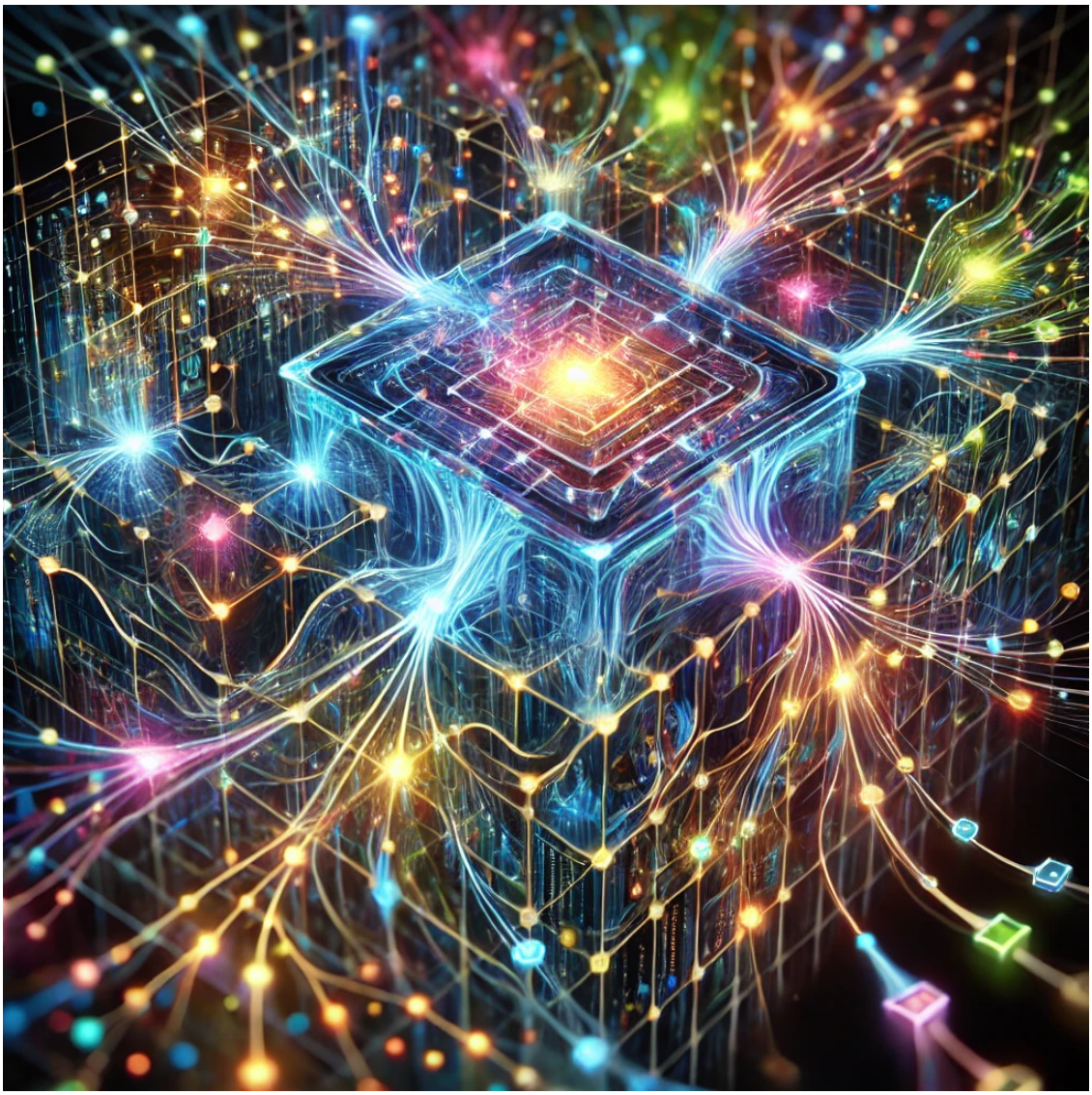
Neural Networks

University at Illinois of Chicago



Virginia Tasso

A.A. 2024-25

Contents

1 Lecture 1

Why Study Neural Networks?

- The human brain is very successful at certain tasks that prove to be difficult to accomplish on a classical computer, such as learning and pattern recognition. For example, consider the generic classification task: when you see a picture of an apple, you immediately recognize it as an apple. Similarly, you understand what someone says when they speak. Computers, however, often struggle with these tasks. For instance, even with modern technology, most commercial speech recognition systems cannot perfectly understand speech, especially with accents. This motivates us to study how the brain, or the human neural network, functions.

- Why is the human brain better at certain tasks compared to computers? Could it be about speed? No, because while a transistor can switch in nanoseconds, it takes a few milliseconds for your brain to process a stimulus, such as pricking your leg with a needle. Hence, the difference is not necessarily due to speed. What the brain lacks in speed, it compensates for in the number of processing elements and the massively interconnected and parallel nature of its operations, unlike the serial operations of classical computers. Additionally, the human brain is adaptive; its functionality evolves through learning, whereas the physical structure of a CPU in a computer is fixed and does not change over time.

Summary:

- **Classical computer:** Almost operates in serial (e.g., running a program while ignoring pipelining), but each operation can be performed extremely fast (less than nanoseconds per operation).

- **Neural network:** Operates massively in parallel, and may be more powerful than a classical computer, despite the fact that a given neuron is relatively "slow" (a neuron can fire at most hundreds of times per second, which translates to milliseconds per operation).

- **Motivation:** By imitating the brain's operation (through artificial neural networks), we may be able to solve complex tasks that classical computers struggle to perform. This is arguably the most important motivation for advancements in this field.

- **Other motivation:** To understand how the brain and nervous system function.

Neural Networks Summary:

1. Neural networks are massively parallel distributed processors composed of simple processing units (mathematical neurons) and the connections between them.

2. Neural networks can store and utilize experiential knowledge through a learning process.

An artificial neural network resembles the brain in two key respects:

- Knowledge is acquired by the network from its environment through a learning process.
- Inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

2 Lecture 2

Neural Networks are characterized by **non linearity** and by a **learning component**.

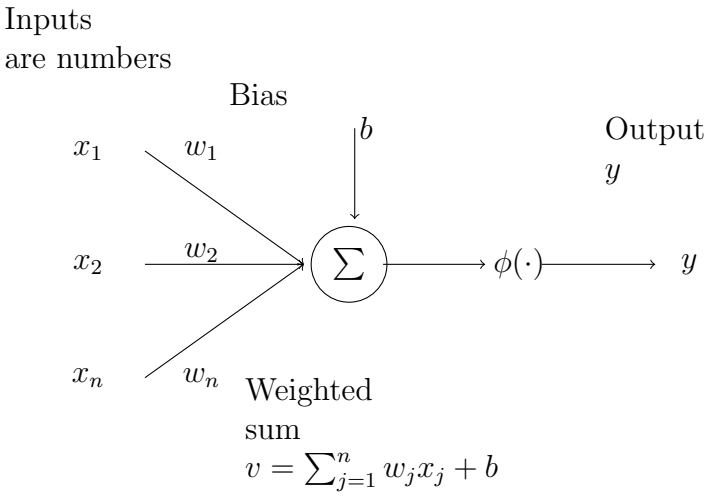
It is useful to understand what a **biological neuron** is, since Neural Networks get inspiration from them. To make it simple: some excitation comes in and the neuron may be activated, if there is enough excitation. The more it is used, the stronger it becomes (**Hebbian Algorithm**).

- n neurons in the brain: 10^{11}
- n neurons in the whole nervous system: 10^{12}
- n synapses: 10^{14}

In our body there are different types of neurons: sensory, internal, motor... and there are special architectures that adapt to different tasks.

Ex. when we born our brain is not fine tuned yet, we must give input data to it $\rightarrow$ **language learning**

2.1 Mathematical Model of the Neuron



The inputs are pulses, chemical perturbations (in biology). In our context, they are basically numbers.

The inputs are going to be weighted. They don't have all the same importance. **Weights** represent the strength of the connection (synaptic weight).

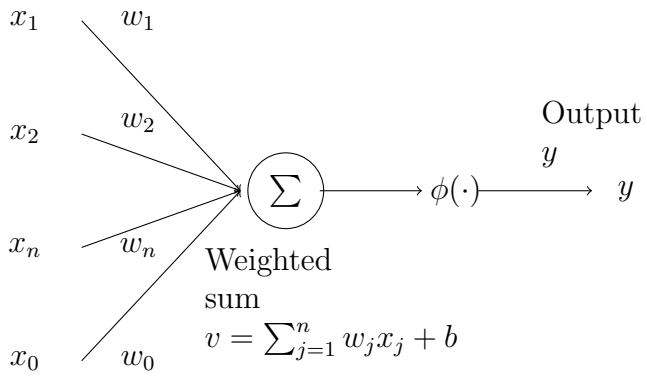
The **activation function** controls whether the input is strong enough.
The **Bias** is an internal excitation.

$$v = \sum_{j=1}^{n} w_j x_j + b$$

This is an affine equation (linear + constant).

A lot of nature is linear. If the input is scaled, the output is also scaled. If I add inputs, the output gets added up. This is the essence of linearity.

Inputs
are numbers



If we add an input x_0 we can remove the bias and have a linear equation, but usually, we keep the bias separated.

The output y is the result of a final **activation function** ($y = \phi(v)$).

$$y = \phi(v) = \phi(\sum_{j=1}^{n} w_j x_j + b) \tag{1}$$

This is what is doing the whole neuron.

2.2 Types of activation functions

For the moment, we will focus on "squashing" type of function, i.e. It limits the amplitude range of the neuron output.

1. **Step function:** threshold function, Heaviside. As long as the input is negative, nothing happens, then something happens.

$$\phi(v) = \begin{cases} 1 & \text{if } v \geq 0, \\ 0 & \text{if } v < 0. \end{cases} \tag{2}$$

   We can have flexibility and change where the jump is happening. We do that with the bias: the threshold at which the bias occurs.

How do we define the weights? We should design them to create a specific behavior. The problem with the step function is that it becomes a discrete problem and finding the proper weights becomes a long process of trial and error → **optimization algorithms**, to gradually change the weights and see what happens. This is the motivation behind the **sigmoid function.**

2. **Sigmoid function:** "smoothed" threshold function

$$\phi(v) = \frac{1}{1 + e^{-aV}}, a > 0 \tag{3}$$

When **a** becomes larger, the function becomes sharper; it means that it becomes less differentiable (so more unstable, because of both large, at transition, and very small, gradients); if **a** becomes closer to 0, we will have a smoother function.

Additionally, the sigmoid function has a derivative easy to compute:

$$\phi'(v) = a\phi(v)(1 - \phi(v)) \tag{4}$$

Step and sigmoid function have outputs $\in [0, 1]$, but sometimes we may want to have outputs $\in [-1, 1]$ → we can change both the step and sigmoid function.

