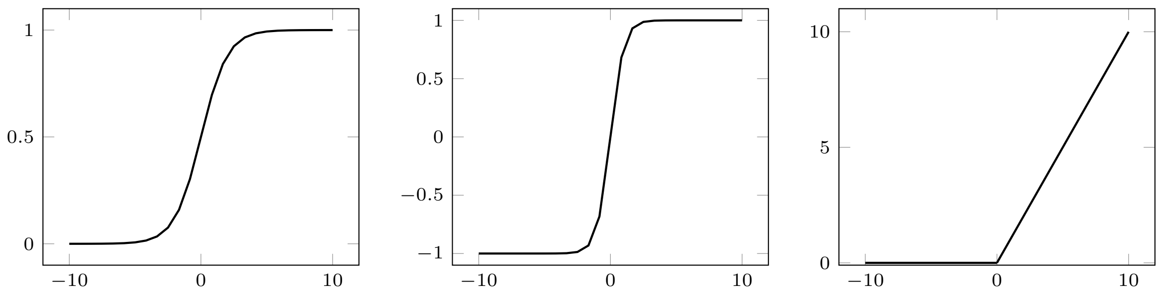
$$A = \begin{bmatrix} a_1^T \\ a_2^T \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{29}$$

giving

$$\begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} = \sigma\left(Ax + b\right) = \sigma\left(\begin{bmatrix} a_1^T x + b_1 \\ a_2^T x + b_2 \end{bmatrix} \right) = \begin{bmatrix} \sigma(a_1^T x + b_1) \\ \sigma(a_2^T x + b_2) \end{bmatrix}. \tag{30}$$

2.1.3 Activation Functions

Thus far we have only considered the logistic function as operating on linear combinations of the input features. The effect of the logistic function is to produce an elementwise non-linear transformation of the features. Transformations



(a) logistic (sigmoid), $1/(1 + e^{-z})$

(b) hyperbolic tangent, $\tanh(z) = (e^{2z} - 1)/(e^{2z} + 1)$

(c) rectified linear unit (ReLU), $\max\{0, z\}$

Figure 15: Common activation functions.

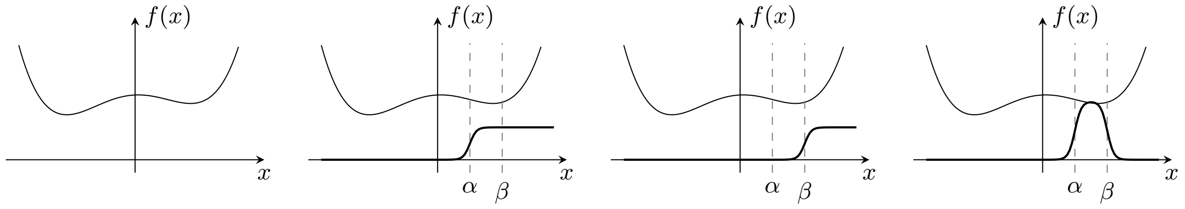


Figure 16: Visual argument for the universal approximation theorem. Shown from left to right: (a) A function that we wish to approximate. (b) An interval in the domain and sigmoid curve shifted to the start of the interval. (c) Another sigmoid shifted to the end of the interval. (d) Subtracting the second sigmoid from the first and scaling to the value of the function in the interval give an hump-shaped approximation to the function within the interval. This can be repeated by subdividing the entire domain into small intervals to approximate the whole function.

other than the logistic function are also possible. These go under the collective name of **activation functions**. Other common activation functions include the hyperbolic tangent function,

$$\begin{aligned}\sigma(z) &= \tanh(z) && (31)\\ &= \frac{\exp(2z)-1}{\exp(2z)+1} && (32)\end{aligned}$$

and the rectified linear unit (ReLU),

$$\sigma(z) = \max\{0, z\} \tag{33}$$

These activation functions are plotted in Figure 15. For obvious reasons quantities $z = Ax + b$ and $y = \sigma(z)$ are often termed pre-activations and post-activations, respectively, on input x. Many other activation functions have been proposed in the literature, e.g., leaky ReLU, gated linear units (GLU), GELU, SELU, radial basis, sinusoidal, etc.

