Deep Learning Lecture Notes

J. Adamczyk

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

February 6, 2024

Contents

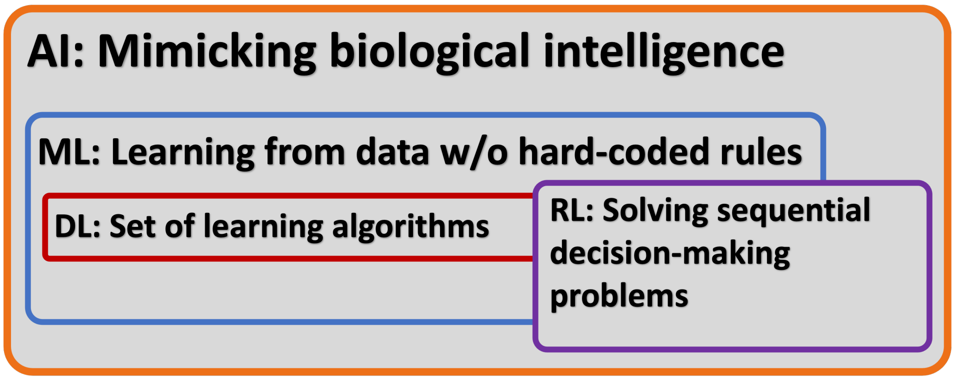

AI: Mimicking biological intelligence
ML: Learning from data w/o hard-coded rules
DL: Set of learning algorithms
RL: Solving sequential decision-making problems

Figure 1: Relationship between the fields of AI, ML, DL, and RL.

1 Introduction

1.1 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 4) 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 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 [8, 5, 1].

1.2 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:

"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.

2 History

TODO: add citations to all of these.

2.1 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.

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

- 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

2.2 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).

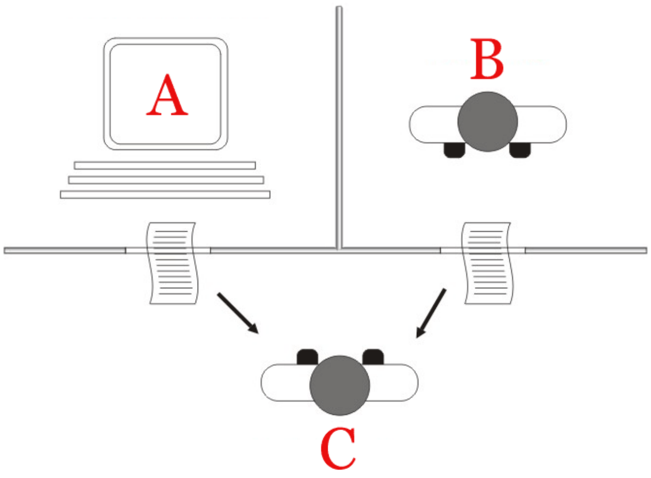


Figure 2: Stolen from wiki. Schematic setup of the Turing test. Interrogator is C.

2.2.1 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.

2.3 Functions

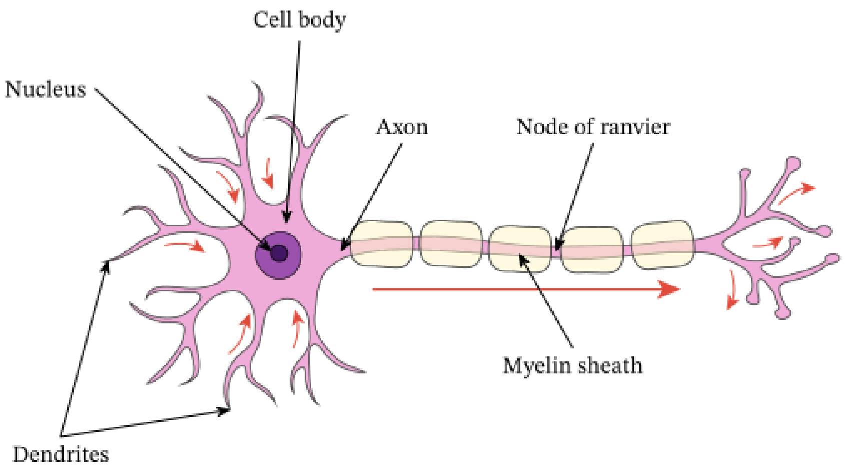
As discussed above, the main idea of machine learning is to have a program learn a particular[3] mathematical function. This being said, we must consider how to encode an arbitrary mathematical function. In the language of linear algebra, if we want a map from $\mathbb{R}^n \to \mathbb{R}^m$, then we 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...)