1. **Sign:**

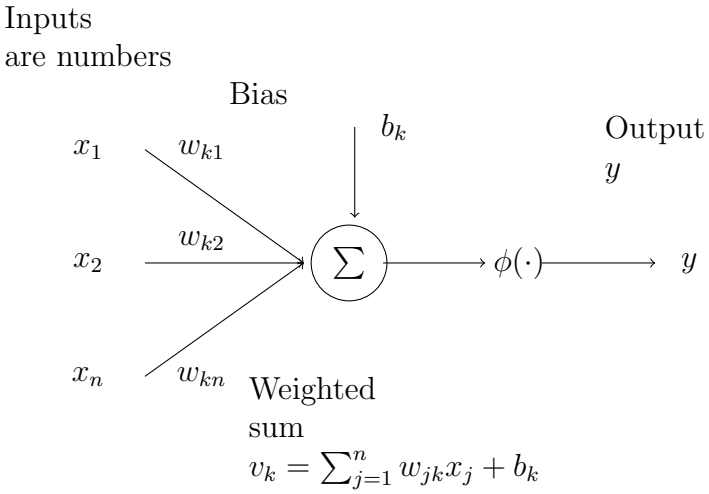
$$\phi(v) = \begin{cases} -1 & \text{if } v < 0, \\ 0 & \text{if } v = 0, \\ 1 & \text{if } v > 0 \end{cases} \tag{5}$$

2. **Hyperbolic Tangent:**

$$\phi(v) = \frac{e^{aV} - e^{-aV}}{e^{aV} + e^{-aV}} \tag{6}$$

2.3 Neural Network architectures

Up to now, we have talked about just one neuron, but we need to connect them. Let's name the first neuron, neuron **k**. We need to label all the neuron to book keep them



All the k's tell you that these units belong to that neuron.

3 Lecture 3

In the neural network, the **activation function** is what is creating the **non-linearity**

$$\phi(v) = \begin{cases} 1 & \text{if } v \geq 0, \\ 0 & \text{if } v < 0. \end{cases} \tag{7}$$

This is an idealization of what is happening inside the neuron, but it is quite rough, sometimes it is better to use a smoother function, like the sigmoid:

$$\phi(v) = \sigma(v) = \frac{1}{1 + e^{-aV}} \tag{8}$$

a is a parameter that defines the sharpness of the transition.

There are also variants that bring the step and the sigmoid functions in the range [-1, 1], and they are the **sign** and the **hyperbolic tangent**.

Also, there are **non squashing** types of functions:

Rectified Linear Unit (ReLu):

$$\phi(v) = \begin{cases} v & \text{if } v \geq 0, \\ 0 & \text{if } v < 0. \end{cases} \tag{9}$$

3.1 Feed Forward Neural Network

K different neurons, each one with its weights and biases. We index the neurons and the inputs are the same for each neuron. Each neuron produces its output y_k.

w_{ki} is the weight that connects neuron k to input i. All this bunch of neurons form a single layer. So this is a **single layer fully connected feed-forward neural network**, but we can have multiple layers.

The information flow is only over one direction, i.e. the input layer of source nodes projects directly onto the output layer of neurons. There is no feedback of the network's output to the network's input. We thus say that the network in Fig. 1 is of feed forward type.

If, for example, we have two layers, we no longer see the output of the first layer, so this becomes an **hidden layer**, while the second layer (in this case) is the output layer.

$$x = \begin{pmatrix} x1 \\ x2 \\ ... \\ x_n \end{pmatrix}$$

it is a vector

$$W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ w_{31} & \dots & & \\ w_{k1} & \dots & \dots & \dots w_{kn} \end{pmatrix}$$

All the weights are inside a matrix, every row is associated to one neuron, while every column is related to an input.

$$b = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_k \end{pmatrix}$$

One bias for each neuron

$$U = \begin{pmatrix} u_{11} & ... & u_{1k} \\ u_{21} & ... & u_{2k} \end{pmatrix}$$

In this case, 2 neurons, k inputs.

$$c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

$$y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

All these vectors and matrices are parameters, i.e. everything needed to define the neural network.

3.2 Linear Algebraic Representation

Any linear operator can be written as **inner products**.

$$\sum_{i=1}^{n} w_{1i}x_i + b_1 = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b_1$$

Repeat per neuron 2 (second row of the matrix)

$$\sum_{i=1}^{n} w_{2i}x_i + b_2 = \begin{pmatrix} w_{21} & w_{22} & \dots & w_{2n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b_2$$

...

Repeat per neuron k

$$\sum_{i=1}^{n} w_{ki}x_i + b_k = \begin{pmatrix} w_{k1} & w_{k2} & \dots & w_{kn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + b_k$$

Single Layer

$$z = \phi(Wx + b)$$

Two Layers

The second layer does the same thing, not to x , but to z

$$y = \phi(Uz + c) = \phi(U(\phi(Wx + b) + c)$$

It is extremely important to bookkeep dimensions:

$$x \in R^n \tag{10}$$
$$W \in R^{k \times n} \tag{11}$$
$$b \in R^k \tag{12}$$
$$z \in R^k \tag{13}$$
$$U \in R^{2 \times k} \tag{14}$$
$$c \in R^2 \tag{15}$$

In this case the **naming** is: n-k-2 fully connected feed forward neural network (each dash is a layer).

N.b A neural network is called fully-connected when we allow all weights; sometimes we can decide to force some of them to zero: in this care connections are removed and the network is not fully-connected anymore.

Example: Convolutional Neural Networks. If some outputs feed back (after some delay) as inputs, the network is not feed forward.

Example: LSTM (Long Short Term Memory)

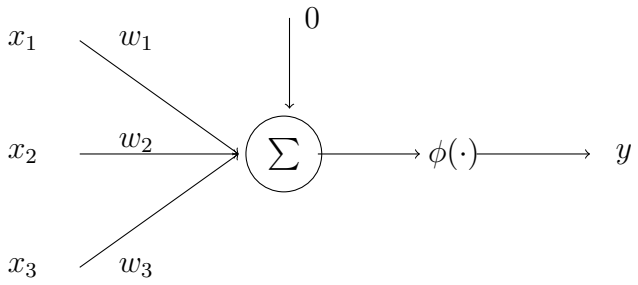
3.3 Examples:

a) **Majority Applications**

- n = 3 inputs $x \in \{-1, +1\}^3$
- Activation Function: sign: $\phi(v) = \begin{cases} +1 & \text{if } v > 0 \\ 0 & \text{if } v = 0 \\ -1 & \text{if } v < 0 \end{cases}$

- Goal: design a simple neuron that outputs the value that most appears in the input.

  E.g. $x = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix} \rightarrow$ y = -1



$$y = \text{sign}(x_1 + x_2 + x_3) = \text{majority}(\{x1, x2, x3\}$$

This neural network has 100% accuracy. If there are more +1 the result is going to be positive, otherwise it is going to be negative.

b) **Logical Statements**

c) **Classification:** e.g. Image representing a 3, the neural network needs to provide 3 as output.

We need to break the image into sth that the NN can take as input → we need to **vectorize** it.

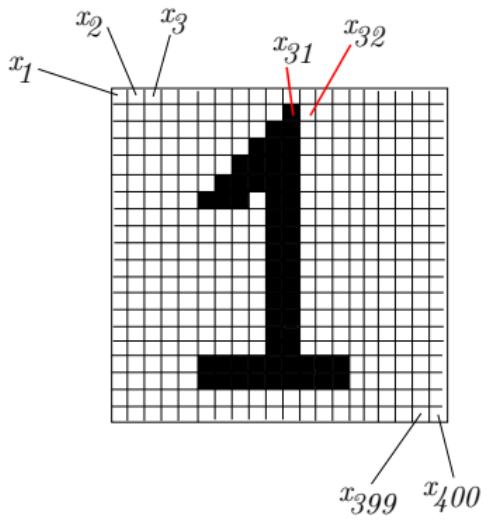


Figure 1: An image of a digit. Each pixel corresponds to one input node

One may design a feedforward neural network for digit classification. We are given 20 pixel by 20 pixel images of handwritten digits. Our job is to find, for each given

image, the actual digit corresponding to the image. Suppose that each pixel assumes binary values, i.e. the images are black and white only with no grayscale. We can then consider a feedforward neural network with 400 nodes in the input layer (labeled as $x_1, x_2, \ldots, x_{400}$), some large number of nodes (say 1000 nodes) in a hidden layer, and 10 nodes in the output layer giving outputs $y_0, \ldots, y_9$. Ignoring the biases, this fully-connected network has:

$$400 \times 1000 + 1000 \times 10 = 410000$$

parameters (synaptic weights) that we can optimize.

Neurons in a neural network can act as **shape-detecting units**, extracting **salient features** from the input. To represent this layer, we might need 1000 weights. For example, we could have one neuron specialized in detecting the digit '1', similar to how specialized areas in the brain work. While multiple neurons might be activated, the network can only predict one number at a time. To resolve this, at the end of the neural network, we can apply an operation called ARGMAX. This produces a one-hot representation, where the '1' indicates which neuron 'won' the prediction.

In other scenarios, using a SOFTMAX layer might be beneficial, which outputs a probability distribution. This helps determine the network's confidence in its prediction, offering insight into how wrong or uncertain the network might have been.