2.2 The Universal Approximation Theorem

The celebrated universal approximation theorem [7, 16] states that for any continuous function $f : \mathbb{R}^n \to \mathbb{R}$ and well-behaved activation function $\sigma : \mathbb{R} \to \mathbb{R}$ (such as the logistic function), then there exists parameters $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, and $d \in \mathbb{R}$ such that the function $\hat{f}$ defined by

$$\hat{f}(x) = c^T\sigma(Ax + b) + d \tag{34}$$

approximates the function f everywhere. That is, $|\hat{f}(x) - f(x)| \leq \epsilon$ for all x.

A visual argument for why the universal approximation theorem holds true is shown in Figure 16. The argument goes as follows. Suppose we have some function f. For simplicity we will assume that the function is defined over the reals (i.e., $n = 1$). Then we can break the domain of the function into small mutually exclusive and exhaustive intervals. For each interval $(\alpha, \beta]$ we can define two logistic functions, the first shifted to the start of the interval and the second shifted to the end of the interval. For example,

$$\sigma(x - \alpha) \quad \text{and} \quad \sigma(x - \beta) \tag{35}$$

Subtracting the second logistic from the first produces a hump-shaped function centred on the interval. We can scale this hump to approximate f over the interval. Let γ be the value of the function, say, at the midpoint of the interval, i.e., $\gamma = f\left(\frac{\alpha+\beta}{2}\right)$. Then the expression,

$$\gamma\left(\sigma(x - \alpha) - \sigma(x - \beta)\right) \tag{36}$$

is a good approximation to the function f in the interval $(\alpha, \beta]$. We can make the edges of the hump arbitrarily sharp using temperature scaling,

$$\gamma\left(\sigma\left(\frac{x-\alpha}{\tau}\right) - \sigma\left(\frac{x-\beta}{\tau}\right)\right) \tag{37}$$

Repeating this process for all of the intervals give the approximation for f everywhere, completing the argument.

The universal approximation theorem is a nice theoretical result—we can approximate any reasonable function with a two-layer network (without the final activation function). But the theorem suffers from two practical shortcomings. First, it does not tell us how many parameters we need in the network, and experience suggests that for two-layer networks we need a very large number of parameters to approximate functions well. Second, it does not tell us how to find the parameter values in an efficient way. In turns out that we can reduce the number of parameters by going to deeper networks, i.e., more than two layers. Moreover, learning appears to be easier in deeper networks, although the theory here is not currently well understood. In practice, we often choose activation functions that violate the assumptions of the universal approximation theorem, such as ReLU, and this doesn't seem to affect the ability for deep networks to learn. Indeed, it often learns better.

A topic that is not taught much, but which is important for understanding learning and comparing trained networks is **identifiability**. This refers to whether we are able to uniquely identify a multi-layer perceptron's parameters from the function that it produces, i.e., its input-output relationship. The answer is that we cannot, i.e., the parameters of a neural network are never unique in that there exists a different set of parameters that produce exactly the same input-output mapping. We can see one specific example of how this may happen as follows. Let $P \in \mathbb{R}^{m\times m}$ be a permutation matrix and consider the two-layer perceptron from Figure 14, where we omit the final activation function for brevity. Then,

$$\begin{aligned}
y &= c^T\sigma(Ax+b)+d && (38)\\
&= c^T\sigma(P^{-1}PAx+P^{-1}Pb)+d && \text{(since } P^{-1}P=I\text{)} \quad (39)\\
&= c^TP^{-1}\sigma(PAx+Pb)+d && \text{(since } \sigma \text{ is applied elementwise)} \quad (40)\\
&= \tilde{c}^T\sigma(\tilde{A}x+\tilde{b})+d && (41)
\end{aligned}$$

Therefore, parameters $\{A, b, c, d\}$ and $\{\tilde{A}, \tilde{b}, \tilde{c}, d\}$ produce exactly the same outputs.