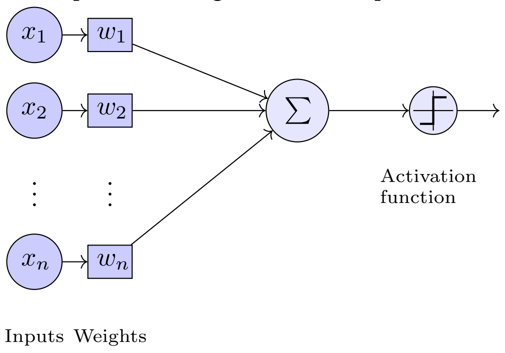
3 Simple Architectures

Motivated from the structure of brain cells, as seen in Fig. 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`

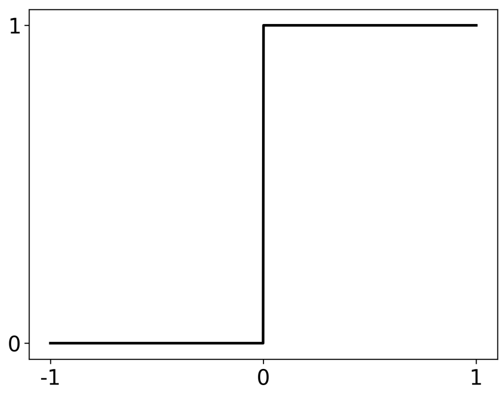


Figure 4: A simple activation function, the Heaviside Function.

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. 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 $\{0, 1\}$ 4.

3.1 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:

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

where Θ is the Heaviside step function – the simplest possible "activation" or "threshold" function, shown in Fig. 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) 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
```

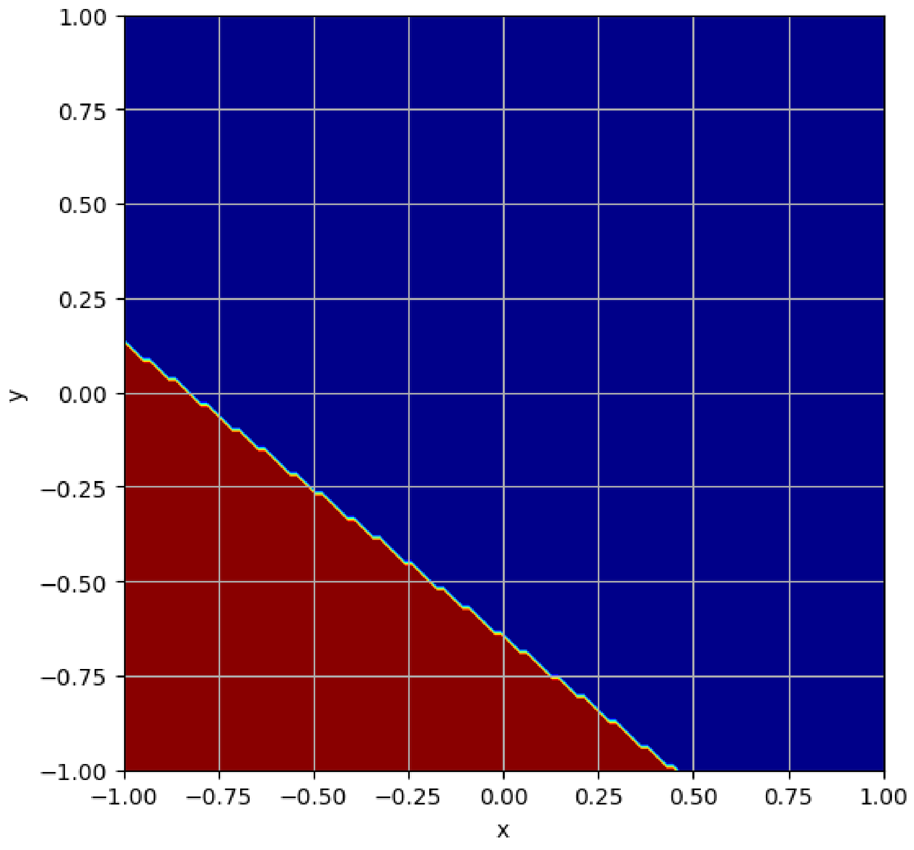
```
[ ]: torch.Size([2, 1])
```