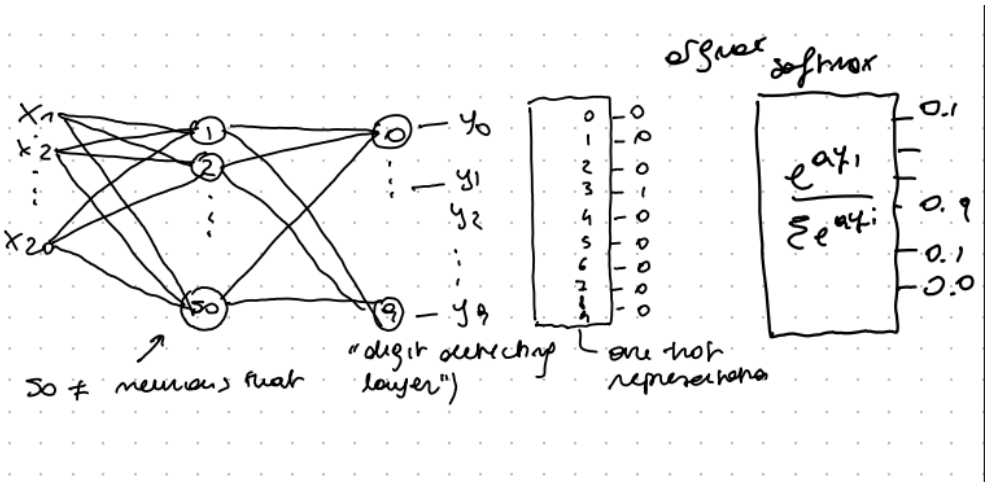


Figure 2: Graphical Representation of the neural network to perform digit classification

- Each node in the input layer will correspond to one pixel in the image, as illustrated in Fig. 2. For example, if the image in Fig. 2 is to be fed to the network as input, the inputs would be $x_0 = 0, x_1 = 0, \ldots, x_{30} = 0, x_{31} = 1, x_{32} = 0, \ldots, x_{399} = 0, x_{400} = 0$, i.e., we use 0 to represent a white pixel, and we use 1 to represent a black square.
- It is possible to design the network (choose the weights) such that when some image of digit 0 is fed to the network, the output will be $y_0 = 1, y_1 = \ldots, y_9 = 0$. Likewise,

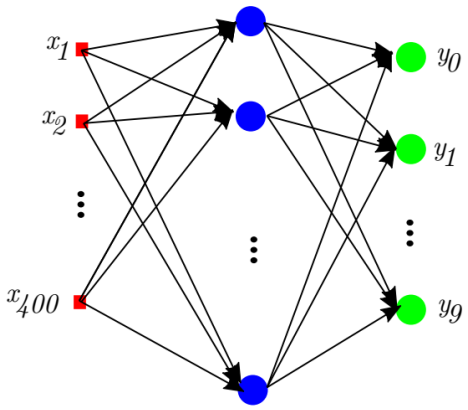


Figure 3: The 2-layer feedforward neural network for digit classification

for any $i \in \{0, \ldots, 9\}$, when some image of digit i is fed to the network, the output will be $y_i = 1$ and $y_j = 0, \forall j \neq i$.

- This is usually done via supervised learning. We prepare (or usually acquire!) a large number of 20×20 training images, each of which looks like Fig. 2 but of course with different digits, and differently written versions of the same digit. See Fig. 4 for some sample portion of such a training set. Each training image has a label that specifies the actual digit corresponding to the image. For example, in Fig. 4, the images in the first row will be labeled as 0, the second row as 1, and so on.

4 Lecture 4

4.1 Approximation theorems and learning algorithms

- Can we always find weights and biases (parameters) to do what we want to do?
- Given enough neurons and layers, almost every function can be approximated (as accurately as we want)
- with logic (binary inputs/outputs) the **DNC/CNF** theorem shows that 2 layers are enough.

  The DNF/CNF theorem states that every Boolean formula can be transformed into either a **Disjunctive Normal Form (DNF)** or a **Conjunctive Normal Form (CNF)**. In DNF, the formula is expressed as a disjunction (OR) of conjunctions (AND) of literals, such as $(x_1 \wedge \neg x_2) \vee (x_3 \wedge x_4)$. Conversely, in CNF, the formula is expressed as a conjunction (AND) of disjunctions (OR) of literals, for example, $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_4)$. This theorem is important because it guarantees that any Boolean function can be represented in one of these two canonical forms, which is useful in logic, computer science, and digital circuit design.

- But how do we set these parameters to achieve the desired behavior? Manual adjustment is impractical, except in simple cases. Instead, **learn using guidance from data**.

$$y_{\text{params}}(x) = \phi(U\phi(Wx + b) + c)$$

This output ideally is equal to our target y, but this is not always the case. **Goal:** Minimize the number of mistakes. Mathematically we have a mistake when $y_{\text{params}} \neq y$.

What we can do is to look at the whole dataset and count how many times mistakes are made.

$$\sum_{(x,y)\in\text{data}} \{y_{\text{params}} \neq y\}$$

Typical algorithm: Local search + gradual data.

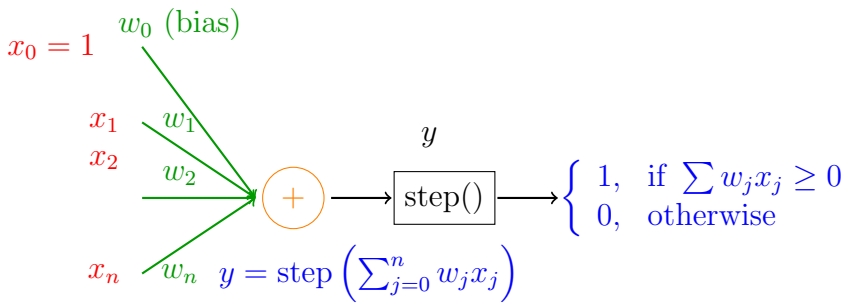
1. Initialize parameters **randomly**
2. for each (x,y) in the data
   - Feed x to NN and get $y_{\text{params}}(x)$
   - Look in the neighborhood of params to make $y_{\text{params}}(x)$ closer to y. Explore a "neighborhood" around the current parameter values, to see if they lead to an improved solution. If a perturbation leads to a better outcome (fewer mistakes), update the parameters accordingly.
3. Repeat 2. until parameters converge

Steps 2. and 3. represent an **epoch**, i.e. one full pass over the dataset.

4.2 Perceptrons

This is the simplest situation we are going to consider.

Consider the **McCullock-Pitt (1945)** neuron, with bias and weights.



What can it do?

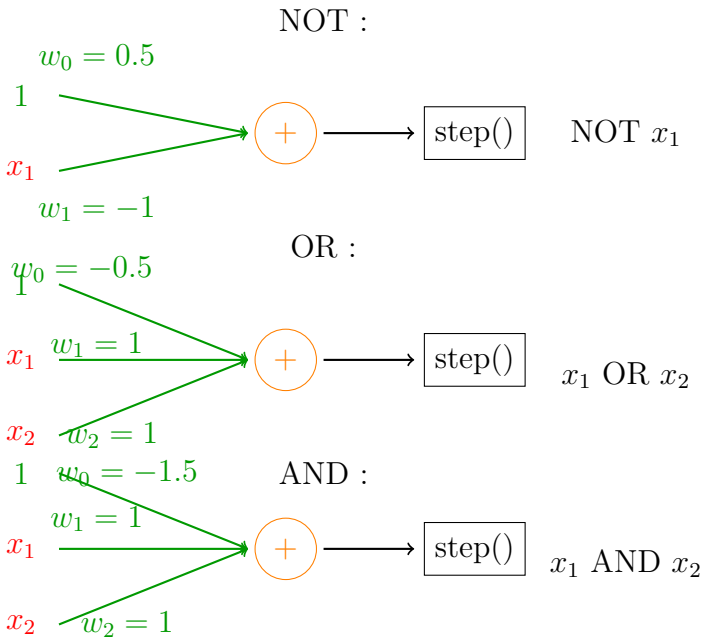
- **Recall:** $\begin{bmatrix} w_0 & w_1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = 0$ This is a line that passes through the origin.

  $\begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = 0$ This is a plane that passes through the origin

- In higher dimensions, this is called **hyperplane**: $\begin{bmatrix} . & . & w & . & . \end{bmatrix} \begin{bmatrix} . \\ . \\ x \\ . \\ . \end{bmatrix} = 0.$

  $\begin{bmatrix} . & . & w & . & . \end{bmatrix} \begin{bmatrix} . \\ . \\ x \\ . \\ . \end{bmatrix} \geq 0$ is a **halfspace**

- It can model **logic gate**



- It can classify using an hyperplane as **decision boundary**, which is the boundary with which we decide to switch from 0 to 1. This classifier is a linear separator

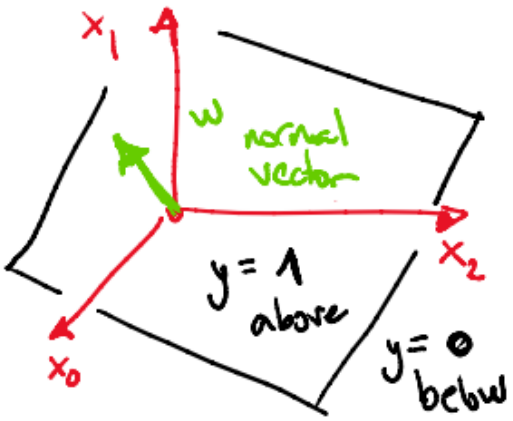


Figure 4: The image illustrates the decision boundary of a hyperplane in a 3D space. The axes x_0, x_1, and x_2 represent input features, and the green vector w is the normal vector to the plane, representing the weight vector of a binary classifier (e.g., a perceptron). Points above the plane are classified as $y = 1$, while points below it are classified as $y = 0$. The plane itself serves as the decision boundary between these two classes.

5 Lecture 5

Last time: Approximation theories and learning theories. Percpetrons → one of the simplest algorithms

5.1 Example

$$\neg x_1 \wedge \neg x_2 ... \wedge \neg x_n \wedge x_{n+1} \wedge x_{n+2} ... \wedge x_{n+m}$$

This is a logic description, but we can move towards an arithmetic description:

$$(1 - x_1) + (1 - x_2) + ... + (1 - x_n) + (x_{n+1}) + ... + (x_{n+m})$$

If I add these guys, depending on their values, I can obtain a different range of results: the smallest one is zero, while the biggest is n+m.

Now we want to **threshold** it (i.e. **adjust the bias**), to build an AND operator.

$$(1 - x_1) + (1 - x_2) + ... + (1 - x_n) + (x_{n+1}) + ... + (x_{n+m}) \geq n + m - \frac{1}{2}$$

If it is an AND the output should be 1 only when they are all 1, so when the sum is equal to n+m:

$$\underbrace{-(m + \frac{1}{2})}_{w_0} \cdot x_0 + \underbrace{(-x_1)}_{w_1} + \underbrace{(-x_2)}_{w_2} + \cdots + \underbrace{(-x_n)}_{w_n} + \underbrace{x_{n+1}}_{w_{n+1}} + \cdots + \underbrace{x_{n+m}}_{w_{n+m}}$$

Because NOT and either AND or OR together are **weighted gates**, **multiplayer perceptron** can represent **any** logical operator. How many layers are sufficient to implement

any operator? **We need 2 layers**, thanks to the **DNF/CNF** formula. May it not be the most efficient way, to implement a table, that's why the concept of **depth** is then introduced.

If we make a **truth table**, we need many entries, while with depth, fewer neurons are needed.

Is 1 layer sufficient? Answer: NO.

5.2 Perceptron Learning Algorithm

Given a desired behavior described by a function $y = f(x)$, the goal is to find a set of parameters $\mathbf{w}$ such that the perceptron's output $y_{\mathbf{w}}(x)$ closely approximates $f(x)$. Manually determining these weights is impractical for complex datasets, necessitating an automatic learning approach.