2.3 Multi-class and Multi-label Classification

We now extend our discussion from binary classification to multi-class and multi-label classification. In **multi-class** or K-way classification we are interested in assigning one label out of a predefined discrete set to each data instance, such image classification on the ImageNet dataset. For this task we are given a set of training examples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$ composed of features $x^{(i)} \in \mathbb{R}^n$ and target labels $y^{(i)} \in \{1,\dots,K\}$. Note that while the target labels are often expressed as integers for compactness, they map to semantic categories such as *dog*, *cat* or *duck*. Another popular encoding is one-hot encoding where the targets are represented by a K-length vector with a one for the element associated with the class and zeros elsewhere, e.g., $(1,0,0)$ for *dog*, $(0,1,0)$ for *cat*, etc. Be aware that authors are often sloppy and the encodings are used interchangeably so that $y^{(i)}$ is sometimes an integer and sometimes a one-hot encoding depending on the context.

We can extend the logistic function to the multi-class logistic,

$$P(y=k \mid x) = \frac{\exp(a_k^Tx+b_k)}{\sum_{j=1}^K \exp(a_j^Tx+b_j)} \qquad \text{for } k=1,\dots,K \tag{42}$$

parametrized by $\{(a_k,b_k) \mid k=1,\dots,K\}$. The function that takes a vector $z=(z_1,\dots,z_K)\in\mathbb{R}^K$ and returns $\left(\frac{\exp z_1}{Z},\dots,\frac{\exp z_K}{Z}\right)$ with $Z=\sum_{k=1}^K \exp z_k$ is called **softmax** and the z_k are called **logits**. The variable Z is called the **partition function** in machine learning and ensures that the output vector sums to one. In the context of the multi-class logistic we have $z_k = a_k^Tx+b_k$. Softmax has the property that $\arg\max_k\left\{\frac{\exp z_k}{Z}\right\} = \arg\max_k\{z_k\}$, i.e., the index of the component that maximizes softmax is the largest logit. This means that when we want to use the model to make predictions (rather than estimate probabilities) then we can simply take the most likely prediction as the label corresponding to the largest logit.

If instead of restricting a data instance to just one label, we allowed the instance to take a set of labels, then the task becomes a **multi-label** classification problem. This is the same as having a set of K binary classification problems, and is typical of tasks like attribute classification. Once again we are given a set of training examples, $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$, composed of features $x^{(i)} \in \mathbb{R}^n$, but now the target labels are binary vectors, $y^{(i)} \in \{0,1\}^K$ indicating true or false for each of the categories. We model the problem using a set of K sigmoid functions,

$$P(y_k=1 \mid x) = \frac{\exp(a_k^Tx+b_k)}{1+\exp(a_k^Tx+b_k)} \qquad \text{for } k=1,\dots,K \tag{43}$$

parametrized by $\{(a_k,b_k) \mid k=1,\dots,K\}$.

Both multi-class and multi-label problems can be solved using multi-layer perceptrons where the only difference is the final activation function, i.e., softmax or sigmoid. With parameters $A\in\mathbb{R}^{m\times n}$, $b\in\mathbb{R}^m$, $C\in\mathbb{R}^{K\times m}$, and $d\in\mathbb{R}^K$, we can write,

$$\begin{aligned}
y^{\text{multi-class}} &= \mathbf{softmax}(C\sigma(Ax+b)+d)\\
y^{\text{multi-label}} &= \mathbf{sigmoid}(C\sigma(Ax+b)+d)
\end{aligned}$$

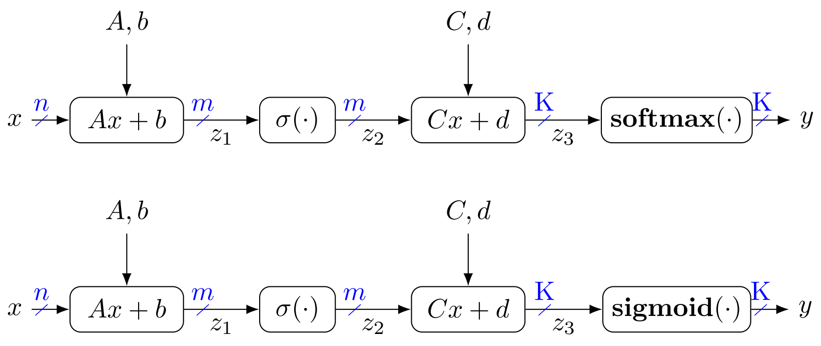


Figure 17: Multi-class (top) and multi-label (bottom) problems solved using a multi-class perceptron.

and depict graphically as shown in Figure 17.