```
[ ]: 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')
```

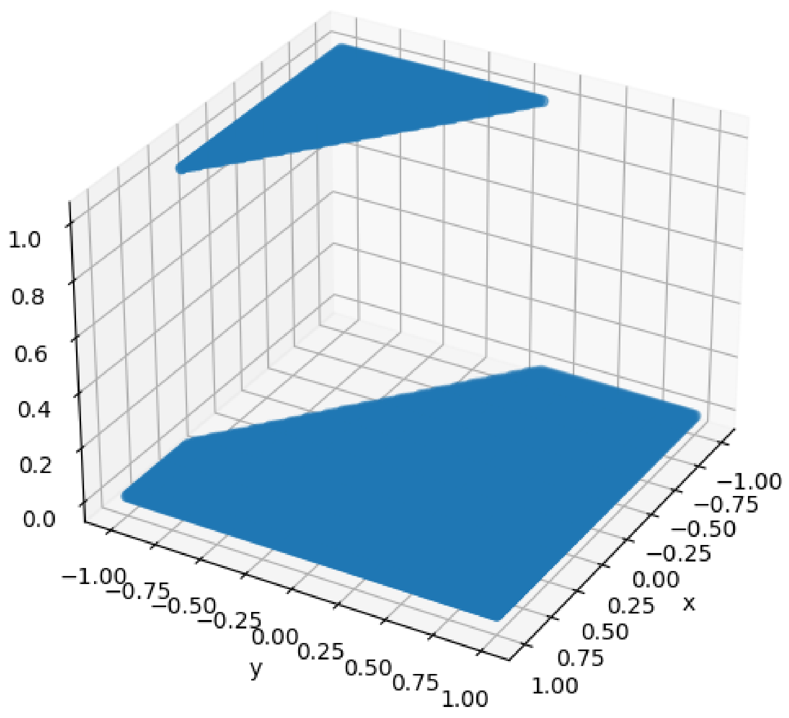
```
[ ]: Text(0, 0.5, 'y')
```

1



[ ]:
```python
# 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()
```

2



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

3

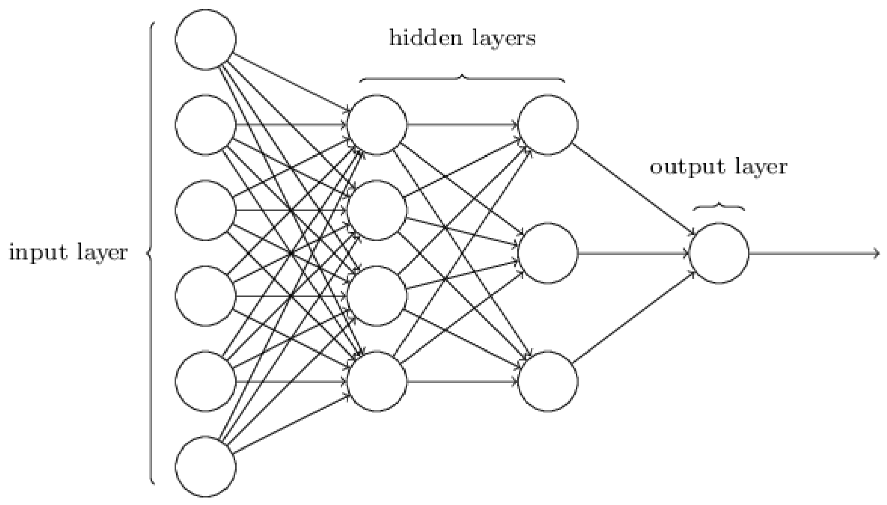


Figure 5: stolen from https://github.com/rcassani/mlp-example

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

3.2 Multi-Layer Perceptron

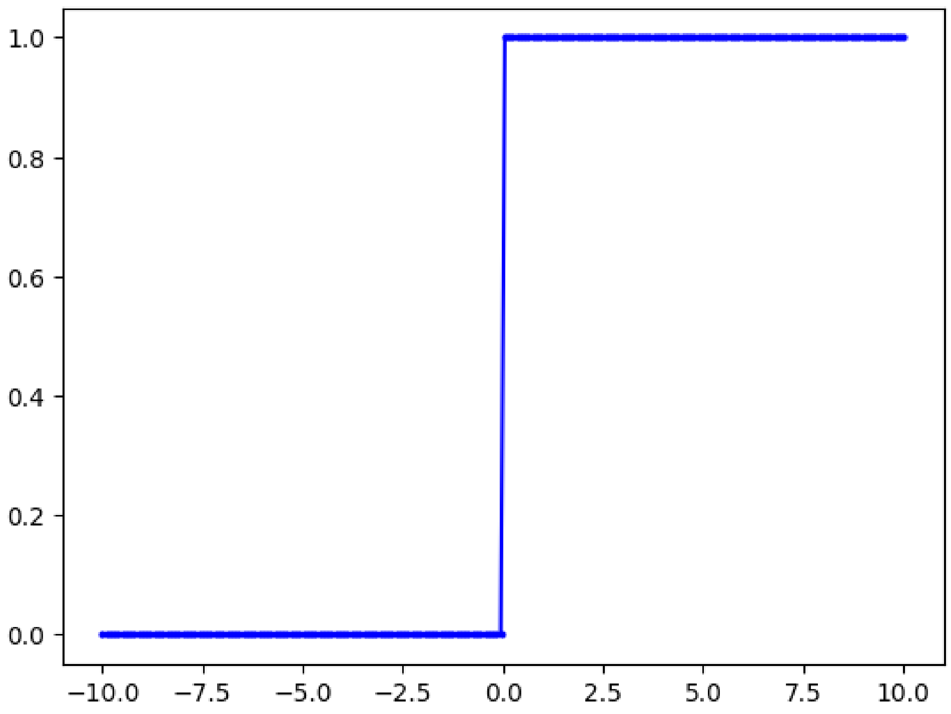
We want to build a more sophisticated version of the Perceptron which is hopefully more useful. Ideally, we'd like a network which can be trained fast, and is universal (we will make this statement precise later) – essentially meaning that we should be able to approximate **any** (sufficiently well-behaved) 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])
```

