Deep Learning Lecture Notes

J. Adamczyk

`https://github.com/JacobHA/deep-learning`

March 25, 2024

Contents

Chapter 1

Introduction

1.0.1 Course Outline

These notes will serve as lecture material for a mini-course on the subject of deep learning. We first discuss some basic history before diving into a motivating example (the Perceptron) which forms the foundation of modern deep learning architectures. We are inspired by Richard Bellman (the father of dynamic programming) to consider the history and applications before the "meat" of the subject:

> "A person who claims the distinction of being well-educated should know the origins and applications of [their] field of specialization."
> — R. Bellman, p.1 of *Introduction to the Mathematical Theory of Control Processes*

Thus we shall first discuss the history of machine learning. We then introduce the groundwork for such algorithms. At this point, we will have enough material under our belts to begin analyzing some interesting applications of the material. Here (section 1.3) we shall see the remarkable algorithms for which deep learning is responsible.

Concluding the introduction to deep learning, we consider some more advanced architectures, relevant for memory-based systems (RNN/LSTM) and generalized pretrained transformers (GPTs), a popular architecture for current LLMs.

In the second section, we move on to deep reinforcement learning, a modern framework for solving decision-making processes in a data-oriented manner.

In section 3 we delve into the mathematical intricacies of deep learning, connecting to probability theory, quantum field theory, and differential geometry (maybe).

In Figure 1.1 you can find the relationship between Artificial Intelligence (AI), Machine Learning (ML), Deep Learning (DL), and Reinforcement Learning (RL). We will primarily focus on the latter two: DL and RL. For further reading on other subjects we recommend the likes of [13, 8, 2].

1.0.2 What is Deep Learning?

Because of the advances of machine learning, software development has taken a new form. To help explain the distinction between old and new software development, we turn to Andrej Karpathy:

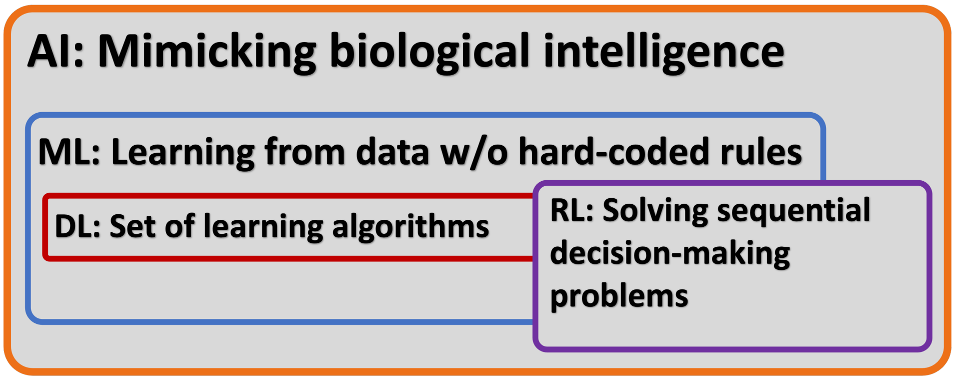

AI: Mimicking biological intelligence
ML: Learning from data w/o hard-coded rules
DL: Set of learning algorithms
RL: Solving sequential decision-making problems

Figure 1.1: Relationship between the fields of AI, ML, DL, and RL.

"To make the analogy explicit, in Software 1.0, human-engineered source code (e.g. some .cpp files) is compiled into a binary that does useful work. In Software 2.0 most often the source code comprises 1) the dataset that defines the desirable behavior and 2) the neural net architecture that gives the rough skeleton of the code, but with many details (the weights) to be filled in. The process of training the neural network compiles the dataset into the binary — the final neural network. In most practical applications today, the neural net architectures and the training systems are increasingly standardized into a commodity, so most of the active "software development" takes the form of curating, growing, massaging and cleaning labeled datasets."
— Andrej Karpathy `https://karpathy.medium.com/software-2-0-a64152b37c35`

Andrej Karpathy is a big name in the field (ImageNet, RNNs): he founded OpenAI (2015-17), joined Tesla as the head engineer of self-driving before returning to OpenAI in 2023.

Before diving into the content, we would like to set some expectations about what machine learning can and cannot do. In principle[1], deep learning can solve any problem (with sufficient data) which can be posed as a well-defined function (a function which maps pixel intensities to classes of cats or dogs, a function mapping the current word to the next word in a sentence, a function mapping the electronic properties of a material to its thermal properties, etc.). Deep learning (Programming 2.0) is able to learn functions from data, without requiring hand-specified rules. Trading off hard-coded rules for (immense amounts of) data can free the engineer from worrying about the possible complexities of the problem at hand. Stated another way, machine learning can mask the true underlying model from the researcher. To some, this may be the interesting part of the problem. If this is the case, DL may not satisfy you, but it may at least provide some insights along the way.

It should go without saying that *learning from data requires (good) data.*[2] This can come as a shock to those who think DL might act as a silver bullet against whatever scientific problem they may be working on. Additionally, learning *successfully* from data often requires a significant amount of time and engineering efforts in properly designing datasets, architectures, and algorithm parameters. To rebut this, we should mention that success can be found by transfer learning (re-using a high-performing model from one domain in another). We discuss the details of transfer learning in section **??**.

However, it is worth saying that with the progress witnessed in the past decade, it is not unreasonable to think we may be surprised by progress on tasks currently seeming impossible.

[1]barring many technical assumptions
[2]Garbage in = Garbage out.

1.1 History

TODO: add citations to all of these.

1.1.1 Timeline

In this section we include a quick timeline of the important advances of the state-of-the-art (SOTA) for the machine learning community. This list is not meant to be exhaustive. The reader may wish to refer back to this timeline after reviewing the details of some of the advances in the subsequent sections.

- 1642 - Blaise Pascal invents the first mechanical calculating machine
- 1837 - First design of a programmable machine (Charles Babbage & Ada Lovelace)
- 1943 - Warren McCulloch & Walter Pitts, theoretical foundation for NNs, draw parallel to BNN
- 1950 - Imitation Game / Turing Test
- 1955 - The term "AI" is coined during Dartmouth conference (John McCarthy)
- 1958 - Physical implementation of the perceptron (from 1943) image classification F. Rosenblatt
- 1969 - M. Minsky "Perceptrons" book showed impossible to learn XOR gate, or non-linear decision boundaries (let's discuss the theory in details later)
- 1969 - 1980 AI winter
- 1986 - Backpropagation (Rumelhard, Hinton, Williams) Nature paper
- 1997 - DeepBlue (SOTA expert system i.e. Programming 1.0) with smart pruning, 200M pos/sec!
- >2000: explosion (Due to the rapid growth of the field, I unfortunately cannot include everything or even a respectable comprehensive list, so just a few interesting favorites are included below)
- 2012 - GANs
- 2015 "Human-level control through deep reinforcement learning" (Atari) mastering without knowledge of rules!
- 2016 - PyTorch released
- 2021/2022 - CLIP / DALL-E
- 2022 - ChatGPT and LLMs
- 2023 -
- Present Day - You (yes you, the reader) make a state-of-the-art algorithm, blowing away the AI community

1.1.2 Introduction

Let us begin our journey of the development of machine intelligence, by first posing the question of what it means for a machine to **be** intelligent. To preface this, we must be sufficiently humble in admitting that we are unable to classify biological intelligence. This question was first (citation needed) considered by Alan Turing in 1950. To address the problem of deciding when a machine has reached a level of "human intelligence" (whatever that might mean), he posed the following gedanken experiment (the astute reader notes that a mere 70 years later, this thought experiment was a physically implementable experiment, with profound consequence).

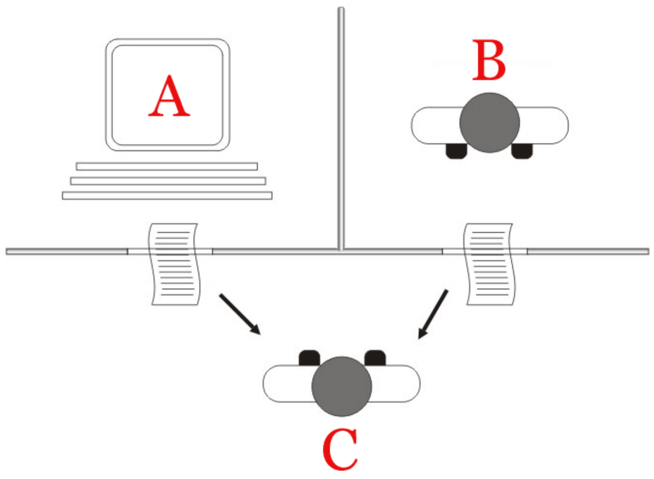


Figure 1.2: Stolen from wiki. Schematic setup of the Turing test. Interrogator is C.

The Turing Test

Put a human in one room and a machine in another isolated room. The only information that passes into or out of this room is too and from the "Interrogator" (a qualified human).

The role of the Interrogator is to submit the same question to both unmarked rooms (the Interrogator does not know who is in which room). The human and machine will both calculate and return a response to the question. If the Interrogator cannot distinguish whether the solution originated from a human (i.e. the machine can "successfully" answer any question), then we say the machine has achieved a level of human intelligence.

1.1.3 Functions

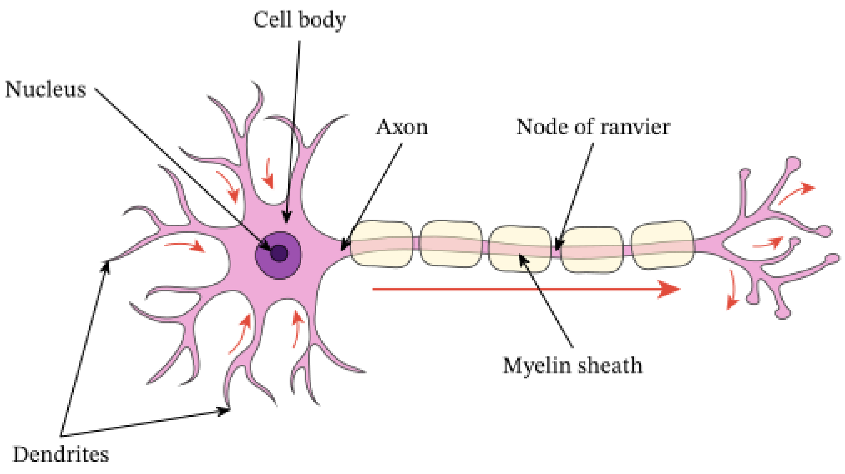
As discussed above, the main idea of machine learning is to have a program learn a particular[3] mathematical function. This being said, we must consider how to encode an arbitrary mathematical function. In the language of linear algebra, if we want a map from $\mathbb{R}^n \to \mathbb{R}^m$, then we could use a linear transformation $A : \mathbb{R}^n \to \mathbb{R}^m$, where A is an $n \times m$ matrix. Although linear transformations are of extreme importance, they are not very general (there are many more non-linear functions than there are linear ones). More importantly, the problems we care about are often not solvable by linear means. If they were, you don't need to apply all the machinery of deep learning! (Use linear methods such as SVM, linear regression, PCA, etc.)

So how *should* we think of general functions from $\mathbb{R}^n \to \mathbb{R}^m$? Well, the brain provides some inspiration for a new way of constructing functions. In the next section, we will elucidate this inspiration, starting with the simplest model (which turns out to be linear, but be patient...)

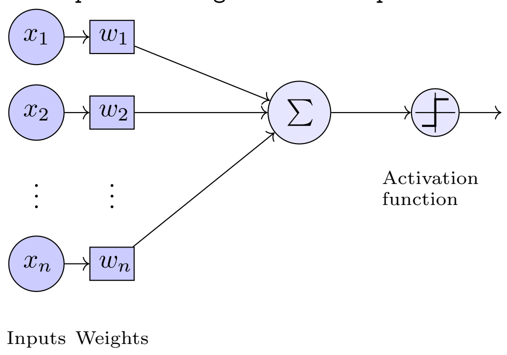
1.2 Simple Architectures

Motivated from the structure of brain cells, as seen in Fig. 1.3a, the simplest possible computational model that arises is the Perceptron. In the Perceptron, an input signal is sent into a "computation node". If the computation node is excited ("activated"), it will electrically propagate the signal as output. This single neuron can be considered as a**biological** neural network. This is where the terminology "artificial" neural network (ANN) comes from. Hereon, we will simply refer to our ANNs as NNs (neural nets).

[3]i.e. one specified by the labeled data



(a) Stolen from `https://www.nagwa.com/en/explainers/494102341945/`



(b) Stolen from `https://tex.stackexchange.com/questions/104334/tikz-diagram-of-a-perceptron`

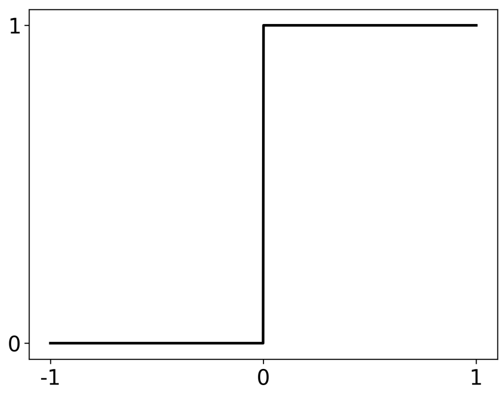


Figure 1.4: A simple activation function, the Heaviside Function.

We are dealing with a function, who takes a single data point[4] as its input, and will output a binary value, zero or one.

We will not concern ourselves yet with the specifics of training such a network, we will first get familiar with its inner workings and functional form. At the end of the day, the Perceptron (shown in Fig. 1.3b) will (hopefully) learn to distinguish two classes: ON/OFF, YES/NO, 0/1. This is seen by the possible outputs of the network (which should be thought of as a function): the Heaviside step function's range is simply $\{0, 1\}$ 1.4.