There are several options for loss functions that train the models. Remember it is the loss function that tells the optimizer what to do and is a function of the model's parameters, collectively denoted by θ in the sequel. Unfortunately, the thing that we really care about, e.g., reducing the number of misclassified training examples, is not translatable into a loss that's easily optimized. Specifically, we seek a loss function that is differentiable. As such, a surrogate loss function is used.

The loss function that counts the number of misclassified examples is called the **0-1 loss**. It can be expressed mathematically as,

$$\ell^{\text{0-1}}(\theta) = \begin{cases} 0, & \text{if } f(x;\theta) \geq 0 \text{ and } y = 1 \\ 0, & \text{if } f(x;\theta) < 0 \text{ and } y = 0 \\ 1, & \text{otherwise} \end{cases} \tag{44}$$

but as mentioned above it is non-differentiable so difficult to optimize.

Standard loss functions that are differentiable are **mean square error** (MSE),

$$\ell^{\text{mse}}(\theta) = \frac{1}{2}\|f(x;\theta) - y\|^2 \tag{45}$$

for regression problems, and **negative log-likelihood** (NLL),

$$\ell^{\text{nll}}(\theta) = -\log P(y \mid x;\theta) \tag{46}$$

for classification problems. The latter is often called **cross entropy** loss when the probability is modeled by a multi-class logistic,

$$P(y \mid x;\theta) \propto \exp(f(x;\theta)) \tag{47}$$

or **binary cross entropy** loss when applied to K independent binary variables such as for multi-label problems,

$$-\sum_{k=1}^{K} y_k \log P_k(1 \mid x;\theta) + (1 - y_k)\log P_k(0 \mid x;\theta) \tag{48}$$

Note that all these loss functions decompose as a summation over training examples, $(x, y) \sim \mathcal{D}$.

2.4 Gradient Descent Optimization

We now turn our attention to the algorithm used to optimize the loss function—gradient descent. Let us begin by reviewing the definition of a gradient from multi-variate calculus. Let $f : \mathbb{R}^n \to \mathbb{R}$ be some function, fix some $x \in \mathbb{R}^n$, and consider the expression

$$\lim_{\alpha \to 0} \frac{f(x + \alpha e_i) - f(x)}{\alpha} \tag{49}$$

where $e_i = (0, \ldots, 1, 0, \ldots)$ is the i-th canonical vector.

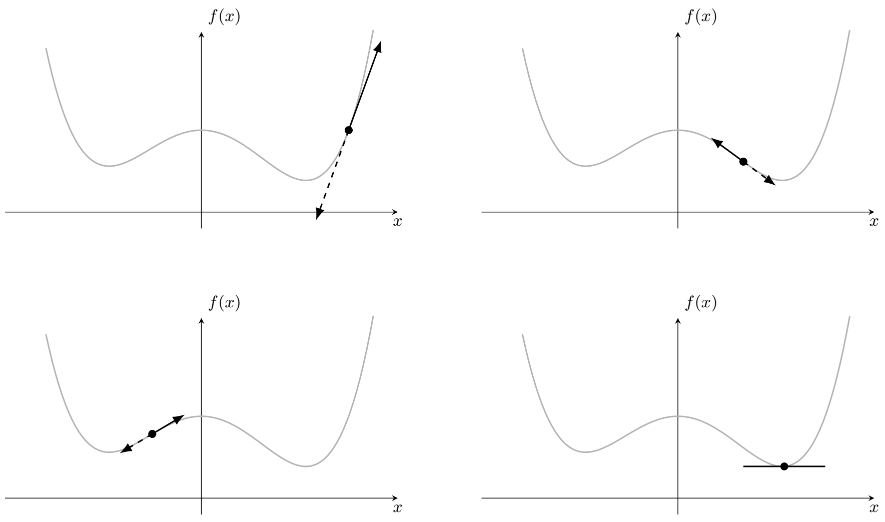


Figure 18: Gradient (solid arrow) shown at various points on the function. The negative gradient (dashed arrow) always points in a direction that reduces the function value, except at stationary points (i.e., local maxima, local minima and saddle points), where the gradient is zero. (Technically speaking, the gradient is the direction indicated in the figure projected onto the x-axis, and the arrow depicts an affine approximation to the function at the given point.)