Assumption: There exists a weight vector $\mathbf{w}$ such that:

$$y_{\mathbf{w}}(x) = \text{step}(\mathbf{w}^T\mathbf{x}) = y$$

where $\mathbf{x} = [x_0, x_1, \ldots, x_n]^T$ includes the bias term $x_0 = 1$.

Jargon: We define **classes**, which are the regions where $y = 0$ and $y = 1$. These classes represent higher-level abstractions of the data, encapsulating the concept of classification.

In **supervised learning**, the model is trained using labeled data, where each data point is paired with its corresponding label.

Algorithm: Perceptron Learning Algorithm Input:

$$\text{Data} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\} \quad \text{where} \quad \mathbf{x}^{(i)} \in R^n, \quad y^{(i)} \in \{0, 1\}$$

Output:

$$\text{Weight vector } \mathbf{w} \in R^{n+1}$$

including w_0 for the bias term.

Procedure:

1. **Initialization**:
   - Initialize weights $\mathbf{w} = [w_0, w_1, \ldots, w_n]^T$ to small random values or zeros.
2. **Training Loop**:
   - **While** there exist misclassified examples in the training data:
     (a) **For each** $(\mathbf{x}, \mathbf{y}) \in$ Data:
     - **Compute** the perceptron's output:

       $$\hat{y} = \text{step}(\mathbf{w}^T\mathbf{x})$$

     - **Update Rules**:

       $$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot \mathbf{x} \cdot (y - \hat{y})$$

       where η is the learning rate, a small positive constant that controls the magnitude of weight updates.

- **Interpretation**:
  - If $\hat{y} = 0$ and $y = 1$: The perceptron needs to increase the weighted sum to correctly classify the example. Hence, $\mathbf{w}$ is updated in the direction of $\mathbf{x}$.
  - If $\hat{y} = 1$ and $y = 0$: The perceptron needs to decrease the weighted sum to correctly classify the example. Hence, $\mathbf{w}$ is updated in the opposite direction of $\mathbf{x}$.

3. **Termination**:

   - The algorithm terminates when either:
     - All training examples are correctly classified.
     - A predefined maximum number of epochs (iterations) is reached to prevent infinite loops in the case of non-linearly separable data.

Learning Rate (η):

- Determines the size of the weight updates.
- Affects the speed and stability of convergence.
- Typical values range between $0 < \eta \leq 1$.

Epoch:

- One full pass over the entire training dataset.
- Multiple epochs are often necessary for the algorithm to converge.

Theorem (Convergence of the Perceptron Algorithm):
If the training data is linearly separable, the perceptron learning algorithm is guaranteed to converge to a weight vector that correctly classifies all training examples within a finite number of updates.

Proof Sketch: The convergence theorem is based on the idea that each update moves the weight vector closer to a separating hyperplane. Assuming the data is linearly separable, there exists a hyperplane that separates the classes. The algorithm will not make updates indefinitely because it progressively reduces the number of classification errors.

5.3 Limitations of the Perceptron

While the perceptron is a fundamental building block in neural networks and offers valuable insights into binary classification, it has several inherent limitations that restrict its applicability in more complex scenarios.

5.3.1 Linearly Separable Data Only

- **Constraint**: The perceptron can only solve classification problems where the two classes are **linearly separable**, meaning there exists a hyperplane that can perfectly separate all instances of one class from the other.
- **Implication**: For datasets that are **not linearly separable**, the perceptron algorithm fails to find a solution that correctly classifies all training examples. A classic example is the *XOR* problem, where no single hyperplane can separate the classes.

5.3.2 Convergence Issues

- **Non-Separable Data**: If the training data is not linearly separable, the perceptron algorithm does not converge. It will continue to adjust weights indefinitely without finding a satisfactory solution.
- **Oscillations**: In cases where data is barely separable or near the boundary of separability, the algorithm may oscillate between different weight vectors without settling.

5.3.3 Limited Expressiveness

- **Single Layer**: The perceptron is a single-layer network, which limits its capacity to model complex, non-linear relationships in data. It cannot capture interactions between features that require multiple layers or non-linear transformations.
- **Inability to Learn Complex Patterns**: More intricate patterns and dependencies in data necessitate multi-layer architectures (e.g., Multi-Layer Perceptrons) that can learn hierarchical feature representations.

5.3.4 Binary Output

- **Output Restriction**: The basic perceptron is inherently a binary classifier, outputting either 0 or 1. Extending it to handle multi-class classification requires additional mechanisms, such as using multiple perceptrons (one-vs-rest strategy) or adopting different network architectures.
- **Lack of Probabilistic Interpretation**: The perceptron's output does not provide a probability measure of class membership, limiting its use in applications where uncertainty quantification is important.

5.3.5 Sensitivity to Input Scaling

- **Impact**: The perceptron's performance is sensitive to the scale of input features. Features with larger scales can dominate the decision boundary, leading to suboptimal classification.
- **Solution**: Implementing feature scaling techniques, such as normalization or standardization, can mitigate this issue by ensuring that all features contribute equally to the decision-making process.

5.3.6 No Probabilistic Output

- **Limitation**: The perceptron provides a hard binary decision without expressing uncertainty or confidence levels in its predictions.
- **Implication**: In applications where understanding the confidence of predictions is crucial (e.g., medical diagnosis), the perceptron's binary output is insufficient.

5.3.7 Difficulty in Choosing the Learning Rate

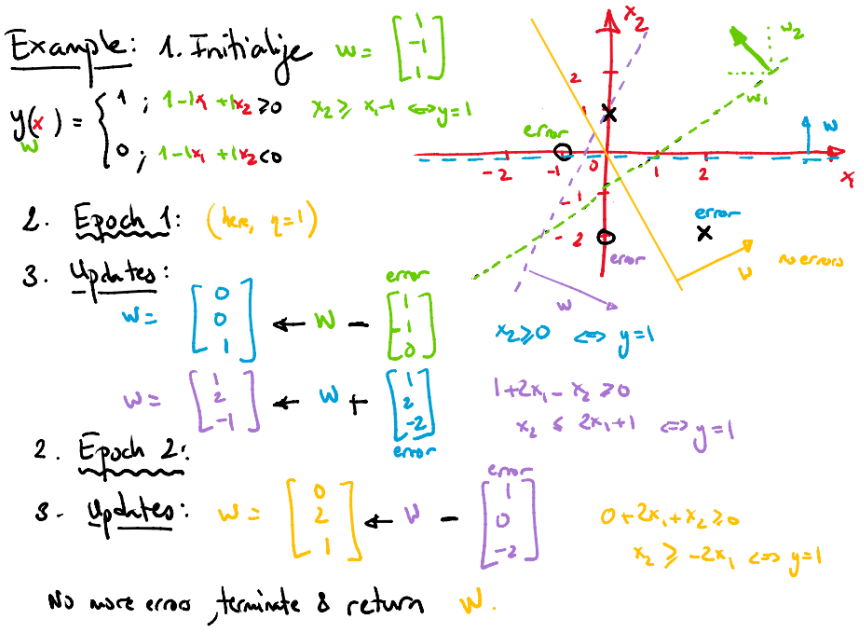
- **Challenge**: Selecting an appropriate learning rate η is critical. A rate that is too high can cause the algorithm to overshoot the optimal solution, while a rate that is too low can lead to slow convergence.
- **Solution**: Techniques such as learning rate schedules, adaptive learning rates, or using algorithms like Momentum can help in selecting and adjusting the learning rate dynamically during training.

5.3.8 No Margin Maximization

- **Limitation**: The perceptron algorithm does not inherently maximize the margin between classes. This can result in a decision boundary that is sensitive to noise and outliers.
- **Implication**: Models that maximize the margin, such as Support Vector Machines (SVMs), often generalize better to unseen data.

6 Lecture 6

6.1 Example



In this example, weights are initialized to

$$\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

In this case, the weight vector is pointing in the negative direction for x_1 and in the positive direction for x_2 and x_0 controls the shift from the origin.
This separator is making errors, so we have to change the weights every time we notice an error.
Epoch 1: weights are updated using the error point: the first error point considered is $(-1, 0)$, which has been classified as 1 but should have been classified as 0, so we subtract that point from the weight vector. We obtain

$$\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}$$

We obtain a vector that is still making errors. The point $(2, -2)$ is classified as 0 but should be classified as 1, so we add this point to the weight vector:

$$\begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ -2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ 1 \end{bmatrix}$$

6.2 Supervised Learning

In **supervised learning** we are dealing with data (x, y), where y is a sort of supervision. There is not a specific task, but it is defined according to how x behaves.

$$x \in \chi \text{ (\textbf{feature space:} context where we perform the task)}$$

$$y \in Y \text{ \textbf{(label space:} target)}$$

- y can be **discrete** and in this case we talk about **classification**, but it can also be **continuous** and in this case we talk about **regression**.
- **Task:** predict/imitate y using only x.
- **How?** Use a neural network: $y_w(x) = f(x, w)$ (function of x, with parameter w). This gives us a predictor, which is a function of x parametrized by w.
- **Loss:** measures how far y and f(x,w) are:
  - **0-1 loss:** $l(y, f) = 1\{y \neq f\}$
  - **squared loss:** $l(f, y) = ||y - f||^2 \rightarrow$ euclidean distance between them
- **Risk:** Average loss over a dataset

$$\text{Data} = \{(x, y), ...\}$$

$$\text{Risk: } \frac{1}{|\text{Data}|} \sum_{(x,y)\in \text{Data}} l(y, f(x, w)) \tag{16}$$

  The risk is a property of the data and of the weights. The risk for the perception is the rate of mistakes and it represents the probability of making a mistake.

- **Goal:** Minimize (empirical) risk. Make as few mistakes as possible, remaining as close to y as possible $\rightarrow$ search the space fo w, to find the one that gives the smallest R(w).
- **Types of algorithms:**
  - **Online:** We use these algorithms when data streams in. Weights are updated **along the way**. (usually) don't revisit data points. If we forgo epoch and get new data, the perceptron will be online.
  - **Batch:** data is available in whole, so we don't have to look at each point singularly, but we can have a global perspective. (Usually) pass over data multiple times (epochs).
  - **Mini batch:** Process data gradually in small batches.

7 Lecture 7

7.1 Linear Regression with Squared Loss

- Data: $\{(x, y), ...\}$, $x \in R^n, y \in R^m$ (x is an n-dimensional vector to predict an m-dimensional vector).
- **Predictor:** $f(x, W) = Wx$, $W \in R^{nxm}$ (this is a single neuron with $\phi(v) = v$, identity activation).

- **Goal:** Minimize $R(w) = \sum_{(x,y)\in \text{Data}} ||y - Wx||^2$

  How do we solve a problem like this? Analytic solution: Optimality condition, because we are unconstrained, we want:

$$\nabla_w R = 0$$

  This is not an equation, but $n \cdot m$ equations.