1.2.1 The Perceptron Model

The simplest neural network (NN) is the Perceptron, a simplified mathematical implementation of the aforementioned biological neuron. Mathematically, the Perceptron model has the following form:

$$f_{\{w_j, b\}}(x_i) = \Theta\left(\sum_j w_j x_j + b\right) \in \{0, 1\}. \tag{1.1}$$

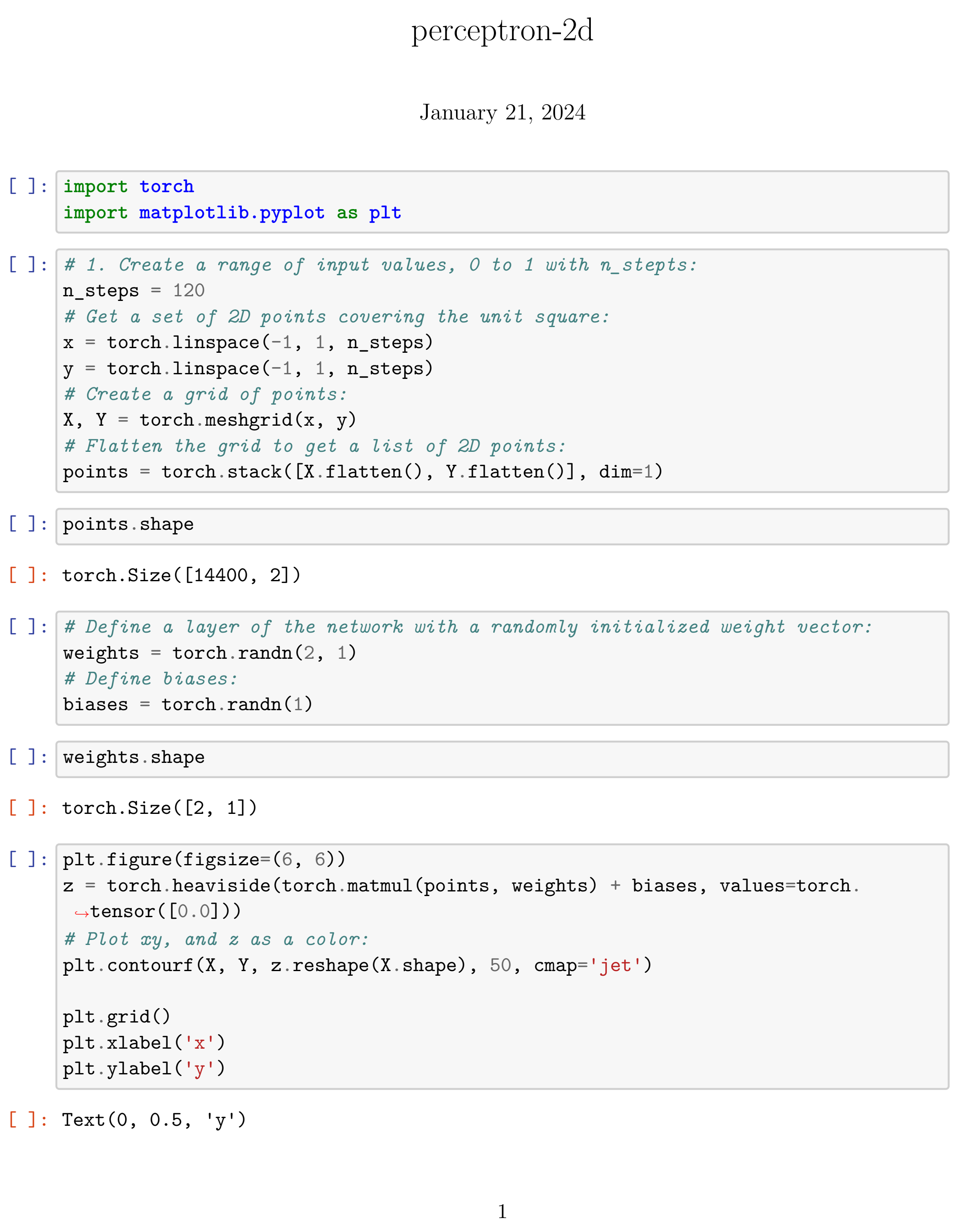
where Θ is the Heaviside step function – the simplest possible "activation" or "threshold" function, shown in Fig. 1.4. As the name suggests, this function dictates whether the neuron is activated, based on whether a potential threshold is surpassed. The threshold is given by the "bias" parameter, b above.[5] We will discuss the training of such a network in detail in later sections, but one should for now keep in mind: The Perceptron outputs a binary value and thus can "answer" yes/no questions (e.g. "should I buy this house given this data?" or "is this configuration of a system going to fail?" etc.). The predicted output ($f(x_i)$ above) will be compared to some ground truth output (a "labeled" data point provided by the user). The difference between the predicted value and the true value will be used to train the network, until (hopefully) the network's error is below a given tolerance. At this point, when all labelled data has been learned (but *not* memorized), one can feed a novel data point (whose ground truth value is unknown) to the Perceptron, and receive the model's best guess for the binary answer.

At the end of the day, one must keep in mind that neural networks are simply providing a best guess for a solution based on the data that it has seen (too anthropomorphic).

[4]The input "data point" is problem-specific, and may mean a set of pixel intensities, a stock price, or weather data.

[5]More accurately, the threshold is $-b$, since the argument of the Θ function in Eq. (1.1) becomes zero when the dot product $\sum_j w_j x_j = -b$. As soon as the dot product exceeds $-b$, the neuron is activated.

Linear decision boundaries are achievable, as can be seen by this randomly initialized Perceptron:



perceptron-2d

January 21, 2024

```
[ ]: import torch
     import matplotlib.pyplot as plt
```

```
[ ]: # 1. Create a range of input values, 0 to 1 with n_stepts:
     n_steps = 120
     # Get a set of 2D points covering the unit square:
     x = torch.linspace(-1, 1, n_steps)
     y = torch.linspace(-1, 1, n_steps)
     # Create a grid of points:
     X, Y = torch.meshgrid(x, y)
     # Flatten the grid to get a list of 2D points:
     points = torch.stack([X.flatten(), Y.flatten()], dim=1)
```

```
[ ]: points.shape
```

```
[ ]: torch.Size([14400, 2])
```

```
[ ]: # Define a layer of the network with a randomly initialized weight vector:
     weights = torch.randn(2, 1)
     # Define biases:
     biases = torch.randn(1)
```

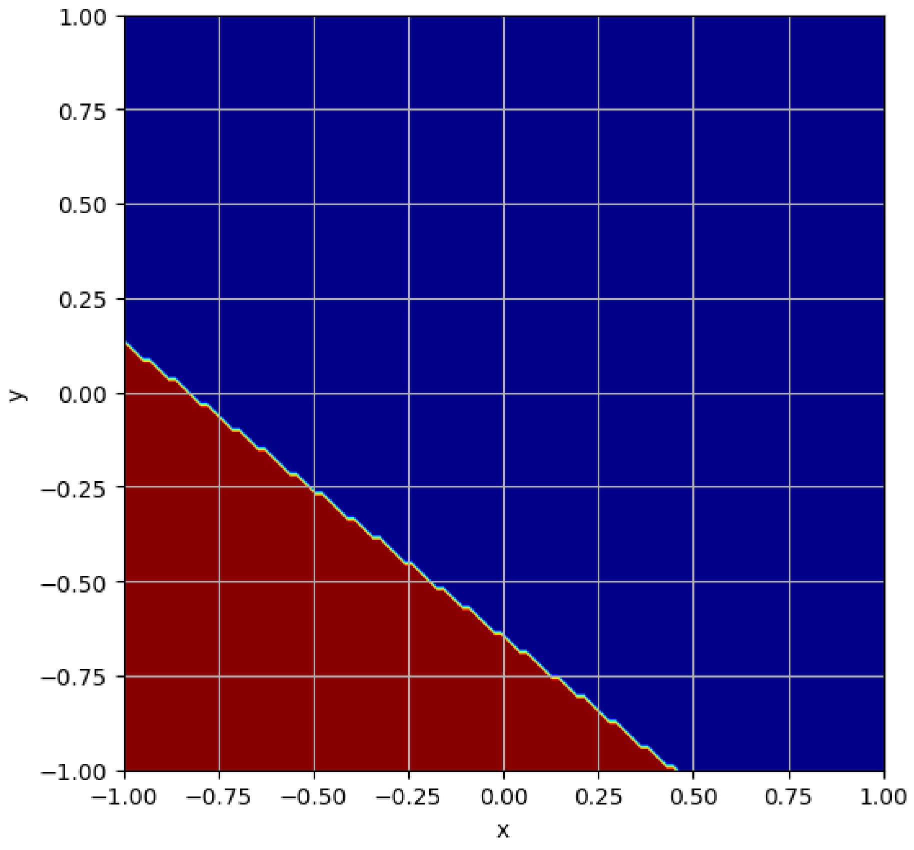
```
[ ]: weights.shape
```

```
[ ]: torch.Size([2, 1])
```

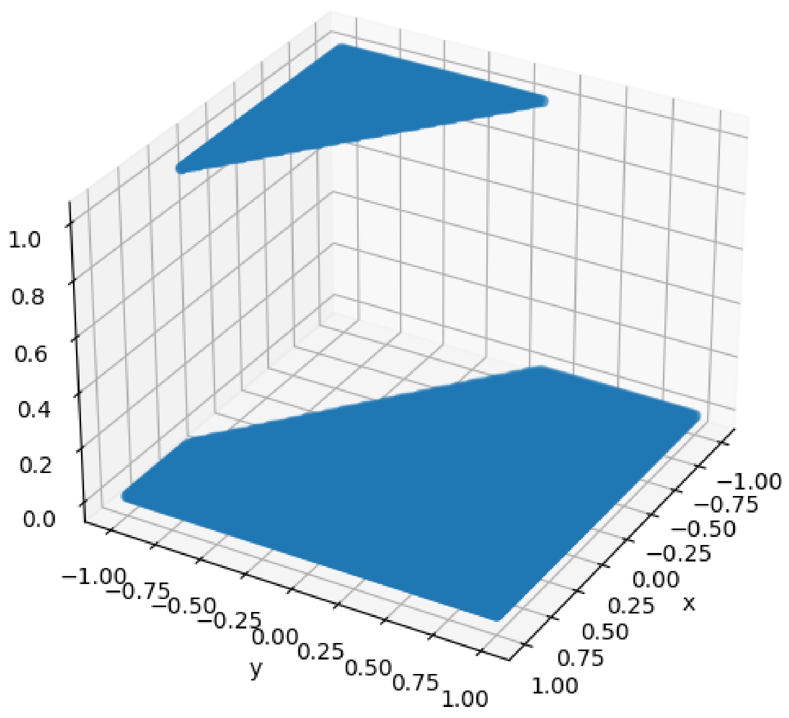
```
[ ]: plt.figure(figsize=(6, 6))
     z = torch.heaviside(torch.matmul(points, weights) + biases, values=torch.
      ↪tensor([0.0]))
     # Plot xy, and z as a color:
     plt.contourf(X, Y, z.reshape(X.shape), 50, cmap='jet')

     plt.grid()
     plt.xlabel('x')
     plt.ylabel('y')
```

```
[ ]: Text(0, 0.5, 'y')
```