If the limit exists, it is called the i-th **partial derivative** of f at x and denoted by $\frac{\partial f(x)}{\partial x_i}$. The **gradient** of $f(x)$ with respect to $x \in \mathbb{R}^n$ is the vector of partial derivatives,

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{50}$$

Figure 18 shows an example one-dimensional function. Several points on the function are highlighted, and the gradient at those points depicted by a solid arrow. The function decreases in the negative gradient direction. This is always true unless the gradient is zero (depicted by a horizontal line in the figure), which will occur at local maxima, local minima and saddle points. We can make use of this property to optimize the function.

The **gradient descent algorithm** is an iterative algorithm that takes steps in the negative gradient direction (of the loss function with respect to parameters) to seek a (local) minimum for the function. It works for smooth functions with any number of parameters. The steps can be expressed mathematically as,

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla \ell(\theta^{(t)}) \tag{51}$$

where η is the (iteration dependent) step size or step length or learning rate that controls how big a change is made to the parameters during each iteration. Taking too big a step (i.e., very large η) may result in overshooting the minimum, whereas taking too small a step may result in very slow progress. Ideally, η is chosen by searching along the negative gradient direction for a minimum value of the function, but this is expensive to do, amounting to solving a one-dimensional optimization problem. So instead of finding the best η, in deep learning we resort to choosing η via a fixed **learning rate schedule**.

Pseudo-code for the gradient descent algorithm is shown below:

```python
def gradient_descent(loss, theta0):
    """Generic gradient descent method."""

    theta = theta0
    for t in range(max_iters):
        dtheta = -1.0 * gradient(loss, theta)   # descent direction is negative gradient
        eta = line_search(loss, theta, dtheta)  # choose step size by line search
        theta = theta + eta * dtheta            # update

    if (converged()):                           # check stopping criteria
        break

    return theta
```

The algorithm terminates when some **stopping criterion** is obtained, usually of the form $\|\nabla \ell(\theta)\|_2 \leq \epsilon$, or when a maximum number of iterations is reached.

Gradient descent is very simple, we just need the gradient of the loss ℓ with respect to the parameters θ at each iteration. However, it can be very slow in its vanilla form. In later lectures we will discuss methods used in deep learning to speed up the algorithm.

The following code shows what gradient descent code looks like in PyTorch.

```
 1    dataloader = ...    # way of loading batches of input-label pairs for training
 2    model = ...         # definition of our prediction function
 3
 4    criterion = torch.nn.CrossEntropyLoss(reduction='mean')
 5    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
 6
 7    for epoch in range(max_epochs):
 8        for iteration, batch in enumerate(dataloader, 0):
 9            inputs, labels = batch
10
11            # zero gradient buffers
12            optimizer.zero_grad()
13
14            # compute model outputs for given inputs
15            outputs = model(inputs)
16
17            # compute and print the loss
18            loss = criterion(outputs, labels)
19            print(loss.item())
20
21            # compute gradient of the loss wrt model parameters
22            loss.backward()
23
24            # take a gradient step
25            optimizer.step()
```

The code starts with some routines for loading the data, defining the model, specifying the criterion or objective function, which is the cross entropy loss in this case, and choosing the optimization algorithm (Lines 1–5). The code the enters the optimization loop (Lines 7–25) where is processes data in batches. We will discuss the mechanics of epochs (once through the dataset) and iterations (one gradient update step) in the next lecture. For now it is sufficient to understand that each iteration the code zeros out the memory used to store gradients (Line 12), computes the models output from its input (Line 15), and calculates the loss by comparing the model's output to the desired or target output (Line 18). This is called the forward pass. It then computes all necessary gradients via a so-called backward pass (Line 22). These gradients are accumulated into buffers associated with the model parameters, which is why it is important to zero out these buffers at the start of the loop. Last, the code updates the model parameters by taking a gradient step (Line 25).