  Let's consider a single data point to start:

$$l = (\sum_{k=1}^{m} y_{k'} - \sum_{j'=1}^{n} w_{kj'} x_{j'})^2 \tag{17}$$

  This is the loss of a single point, on m neurons. To minimize the loss, we need to compute the gradient of the loss with respect to each weight w_{kj}

$$\frac{\partial l}{\partial w_{kj}} = -2y_k x_j + 2\sum_{j=1}^{n} w_{kj} x_j x_j \tag{18}$$

  We are making the derivative with respect to a particular j. in terms of vectorial operation is an **outer product**, so it takes two vectors as input and gives a matrix as output. Represents the contribution of the target value y_k and the input feature x_j to the gradient. It essentially measures how much the current prediction deviates from the target.

  This matrix is multiplied by matrix **W**. For every **w** we find the gradient and put it in a matrix. Represents the contribution of the current weights and input features to the gradient. It accounts for the influence of the weight $\mathbf{w_{kj}}$ in reducing the loss.

$$\nabla_w l = -2 \cdot y \cdot x^T \tag{19}$$

$$\begin{bmatrix} ... \\ y \\ ... \end{bmatrix} \begin{bmatrix} ... & x^T & ... \end{bmatrix}$$

The rank of this matrix is 1

$$\nabla_w l = -2 \cdot y \cdot x^T + 2 \cdot W \cdot x \cdot x^T$$

This is the gradient of the loss for a single data point.

The risk is the sum of the losses of the data points. We want to find the derivative of that. But since the derivative is a linear operator, the derivative of the sum is the sum of the derivatives → the gradient of the sum is the sum of the gradients.

$$R(w) = \sum_{(x,y)\in \text{Data}} l(y, Wx) \rightarrow \nabla_w R = \sum_{(x,y)} \nabla_w l$$

$$\nabla_w R = \sum_{(x,y)} -2 \cdot y \cdot x^T + 2 \cdot W \cdot x \cdot x^T$$

$$-2 \cdot \begin{bmatrix} \dots & \dots. & \dots \\ y & y & y \\ \dots & \dots. & \dots \end{bmatrix} \cdot \begin{bmatrix} \dots & x^T & \dots \\ \dots & x^T & \dots \\ \dots & x^T & \dots \end{bmatrix} + 2 \begin{bmatrix} W \end{bmatrix} \cdot \begin{bmatrix} \dots & \dots & \dots. \\ x & x & x \\ \dots & \dots & \dots \end{bmatrix} \cdot \begin{bmatrix} \dots & x^T & \dots \\ \dots & x^T & \dots \\ \dots & x^T & \dots \end{bmatrix}$$

$$(m \cdot \text{Data}) \cdot (\text{Data} \cdot n) + 2 \cdot (m \cdot n) \cdot (n \cdot \text{Data}) \cdot (Data \cdot n)$$

In terms of data matrices:

$$\nabla_w R = 0 \iff -2 \cdot y \cdot x^T + 2 \cdot w \cdot x \cdot x^T = 0$$

$$-2 \cdot y \cdot x^T + 2 \cdot w \cdot x \cdot x^T = 0 \Rightarrow W = y \cdot x^T \cdot (x \cdot x^T)^{-1}$$

We are trying to solve $W \cdot x \sim y$, but the problem is that we can't always invert X, because it isn't always a square matrix. We can almost invert it with $X^T \cdot (X \cdot X^T)^{-1}$, which is the **Moore-Penrose Pseude-Inverse**.

This solution provides the exact minimum of the empirical risk for linear regression, **without approximation**, computed directly, **without iterative procedured**.

8 Lecture 8

8.1 Gradient Descent

Gradient Descent is an iterative optimization algorithm used to minimize the empirical risk bu updating the weights **W** in the direction opposite to the gradient of the loss function.

Under certain conditions (e.g. convexity of the loss function, appropriate learning), GD converges to the same solution as the analytical method.

There are some variants of the GD:

- **Batch Gradient Descent:** use the entire dataset to compute the gradient at each step
- **Stochastic Gradient Descent (SGD):** uses one data point to compute the gradient, leading to noisy but faster updates.
- **Mini-Batch Gradient Descent:** Uses a subset (mini-batch) of data points for gradient computation. It is a trade-off.

- **Goal:** minimize over w $R(w) = \sum_{(x,y)\in \text{data}} l((x,y),w) = l(y, f(x,W))$
- **Idea:** in general the optimal condition is to set the derivative to 0. In multivariate case we should change the notion of derivative to the notion of **gradient**, which is a collection of derivatives. However, we are not going to find a good solution by setting it to 0! We are going to reduce it step by step to 0.

Let's call the amount of change **delta**. At each step we change the weights by an amount Δw.

$$w' = w + \Delta w \tag{20}$$

$$R(w') = R(w + \Delta w) = R(w) + \nabla R(w)^T \Delta w + O(||\Delta w||^2) \tag{21}$$

We want to move in such a way to make this function as negative as possible. How should we choose Δw to get the best reduction?

$O(||\Delta w||^2)$ represents other terms that are smaller than the norm squared of Δw

To minimize $a^T v$, $v = \frac{-a}{||a||_2} \rightarrow ||v||_2 \leq 1$.

Let $\Delta w \propto -\nabla R(w)$

$$w' = w - \eta \nabla R(w) \tag{22}$$

It's like rolling down the hill, but without momentum

8.1.1 1-D example

$R(w) = w^4$ The minimum of this function is 0.

$$\frac{dR}{dw} = 4w^3, \eta = \frac{1}{8}$$

$$w(0) = 1$$

,

$$w(1) = w(0) - \nabla R(w(0)) \cdot \eta = \frac{1}{2}$$

$$w(2) = w(1) - \nabla R(w(1)) \cdot \eta = \frac{1}{2} - 4 \cdot \frac{1}{2}^3 \cdot \frac{1}{8} = \frac{7}{16}$$

Now us smaller than $\frac{1}{2}$, but the algorithm has slowed down. What happens if we continue? The algorithm at each step is slowing down more and more, but this does not work for all *eta*. If $\eta < 0$, there will be oscillations, and the algorithm may not converge. If η is small enough, always diminishes since bounded from below (by 0). Will it converge to 0? Yes, think about it.

It is extremely important **to have η small enough**.

8.2 Widrow-Hoff LMS Algorithm

The LMS algorithm is an iterative method for adjusting the weights of a linear model to minimize the mean squared error (MSE) between the predicted outputs and the actual target values. It operates in an online or incremental fashion, updating the weights as each data point is processed.

The Widrwo-Hoff LMS is very similar to the PTA. Note that this is not a true gradient descent, and may not converge to the globally optimal solution (unlike the true gradient descent does when η is taken sufficiently small). In fact, the solution will move randomly around the globally optimal solution.

The update now becomes:

$$w = w + \eta \cdot \sum_{(x,y)} (y - w^T \cdot x) \cdot x$$

This is one pass across the data, so it is a big **batch**.
Now we change the batch algorithm into an online algorithm (one data point at a time).

For each (x, y) $\in$ Data: $w = w + \eta \cdot (y - w^T \cdot x) \cdot x$

This algorithm looks like the perceptron, which is in some sense a gradient descend algorithm.

What would neurons satisfy to implement the gradient descend? The activation function must be **differentiable**.

8.3 Delta Rule

We have to make sure our loss is **differentiable**, otherwise we can't apply the **gradient descend algorithm**.

The **Delta Rule** can be seen as a special case of Gradient Descent, specifically used for training single-layer neural networks(perceptions), applied to the **mean squared error** cost function. So, while GD is a general algorithm applicable to a wide range of optimization problems, the delta rule is focused on adjusting weights in a perception to minimize error.

$$w \leftarrow w + \eta \cdot (d - y) \cdot x \tag{23}$$

Where d is the desired target, while y is the obtained output.

The equation for GD is:

$$w \leftarrow w - \eta \cdot \nabla_w R(w) \tag{24}$$

Squared Loss for Simple neuron with differentiable $\phi(.)$

$$R(w) = \sum_{(x,y)} (y - \phi(W^T x))^2$$

Apply GD to it

$$\nabla R(w) = \sum_{(x,y)} 2 \cdot (y - \phi(w^t \cdot x)) \cdot (-\phi'(W^T \cdot x) \cdot x)$$

$$w' \leftarrow w - \eta \nabla R(w) x' \leftarrow w + \eta \sum_{(x,y)} 2 \cdot (y - \phi(w^t \cdot x)) \cdot \phi'(W^T \cdot x) \cdot x$$

One data point at a time

$$\phi(v) = \frac{1}{1 + e^{[} - av]} \tag{25}$$

$$\phi'(v)) \frac{-e^{-av}}{1 + e^{-av}} = \phi(v)(1 - \phi(v)) \tag{26}$$

9 Lecture 9

Last time: Gradient Descent
Minimize$_w$ R(w) = $\sum_{(x,y) \in \text{Data}} l(y, f(x, w))$

$$w' \leftrightarrow w - \eta \nabla R(W)$$

The minus sign is because we are moving towards the direction where the gradient is decreasing.

$$f(x, w) = Wx \tag{27}$$

$$l(y, f(x, w)) = ||y - f(x, w)||^2 \tag{28}$$

$$w' \leftarrow w - \eta \nabla R(w) = w - 2 \cdot \eta (y - \sum_{(x,y) \in \text{data}} (y - wx) \cdot (-x)) \tag{29}$$

$$w' \leftarrow w + 2 \cdot \eta (y - \sum_{(x,y) \in \text{data}} (y - Wx)) x^T \tag{30}$$

The delta method allows to put in an activation function.
How do we update weights in a computationally efficient way?

9.1 Back Propagation

A feed forward neural network accepts an input $\mathbf{x}$ and produces an output $\hat{\mathbf{y}}$; information flows forward thought he network. The input $\mathbf{x}$ provides the initial information that then-propagates up to the hidden units at each layer and finally produces $\hat{\mathbf{y}}$. This is the **forward propagation**. After this forward pass, a cost function is produced (error between desired value and output value). The **back-propagation** algorithm allows information from the cost to then flow backward through the network in order to compute the gradient. Backpropagation is just a method for computing the gradient, using the **chain-rule**, while gradient descents performs learning using this gradient.