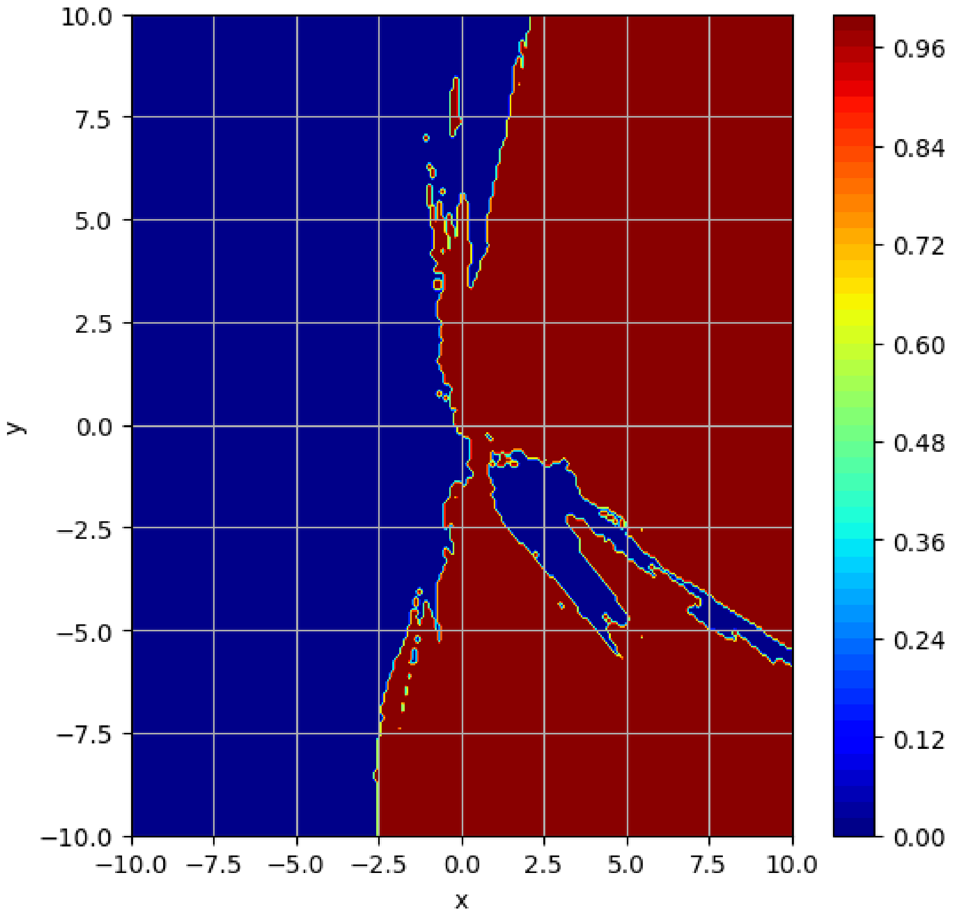
[ ]:
```
# Define weights for second layer:
w_layer2 = torch.randn(hidden_dim, 1)
# Define biases:
b_layer2 = torch.randn(1)
```

[ ]:
```
# 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)
```

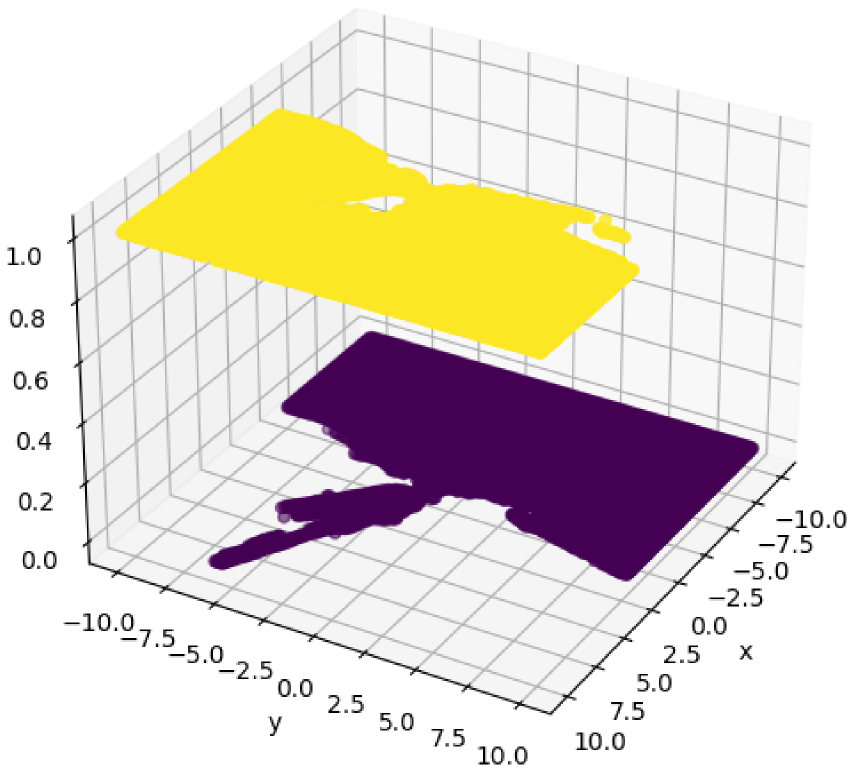
[ ]:
```
plt.figure(figsize=(6, 6))
z = out_layer2
# Plot xy, and z as a color:
# plt.scatter(points[:, 0], points[:, 1], c=z[:, 0])
# use interpolation:
plt.contourf(X, Y, z.reshape(X.shape), 50, cmap='jet')

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)
```



```
[ ]:
```