1



[ ]:
```
# Plot x,y,z on a 3D plot:
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(points[:, 0], points[:, 1], z[:, 0])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
# Rotate the plot:
ax.view_init(30, 30)
# make interactive:
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

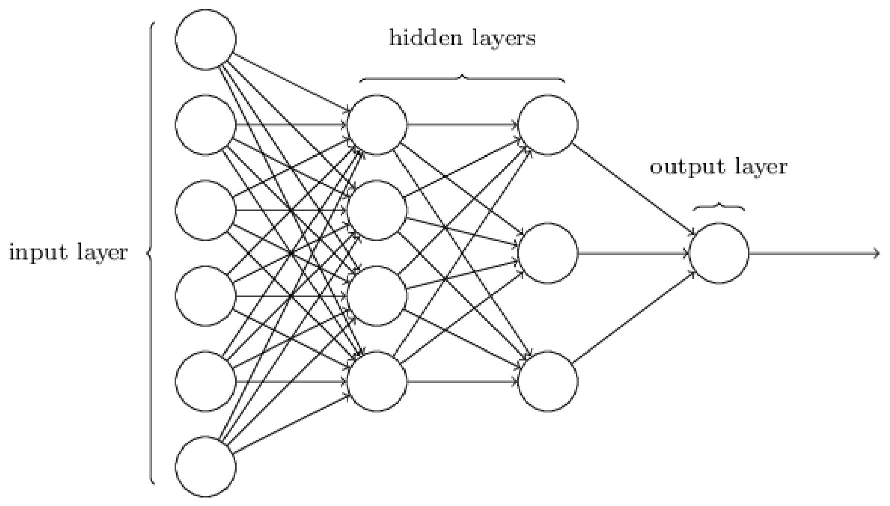


Figure 1.5: stolen from https://github.com/rcassani/mlp-example

However, the Perceptron is not capable of choosing which is the "best" hyperplane (later, SVMs solve this in the 90s - citation!) Additionally, and most importantly, it is limited to linear boundaries. Many problems of interest have complex nonlinear relationships, and thus the Perceptron is unsuitable. Another issue is that if there is no linear boundary, the learning algorithm will not terminate (there is no fixed point in gradient descent) even in the infinite data limit.

Many other models emerged (hopfield, boltzman, what else?) but one emerged as a winner for speed, expressiveness, and widespread utility: The Multi-Layer Perceptron

1.2.2 Multi-Layer Perceptron

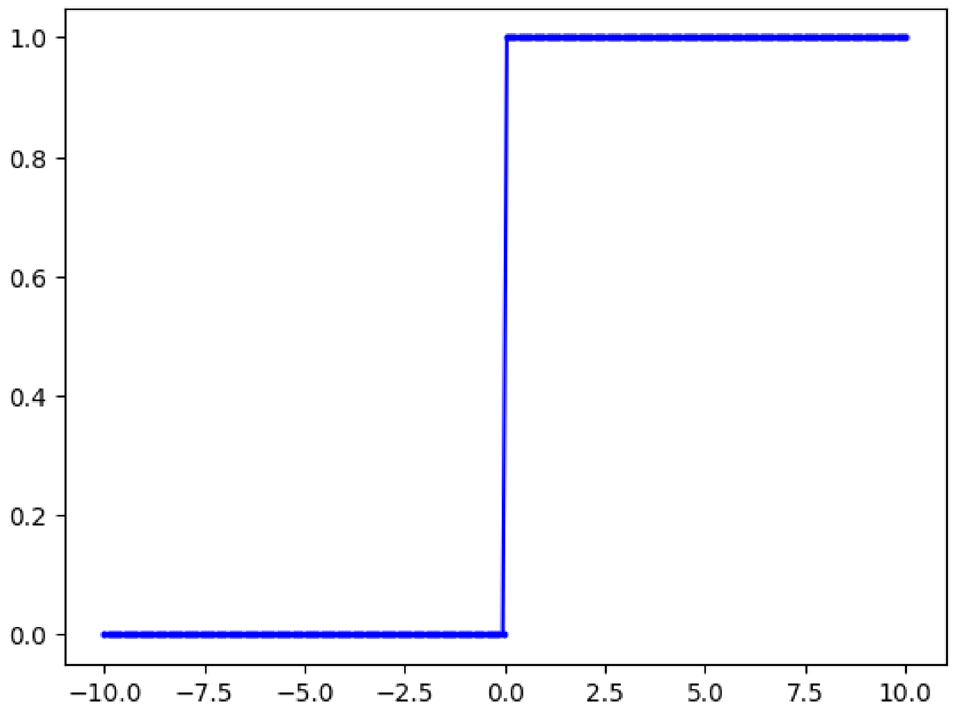
We want to build a more sophisticated version of the Perceptron which is hopefully more useful. Ideally, we'd like a network which can be trained fast, and is universal (we will make this statement precise later) – essentially meaning that we should be able to approximate **any** (sufficiently well-behaved) function. Although it took a significant time to emerge, one (now obvious) solution to making the network more complex would be to stack them together sequentially. This allows nonlinearities to propagate through multiple layers (Find image showing "folding" and curving). Beyond stacking layers, we can also introduce different modes of non-linearity. These are deemed activation functions, and several examples are given below. These days, we use smoother activation functions (compared to Heaviside step function used previously). The reasoning is twofold: (1) to allow "partial" information to be propagated, not so lossy, (2) to be differentiable (almost everywhere) allows for simple training.

Despite its relatively simple construction, the MLP is still a widely used (base) architecture for machine learning models. For example, many RL problems that we will see in the later chapter will use a simple MLP to model the "value function". Their architecture remains mostly unchanged too. The deepest models are typically only 10s of layers deep. Anything beyond this can lead to vanishing gradients or covariance explosion (we'll discuss this in detail later).

Interestingly, we should note that all neurons (network nodes) are identical in nature: a neuron takes a linear combination of the inputs, passes this weighted sum through an activation threshold function, and passes this value to the next layer. This is not necessary. In fact, more advanced architectures can combine many types of neurons in a single model. Though usually, the neuron type is consistent across a given layer. The neuron we have depicted is not the only “flavor” imaginable. One might also construct convolutional, recurrent, spiking, or noisy neurons. We’ll explore some of these flavors in the subsequent sections.

As a side note: We began this section with inspiration from biological models. These days, the majority of machine learning research has shifted away from biologically-plausible models, in lieu of powerful, efficient, and trainable models.

Now let’s see how these more complicated (importantly, non-linear) networks look through a simple piece of code. We’ll use two-dimensional inputs, with a step function at the output (for classification of two distinct classes), similar to the example shown above for the linear model. As you can see, the code is randomly initializing weights and biases, and this is not a very clean way of doing things. In the next section we’ll see how to “actually” set up the model with PyTorch. As usual, all of the code shown is available at `https://github.com/JacobHA/deep-learning`.



[ ]:
```
# Pass the input through the layer:
out_layer1 = torch.matmul(points, w_layer1) + b_layer1
# Apply a non-linear activation function:
out_layer1 = activation(out_layer1)
out_layer1.shape
```

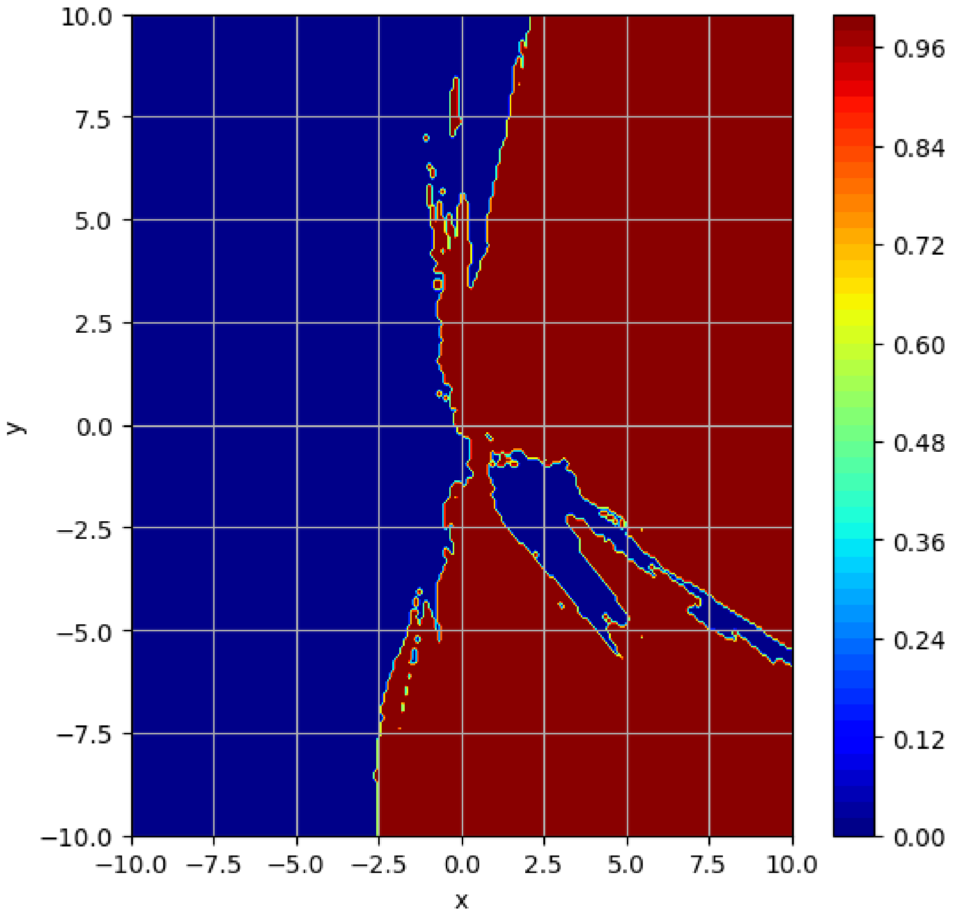
[ ]:
```
torch.Size([40000, 400])
```

[ ]:
```
# Define weights for second layer:
w_layer2 = torch.randn(hidden_dim, 1)
# Define biases:
b_layer2 = torch.randn(1)
```

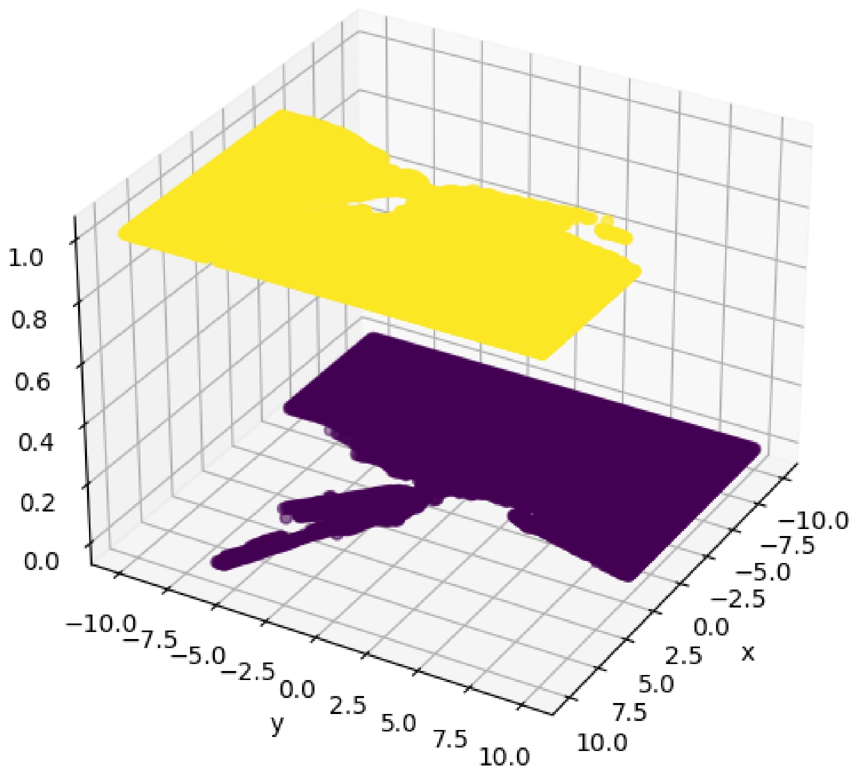
[ ]:
```
# Pass the output of the first layer through the second layer:
out_layer2 = torch.matmul(out_layer1, w_layer2) + b_layer2
# Apply a non-linear activation function:
out_layer2 = activation(out_layer2)
```

[ ]:
```
plt.figure(figsize=(6, 6))
z = out_layer2
# Plot xy, and z as a color:
# plt.scatter(points[:, 0], points[:, 1], c=z[:, 0])
# use interpolation:
plt.contourf(X, Y, z.reshape(X.shape), 50, cmap='jet')

plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar()
```

```
[ ]: <matplotlib.colorbar.Colorbar at 0x7f9e596c9a60>
```



```
[ ]: # Plot x,y,z on a 3D plot:
     from mpl_toolkits.mplot3d import Axes3D
     fig = plt.figure(figsize=(6, 6))
     ax = fig.add_subplot(111, projection='3d')
     ax.scatter(points[:, 0], points[:, 1], z[:, 0], c=z[:, 0])
     ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
# Rotate the plot:
ax.view_init(30, 30)
```



```
[ ]:
```

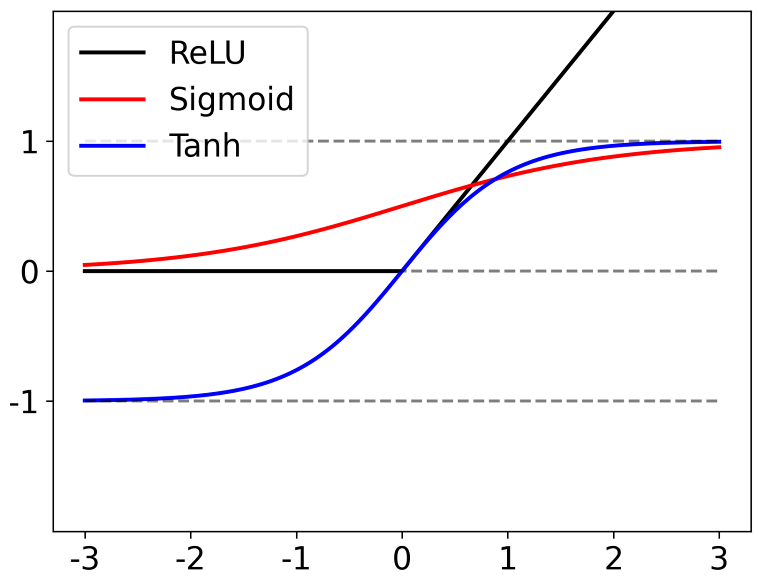
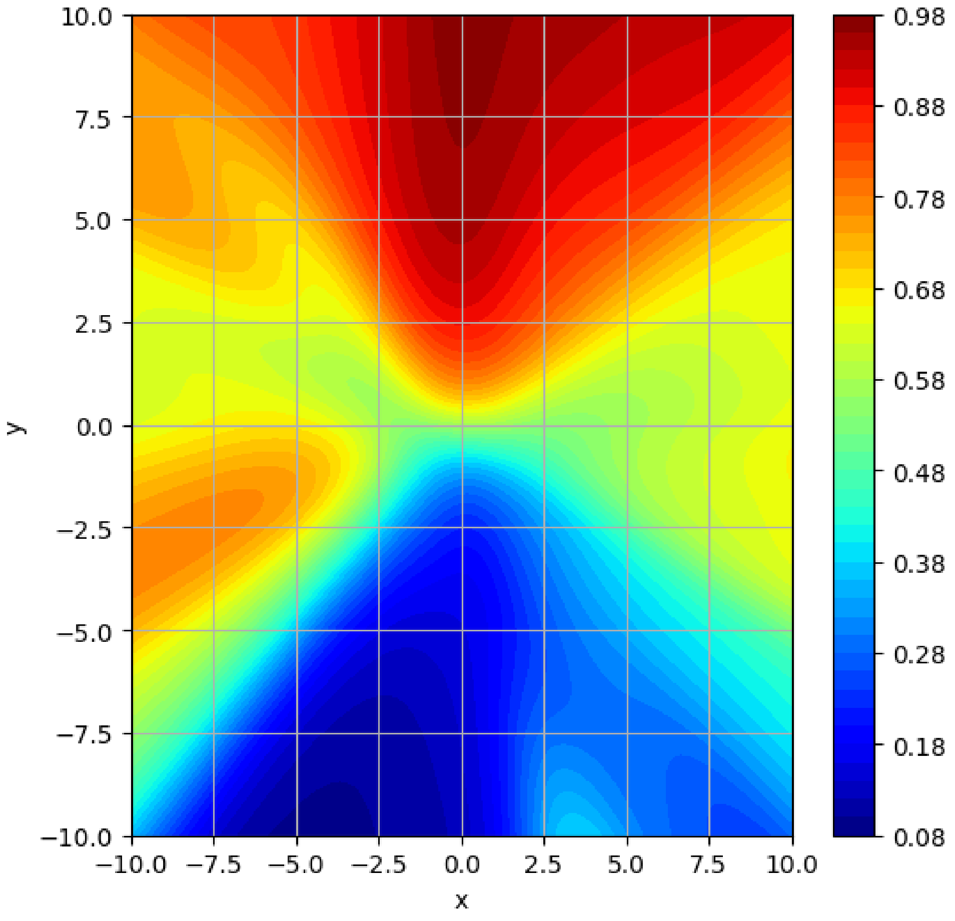


Figure 1.6: Commonly used activation functions. These generalize the step function in Fig. 1.4 by introducing nonlinearities with non-trivial derivatives. Note that some activation functions (e.g. Tanh and ReLU) yield bounded outputs. Thus, one must choose wisely the activation at the output layer so that the true values have a chance at being learned.

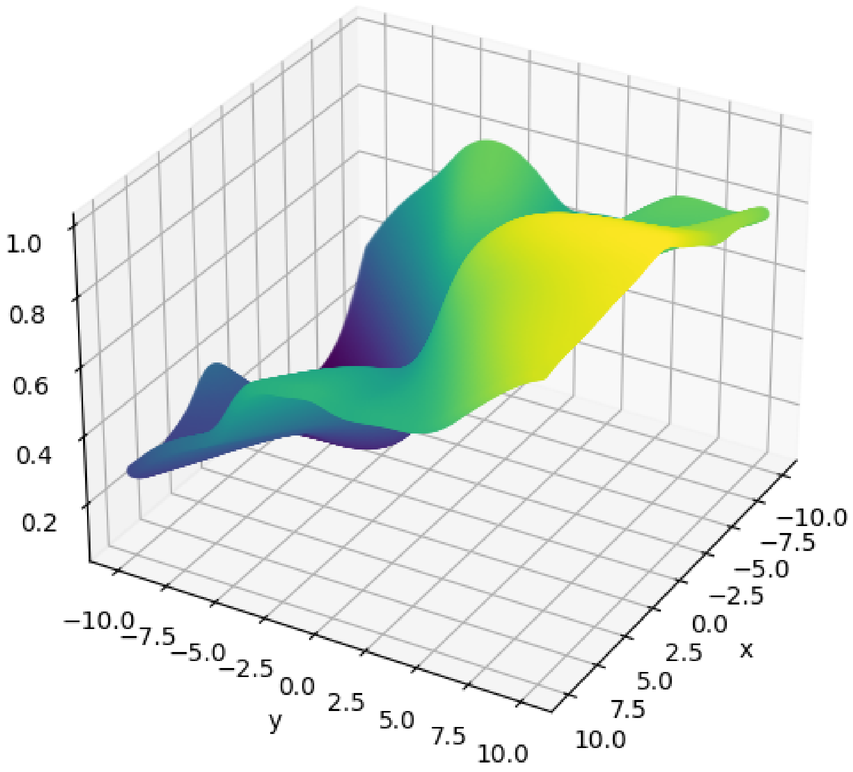
Next, I'll change the output activation function from the step function to the sigmoid (cf Fig. 1.6). Now the output is in the range $y \in (0, 1)$, and thus can be thought of as a *probability* for a given input vector belonging to a certain class (e.g. what is the probability that the input image contains a cat vs. not a cat?)