The **backpropagation** is an algorithm to compute the gradient. So far we have been computing gradients with 1 layer of neurons. What if we have a network of interconnected neurons? **We are going to apply the chain rule multiple times.**

$$\frac{\partial R}{\partial w_k} = \sum_{(x,y)} \frac{\partial l}{\partial w_k} = \sum_{(x,y)} \frac{\partial l}{\partial f} \cdot \frac{\partial f}{\partial w_k} \tag{31}$$

since it is a linear operator, we can bring the gradient inside the sum. Equation 31 works if we have just one output. If we have multiple outputs, we have to repeat the same things for all the i outputs.

$$\frac{\partial R}{\partial w_k} = \sum_{(x,y)} \cdot \frac{\partial l}{\partial w_k} = \frac{\partial l}{\partial f_i} \cdot \frac{\partial f_i}{\partial w_k} \tag{32}$$

To solve the equation we need $\frac{\partial \mathbf{f_i}}{\partial \mathbf{w_k}}$.

This f is a function of t_k, which is a function of v_k, which is a function of $w_k \rightarrow$ **proof by induction**.

$$\frac{\partial f}{\partial w_k} = \frac{\partial}{\partial w_k} f(k_1, t_k, \dots) \tag{33}$$

con $t_k = \phi(v_k)$ e $v_k = w_k x + \dots$

$$= \left[\frac{\partial f}{\partial t_k}\bigg|_{t_k=\phi(v_k)} \right] \cdot \phi'(v_k) \cdot x \tag{34}$$

dove $\delta_k = \frac{\partial f}{\partial t_k}$

$$\frac{\partial f}{\partial \xi}\bigg|_{\xi=\phi(u)} = \frac{\partial f}{\partial \xi} f(x, t_{k_1}, \dots, t_{k_m}, \dots) \tag{35}$$

where,

$$t_{k_1} = \phi(w_1 x + \dots) \tag{36}$$

$$t_{k_2} = \phi(w_2 \xi + \dots) \tag{37}$$

$$t_{k_m} = \phi(w_m \xi + \dots) \tag{38}$$

$$t_{k_n} = \phi(w_n \xi + \dots) \tag{39}$$

To calculate $\frac{\partial \mathbf{f_i}}{\partial \mathbf{w_k}}$, we first perform a **forward pass**, to get all the outputs (u, ξ, v, t), then we perform a **backward pass**.

ξ feed into w in the forward pass, while δ feeds into w in the backward pass- To find all the gradients we need to compute the forward signal with the backward signal.

$$\textbf{Gradient of W} = \textbf{forward signal} \cdot \textbf{backward signal}$$

If we have multiple neurons, each one with its ξ we just have to index each ξ.

10 Lecture 10

With the gradients computed via backpropagation, we can perform **gradient descent** to find the optimal weights $\mathbf{w}$ that minimize the risk (loss) function. In this lecture, we will delve deeper into the backpropagation algorithm, elucidating the mathematical foundations and providing a step-by-step explanation of the gradient computations.

10.1 Overview of Gradient Descent

Gradient descent is an optimization algorithm used to minimize the loss function $R(\mathbf{w})$. The core idea is to iteratively update the weights in the opposite direction of the gradient of the loss function with respect to the weights.

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} R(\mathbf{w})$$

where:

- η is the learning rate.
- $\nabla_{\mathbf{w}} R(\mathbf{w})$ is the gradient of the loss function with respect to the weights.

10.2 Computing Gradients with Backpropagation

Backpropagation efficiently computes the gradients of the loss function with respect to each weight by applying the chain rule of calculus. Let's explore the mathematical formulation in detail.

10.2.1 Basic Gradient Computation

Consider a neuron with input ξ, weight w_k, bias b, activation function ϕ, and output t_k:

$$t_k = \phi(w_k \xi + b)$$

The loss function ℓ depends on the output t_k, the input x_i, and possibly other parameters. To perform gradient descent, we need to compute the partial derivative of the loss with respect to each weight w_k.

$$\frac{\partial \ell}{\partial w_k} = \frac{\partial \ell}{\partial t_k} \cdot \frac{\partial t_k}{\partial w_k}$$

Applying the chain rule:

$$\frac{\partial \ell}{\partial w_k} = \left[\left. \frac{\partial \ell}{\partial t_k} \right|_{t_k = \phi(w_k \xi + b)} \right] \cdot \phi'(w_k \xi + b) \cdot \xi$$

Introducing the notation δ_{t_k}:

$$\delta_{t_k} = \frac{\partial \ell}{\partial t_k}$$

Thus, the gradient becomes:

$$\frac{\partial \ell}{\partial w_k} = \delta_{t_k} \cdot \phi'(w_k \xi + b) \cdot \xi$$

10.2.2 Aggregating Gradients for the Risk

When considering the entire dataset, the risk $R(\mathbf{w})$ is typically the average loss over all training examples. The gradient of the risk with respect to a weight w_k is the sum of the gradients of the individual losses:

$$\frac{\partial R}{\partial w_k} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \ell_i}{\partial w_k}$$

For simplicity, we often omit the scaling factor $\frac{1}{N}$ in gradient descent updates, as it can be absorbed into the learning rate η.

10.2.3 Forward Pass

During the forward pass, we compute the activations of each neuron in the network:

$$\mathbf{t} = \phi(\mathbf{W} \xi + \mathbf{b})$$

where:

- $\mathbf{W}$ is the weight matrix.
- ξ is the input vector.

- $\mathbf{b}$ is the bias vector.
- ϕ is the activation function applied element-wise.

10.2.4 Backward Pass

After computing the forward pass, we perform the backward pass to calculate the gradients.

Gradient with Respect to Weights For a weight matrix $\mathbf{W}$, the gradient of the loss with respect to $\mathbf{W}$ is:

$$\nabla_{\mathbf{W}}\ell = [\phi'(\mathbf{W}\xi + \mathbf{b})] \circ (\nabla_{\mathbf{t}}\ell) \cdot \xi^T$$

Gradient with Respect to Inputs To propagate the gradient to the previous layer, we compute the gradient of the loss with respect to the input ξ:

$$\nabla_{\xi}\ell = \mathbf{W}^T \cdot (\phi'(\mathbf{W}\xi + \mathbf{b}) \circ \nabla_{\mathbf{t}}\ell)$$

Simplifying the notation with δ:

$$\delta = \phi'(\mathbf{W}\xi + \mathbf{b}) \circ \nabla_{\mathbf{t}}\ell$$

Thus,

$$\nabla_{\xi}\ell = \mathbf{W}^T \cdot \delta$$

This δ is then used to compute gradients for the preceding layers, enabling the backward propagation of errors.

10.3 Handling Biases in Backpropagation

Bias terms are treated similarly to weights but do not have an associated input ξ. Instead, their input is always 1.

10.3.1 Gradient with Respect to Bias

Given the neuron activation:

$$t = \phi(\mathbf{w}^T\xi + b)$$

The gradient of the loss with respect to the bias b is:

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial t} \cdot \frac{\partial t}{\partial b} = \frac{\partial \ell}{\partial t} \cdot \phi'(\mathbf{w}^T\xi + b)$$

Using the δ notation:

$$\frac{\partial \ell}{\partial b} = \delta_t \cdot \phi'(\mathbf{w}^T\xi + b)$$

10.4 Linear Algebraic Expressions in Backpropagation

Expressing backpropagation in matrix form allows for efficient computation, especially with modern hardware optimizations.

10.4.1 Neuron Activation in Matrix Form

$$\mathbf{t} = \phi(\mathbf{W}\xi + \mathbf{b})$$

where:

- $\mathbf{W} \in R^{m \times n}$ is the weight matrix connecting n inputs to m neurons.
- $\xi \in R^n$ is the input vector.
- $\mathbf{b} \in R^m$ is the bias vector.
- $\mathbf{t} \in R^m$ is the output vector after activation.

10.4.2 Loss Function

$$\ell(\mathbf{t}) = \ell(\phi(\mathbf{W}\xi + \mathbf{b}))$$

10.4.3 Gradient with Respect to Weight Matrix

$$\nabla_{\mathbf{W}}\ell = [\phi'(\mathbf{W}\xi + \mathbf{b})] \circ (\nabla_{\mathbf{t}}\ell) \cdot \xi^T$$

where $\circ$ denotes element-wise multiplication.

10.4.4 Gradient with Respect to Input Vector

$$\nabla_{\xi}\ell = \mathbf{W}^T \cdot (\phi'(\mathbf{W}\xi + \mathbf{b}) \circ \nabla_{\mathbf{t}}\ell)$$

10.4.5 Delta Rule Representation

Using the delta rule, we represent the error term δ for each neuron:

$$\delta = \phi'(\mathbf{W}\xi + \mathbf{b}) \circ \nabla_{\mathbf{t}}\ell$$

Thus, the gradient update for the weights becomes:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot (\delta \cdot \xi^T)$$

10.5 Weight Update Equations

After computing the gradients, we update the weights and biases using gradient descent.

10.5.1 Weight Update

$$\mathbf{W} \leftarrow \mathbf{W} - \eta\nabla_{\mathbf{W}}\ell$$

10.5.2 Bias Update

$$\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} \ell$$

10.5.3 Choosing the Learning Rate

The learning rate η controls the size of the weight updates:

- **Too Small**: Training can be very slow.
- **Too Large**: May cause the algorithm to overshoot minima or diverge.

Techniques like learning rate scheduling, adaptive learning rates (e.g., Adam, RMSprop), and learning rate decay are often employed to address these issues.

10.5.4 Batch vs. Stochastic vs. Mini-batch Gradient Descent

- **Batch Gradient Descent**: Uses the entire dataset to compute gradients.
- **Stochastic Gradient Descent (SGD)**: Uses a single training example to compute gradients.
- **Mini-batch Gradient Descent**: Uses a subset (mini-batch) of the dataset to compute gradients.

Mini-batch gradient descent is commonly used as it balances the efficiency and convergence stability of batch and stochastic methods.

10.6 Summary of Backpropagation Steps

1. **Forward Pass**:
   - Compute activations for each layer.
   - Compute the output of the network.
   - Calculate the loss using the loss function.
2. **Backward Pass**:
   - Compute the gradient of the loss with respect to the output.
   - Propagate the gradients back through the network using the chain rule.
   - Compute gradients with respect to weights and biases at each layer.
3. **Update Weights and Biases**:
   - Adjust the weights and biases using the computed gradients and the learning rate.
4. **Iterate**:
   - Repeat the forward and backward passes for multiple epochs until convergence.