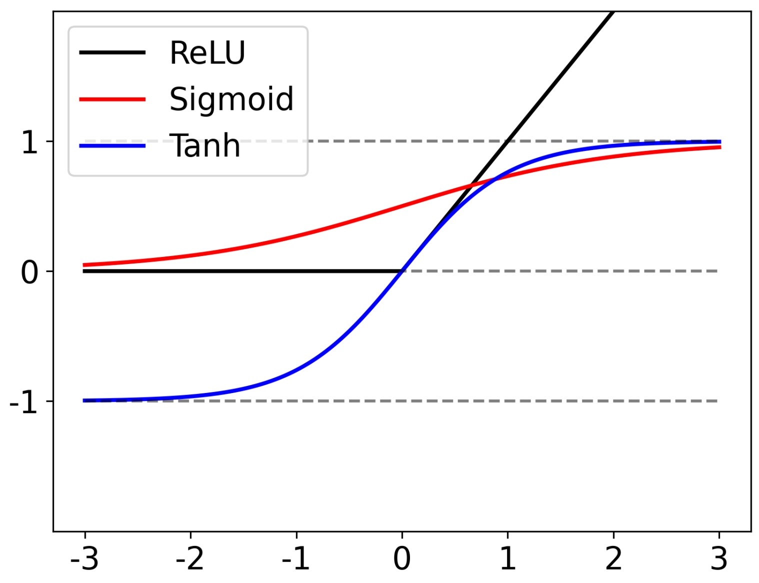


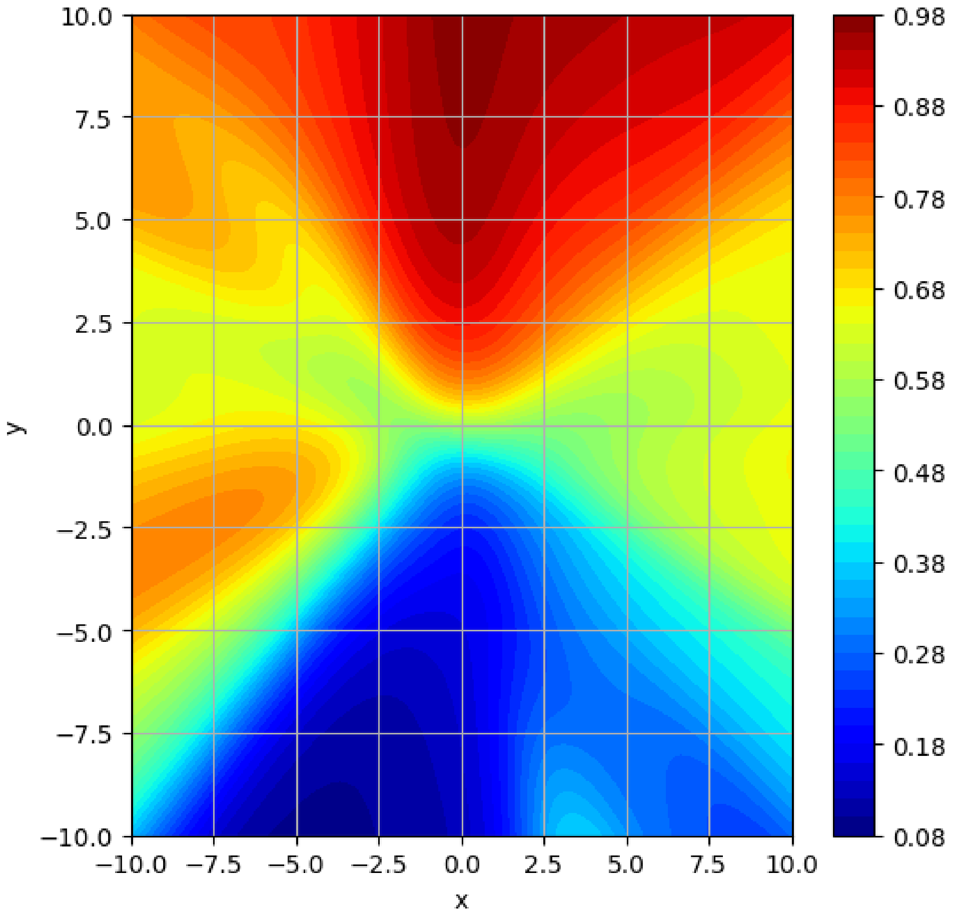
Figure 6: Commonly used activation functions. These generalize the step function in Fig. 4 by introducing nonlinearities with non-trivial derivatives. Note that some activation functions (e.g. Tanh and ReLU) yield bounded outputs. Thus, one must choose wisely the activation at the output layer so that 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. 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?)