```
# use interpolation:
plt.contourf(X, Y, z.reshape(X.shape), 50, cmap='jet')

plt.grid()
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar()
```

```
[ ]: <matplotlib.colorbar.Colorbar at 0x7f9e61860760>
```



```
[ ]: # Plot x,y,z on a 3D plot:
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(points[:, 0], points[:, 1], z[:, 0], c=z[:, 0])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
# Rotate the plot:
ax.view_init(30, 30)
```



```
[ ]:
```

History

The MLP was first introduced by Frank Rosenblatt as early as 1960 (include figs), without a learnable hidden layer (i.e. only input and output layers had adjustable values). This took forever to train, but was capable of solving non-linear decision problems.

Universal Approximation

Not only is the MLP a useful way of parametrizing more complicated functions, it is actually be proven to be a *universal* function approximator. Loosely speaking, this means that *any* function can be parameterized by a (sufficiently large) MLP with just one hidden layer.

A bit more formally,

Universal Approximation Theorem (informal)

Given any smooth function on a compact subset of $\mathbb{R}^m$, $f : \mathbb{R}^m \to \mathbb{R}^n$, for any $\varepsilon > 0$, there exists an MLP with a single hidden layer of dimension $d < \infty$ such that for all x,

$$||f(x) - g(x)||^2 < \varepsilon \tag{1.2}$$

where $g : \mathbb{R}^m \to \mathbb{R}^n$ is a (properly parameterized) MLP.

This wonderful theorem, and more recent variants of it [**<empty citation>**] provide an existence theorem. This guarantees that any function (the solution to our supervised learning task) can be described as an MLP. However, this is at the price of (a) extremely large hidden dimensions (e.g. too large to fit on a computer) and more importantly (b) the theorem does not specify how to *construct* (i.e. set the weights) of the network.

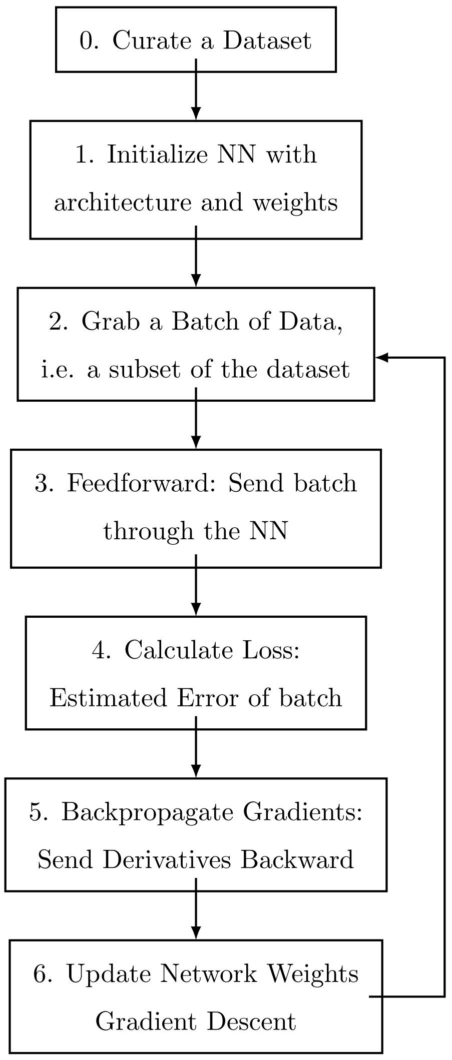
Nevertheless, this is an important theoretical guarantee (there are not very many in deep learning as of today). This allows us to (somewhat, based on caveats above) confidently use MLPs as an architecture to solve our problems of interest.

1.3 Training

We will now dive into the theory that supports the training of neural network architectures.

The uninitiated (or uninterested) should note that although the development of this theory was a necessary step in advancing deep learning, it mostly happens under the hood, without the practitioner needing to know the derivations nor calculations. Modern libraries such as PyTorch, Tensorflow, TinyGrad, [insert your favorite deep learning library here], etc. will take care of automatic differentiation, through a widely used “autodiff” package. Nevertheless, we believe that understanding the calculus of backpropagation will assist the reader in a deeper comprehension of the training process. It may also be of considerable use when the reader invents their own novel architecture, whose training framework does not immediately fit the status quo.

As a high-level overview, training a neural net requires the following steps:


0. Curate a Dataset
1. Initialize NN with architecture and weights
2. Grab a Batch of Data, i.e. a subset of the dataset
3. Feedforward: Send batch through the NN
4. Calculate Loss: Estimated Error of batch
5. Backpropagate Gradients: Send Derivatives Backward
6. Update Network Weights Gradient Descent

In the following sections, we will dive into the details of each of these segments in the training loop. For those uninterested in the details of calculation, this surface-level knowledge of training may be sufficient. Nevertheless, we believe providing such details will provide a deeper understanding of the process. We merely state this to remind the reader not to get lost in the weeds.

1.3.1 Step 0: Datasets

We'll consider a labelled dataset, denoted $\mathcal{D} = \{x_i, y_i\}_{i=1}^{N}$, where $N \gg 1$ is the total size of the dataset[6]. Curating a dataset often entails things like: deleting and cleaning spurious data with missing labels or corrupted values; cropping or greyscaling images to ensure consistency; re-weighting classes to ensure low bias; etc.

1.3.2 Step 1: Network initialization

Although we have discussed the architecture, or structure of the network that we wish to train, we have not fully specified its initialization. That is, what values of the weights θ should we choose before training? [7] has a good discussion in Chapter 8 on initialization, here I will try to give a brief overview of some of the main ideas. As discussed there, initialization still remains an open problem, especially with respect to convergence

[6]For now, we won't worry about splitting the full dataset into train/test/validation sets, but this is important when it comes to training and deploying models.

rates and even more so the generalization capability. It is useful to keep in mind that specifying an initialization of the weights is a way of encoding a prior belief into the network.

Symmetry Breaking

There has been a considerable amount of theoretical analysis in recent years studying how to initialize deep neural networks [14]. [7]The simplest property required for training (by a deterministic algorithm) is that there must not be symmetry in the parameters at a given layer. Because of the mechanism of backpropagation (to be discussed in section 1.3.6), the NN weights must not all be initialized to the same values. Otherwise, upon training, the NN will not update in a meaningful way.

Thus, one way of setting the weights would simply be to initialize them randomly in some bounded interval. This would ensure (with high probability) that the weights are sufficiently distinct to allow for efficient training.

Distributions

A popular distribution from which to draw the initial weights is defined by the "Xavier Glorot initialization" [6]: $\theta^{(l)} \sim \mathcal{N}\left(0, \sqrt{2/(n_l + n_{l+1})}\right)$ where n_l is the number of neurons in layer l. [8]

Vanishing and Exploding Gradients

One of the primary issues in training deep neural networks is that of vanishing or exploding gradients. This problem derives from the fact that deep nets are compositions of many functions. Thinking of each function in the chain of composition as a matrix multiply (i.e. only look at weights and not biases or non-linearities), then the effect of chaining many such matrices together can be roughly described by their largest contribution. The most significant contribution in a matrix multiply, in some sense, is given by the first term in its spectral decomposition, or the largest eigenvalue of the matrix. If we have a chain of many matrix multiplies, we can think of the first-order effect as being attributable to the product of the dominant eigenvalues (and corresponding normalized eigenvectors). If the dominant eigenvalue is on average less than unity, then a chain of many such matrices will result in a vanishingly small output. Similarly, for matrices with dominant eigenvalue typically larger than unity, the output will explode as depth increases. Clearly, this is quite problematic. We certainly want to use deep networks as they are and more efficient to feedforward (and just as expressive) than their very wide counterparts. [9]

To combat this issue, we can choose an initialization of weight matrices such that their dominant eigenvalue is close to one.

1.3.3 Step 2: Sampling batches

In classical or standard gradient descent, there is no batch, and the sample is the entire dataset. Although this may work, it has two issues: (1) we don't have unlimited RAM/VRAM and hence cannot fit an entire batch in the memory of our computer, and (2) it yields a deterministic algorithm for gradient descent. Although point (1) is a practical concern, point (2) is a little more mysterious. We will see later that the randomness given by

[7]Looks interesting: [15]
[8]The Gaussian can also be replaced with a uniform distribution after swapping the 2 for a 6.
[9]This informal discussion is based on the first few chapters of [14].

stochastically sampling from our large dataset $\mathcal{D}$ allows gradient descent to more effectively "explore" the loss landscape.

We will denote the batch size as $B \in [1, |\mathcal{D}|$.

- $B = 1$: "stochastic gradient descent"
- $B = |\mathcal{D}|$: "gradient descent"
- else: "(mini-)batch gradient descent"

When the context is clear, I will use "gradient descent" or GD. The optimal batch size B is not known *a priori*, and thus we must treat it as a **hyperparameter**[10] Hyperparameters are termed such because they are not parameters (as are the weights and biases in the model) and they are not (historically at least, cf. "meta-learning" techniques) learned, and instead must be hand-chosen or otherwise optimized through other search methods.

In many applications, the largest batch-size that does not bottleneck the memory capacity of the hardware is typically used by default, though (especially in Reinforcement Learning) this is not always true and the batch size generally requires tuning.

The batch size is typically chosen to be a divisor of the full dataset size. Then, after enough batches have been sampled, we will have used up the entire dataset without sparing/forgetting any data. Once this has occurred, we say a training "epoch" has passed.

1.3.4 Step 3: Feedforward as matrix multiplication

While introducing neural networks, we have seen the feedforward (or forward pass) of a single data point through the networks architecture as a series of multiplications, additions, and activation functions. Now we will emphasize the viewpoint of the former two operations as a matrix product.

If we carefully write out the function expressed by the (shallow[11]) neural network, we find that it can be expressed as a set of matrix multiplications.

$$f_\theta(x) = \sigma(\mathbb{W}^{(2)}\sigma(\mathbb{W}^{(1)}x + b^{(1)}) + b^{(2)}) \tag{1.3}$$

go over shapes of each structure. Understanding the shapes of data is important for writing and debugging code. The beauty of this equation is that we can understand it a bit from the perspective of linear algebra, and more importantly, it can be efficiently calculated with GPU hardware.

Moreover, when we go to feedforward many data points, the data vector $\vec{x}$ can be recast as a data matrix. Luckily, because of our abstraction in writing matrix products, we don't have to make any changes to the above formula. In this case, the addition of the bias vectors has to be understood as performed on a column-wise basis (the bias vectors must be tiled across the data dimension).

1.3.5 Step 4: Loss Functions

The loss function (also referred to as cost, error, which I'll use interchangeably) determines how well / poorly the network is doing (at predicting the outputs defined by the dataset).

[10]We will meet many other hyperparameters.
[11]We call NNs shallow if there is only one hidden layer

The choice of loss function is task-dependent. If the task is **classification**, then one wishes to learn to which class an input belongs. Choosing a single class would be too "brittle" and throw away too much information when the NN predicts incorrectly. So to address this task, we use neural nets which learn (discrete) probability distributions over the set of classes. This way, the neural net can assign some probability to the input belonging to class A,B and C.

In the previous examples, the neural networks we considered had only a single output value, $y \in \mathbb{R}$. To provide a probability for say C classes, we require C output nodes in the architecture. insert diagram. With this setup, though, we are not ensured to have a probability distribution over the final layer as desired. How do we solve this? Well, the final layer (before going through any activation function) consists of a set of C real numbers. Our challenge is thus to transform a set of real numbers to a well-defined probability distribution.

Recall that for a vector $x \in \mathbb{R}^C$ to represent a probability distribution, it must satisfy:

$$\forall i: \; x_i > 0, \tag{1.4}$$

$$\sum_{i=1}^{C} x_i = 1. \tag{1.5}$$

So, we must cook up a transformation which sends real numbers to the range $[0, 1]$ in such a way that normalizes the vector as in Eq. (1.5).

One way to enact this operation is via the "softmax" function:

$$\text{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{i=1}^{C} e^{x_i}} \tag{1.6}$$

The exponentiation in the numerator ensures that all real numbers are mapped to positive values, as $e^x > 0$ for all $x \in \mathbb{R}$. The denominator normalizes the vector to satisfy Eq. (1.5).

The other broad class of problems in deep learning is regression, where the goal is to predict a value (as opposed to a discrete class). In such a case, we don't have to worry as much about the activation at the output layer, so long as it yields the values known to be "true" (based on the labels $y \in \mathcal{D}$). For example, if we wish to predict the sale price of a house, then we expect the output of our NN to be a positive value (assuming houses have value). In this case, the ReLU activation at the output would be sensible, as it restricts the output range of the function to be positive. If on the other hand, we expect the output value to be real (positive or negative) we can impose no activation function, leaving a linear layer at the output.

Now that we have discussed the two areas of classification and regression, we are ready to detail the loss functions applicable to these tasks.

Cross Entropy Loss

Cross Entropy Loss for binary classification:

$$L(y, \hat{y}) = -\left(y \log(\hat{y}) + (1 - y)\log(1 - \hat{y})\right)$$

Mean Squared Error (MSE) Loss

MSE Loss for regression:

$$L(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Mean Absolute Error (MAE) Loss

MAE Loss for regression:

$$L(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

1.3.6 Step 5: Backpropagation

I don't want to repeat the amazing tutorial on backprop written by Nielsen: `http://neuralnetworksanddeeplearning.com/chap2.html`, so I'll just give some of the basic results to prepare the reader a bit for that reference.

The initial idea for updating a network to reduce its loss would be to change each parameter by a small amount in some direction. If the network performs worse (bigger loss), make the opposite change. If the networks performs better (smaller loss), accept this change to the parameter. Continue *ad infinitum.* In this algorithm, we are approximating the derivative of the loss with respect to each parameter, and then *descending the gradient.*