A simple example of gradient descent from Boyd and Vandenberghe [2], that also demonstrated why it can be slow, is illustrated in Figure 19. In this example we are optimizing the function,

$$\ell(\theta) = \frac{1}{2}\left(\theta_1^2 + \gamma\theta_2^2\right) \qquad (\gamma > 0) \tag{52}$$

over a two-dimensional variable $\theta \in \mathbb{R}^2$. The constant γ gives us a family of functions. We start at $\theta^{(0)} = (\gamma, 1)$ and use exact line search. This allows us to calculate the iterates determined by the gradient descent algorithm in closed-form:

$$\theta_1^{(t)} = \gamma\left(\frac{\gamma-1}{\gamma+1}\right)^t, \quad \theta_2^{(t)} = \left(-\frac{\gamma-1}{\gamma+1}\right)^t \tag{53}$$

Note that convergence will be very slow if $\gamma \gg 1$ or $\gamma \ll 1$. In the figure we use $\gamma = 10$.

One question remains: how do we calculate the gradients? To finish the lecture we give a worked example for the two-layer perceptron with logistic activation functions. In the next lecture we will discuss gradient calculation for deep learning in much more detail and introduce automatic methods for performing the calculations for us. Nevertheless, it is a valuable exercise to work through computing gradients by hand at least once.

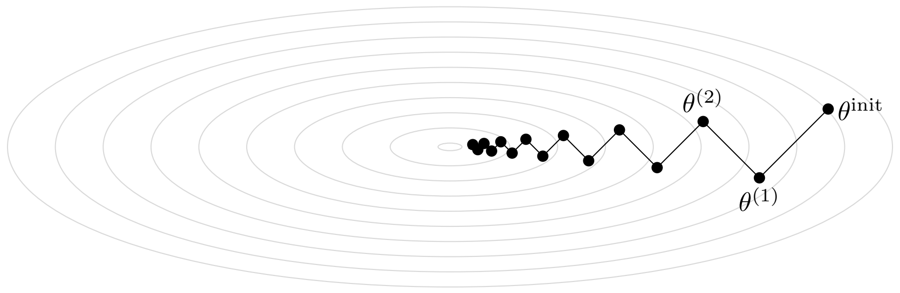


Figure 19: Example of gradient descent for two-dimensional quadratic problem, $\ell(\theta) = \frac{1}{2}(\theta_1^2 + \gamma\theta_2^2)$.

Consider the two-layer perceptron expressed as

$$z = \sigma(Ax + b) \qquad \text{(layer one)} \tag{54}$$

$$y = \sigma(c^T z + d) \qquad \text{(layer two)} \tag{55}$$

where $\sigma(\xi) = (1 + e^{-\xi})^{-1}$ is applied elementwise. We will assume an arbitrary differentiable loss function ℓ that we wish to minimize. As such, we can assume that $\frac{\mathrm{d}\ell}{\mathrm{d}y}$ is given. But we need to compute gradients of the loss for all of our parameters, i.e., $\frac{\mathrm{d}\ell}{\mathrm{d}A}$, $\frac{\mathrm{d}\ell}{\mathrm{d}b}$, $\frac{\mathrm{d}\ell}{\mathrm{d}c}$ and $\frac{\mathrm{d}\ell}{\mathrm{d}d}$, so that we can perform gradient descent.

First, observe that

$$\frac{\mathrm{d}\sigma}{\mathrm{d}\xi} = (-e^{-\xi})(-1)(1 + e^{-\xi})^{-2} \tag{56}$$

$$= \left(\frac{1}{1 + e^{-\xi}}\right)\left(\frac{e^{-\xi}}{1 + e^{-\xi}}\right) \tag{57}$$

$$= \sigma(\xi)(1 - \sigma(\xi)) \tag{58}$$

Starting at the second layer, let $\xi = c^T z + d$. We have, by the chain rule of differentiation,

$$\frac{\partial \ell}{\partial d} = \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\partial y}{\partial d} \tag{59}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\frac{\partial \xi}{\partial d} \tag{60}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y}y(1 - y) \tag{61}$$

since $\frac{\mathrm{d}y}{\mathrm{d}\xi} = y(1 - y)$ by our observation and $\frac{\mathrm{d}\xi}{\mathrm{d}d} = \frac{\mathrm{d}c^T z + d}{\mathrm{d}d} = 1$, and

$$\nabla_c \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y}\nabla_c y \tag{62}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\nabla_c \xi \tag{63}$$

$$= \frac{\mathrm{d}\ell}{\mathrm{d}y}y(1 - y)z \tag{64}$$

where in the last line we, again, used the fact that $y = \sigma(\xi)$ and $\nabla_c \xi = \nabla_c(c^T z + d) = z$.