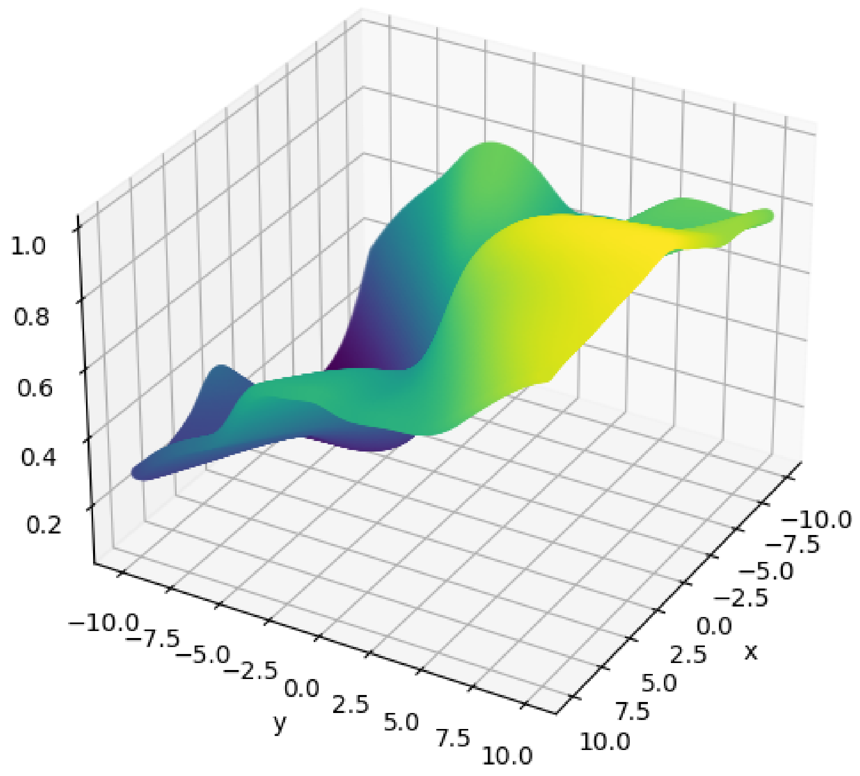
```
# use interpolation:
plt.contourf(X, Y, z.reshape(X.shape), 50, cmap='jet')

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)
```



```
[ ]:
```

3.2.1 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.

3.2.2 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,

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

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.

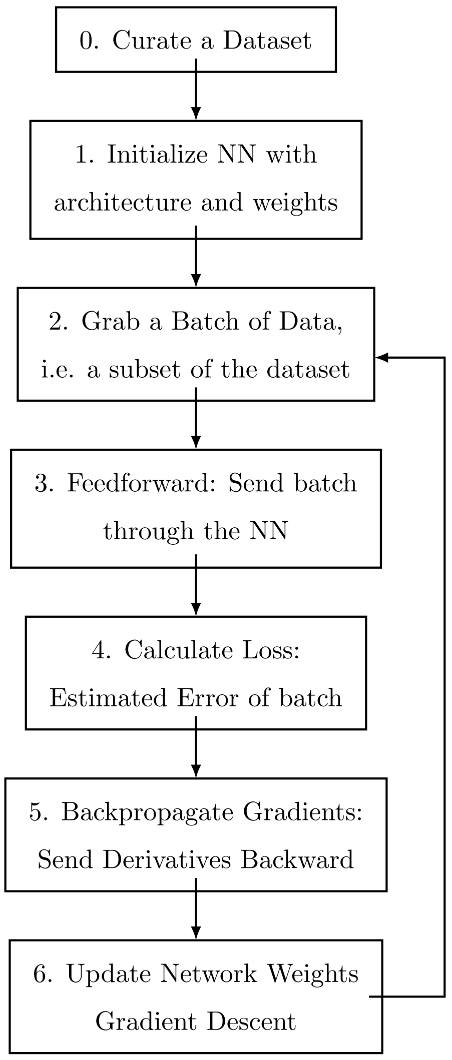
Nevertheless, this is an important theoretical guarantee (there are not very many in deep learning as of today). This allows us to (somewhat, based on caveats above) confidently use MLPs as an architecture to solve our problems of interest.

4 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.

4.1 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.

4.2 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? [4] 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.

4.2.1 Symmetry Breaking

There has been a considerable amount of theoretical analysis in recent years studying how to initialize deep neural networks [9]. [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 **??**), 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.

4.2.2 Distributions

A popular distribution from which to draw the initial weights is defined by the "Xavier Glorot initialization" [3]: $\theta^{(l)} \sim \mathcal{N}\left(0, \sqrt{2/(n_l + n_{l+1})}\right)$ where n_l is the number of neurons in layer l. [8]

4.2.3 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.

4.3 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: [10]

[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 [9].

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.

4.4 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.

$$f_\theta(x) = \sigma(\mathbb{W}^{(2)}\sigma(\mathbb{W}^{(1)}x + b^{(1)}) + b^{(2)}) \tag{3}$$

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).

4.5 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:

$$\forall i: \; x_i > 0, \tag{4}$$

$$\sum_{i=1}^{C} x_i = 1. \tag{5}$$

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. (5).

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

$$\text{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{i=1}^{C} e^{x_i}} \tag{6}$$

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. (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.

4.5.1 Cross Entropy Loss

Cross Entropy Loss for binary classification:

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

4.5.2 Mean Squared Error (MSE) Loss

MSE Loss for regression:

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

4.5.3 Mean Absolute Error (MAE) Loss

MAE Loss for regression:

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

4.6 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.

4.6.1 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 $f(g(\vec{x}))$

$$\frac{d}{dx}f(g(x)) = \frac{df}{dg}\frac{dg}{dx} \tag{7}$$

4.7 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.

4.7.1 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.

4.7.2 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.