This idea, though correct in spirit, does not "scale" [12] to large networks with many parameters. Instead, we can compute the derivatives of the weights *exactly* rather than approximately. Then, we will use the idea of gradient descent to descend the loss function to (hopefully) the global minimum. In order to calculate the derivatives of parameters, we will need a bit of calculus.

Chain Rule

The chain rule from differential calculus is the backbone of the backprop algorithm. As a reminder, the chain rule for a multivariable function $f(g(\vec{x}))$

$$\frac{d}{dx}f(g(x)) = \frac{df}{dg}\frac{dg}{dx} \tag{1.7}$$

1.3.7 Computation Graph

Neural networks can be conceptualized as computational graphs, where nodes represent mathematical operations, and edges represent the flow of data. Understanding the computation graph is crucial for performing the backward pass (backpropagation) in a neural network.

Directed Acyclic Graph (DAG)

A neural network must be a Directed Acyclic Graph (DAG) to facilitate the backpropagation process effectively. A DAG is a graph that consists of nodes connected by directed edges, and importantly, it has no cycles. This acyclic nature is essential for the well-defined backward flow of gradients from the output to the input.

[12] we will use this term a lot in later chapters. A method is said to 'scale" if it "survives" (i.e. still works well, or better) as the size of the network increases.

- **Gradient Descent Update:** Backpropagation involves computing the gradients of the loss function with respect to the network parameters. The acyclic structure ensures that the backward flow of gradients is well-defined and finite.
- **Stopping Gradient Accumulation:** In a DAG, there is a clear endpoint where the input nodes are reached. This allows the network to know when to stop accumulating the gradient during backpropagation. Without cycles, the gradient computation process terminates, preventing an infinite loop.
- **Parameter Updates:** Gradients computed during backpropagation are used to update the parameters of the neural network using optimization algorithms like gradient descent. The acyclic structure guarantees a clear and finite path for updating parameters.

Recurrent Neural Networks (RNNs)

It's worth noting that Recurrent Neural Networks (RNNs) introduce cycles in the graph due to their recurrent connections. While RNNs violate the acyclic property, they have mechanisms to handle this, such as unrolling the recurrent connections for a fixed number of time steps during training. We'll discuss the details of this in a later section.

In summary, the acyclic nature of the computation graph is a fundamental requirement for effective backpropagation in neural networks. It ensures a well-defined and finite process of computing gradients, enabling the network to learn and update its parameters efficiently.

Gradient Accumulation

To calculate the derivative of the loss with respect to a parameter in the network, we will have to step backwards from output layer toward the input layer. In stepping from the output to the input, the first (or zeroth) step is calculating how the loss affects the output layer. All of these calculations are done symbolically, with pre-activation values being stored from the forward pass (step 3).

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{1.8}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{1.9}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{1.10}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{1.11}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{1.12}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{1.13}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{1.14}$$

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}}, \tag{1.15}$$

1.3.8 Step 6: Gradient Descent

Now that we have computed the gradients, we need to improve the network (reduce the loss). To do so, we will descend the gradient as previously discussed. One possibility is to calculate the loss for the entire dataset, $\mathcal{L}(\mathcal{D})$. Then, we descend this gradient by calculating new weights based on:

$$\theta_{i+1} = \theta_i - \alpha\nabla_\theta\mathcal{L}(\mathcal{D}) \tag{1.16}$$

where α is the step-size or **learning rate** of the gradient descent algorithm. The learning rate pushes the parameters in the correct direction by a certain small amount, α. The learning rate is our first hyperparameter (named as such to distinguish from the parameters inside the model). We will soon see that the choice of α is very important for the convergence of a training algorithm. It will be the job of the user to choose a good learning rate through some experimentation.

The reason that the learning rate is a small value ($\ll 1$) is because the loss function is not linear (in parameter space) and thus the gradient will only approximate the loss function near the evaluation point [13]

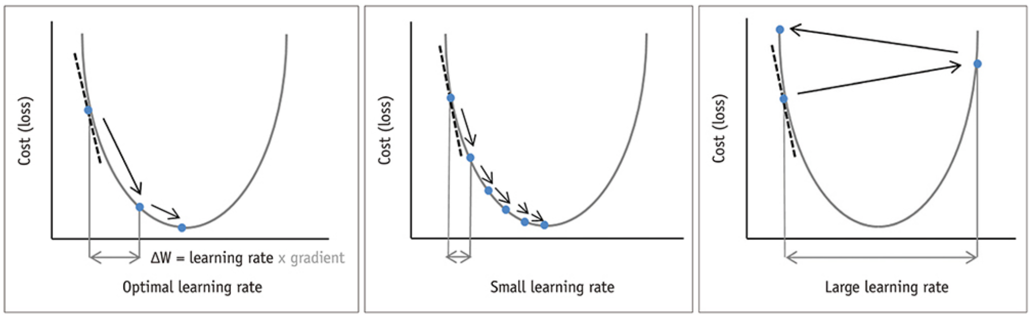


Figure 1.7: Gradient descent for various learning rates. Stolen from `https://analyticsindiamag.com/how-to-use-learning-rate-annealing-with-neural-networks/`

To help imagine how difficult the problem of loss minimization truly is, imagine you are the NN. The only

[13] As in Taylor expansion analysis, it is possible to include a second derivative term or higher. These techniques are known as "higher order" methods and do not seem very common in practice.

input signal you receive is a single number (the loss) and a hyperplane (set of derivatives corresponding to each weight). You don't have access to the rest of the loss function landscape. So how can you decide where to go (i.e. how to change your weights)? One technique, as in a dark room, is to move very slowly, moving away from any bumps or walls you encounter. On top of this, the loss function is extremely high-dimensional (one dimension for each parameter), making our intuition about optimization fail drastically [14].

Stochastic (Minibatch) Gradient Descent

Rather than calculating the loss over the entire dataset, we will instead take a sample, or batch, of the dataset. We calculate the loss only over this smaller batch. This will be much faster than calculating the loss for the whole dataset, but it will not give as accurate a derivative. However, randomly sampling at the price of inaccurate derivatives can allow the optimization procedure to "explore" the loss landscape by stochastically moving about rather than deterministically, as it would on the full dataset. Thus, stochastic gradient descent (SGD) is not only more computationally efficient, but can also lead to better (i.e. lower loss) solutions. Moreover, SGD has been found to prefer wider minima of the loss landscape, which is preferable from the viewpoint of stability and generalization.

1.3.9 Momentum

The previous method of gradient descent does not incorporate any "memory"; it simply calculates a new gradient at the new position (in parameter-space). However, when the learning rate is set too high, this can cause the parameter to "chatter" or bounce around near a local optimum. One method to alleviate this, of course, would be to reduce the learning rate. However, this could increase the time for convergence to an intolerable value. Instead, the gradient descent algorithm could have some memory of the previous state (previous parameters and gradient) and use this to gain a better estimate for the loss minimum. Another difficulty with standard gradient descent is the occurence of "barren plateaus": large stretches of the loss landscape which have essentially no gradient. This could be a flat patch or a saddle point. Both are problematic for standard gradient based techniques.

This is precisely what momentum-based methods aim to counteract. By tracking the previous gradient, there is a momentum term [15] included in Eq. (1.16):

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta \mathcal{L} \tag{1.17}$$

[14]or my intuition, at least

[15]the physical analogy to momentum is not perfect, especially because we are in discrete time. You can instead think of this as maintaining a rolling average of gradient values.

1.3.10 Adam: Adaptive Momentum

It is difficult to visualize high dimensional loss functions, so we typically start with lower dimension (e.g. 1- or 2-dimensional space). In low dimensions, we can easily imagine gradient descent getting trapped in the various local minima arising from all sorts of bumps in the loss landscape. However, this intuition is entirely incorrect in high dimensional spaces, where deep learning takes place. In fact, saddle points are much more likely than local minima in high dimension. An easy way to understand this intuitively is to imagine that at every point, there is some probability p for the function to be concave up (and thus probability $1-p$ to be concave down). Recall that a local minima corresponds to **all coordinate directions** having a positive second derivative (concave up). Based on our setup, this would only happen with probability p^N where N is the number of parameters in the model, these days $N \gg 10^6$. For any non-zero probability p, we see that this phenomenon (occurence of local minima) is extremely unlikely, because it would require all N dimensions to be concave up. If even one is concave down, then we will get a saddle point (Fig. 1.8) instead of a minimum. Thus, in high-dimensional space, the real issue is saddle points, not local minima. Thus, we need techniques which can easily escape such saddle points. Enter momentum...

(From Andrew Ng's Coursera lectures)

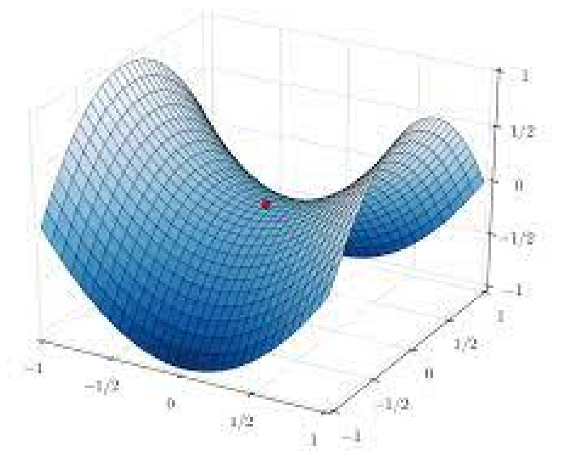


Figure A

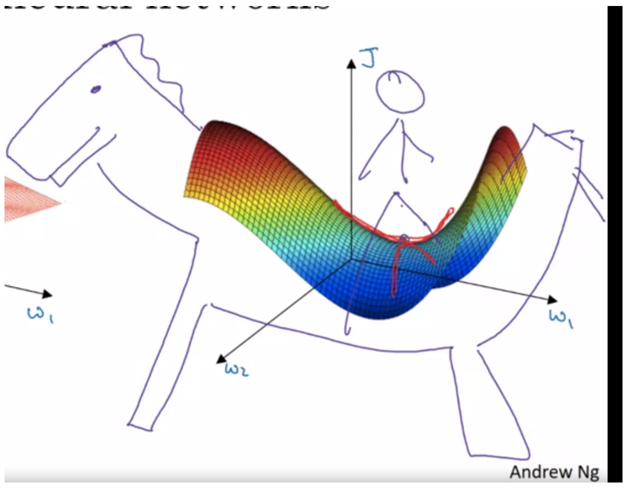


Figure B

Figure 1.8: Saddle points are points in a mathematical function where the surface curves up in one direction and down in the other, resembling a saddle. Figure B shows Andrew Ng's rendering of the eponymous "saddle" point with hand-drawn horse and rider.

One major breakthrough in training algorithms was given by RMSProp, where each parameter in the model will have its own learning rate, which is dynamically updated based on past gradients. Adam (adaptive momentum) is a different version of momentum-based methods which has seen amazing success in practice [16] Momentum allows the optimization algorithm to skim past "historically" steep directions to more quickly find optima and escape barren plateaus.

[16] As of writing, the Adam paper [12] has over 160k citations.

$$\vec{\nu}_t = \beta_1 \vec{\nu}_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L} \tag{1.18}$$

$$\vec{s}_t = \beta_2 \vec{s}_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L})^2 \tag{1.19}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\vec{\eta}_t}{\sqrt{\vec{s}_t} + \epsilon} \nabla_\theta \mathcal{L} \tag{1.20}$$

The rolling average of the gradient and its square are tracked, and are used to devise an adaptive parameter-wise learning rate. The operations in Eq. (1.20) are performed element-wise. (I emphasize the vectorial nature to show from where the "parameter-wise" comes)

1.3.11 Newer methods

A new method, LION [4], has entered the arena of optimization algorithms. Though it is still too early to say if this will be as successful as e.g. Adam, it appears to perform quite well on image-based techniques. Notably, this optimization algorithm's code was not hard-coded but *learned* through a reinforcement learning algorithm. Perhaps in the future, other algorithms will similarly be discovered.

1.3.12 Concluding Remarks

To conclude this chapter on training NNs, consider the phenomenon of overfitting. Ilya Sutsekver puts it nicely:

"Overfitting is when your model is somehow very sensitive to the small random unimportant stuff in your training dataset. So if you have a small model and a big dataset, the small model is kind of insensitive to the noise...
Suppose you have a huge neural network (huge number of parameters). Now let's pretend everything is linear. Then there is this big subspace where the neural net achieves zero error. SGD is going to approximately find the point with the smallest norm in that subspace. That can also be proven to be insensitive to the small randomness in the data when the dimensionality is high. But when the dimensionality of the model is equal to the dimensionality of the data, then there is a one-to-one correspondence between all the datasets and their models. So small changes in the dataset lead to large changes in the model. "
— Ilya Sutskever, May 8 2020 on Lex Fridman podcast

This point where parameters are on the same order of magnitude as dataset size is the peak of "double descent bump". To go beyond this correspondence where small perturbations are quite harmful, we have to (greatly) increase model size, so that params $\gg$ data.

need to discuss train/test/validation sets and softmax function more clearly. softmax=prob, tanh/sigmoid=squashing

Chapter 2

Advanced Architectures

In this chapter, we'll shift away from the MLP toward more advanced architectures, used in real-world applications. We will begin with the Convolutional Neural Network (CNN or ConvNet) used in image-based tasks (Sec. 2.1. Then, we'll study Residual Networks (Sec. 2.2, Recurrent Neural Networks, Long Short-Term Memory, and Transformers as solutions for sequential (time-ordered) problems.

No free lunch theorem, and priors about datasets / good models

2.1 Convolutional Neural Nets

2.1.1 Motivation

Problem Suppose we are given the task of training a neural network on a set of image data. E.g. "Given a 256×256 image, predict if the image contains a cat, dog, or neither." How can we go about this? Perhaps we can try to use our MLP? Let's then set up an MLP with the correct number of input dimensions. For simplicity, let's use the same number of nodes in the two hidden layers as there are in the input. We will have three output nodes corresponding to the probability of the image containing a "cat", "dog", and "neither".

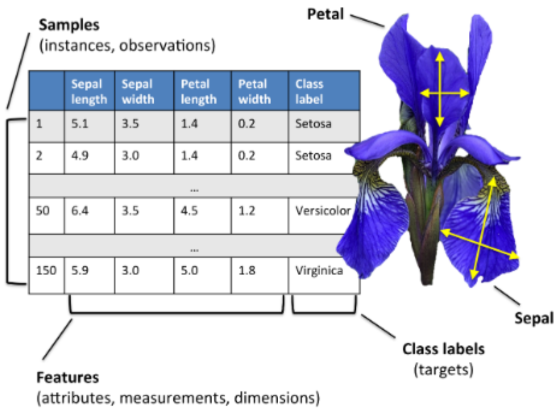
A simple calculation shows how many parameters would be in this model. First, for a color image (RGB), there are three channels, therefore $256*256*3 = 196608$ input dimensions. For an MLP, we have fully connected weights:

$$196608 * 196608 + 196608 * 196608 + 196608 + 196608 = 77309804544 = 77B \tag{2.1}$$

where the last two additions are for the biases on each neuron. So a total of 77 billion parameters (even without connecting to the 3 output nodes)... seems like a lot! (This was only for 3 channels, more possible, dependent on application e.g. medicine, fluorescence microscopy, weather data.) Maybe we can remove the color channels, and just use black and white images, so the figure of interest is $256 * 256 = 65536$, leading to a total parameter count of