10.7 Visual Illustration

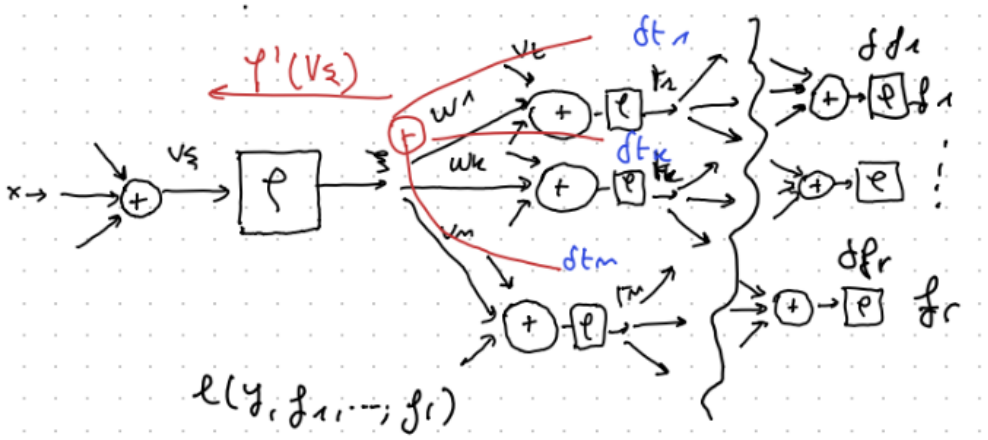


Figure 5: Illustration of the Backpropagation Process

Figure 5 shows the flow of information during the forward and backward passes. During the forward pass, activations are computed layer by layer. In the backward pass, error gradients are propagated from the output layer back to the input layer, updating weights and biases accordingly.

11 Lecture 11

Last time: backpropagation

Forward:

$$\ell(_l) = l(\phi(w_l \xi_{l-1} + b_l)) \tag{40}$$

$$\phi(w_l \xi_{l-1} + b_l = xi_l \tag{41}$$

$$w_l \xi_{l-1} + b_l = v_l \tag{42}$$

Backward:

$$\nabla \xi_{l-1} \ell = w_l(\phi'(vl) \nabla \xi_l \ell) = w_l^T \delta_l \tag{43}$$

Gradients:

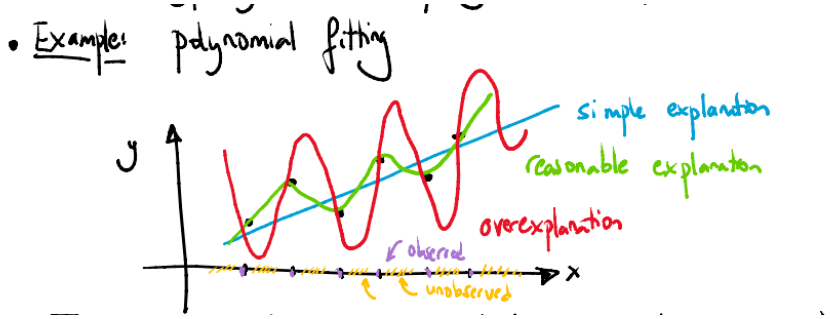
$$\nabla_w \ell = (\phi'(v_l) \nabla \xi_l \ell) \xi_l^t = \delta_l \xi_{l-1}^T \tag{44}$$

$$\nabla_b \ell = \delta_l \tag{45}$$

11.1 Overfitting

It is the tendency to **overexplain** the training data. Overexplaining means adding too much complicated and useless details.

Example: polynomial fitting



The opposite of overfitting is **underfitting**, when your model is to simple and it is not able to fit your data; on the other hand, when overfitting occurs, your model is perfectly capable of fitting your data, but lacks of generalization.

If you use a simple explanation you have a small variance, but high bias. With more complicated explanations, you lower the bias because you are able to catch the complexity of your data, but you increase the variance.

Typical problem to this: high sensitivity to changes in data. When learning, we have been focusing on observed contexts, but we mostly care about performance on new **unobserved contexts**. Performing well in unobserved contexts is called **generalization** and overfitting usually prevent it.

How do we quantify it?

Training Data $= \{(x, y), \dots\} \rightarrow$ use this to build the model

Testing Data $= \{(x', y'), \dots\} \rightarrow$ use this to assess generalization, **but** we must be very careful! We can't reuse test data, otherwise we'll overfit them too.

The train risk is not a good indicator of the model generalization capability; we must also look at the test risk; when it starts to increase, that's when overfitting occurs. Generally, if we have more data available, we can afford trying to overexplain it, but at a certain point overfitting will start again.

11.1.1 Observations

- stopping earlier can alleviate overfitting. Why? The neural network won't fit the noise.
- How? By staying close to its initialization. **Inductive bias** $\rightarrow$ preferring simple explanations.
- More data tends to help, but often it is not an option.
- **early stopping**, but we don't want to look at the test data

11.2 Cross-validation

Since training risk is not a good **proxy** to test the risk, we could hold out some of the training data for **validation**.

Training Data: use it to train the model (often.
Validation Data: use it to check if okay (sometimes)
Test Data: test on this (once).

What is validation good for? You can choose what epoch to stop at, and it allows you to choose among possingle architectures, learning rates... (you choose the combination that provides the lowest validation risk). These choices are **coarser** than choosing weights and biases: they are called **hyperparameters**. Validation is used to adjust these parameters.

A common way to perform cross validation is the **k-fold cross-validation**: data are divided into k blocks. The model is trained on k-1 blocks and validated on block k. The process is repeated k times. The obtained values are averaged, to obtain a more accurate loss and a less noisy proxy. The extreme case for this is **leave-one-out cross-validation (LOOCV)**

12 Lecture 12

12.1 Regularization

Cross-validation helps to **monitor** overfitting, while regularization helps to **prevent** it. It is a concept that comes from physics, from **inverse problems**.

Regularization: collapse the possible explanations in fewer explanations. We need to be able to favor some of the solutions over the others. Ideally, we would like to favour 1. In physics we tend to favour simpler ones → **Occam's Razor** (choosing simple explanations). For example, in physics we prefer low energy because it is more likely for a system to go to a lower energy state. For polynomials and nueral networks we would favour **smaller weights** ("low energy" weights).

$$\min_w R(w), \text{ for } ||w||_2^2 \leq C \tag{46}$$

We want the **euclidean norm** to be smaller than a threshold. This condition is the regularization part.

In a polynomial, when weights are smaller, we obtain a **smoother curve**. This is a version of multiple objective minimization and usually we see another version of the problem.

$$\min_w R(w), \text{ for } ||w||_2^2 \leq C \longleftrightarrow \min_w R(w) + \lambda||w||_2^2 \tag{47}$$

This is a **regularized empirical risk minimization**: a penalty term is added to penalize too large weights. This is called **weight decay**. L^2 regularization is also known as **ridge regression** or **Tikhonov regularization**. The smoothness prevents overfitting the noise, so the empirical risk and regularization become better representatives of the test set. Together, they better track the empirical risk.

How does this affect backpropagartion?

$$\nabla_w(R(w) + \lambda||w||_2^2) = \nabla_w R + 2\lambda \cdot w \tag{48}$$

$$w \leftarrow w - \eta(\nabla_w R + 2\lambda w) \tag{49}$$

$$w \leftarrow (1 - 2\eta \cdot \lambda) \cdot w - \eta \nabla_w R \tag{50}$$

$$1 - 2\eta \cdot \lambda = \epsilon \tag{51}$$

Usually, ϵ is smaller than $1 \rightarrow$ in front of w we have something smaller than 1 and this term in ML is called weight decay. The addition of the weight decay term has modified the learning rule to multiplicative shrink the weight vector by a constant factor on each step.

$$w = (w_1, w_2, w_3) \tag{52}$$

$$||w||_2 = (w_1^2 + w_2^2 + w_3^2)^{\frac{1}{2}} \tag{53}$$

$$||w||_p = (w_1^p + w_2^p + w_3^p)^{\frac{1}{p}} \tag{54}$$

$$||w||_1 = |w_1| + |w_2| + |w_3| \tag{55}$$

If we change the L^2 norm with the L^1 norm, we obtain:

$$\min_w R(w) + \lambda||w||_1 \tag{56}$$

$$\nabla_w(R(w) + \lambda||w||_1) = \nabla_w R + \lambda \cdot \text{sign}(w) \tag{57}$$

L1 norm is defined as the **summation of absolute values** of a vector's all components.

The L^1 norm for **w** to be sparse $\rightarrow$ will produce a sparse solution. It is also known as **Lasso regularization**.

L1 Norm:

- Encourages sparsity in the weight vector . The sparsity property induced by L^1 regularization has been used as **feature selection mechanism**.
- Leads to many weights being exactly zero, effectively performing feature selection.
- The gradient involves the sign of each weight, promoting weights to either stay zero or become non-zero.

The L^1 norm is different from L^2. This regularization contribution to the gradient is no longer scales linearly with each w_i; instead it is a constant factor with a sign equal to $\text{sign}(w_i)$

L2 Norm:

- Encourages smaller, but generally non-zero, weights.
- Does not inherently promote sparsity.
- The gradient is proportional to the weight value, leading to a smooth decay of weights.

Why L1 Norm Introduces Sparsity: The L1 norm creates a geometry with sharp corners (like a diamond shape in two dimensions). When minimizing the loss, the optimization process is more likely to hit these corners where one or more weights are exactly zero. This results in sparse solutions where only a subset of features contribute to the model, enhancing interpretability and reducing overfitting by eliminating irrelevant features.

In contrast, the L2 norm has a spherical geometry, promoting solutions where all weights are small but typically non-zero, distributing the penalty more evenly across all parameters.

L1 regularization forces the model to be "dumber", so instead of simply memorizing stuff, it has to look for simpler patterns from the data.

12.2 Other types of regularizations

- Regularization can be viewed as incorporating prior beliefs about the model parameters. For example, in L2 regularization, we assume that smaller weights are a priori more likely, reflecting a preference for simpler models.
- **Invariance in Vision**:
  - **Concept of Invariance**: In computer vision, certain transformations of an image (such as rotation, translation, or scaling) do not alter the fundamental content. For instance, rotating a picture of a cat still results in a cat. This property is known as **invariance**.
  - **Equivalence Classes**: Due to invariance, a single data point can represent an entire class of equivalent points transformed by invariant operations. For example, an image and its rotated versions belong to the same equivalence class.
  - **Data Augmentation as Regularization**: Instead of modifying the model to be invariant (e.g., using convolutional neural networks that are inherently translation-invariant), another approach is to **augment the data**. This involves generating synthetic data points by applying invariant transformations to existing data. By training on these augmented datasets, the model learns to generalize better, effectively acting as a form of regularization by exposing the model to a wider variety of inputs.
  - **Benefits of Data Augmentation**:
    - Enhances model robustness to variations in input data.
    - Increases the effective size of the training dataset, reducing overfitting.
    - Encourages the model to focus on invariant features rather than spurious details.
- **Resilience in Neural Networks**:

- **Biological Inspiration**: Just as the human brain remains functional even if some neurons are damaged or removed, artificial neural networks can benefit from resilience to ensure reliability and robustness.
- **Dropout Regularization**: A popular technique to introduce resilience involves **dropout**, where during training, each neuron (or weight) is randomly "dropped out" with a probability p. This means setting the neuron's output to zero for that iteration.
  - **Mechanism**:
    - At each training step, independently flip a biased coin with probability p for each neuron.
    - If the coin is heads, set the neuron's output to zero for that forward and backward pass.
    - During testing, use all neurons but scale their outputs by $(1 - p)$ to account for the dropped neurons during training.
  - **Effects**:
    - Prevents the network from becoming overly reliant on specific neurons.
    - Encourages the network to develop redundant representations, enhancing generalization.
    - Acts as a form of ensemble learning, where multiple subnetworks are implicitly trained and averaged.