4.7.3 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).

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{8}$$

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{9}$$

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{10}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{11}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{12}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{13}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{14}$$

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}}, \tag{15}$$

4.8 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:

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta \mathcal{L}(\mathcal{D}) \tag{16}$$

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]

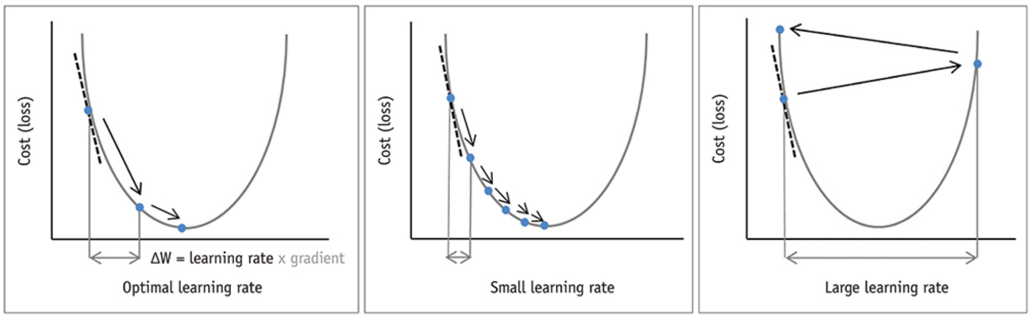


Figure 7: Gradient descent for various learning rates. Stolen from `https://analyticsindiamag.com/how-to-use-learning-rate-annealing-with-neural-networks/`

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].

4.8.1 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. [**<empty citation>**]

4.9 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. (16):

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta \mathcal{L} \tag{17}$$

4.10 Adam: Adaptive Momentum

saddle points are more likely than local minima in high dimension (B/c it would require all 20,000 dimensions to be concave up. if even one is concave down, then we get saddle)

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]

[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.

[16]As of writing, the Adam paper [7] has over 160k citations.

Momentum allows the optimization algorithm to skim past "historically" steep directions to more quickly find optima and escape barren plateaus.

$$\vec{\nu}_t = \beta_1 \vec{\nu}_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L} \tag{18}$$

$$\vec{s}_t = \beta_2 \vec{s}_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L})^2 \tag{19}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\vec{\eta}_t}{\sqrt{\vec{s}_t} + \epsilon} \nabla_\theta \mathcal{L} \tag{20}$$

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. (20) are performed element-wise. (I emphasize the vectorial nature to show from where the "parameter-wise" comes)

4.11 Newer methods

A new method, LION [**lion**], 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.

4.12 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.

5 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.

Then, we'll study Recurrent Neural Networks, Long Short-Term Memory, and Transformers as solutions for sequential (time-ordered) problems.

5.1 CNN

5.2 Convolutional Neural Nets

5.2.1 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:

$$196608 * 196608 + 196608 * 196608 + 196608 + 196608 = 77309804544 = 77B \tag{21}$$

where the last two additions are for the biases on each neuron. So a total of 77 billion parameters... 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

$$65536 * 65536 + 65536 * 65536 + 65536 + 65536 = 8.6B \tag{22}$$

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).

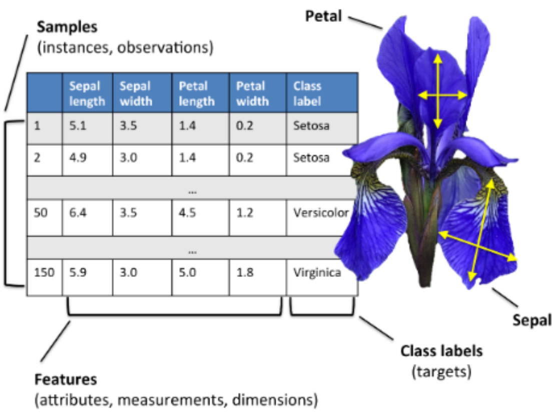
Can we do any better without down-sampling / decreasing the model complexity (e.g. we could just use a shallow network on a tiny image)?

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! Examples shown in Figure 9

This 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: For inspiration, let's turn to how a biological neural network (brain) processes and perceive images? A groundbreaking study by Hubel & Wiesel [6] in the 1950s discovered that some neurons positively respond to lines in the optical field of view. Specifically, lines at different angles and positions led to different neurons being activated. This work led to their awarding of the Nobel prize in 1981 for physiology and medicine.

'99 "Linear summation of excitatory inputs by CA1 pyramidal neurons" Neuron (hippocampus, memory)


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 8: Sepal/petal feature extraction

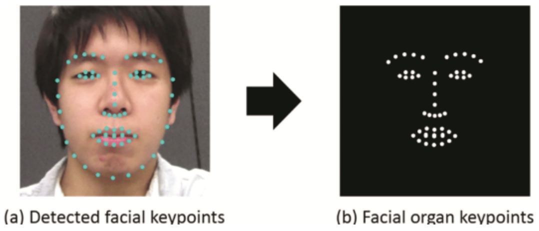


Figure 9: taken from `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L13_intro-cnn__slides.pdf`