$$65536 * 65536 + 65536 * 65536 + 65536 + 65536 = 8.6B \tag{2.2}$$

As a comparison, the Llama large language models (near SOTA in sheer size) exists in the range of 7B-70B parameters, and it is capable of performing extremely complicated tasks (next word prediction / generating interesting text). Can we do any better without down-sampling / decreasing the model complexity (e.g. we could just use a shallow network on a tiny image)?


Samples
(instances, observations)
Petal
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Figure 2.1: Sepal/petal feature extraction

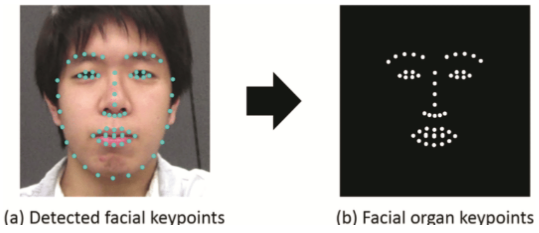


Figure 2.2: taken from `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L13_intro-cnn__slides.pdf`

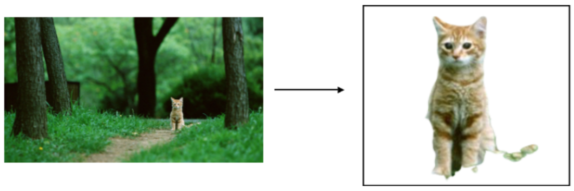


Figure 2.3: Manual pre-processing

Context

As some historical context on this problem, let's first consider what was done by the computer vision (CV) community in the past: feature design, feature detection, and cropping; all by hand! Some examples of hand-coded features are shown in Figure 2.2. As you can see, the feaures are indeed relevant for the problem at hand, making them useful, but highly problem-specific. The invention and calculation of hand-designed features took a lot of time, as expected (requires human-in-the-loop). How can we fully automate this process? (The most difficult missing process is probably feature design.)

Solution

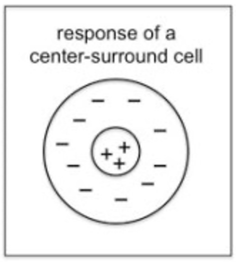
:

Delving into the intricacies of how our biological neural networks process visual information, we turn to the seminal work of Hubel and Wiesel from the 1950s [11]. Their groundbreaking research revealed that specific neurons exhibit heightened activity in response to distinct visual patterns, such as lines, within the field of view. The nuanced activation of different neurons corresponds to variations in line angles and positions. Notably, Hubel and Wiesel's contributions earned them the Nobel Prize in Physiology and Medicine in 1981.

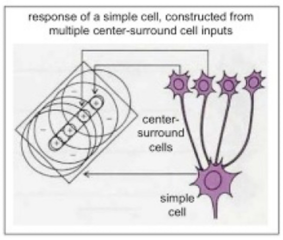
In the realm of artificial neural networks, Convolutional Neural Networks (CNNs) serve as a sophisticated computational paradigm for addressing image-based challenges by learning specialized image features. Drawing inspiration from the hierarchical and specialized architecture observed in the brain's visual processing pathways, CNNs leverage convolutional layers to autonomously learn hierarchical features from images. These layers emulate the local pattern detection akin to the receptive fields in the initial stages of visual processing within the brain.

Moreover, the adaptability of CNNs to discern complex and non-linear relationships echoes the brain's remarkable capability to discern intricate visual features. Studies, such as "Linear summation of excitatory inputs by CA1 pyramidal neurons" [3], underscore the biological neural network's proficiency, particularly in the hippocampus region, in distinguishing linearly-separable features. This finding underscores the connection or inspiration between biological and artificial neural networks. However, modern CNNs are not so similar to their biological correspondence, due to engineering requirements (need for fast, general-purpose architectures that aren't limited by biology).

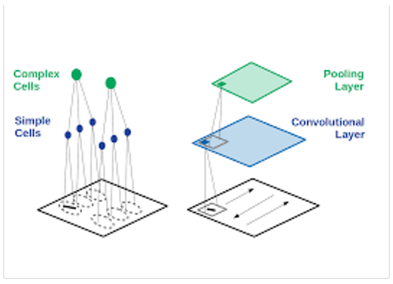
In summary, the amalgamation of insights from biological neural network research, exemplified by the work of Hubel and Wiesel, with advancements in artificial neural networks, notably CNNs, has engendered powerful solutions to image-based problems. This synergy empowers neural networks to autonomously learn and extract hierarchical features, positioning them as formidable tools in tasks ranging from image classification to object detection.



(a) Simplest receptive field is based on clumps of similar responses


response of a simple cell, constructed from multiple center-surround cell inputs
center-surround cells
simple cell

(b) Lining up many center-surround cells yields filters resembling lines in a given direction and position.



(c) Computationally, stacking simple to complex cells is analogous to the convolutional filter performed by CNNs.

(ex input graph)

Filters

The main idea of automatically learned features are two-dimensional[1] **filters**. A filter is a single feature, which can be thought of as a miniature image, which will be slid across (convolved with) the input image. If the filter "lines up" or "agrees" with the input image at some location, then it will excite "the filter neuron", and propagate this information downstream. More specifically, the presence of a filter is based on a pixel-wise dot product between the filter (maybe a small line or circle, to be more concrete) and the input image. The result of such images "lining up" is a non-zero entry in the corresponding output. This is a tough concept to understand and proper visualizations are required. As seen in Figures 2.5, 2.6, the output (similarity/"lining up") of a small patch (filter) results in a single pixel intensity at an abstract image downstream (deeper in the network). Each filter, after sliding over the input image, constructs a single "abstract image", one pixel at a time (a single blue sheet in Fig. 2.5). If one has $n1$ filters at the first convolutional layer, this will thus correspond to $n1$ new

[1]Convolutional nets are not limited to two-dimensional image data, they can be used on 1D timeseries, and multi-channel (e.g. microscopy) data.

"abstract images". These are stacked together into $n1$ channels. This operation of "convolving" a filter against an image is used to construct deeper (more abstract) images. Notice that deeper in the CNN, the (abstract) images become smaller, due to the compression arising from a convolution (a single 3×3 miniature-image filter is sent to a single pixel in the output abstract image). Deeper layers can also yield smaller images because of downsampling due to pooling layers[2] Finally, notice that the abstract images are becoming "longer" in the sense of larger number of channels. This is due to the use of an increasing number of filters in deeper convolutional layers, which is a common technique.

Convolutional layers can be thought of as applying logical operations or relations between smaller features. In the early layers, the network may learn to detect simple features like edges—horizontal, vertical, and diagonal lines. As these lines are extracted, subsequent layers combine them to recognize more intricate patterns. Imagine a scenario where a horizontal line and a vertical line converge within a receptive field. In this case, the network might learn to respond strongly to the presence of a corner, as corners often manifest when lines intersect.

As we progress through the network's layers, the concept of corners can become part of even more sophisticated representations. These representations might involve the arrangement of corners into shapes like 'L' or 'T', which, in turn, contribute to the recognition of more complex structures such as the contours of objects or the boundaries of distinct regions within an image.

At a higher hierarchical level, the network could assemble these contours to identify specific objects. For instance, the combination of corners and edges might contribute to recognizing the contours of a door or the outline of a window in an image. The network, through its hierarchical learning process, effectively transforms low-level features like lines and corners into abstract, high-level representations that capture the semantics and context of the input.

In essence, the hierarchical nature of convolutional layers allows the network to understand increasingly abstract concepts by combining and building upon simpler features. This process mirrors how our brains naturally progress from perceiving basic elements to comprehending complex objects in visual scenes, showcasing the power of convolutional neural networks in capturing hierarchical structures in visual data.

2.1.2 Computation of Convolution

The filters must have same number of channels as the input for the convolution to be well-defined.

In convolutional neural networks, the convolution operation is a fundamental building block that enables the extraction of features from input data. One critical consideration is the matching of the number of channels between the filters (also known as kernels) and the input data. This constraint ensures that the convolution operation is well-defined and computationally valid.

Consider an RGB image as an example, where each pixel has three color channels (red, green, and blue). If a convolutional filter is designed to capture specific patterns within these color channels, it must possess the same number of channels as the input image to perform a valid convolution.

Mathematically, the convolution operation involves sliding the filter over the input image, channel by channel, and computing the element-wise product of the filter values and the corresponding pixel values in the input.

[2]It is worth noting that pooling is not as prevalent in modern architectures due to decreased computational demands. Nevertheless, when employed, pooling effectively downsamples feature maps by applying operations such as maximum, average, or other global functions over local patches of pixels.

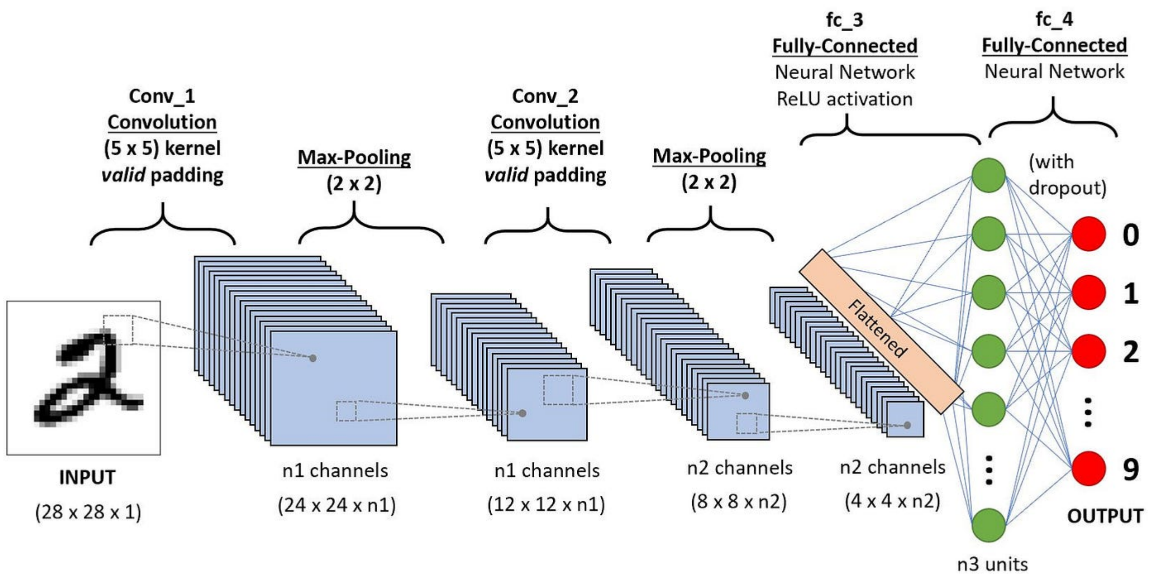


Figure 2.5: Visualization of a typical deep CNN. Notice that pooling downsamples images, and n_1 (number of filters in layer 1) is equal to number of channels in the input to layer 2. Typically, the penultimate layer is fully connected (cf. MLPs in 1.2.2) after being flattened (cf. the MLP application to MNIST in **??**).

These products are then summed to generate a single value in the output feature map. This process is repeated for each location in the input, producing a complete feature map that highlights specific patterns learned by the filter.

The constraint of having the same number of channels ensures that the convolution operation is performed coherently across all channels of the input. It allows the filter to capture patterns and relationships within each channel simultaneously, facilitating the extraction of complex features and preserving spatial information.

Moreover, modern CNN architectures often employ multiple filters in each convolutional layer, each responsible for capturing different features. These filters are typically small in spatial extent but extend across all channels. The combination of multiple filters with the same number of input channels enriches the network's capacity to learn diverse and hierarchical representations from the input data.

In implementation, frameworks like TensorFlow or PyTorch handle these channel matching operations seamlessly. Convolutional layers are designed to enforce the channel compatibility constraint, ensuring that the number of input channels matches the number of channels in each filter. This meticulous attention to channel alignment is crucial for the coherent and effective learning of hierarchical features in convolutional neural networks.

When writing your model, you can choose the number of "convolutional filters" in a given layer. This number will control the number of channels in the proceeding layer. This is because the result of a single filter being applied to an input image is a single 2D image. Multiple filters can thus be thought of as multiple 2D images. When stacked together, this is equivalent to a 3D image (2D with channels equal to number of filters).

The important takeaway about convolutional neural networks being a more efficient (less parameters) model for images, is through the lens of a "prior":

Definition 1 (Prior)**.** A prior is a probability distribution over the parameters of the model, indicative of our belief about what is reasonable.

Intuitively, the prior encodes some idea we have for how to perform well on a task. Rather than having

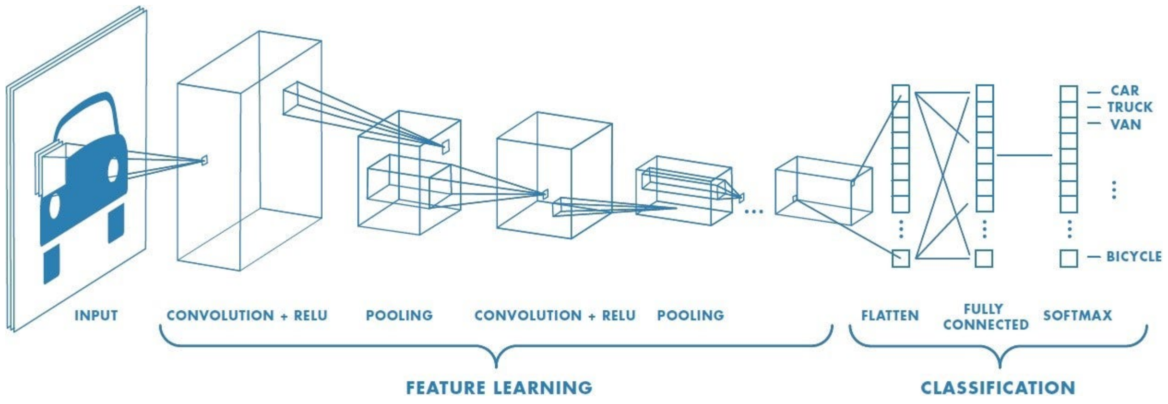


Figure 2.6: You may also see such visualizations of CNNs representing images with many channels as volumes.

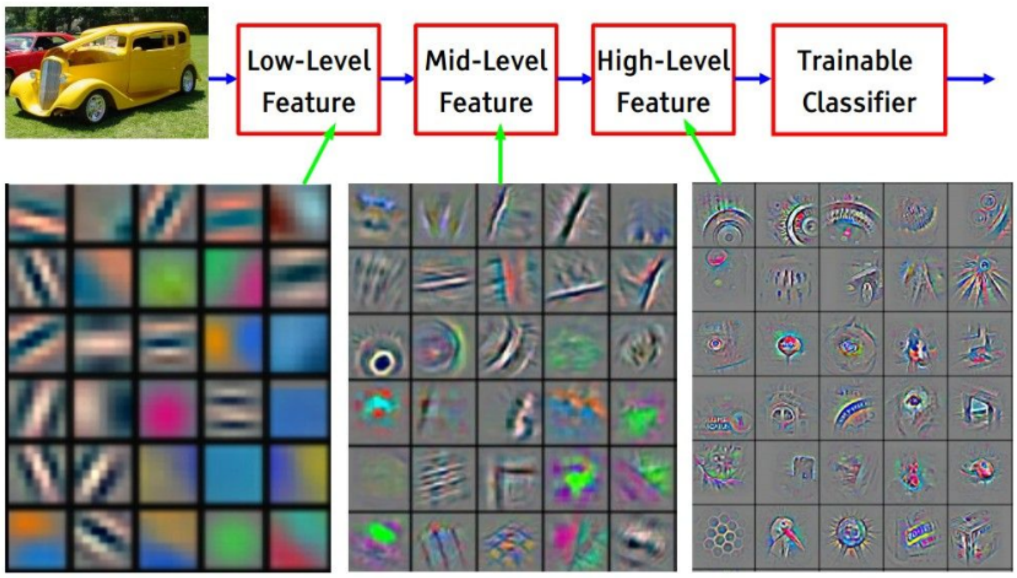


Figure 2.7: Learned filters in an image classification setting. Notice how more complex features emerge in the "abstract" deeper convolutional layers.

huge width of fully connected (dense) layers, one can instead propose specific more sparse architectures. The first example of this is of course the CNN we have just discussed. Instead of performing dense matrix multiplies throughout, there is an encoded spatial dependence through the use of filters (the prior) which makes the network much smaller. In fact, most of the learnable parameters in modern models (e.g. ResNets) live in the Dense (fully connected) layers, as opposed to the many convolutional layers typically preceding it.

2.2 Residual Networks

Now we will introduce a new architecture that blends well with CNNs: The ResNet (Residual Network).

2.2.1 History

Residual Networks, or ResNets, represent a significant milestone in the evolution of deep neural networks, particularly in the domain of computer vision. Developed by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, ResNets were introduced in their seminal paper titled "Deep Residual Learning for Image Recognition," presented at the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

The motivation behind ResNets stems from the observation that as neural networks grow deeper, they encounter the vanishing gradient problem, where gradients diminish exponentially as they backpropagate through the layers during training. This phenomenon hampers the ability of deep networks to learn meaningful representations, limiting their overall performance.

The key innovation of ResNets lies in the introduction of residual blocks, which incorporate skip connections, also known as shortcut connections or identity mappings. Unlike traditional networks, where each layer learns a direct mapping from input to output, ResNets allow information to bypass certain layers, facilitating the flow of gradients and addressing the vanishing gradient issue.

The residual block architecture is simple but remarkably effective. Instead of trying to learn a desired mapping directly, ResNets learn the "residual" mapping: the difference between input and output. This residual mapping is then added back to the input, enabling the network to focus on learning the residual details rather than the entire transformation. This skip connection mechanism facilitates the training of much deeper networks with hundreds or even thousands of layers.

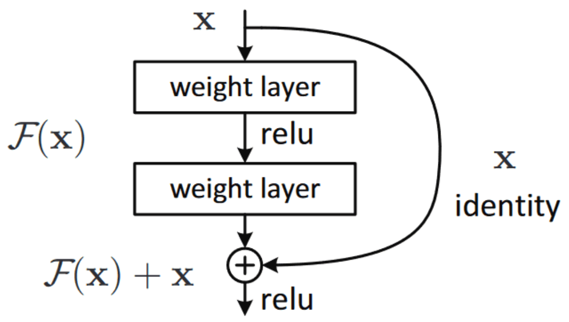