Moving backward onto the first layer, we have

$$\nabla_z \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y}y(1 - y)c \tag{65}$$

following the same pattern as $\nabla_c \ell$. Then, for the first layer's parameters,

$$\nabla_b \ell = \nabla_z \ell \circ z \circ (1 - z) \tag{66}$$

$$\nabla_A \ell = \frac{\mathrm{d}\ell}{\mathrm{d}y}y(1 - y)\left(c \circ z \circ (1 - z)\right)x^T \tag{67}$$

where $\circ$ denotes elementwise product. These expressions use vector operations and can be a little intimidating when seen for the first time. If you find these difficult to interpret, try computing gradients on individual parameters using only scalar-valued derivatives, e.g.,

$$\frac{\mathrm{d}\ell}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\left(\sum_k \frac{\mathrm{d}\xi}{\mathrm{d}z_k}\frac{\mathrm{d}z_k}{\mathrm{d}A_{ij}}\right) = \frac{\mathrm{d}\ell}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}z_i}\frac{\mathrm{d}z_i}{\mathrm{d}A_{ij}} \tag{68}$$

where $z_k = [\sigma(Ax+b)]_k = \sigma(\sum_p A_{kp}x_p + b_k)$ and noting that $\frac{\mathrm{d}z_k}{\mathrm{d}A_{ij}} = 0$ if $k \neq i$.

Notice that we have reused calculations from evaluation of ℓ, i.e., y and z, in our expressions for the gradients. We could just have legitimately written the gradients in terms of the input x only. How we write these expressions leads to an important consideration of memory-vs-compute trade-off that will be discussed in later lectures. Stay tuned.

References

[1] M. Abadi and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge, 2004.

[3] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[4] N. Carion, A. Kirillov, F. Massa, G. Synnaeve, N. Usunier, and S. Zagoruyko. End-to-end object detection with transformers. In *ECCV*, 2020.

[5] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin. Emerging properties in self-supervised vision transformers. In *ICCV*, 2021.

[6] X. Chen and K. He. Exploring simple siamese representation learning. In *CVPR*, 2021.

[7] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems 2*, 1989.

[8] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *NeurIPS*, 2022.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *ACL*, 2019.

[11] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[13] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. Richemond, E. Buchatskaya, C. Doersch, B. Avila Pires, Z. Guo, M. Gheshlaghi Azar, et al. Bootstrap your own latent—a new approach to self-supervised learning. *NeurIPS*, 2020.

[14] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[15] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. In *NeurIPS*, 2020.

[16] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[17] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[18] D. P. Kingma and M. Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[20] J. Li, D. Li, C. Xiong, and S. Hoi. BLIP: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *CVPR*, 2022.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

[22] M. Milakov and N. Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[24] A. v. d. Oord, Y. Li, and O. Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[25] M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski. DINOv2: Learning robust visual features without supervision, 2023.

[26] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *JMLR*, 22(1), Jan 2021. ISSN 1532-4435.

[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*, 2017.

[28] J. Pennington, R. Socher, and C. Manning. GloVe: Global vectors for word representation. In *EMNLP*, 2014.

[29] S. J. Prince. *Understanding Deep Learning.* The MIT Press, 2023.

[30] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision. In *ICML*, 2021.

[31] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, 2022.

[32] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan Books, 1962.

[33] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *NeurIPS*, 2015.

[34] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[35] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. *Stanley: The Robot That Won the DARPA Grand Challenge*, pages 1–43. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-73429-1_1.

[36] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double Q-learning. In *AAAI*, 2015.

[37] V. N. Vapnik. *Statistical Learning Theory.* Wiley-Interscience, 1998.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[39] B. Widrow and M. E. Hoff. Adaptive switching circuits. *IRE WESCON Convention Record*, 1960.

[40] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning.* Cambridge University Press, 2023.