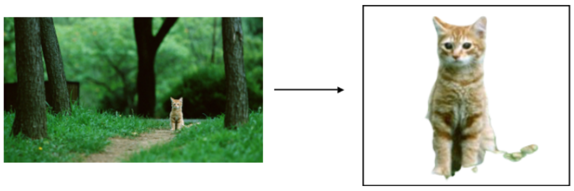


Figure 10: Manual pre-processing

[2] Can learn to distinguish linearly-separable features

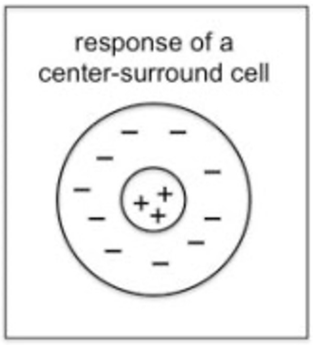


Figure 11: stolen from

(ex input graph)

5.2.2 Filters

The main idea of automatically learned features are two-dimensional[17]**filters**. A filter is a single feature, which can be thought of as a miniature image, which will be slid across the input image. If the filter "lines up" with the input image, 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 image (maybe a line or circle, to be more concrete). When writing your model, you can choose the number of "convolutional filters" in a given layer.

5.3 RNN

5.4 LSTM

5.5 Transformers

6 Real-World Examples

In applications (Next week) we will see how CNNs are used in a wide variety of scientific 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)

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

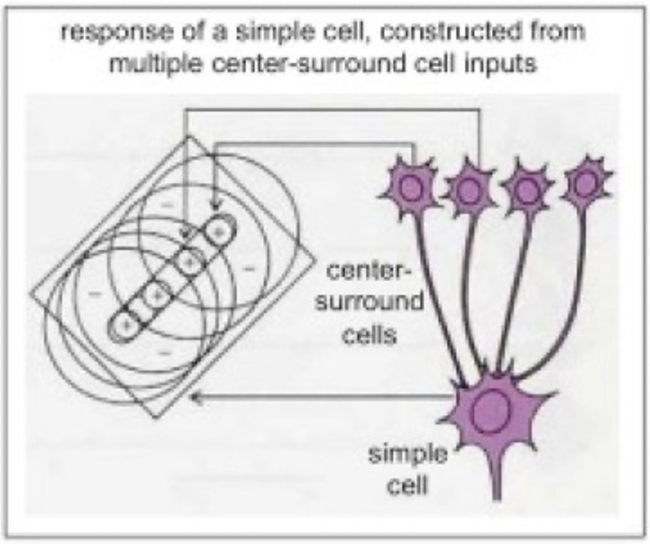

response of a simple cell, constructed from multiple center-surround cell inputs
center-surround cells
simple cell

Figure 12: stolen from

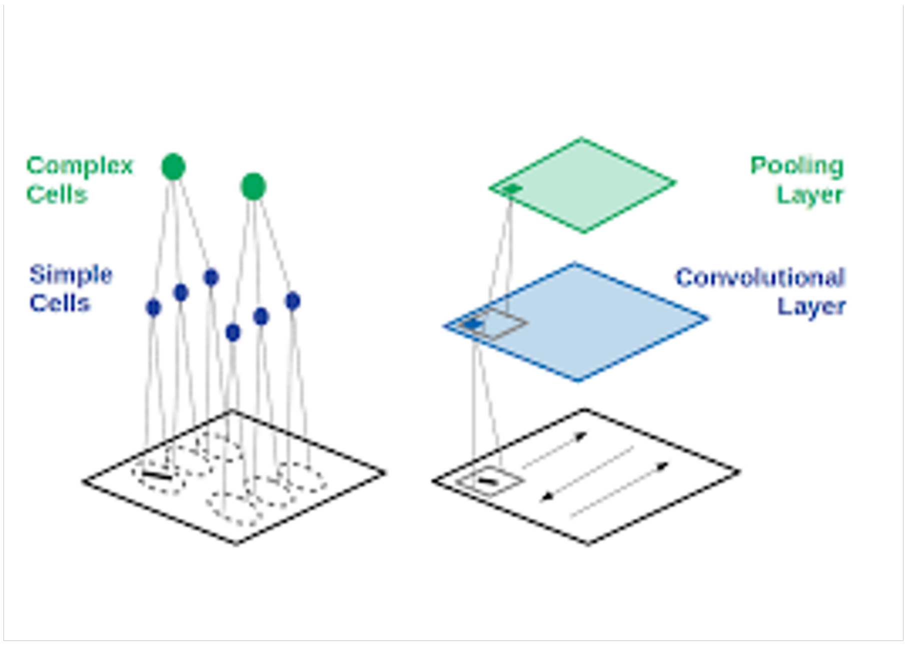


Figure 13: stolen from

- Loss function

6.1 Computer Vision

6.2 Physics

6.3 Biology

6.4 Mathematics

7 Reinforcement Learning

7.1 MDPs

7.2 Bellman Equation

7.3 Examples

8 Advanced Theory

8.1 Classical Learning Theory

8.1.1 Universal Function Approximation

8.2 Concentration Inequalities

8.3 Kakade RL / Complexity Bounds

8.4 Deep Nets as Field Theories

References

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

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

[10] 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.

Andrej Karpathy's YouTube Channel Lex Fridman MIT Lectures