The introduction of ResNets marked a breakthrough in training deep neural networks and significantly advanced the state-of-the-art in image recognition tasks. The architecture not only mitigated the vanishing gradient problem but also enabled the successful training of networks with unprecedented depth. The ability to build deeper networks led to improved feature learning and hierarchical representation, ultimately boosting performance on a variety of computer vision tasks.

2.2.2 Motivation

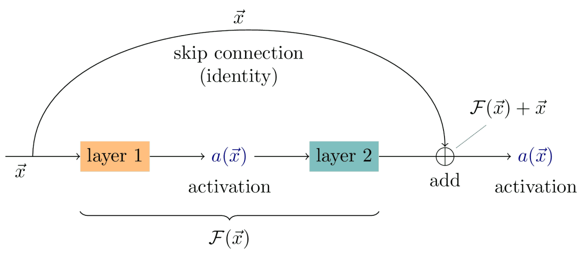
As some further motivation, I'll outline some of the insights from the ResNet paper [9] [3] from 2015. Much of this discussion was derived from this nice video `https://www.youtube.com/watch?v=GWt6Fu05voI` by Yannic Kilcher.

[3]This paper has a whopping 200k citations, evidence of its enormous impact on the field. More advanced architectures such as LSTMs and Transformers utilize the idea of residual connections.

- Insight 1: Deeper networks *should* obtain loss at least as low as a shallow network. (There always exists a deep network (m layers) with same capability as the shallow network (n layers) of same architecture. By construction, just make the final $m - n$ layers equivalent to the identity map. Unfortunately, identity is just as hard to learn as any other map!)
  - If deeper is always better, let's just always use deeper models.
  - However, this was not practical and deeper > shallower was not seen experimentally. After making models even deeper, loss started going back up! (and not because of overfitting, both train and validation loss increased. cf. Fig. 1 of [9].)
  - One reason is because of the vanishing / exploding gradients problem previously discussed.
  - One way to fix such a problem is to pretrain the first few layers, then freeze them and add more layers consecutively. This was done in practice by e.g. GoogLeNet.
- Insight 2: We want models to be initialized to near identity (rather than near zero, as given by the typical initialization schemes (Sec. 1.3.2). So let us create an architecture supporting this prior.
  - Make identity the default function, and learn a deviation or *residual*!
  - This is an interesting prior: We are assuming that the constituent functions of a deep network are (to first order) the identity function. It sounds reminiscent of e.g. the exponential's Taylor expansion. Compare this to the fully connected case, where we are looking at a random matrix, likely far from the identity at initialization.
- Insight 3: A simple way to instantiate this idea algorithmically is through a "skip connection", shown in Fig. 2.8a, 2.8b. $\mathcal{F}$ can be thought of as a correction factor
  - This idea successfully implements a "near-identity" initialization. Similarly, when using weight decay (L2 regularization on weights) the network will be pushed towards the identity map, rather than the "zero" linear transformation map [4]!
  - Note that a ReLU is typically added at the output of this block, making it non-linear and thus not the identity, but the identity is still present within the inner block.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

(a) Figure 2 from [9]



(b) Cleaner version of Figure 2 from [9]

Figure 2.8: Visualizations of the skip connection: the bedrock of the ResNet.

[4]It's instructive to think for a moment about how shrinking solutions toward the identity can be more useful than shrinking them toward zero

Because the architectures of ResNets can be deeper, skinnier (less wide) networks can be employed, further increasing computational efficiency (compare e.g. VGG-19 with 128, 256 or 512 filters per layer to ResNet-34 with 64 or 128 filters per layer).[5] Due to the skip connections, during backpropagation the gradients now have a stronger effect at far-away layers (closer to input) because there is now a simple identity route $f(x) = x$ which the gradient can follow to skip the "computational"/connected blocks. Through the chain rule, such an identity skip connection implies the gradient picks up a factor of $f'(x) = 1$, hence no fear of exploding/vanishing gradients!

2.2.3 Application to CIFAR10

Training a vanilla CNN on CIFAR10 does not give extraordinary results (I was able to get $\sim 75\%$ validation accuracy with four convolutional layers). However, CNNs on MNIST can achieve well beyond 99% validation accuracy. Granted, CIFAR is a much trickier problem (color images, very grainy, less structured). One way to drastically[6] improve performance is through the use of residual connections. For this task, it is good to keep in mind that 95% accuracy is the level of human performance, so coming near this value would already be quite impressive! The code for this can be found here: `https://github.com/JacobHA/deep-learning/blob/master/Code/Lecture4-CIFAR10%2BResNets`.

2.3 Recurrent Neural Networks

2.3.1 Motivation

Thus far, our exploration of neural networks has revolved around mastering the task of learning functions that transform data into fixed-size outputs. Whether it's discerning intricate patterns in images through CNNs or extracting meaningful features from structured data using MLPs, our focus has centered on data conforming to consistent structures. Dor example, we cannot use the same CNN to pass in images of different sizes (we would have to resort to some pre-processing step to ensure all inputs have the same dimensionality).

However, the landscape shifts when we venture into Natural Language Processing (NLP), a domain where the variability in data length becomes a formidable challenge for which our current architectures are useless. Consider the task of sentiment analysis: determining whether a given sentence carries a positive or negative connotation. Herein lies the complexity; sentences can vary significantly in length.

Take, for instance, the sentences "Unfortunately it will rain." and "All students were generously gifted five dollars for attending." One is succinct, containing only a handful of words, while the other is much longer. If we conceptualize our deep learning model as operating on individual words, how do we construct a system capable of seamlessly processing both a concise 4-word sentence and a more elaborate 8-word (or 800-word) counterpart? More generally, how can a model (or general mathematical function) act on inputs of arbitrary dimension?[7]

This is precisely the problem which Recurrent Neural Networks (RNNs) aim to solve[8] RNNs are purpose-

[5]Using larger strides is now more common than max pooling because of this paper.

[6]Keep in mind that as accuracy $\to$ 100%, the difficulty in improving by the same percentage becomes asymptotically more difficult.

[7]Seems like an interesting mathematical question on its own right. Not sure if it is worth thinking further about theoretical consequences / understanding.

[8]Note that like our previous architectures (CNN, ResNet), new architectures are often posed to solve some open problem in a field. Luckily, they often extend their initial use-case and provide inspiration for further development.

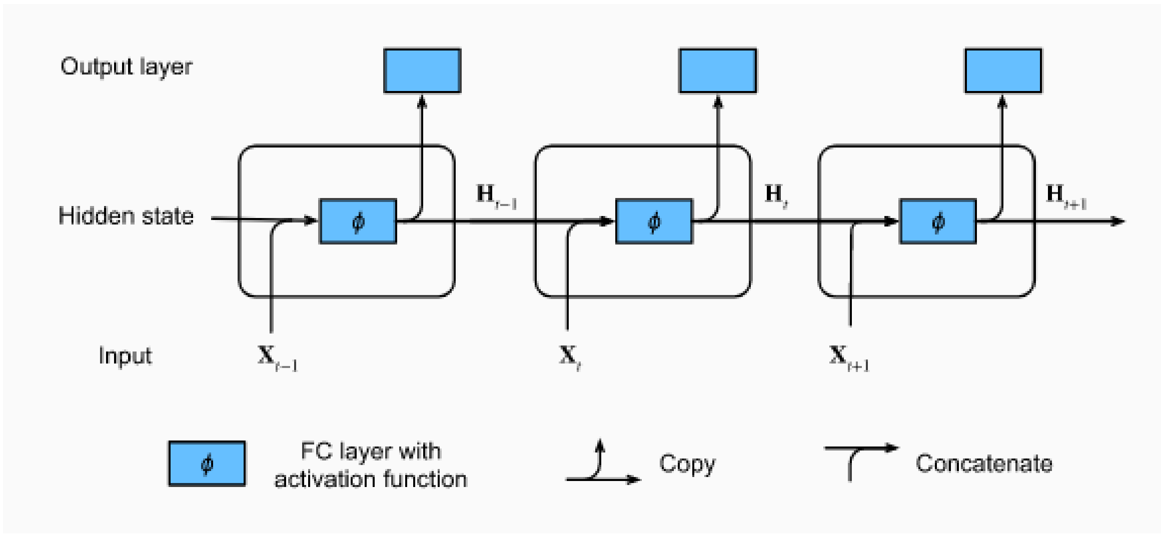


Figure 2.9: An "unrolled" version of the RNN architecture. Stolen from `http://d2l.ai/chapter_recurrent-neural-networks/rnn.html`

built for handling sequential data with varying lengths, making them particularly adept at navigating the dynamical nature of language. Diverging from our earlier models, RNNs harbor the unique ability to retain a memory of past inputs, rendering them exceptionally well-suited for tasks like sentiment analysis where the contextual understanding of words is paramount.

2.3.2 Key Insights

- Insight 1: Feed inputs sequentially. Maintain a hidden state encoding the memory so that inputs are shared across time.
- Insight 2: Keep fully-connected layer(s) frozen, so that training is straightforward

2.4 Long Short-Term Memory

The next step in the progression of sequence modeling is the advent of "memory". The RNN is unable to remember things over long time frames, as the input is continuously mixed into the hidden state at each time step. LSTM

While Recurrent Neural Networks (RNNs) brought significant advancements in handling sequential data, they suffer from a major limitation – the vanishing gradient problem. This issue hinders their ability to learn long-term dependencies within sequences. RNNs process information sequentially, with the hidden state (containing information from previous steps) feeding into the calculation of the current hidden state. However, as the sequence lengthens, the influence of the gradients from earlier steps diminishes exponentially, making it difficult for the network to learn long-range relationships.

Memory in LSTMs:

To address this limitation, Long Short-Term Memory (LSTM) networks were introduced. LSTMs incorporate a concept called cellular memory, allowing them to store information for extended periods and effectively learn long-term dependencies. Unlike RNNs with a single hidden state, LSTMs have an additional internal structure called a cell. This cell acts as a powerful memory unit, selectively remembering and updating information over time.

Forget Gate, Input Gate, and Output Gate:

The core functionality of the LSTM cell lies in three crucial gates: forget gate, input gate, and output gate. Each gate controls the flow of information within the cell, regulating what information is kept, what new information is added, and what information is ultimately exposed to the network.

1. **Forget Gate (f_t):** This gate determines what information from the previous cell state (C_{t-1}) should be forgotten. It takes the previous hidden state (h_{t-1}) and the current input (x_t) as inputs and outputs a vector of values between 0 and 1. A value close to 1 indicates that the corresponding information in the cell state should be retained, while a value close to 0 indicates it can be forgotten. Mathematically, the forget gate can be represented as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.3}$$

   where:

   σ is the sigmoid activation function, ensuring the output is between 0 and 1. W_f and b_f are weight and bias parameters specific to the forget gate.

2. **Input Gate (i_t):** This gate controls the flow of new information from the current input (x_t) and decides what information needs to be stored in the cell state. It also generates a candidate cell state ($\widetilde{C}_t$), which combines the new information with the previous hidden state. Similar to the forget gate, the input gate takes the previous hidden state and the current input as inputs. It outputs two vectors: a forget vector (i_t^f) and a candidate cell state vector ($\widetilde{C}_t$).

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.4}$$
$$\widetilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.5}$$

   where:

   W_i and b_i are weight and bias parameters for the input gate. W_c and b_c are weight and bias parameters for generating the candidate cell state. tanh is the hyperbolic tangent activation function.

3. **Output Gate (o_t):** This gate determines what information from the current cell state (C_t) is used to update the hidden state (h_t), which is ultimately exposed to the network. It considers both the current cell state and the previous hidden state. The output gate outputs a vector of values between 0 and 1, and a final hidden state vector (h_t).

$$o_t = \sigma(W_o \cdot [h_{t-1}, C_t] + b_o) \tag{2.6}$$
$$h_t = o_t \cdot \tanh(C_t) \tag{2.7}$$

where:

W_o and b_o are weight and bias parameters for the output gate, respectively.

Cell State Update: The core memory functionality of the LSTM lies in how these gates interact:

The forget gate decides what to forget from the previous cell state (C_{t-1}) using the forget vector (f_t). The input gate determines what new information to store in the cell state by creating a candidate cell state ($\widetilde{C}_t$). The new cell state (C_t) is formed by combining the information to be kept from the previous state ($f_t * C_{t-1}$) with the new information from the candidate state ($i_t * \widetilde{C}_t$). Mathematically: $C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$.

Finally, the output gate controls what information from the current cell state (C_t) is used to update the hidden state (h_t). The output gate uses the tanh activation function on the current cell state (C_t) and then multiplies it by the output gate vector (o_t) to create the final hidden state (h_t) that is exposed to the network. $h_t = o_t * \tanh(C_t)$

By selectively remembering and forgetting information through these gates, LSTMs can effectively capture long-term dependencies within sequences, overcoming the limitations of RNNs.

Original LSTM: [10].

A cool application of LSTMs to music generation: [5].

2.4.1 GRU

2.5 Issues with Recurrent Networks

The main advantage of RNNs is also their biggest crux. The biggest issue for RNNs is their requirement to process input data one step at a time, which means that computations cannot be done in parallel, and sequential processing must be done until completion of any given input. Thus, there was a need for models capable of *simultaneously* processing entire strings (or vectors, etc.) of variable length. One naïve way to fix this issue is to store an input sequence (of arbitrary length) into a fixed-length abstract "context vector". Then, the entire sequence is represented by a vector of fixed length and can be processed in full (i.e. the entire context / sequence enters the model simultaneously) as in MLPs or CNNs.

Unfortunately, the clear drawback here is that the size of the context vector is now a bottleneck for larger input sequences. As an example, imagine a context vector which can store 1000 bits of information. Now imagine encoding a sentence of 10 tokens[9] into this context vector. Then the context vector is giving roughly 100 bits / token of information to the input sequence. However, if an input sequence happens to have length 40, then suddently the input sequence now has to be compressed to 25 bits / token to fit into the same context length. Thus, this bottleneck arises because fixed-length context vectors may not be able to adequately capture all the information present in longer input sequences.

Moreover, typical tokenization and projection into a fixed-size context length vector leads to loss of temporal order (cf. "bag of words" models). For instance, if we just have a weight matrix that translates tokens to the context vector, then the sentence "The dog chases the cat" will have the same context vector as the sentence "The cat chases the dog", or even "cat dog chases The the", all of which clearly have different meaning (More information about this Word2Vec translation problem here: `https://engineering.purdue.edu/DeepLearn/`

[9]Recall that **tokens** are the result of tokenization of our input vocabulary of characters/word fragments. An in-depth introduction to "tokenization" is provided by Andrej Karpathy here: `https://www.youtube.com/watch?v=zduSFxRajkE`

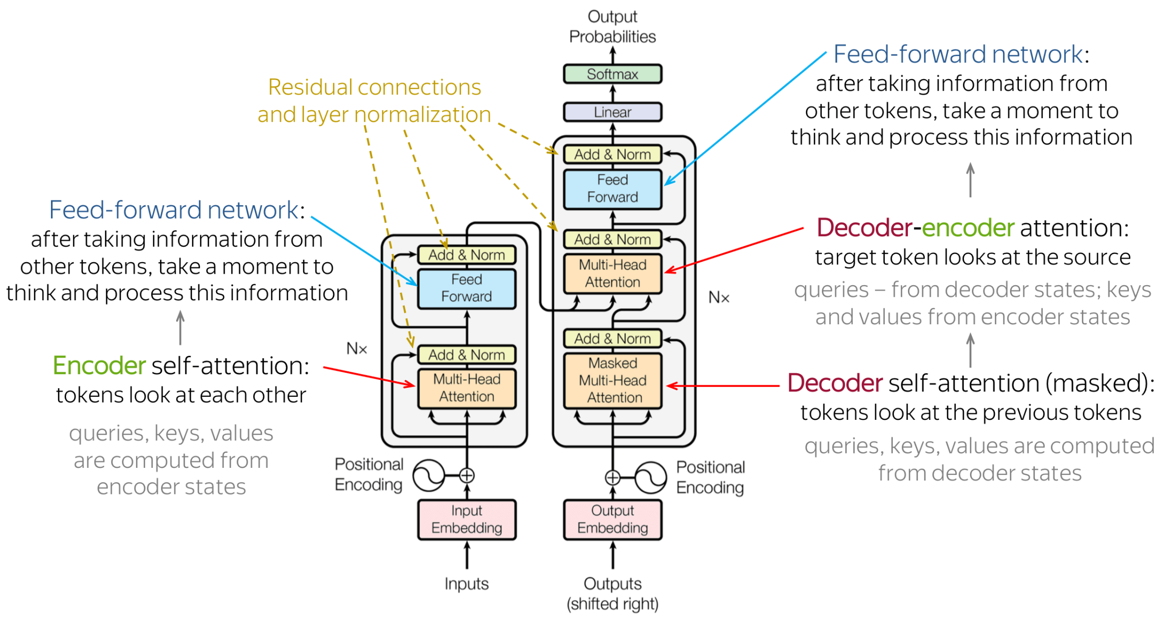

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)
Residual connections and layer normalization
Feed-forward network: after taking information from other tokens, take a moment to think and process this information
Encoder self-attention: tokens look at each other
queries, keys, values are computed from encoder states
Feed-forward network: after taking information from other tokens, take a moment to think and process this information
Decoder-encoder attention: target token looks at the source
queries – from decoder states; keys and values from encoder states
Decoder self-attention (masked): tokens look at the previous tokens
queries, keys, values are computed from decoder states

Figure 2.10: Caption

pdf-kak/Seq2Seq.pdf). To prevent this issue, the concept of "Positional Encoding" has been introduced in the Transformer model (see Section 2.6.1).

With positional encoding of input sequences, a novel functional layer, "attention", can act simultaneously on the distinct positionally-encoded word embeddings to "attend" between arbitrary token pairs. Repeating this at multiple levels (multi-head attention) allows the decoder to attend to many different pairs of tokens simultaneously when predicting the next output token.

Along with some other tricks, these are the key features that make the Transformer model so powerful as a successor to the recurrent architectures we have just studied.

2.6 Transformers

The transformer, depicted below in Figure 2.10, is (one of) the most recent architectures to continue in the direction of generalization and SOTA, especially in Natural Language Processing (NLP) tasks. Its inception can be traced back to the seminal work of Bahdanau et al. [1], where the concept of attention mechanisms began to take shape. However, it was truly popularized and revolutionized by Vaswani et al. [16] with their landmark paper titled "Attention is All You Need". Since then, transformers have become the cornerstone of modern NLP, showcasing unparalleled performance across a multitude of tasks. This surge in popularity is largely attributable to their ability to effectively capture long-range dependencies and relationships within sequential data, thereby surpassing previous architectures in both efficiency and effectiveness. With their capacity for parallelization and scalability, transformers have not only set new standards for state-of-the-art performance but also paved the way for advancements in various other fields beyond NLP. In this context, exploring the underlying principles and mechanisms driving transformers becomes imperative to fully harness their potential and propel the frontier of AI research and application.

2.6.1 Positional Encoding

In traditional natural language processing (NLP) models, such as recurrent neural networks (RNNs), the sequential order of tokens in the input sequence is inherently captured through the recurrence mechanism. However, in Transformer architectures, which do not inherently possess such sequential processing, a mechanism is needed to explicitly encode the position of tokens within the input sequence. This is crucial for preserving the sequential information and enabling the model to differentiate between tokens based on their positions.

In tasks where the order of tokens carries significant semantic meaning, such as language modeling or machine translation, it is essential for the model to be able to distinguish between tokens based on their positions in the sequence. Failure to do so can lead to a loss of important temporal information, ultimately degrading the model's performance on such tasks.

To address this challenge, the Transformer model introduces positional encoding as a way to inject positional information into the input embeddings. Positional encoding is added to the input embeddings before they are fed into the Transformer encoder or decoder. This enables the model to encode not only the identity of each token but also its position within the sequence.

The positional encoding is typically represented as a matrix that contains sinusoidal functions of different frequencies. The dimensions of this matrix correspond to the dimensions of the input embeddings. Each row of the matrix corresponds to a position in the sequence, and each column corresponds to a dimension in the input embeddings. The values in the matrix are computed based on sinusoidal functions of different frequencies and phases.

Mathematically, the positional encoding for a token at position *pos* and dimension i can be represented as follows:

$$PE_{(pos,i)} = \begin{cases} \sin(pos/10000^{2i/d_{\text{model}}}), & \text{if } i \text{ is even} \\ \cos(pos/10000^{(2i/d_{\text{model}})}), & \text{if } i \text{ is odd} \end{cases}$$

where *pos* is the position of the token in the sequence, i is the dimension of the input embeddings, and d_{model} is the dimensionality of the model.

The choice of sinusoidal functions in positional encoding allows the model to learn to attend to different positions in the sequence based on their relative frequencies. Tokens at different positions will have different representations in the positional encoding matrix, enabling the model to differentiate between them based on their positions. The intuition here is that a different sine wave (wave across vector index) is added to each vector representation of an input token, at position "pos" in the sequence. Of course, the 10000 comes into question. What happens when index $i > 10000$? My own intuition breaks down, and we will probably need new models capable of handling much larger sequence lengths (e.g. whatever the newest Gemini model is doing, which doesn't yet have any public information released).

In conclusion, positional encoding plays a crucial role in Transformer architectures by enabling the model to encode positional information into the input embeddings. This allows the model to preserve the sequential order of tokens in the input sequence, thereby enabling effective processing of sequential data in tasks such as language modeling, machine translation, and sequence generation.

2.6.2 Attention

The attention mechanism is at the heart of the transformer network. Let's spend some time looking at examples of how attention works to digest it fully.

While Recurrent Neural Networks (RNNs) have been the workhorse for many sequence-based tasks, they struggle with capturing long-range dependencies within a sequence. This is where Transformers come in, a powerful deep learning architecture that relies heavily on an ingenious mechanism called attention. Unlike RNNs that process information sequentially, Transformers leverage attention to understand the relationships between all elements in a sequence simultaneously. This section dives into the core concept of attention and how it is used in Transformers.

Imagine you're reading a complex sentence. To understand the meaning fully, you don't just read words sequentially; you focus on specific words based on the context of the sentence. Attention works similarly. It allows the model to focus on specific parts of an input sequence (like words in a sentence) that are most relevant to understanding the current element being processed.

Before breaking down the attention mechanism, let's first walk through the "classical" task of querying a database. In a database, keys (such as names) are associated to values (such as addresses). When we want to access certain element(s) of a big database, we will offer a query such as "starts with letter J and last name Adamczyk". Then, the query will be checked against the keys of the database. Finally, the corresponding value(s) will be output to the user that requested a certain query. The attention mechanism is a way of doing this more "softly" or probabilistically.

Here's a breakdown of the key components in the attention mechanism:

- **Query (Q), Key (K), and Value (V) Vectors:** The input sequence is projected into three different vector spaces: query (Q), key (K), and value (V). Intuitively, the query vector can be thought of as a search query. It represents the specific information the model is looking for at a particular point in the sequence. The key vector, on the other hand, captures the essence of each element in the sequence, acting like a description tag that summarizes the element's content. Finally, the value vector holds the actual information associated with each element. By comparing the query vector to the key vectors of all elements, the model can identify which elements are most likely to contain the information it needs. The value vectors of these relevant elements are then used to create a context vector that provides a focused summary of the parts of the sequence that are most important for understanding the current element.

- **Attention Scores**: The model calculates a score for each element in the sequence, indicating how relevant it is to the current query. This score is typically computed[10] using a scaled dot-product operation between the query vector (Q_i) and each key vector (K_j) in the sequence:

  $$\text{Attention Score}(i, j) = \frac{Q_i \cdot K_j}{\sqrt{d_k}} \tag{2.8}$$

  where d_k is the dimension of the key vectors (used for numerical stability). This score essentially measures how well the "key" (content of an element) matches the current "query" (focus of the model).

- **Softmax Function**: The attention scores are then passed through a softmax function. This function transforms the scores into a probability distribution, ensuring that the sum of attention scores across all

[10]though there are other similarity scores, such as cosine similarity, or unscaled dot products

elements equals 1. Each element now has a weight between 0 and 1, representing its relative importance to the current query.

- **Weighted Sum:** Finally, the model computes a weighted sum of the value vectors (V) using the attention scores as weights. This effectively creates a context vector that summarizes the most relevant parts of the sequence based on the current query.

  Mathematically, the attention output (O_i) for a specific position i in the sequence can be represented as:

$$O_i = \sum_j \text{softmax}(\text{Attention Scores}(i, j)) * V_j \tag{2.9}$$

  Note that these softmax'ed attention scores (which now represent a probability distribution) is typically denoted $\alpha_{ij} = \text{softmax}(\text{Attention Scores}(i, j))$

Benefits of Attention

Unlike RNNs, attention allows the model to directly attend to any element in the sequence, irrespective of its distance. This is crucial for capturing long-range dependencies that are essential in tasks like machine translation and question answering.

Attention calculations can be performed for all elements in the sequence simultaneously, making Transformers much faster to train compared to sequential models like RNNs.

The core attention mechanism described above is called *self-attention.* It focuses on relationships within a single sequence. Transformers typically employ a more advanced concept called **multi-head attention**. This involves creating multiple independent attention "heads" that learn different aspects of the relationships within the sequence. The final output is then the concatenation of the outputs from all these heads, allowing the model to capture a richer understanding of the sequence.

The attention mechanism is a fundamental building block of Transformers, enabling them to excel in various NLP tasks. By focusing on relevant parts of a sequence, attention allows Transformers to capture complex relationships and achieve state-of-the-art performance in numerous applications.

- Some useful references:
  - `https://www.jeremyjordan.me/attention/`
  - `https://jalammar.github.io/illustrated-transformer/`
  - `https://nlp.seas.harvard.edu/annotated-transformer/`
  - A deeper mathematical discussion: `https://transformer-circuits.pub/2021/framework/index.html`

Chapter 3

Applications

with any new architecture must devise experiments

Dependence of final performance on:

- Dimension, structure (topology)
- activation function (at each node/layer)
- pooling methods (max mean etc)
- filter size
- data set (difficult/expensive)
- training parameters (hyperam hell)
- Loss function

3.0.1 Computer Vision

3.1 U Nets

3.1.1 Physics

3.1.2 Biology

3.1.3 Mathematics

Chapter 4

Reinforcement Learning

4.0.1 MDPs

4.0.2 Bellman Equation

4.0.3 Examples

Chapter 5

Advanced Theory

5.0.1 Classical Learning Theory

Universal Function Approximation

Concentration Inequalities

PAC Learning

5.0.2 Sample Complexity in RL

5.0.3 Deep Nets as Field Theories

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[2] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control.* Cambridge University Press, 2022.

[3] Sydney Cash and Rafael Yuste. “Linear summation of excitatory inputs by CA1 pyramidal neurons”. In: *Neuron* 22.2 (1999), pp. 383–394.

[4] Xiangning Chen et al. “Symbolic discovery of optimization algorithms”. In: *arXiv preprint arXiv:2302.06675* (2023).

[5] Michael Conner et al. “Music generation using an lstm”. In: *arXiv preprint arXiv:2203.12105* (2022).

[6] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[9] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[10] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[11] David H Hubel and Torsten N Wiesel. “Receptive fields of single neurones in the cat’s striate cortex”. In: *The Journal of physiology* 148.3 (1959), p. 574.

[12] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[13] P Russel Norvig and S Artificial Intelligence. “A modern approach”. In: *Prentice Hall Upper Saddle River, NJ, USA: Rani, M., Nayak, R., & Vyas, OP (2015). An ontology-based adaptive personalized e-learning system, assisted by software agents on cloud storage. Knowledge-Based Systems* 90 (2002), pp. 33–48.

[14] Daniel A Roberts, Sho Yaida, and Boris Hanin. *The principles of deep learning theory.* Cambridge University Press Cambridge, MA, USA, 2022.

[15] Hidenori Tanaka and Daniel Kunin. “Noether’s learning dynamics: Role of symmetry breaking in neural networks”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 25646–25660.

[16] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

Andrej Karpathy’s YouTube Channel Lex Fridman MIT Lectures