13 Lecture 13

Last Time: Regularization.

- **Idea:** reduce the space of neural networks that we can choose. Avoid overexplanation by limiting to **reasonable** explanations → prior knowledge/constraints on weights and biases.

$$\min_w R(w) \text{ for } ||w||_2 \leq C \leftrightarrow \min_w R(W) + \lambda ||w||_2^2 \tag{58}$$

The term in blue is the regularization/penalty

$$\nabla_w [R(w) + \lambda ||w||_2^2] = \nabla_w R(w) + 2\lambda w \tag{59}$$

Another possibility is to use the L1 norm (sum of the absolute values of each coordinate). In this case, the gradient becomes:

$$\nabla_w R + \lambda \text{sign}(w) \tag{60}$$

- **Data augmentation**
- **Drop Out:** At train time, remove edges with probability p, forward/backward. At test use all the edges, but weight them by p.

13.1 Optimization

13.1.1 Stochastic Gradient Descent

Empirical Risk:

$$R(w) = \frac{1}{\text{Data}} \sum_{(x,y)\in \text{Data}} l(y, f(x, w)) \tag{61}$$

True Risk: $R'(w) = E[l(y, f(x, w)]$: this expected value is an average over an infinite number of test sets.

What we really care about is minimizing this. What is the relationship between them? The true risk takes into account an infinite number of data points. The first one is a **sample average**, while the second is a **population average**.

- ***Population Average:*** *is the average of all possible observations in a full population. The population refers to the entire group of individuals or data points you are interested in studying*
- ***Sample Average:*** *average of a subset of the population (a sample). Since it's actually impractical to measure the entire population, we often use a sample to estimate the population's characteristics.*

The expected value of the estimate R(w) is an unbiased estimate of R'(w) $\rightarrow E[(R(w)] = R'(W)$; that means, we would like to do the gradient descent on R'(w), but also $\nabla R(w)$ is an unbiased estimate of $'(w)$.

$$E[\nabla_w R(w)] = \nabla_w R'(w) \tag{62}$$

The gradient of the empirical risk is random, but in expectation is equal to the gradient of the risk.

But, this remains true even for single data points:

$$E[\nabla_w l(y, f(x, w))] = \nabla_w R'(w) \tag{63}$$

or for a minibatch of points:

$$E[\frac{1}{B} \sum_{(x,y)\in B} \nabla_w l(y, f(x, w))] = \nabla_w R'(w) \tag{64}$$

They are all unbiased. What changes is the variance. Single data points have the highest variance, while all the dataset has the lowest. Minibatches represent a trade off.

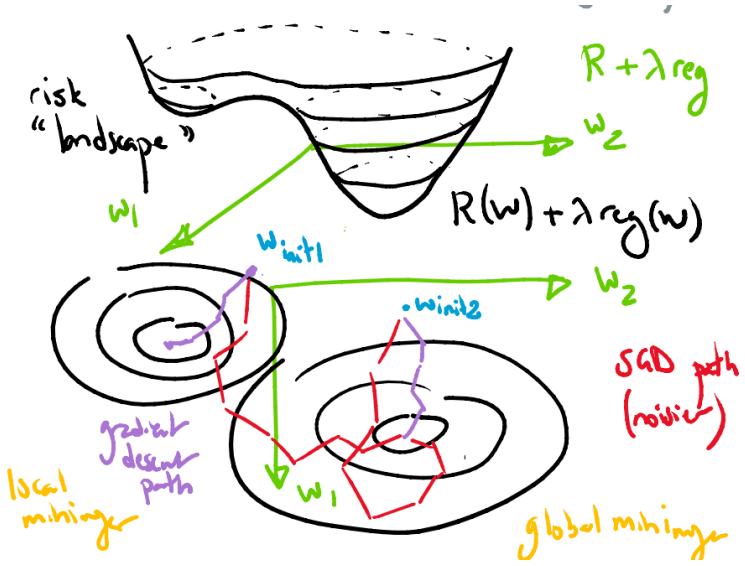
$$\text{Points} \downarrow \quad \text{Variance} \uparrow \quad \text{Noiseness} \uparrow$$

Why would we not choose the lowest variance, i.e. all of the data?

- Computational Advantage: it takes much less time to get an update with small batches
- The justification comes from 2 places:
  - statistics: homogeneous data $\rightarrow$ small batches give a glimpse of the whole
  - optimization: The variance can be helpful to get out of local minima

General Form of SGD:

for epoch in epochs **do**
  $t = 0$; $\sum \nabla = 0$ (sum of the gradients)
  for (x, y) in data **do**
    forward(x)
    backward(xw) (the backward equation gives us the gradients)
    $\sum \nabla = \nabla R + \lambda \nabla_{\text{reg}}$
    $t+ = 1$
    if $(t+1)\%B == 0$ **then**
      $w+ = \eta \frac{\sum \nabla}{B}$
      $\sum \nabla = 0$ (zero out gradients again)
    end if
  end for
end for



It is going to see a noisy version of the gradient, but this noiseness can actually help to kump out of local minima.

13.1.2 Choice of η, the "learning rate"

Because of noiseness in SGD, η has to be adjusted along iterations. In theory, we can schedule η to decay with iterations. In practice don't change η, unless things break down ($R(w) \uparrow$). If something breaks down, adjust η by a factor, for example $\eta' = 0.90\eta$.

Another possibility is to adjust η differently along layers. Later layers tend to have ∇w larger, so we can make $\eta \downarrow$. Neuron by neuron inputs $\uparrow$, so we can make $\eta \downarrow \propto \frac{1}{\sqrt{\text{inputs}}}$ (heuristic).

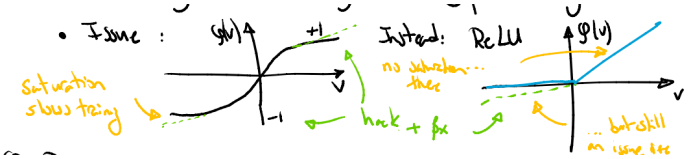
13.1.3 Controlling Stability

Because gradients are repeatedly multiplied during backpropagation, they could vanish, or they could also explode. We can normalize the gradients (across layer or batch), or we can clip them (everything bigger than th is equal to th).

13.2 Heuristic

13.2.1 Choosing activation function

Sigmoid and tanh were popular, but they suffer of vanishing gradients issue. An alternative is represented by ReLu activation function. It has still the problem for x ¡ 0, but for $x \geq 0$ the vanishing gradients problem disappears because this activation function does not saturate. Regaarding the negative part, we can help by adding a tilt in the negative part of the axis.



13.2.2 Preprocess inputs

Ideally, we want our inputs to be uncorrelated/centered, otherwise this may slow things down. What if for example inputs are always + or - together? We could move only in two directions.

We usually preprocess them to make them 0 mean, unit variance and zero covariance. This process is known as **whitening**. How?

$$\mu = \frac{1}{\text{Data}} \sum_{(x,y)\in \text{Data}} x \tag{65}$$

$$C = \frac{1}{\text{Data}} \sum_{x\in \text{Data}} (x-\mu)(x-\mu)^T \tag{66}$$

C is the covariance matrix and it is positive semidefinite, so we can have eigenvalues decomposition.

$$C = UDU^T$$

$$UU^T = I$$

Let $x_{\text{new}} = D^{\frac{-1}{2}} U^T (x-\mu)$.
We divide by the square root of the variance, to obtain an unit variance. Now the covariance matrix is identity.

14 Lecture 14

Last time: Optimization (minibatch SGD). Sometimes we need some Heuristic to make the learning process go well. We need to make sure that there is no redundances. We want to whiten the inputs to make them uncorrelated.

14.1 Initialize the Weights

Intuition: Maintain similar statistics across layer to layer. At least at initialization. Ignore activation, assume white inputs (o mean, covariance identity (I)). If there is no bias, the mean condition will be automatically satisfied: $z = Wx + \not{b}$. The variance of z is computed as follows:

$$C = \frac{1}{\text{Data}} \sum_{x \in \text{Data}} (Wx)(Wx)^T \tag{67}$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} a & b & c \end{bmatrix} = \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix} \tag{68}$$

On the off diagonal, we will have $2 \neq w_{ij}$'s $\rightarrow$ we want them to be zero so we need to make different w_{ij}'s uncorrelated. On the diagonal we have the sum of squares of each row $\rightarrow n\text{Var}(w_{ij}$. How do we make sure that $E[ww^T] = I$? If we make the guy on the diagonal equal to 1, that will be the case $\rightarrow$ we need $\text{Var}(w_{ij}) = \frac{1}{n}$. The easiest way to do that is:

$$w_{ij} \sim \frac{N(0,1)}{\sqrt{\text{inputs}}} \tag{69}$$

We sample w_{ij} to be random variables with gaussian distribution and divide by the square root of the number of inputs. This procedure is really helpful, specifically at the beginning of training.

14.2 Convolutional Neural Networks

A linear system takes a signal and produces an output linearly proportional to the input. It is not necessarily true that if we shift the input in time, the output will be shifted in time too (this is not true for all linear systems). This is the idea behind **Convolution**. We feed into the system an impulse and we look how the system will behave because of this impulse.

Convolution is time-invariant, meaning that the output depends only on the relative positions of the input functions, not on their absolute positions.

In image processing (but also in signal processing), we can use convolution to perform **filtering**.

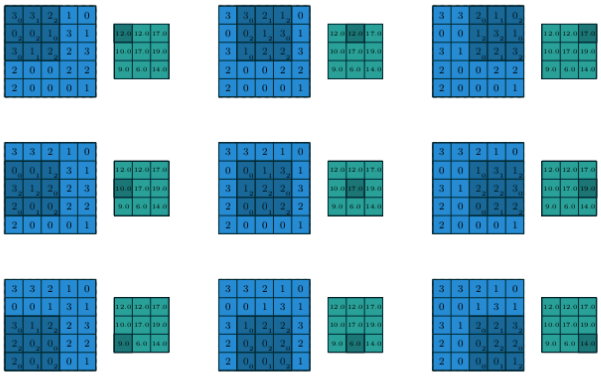


Figure 1.1: Computing the output values of a discrete convolution.

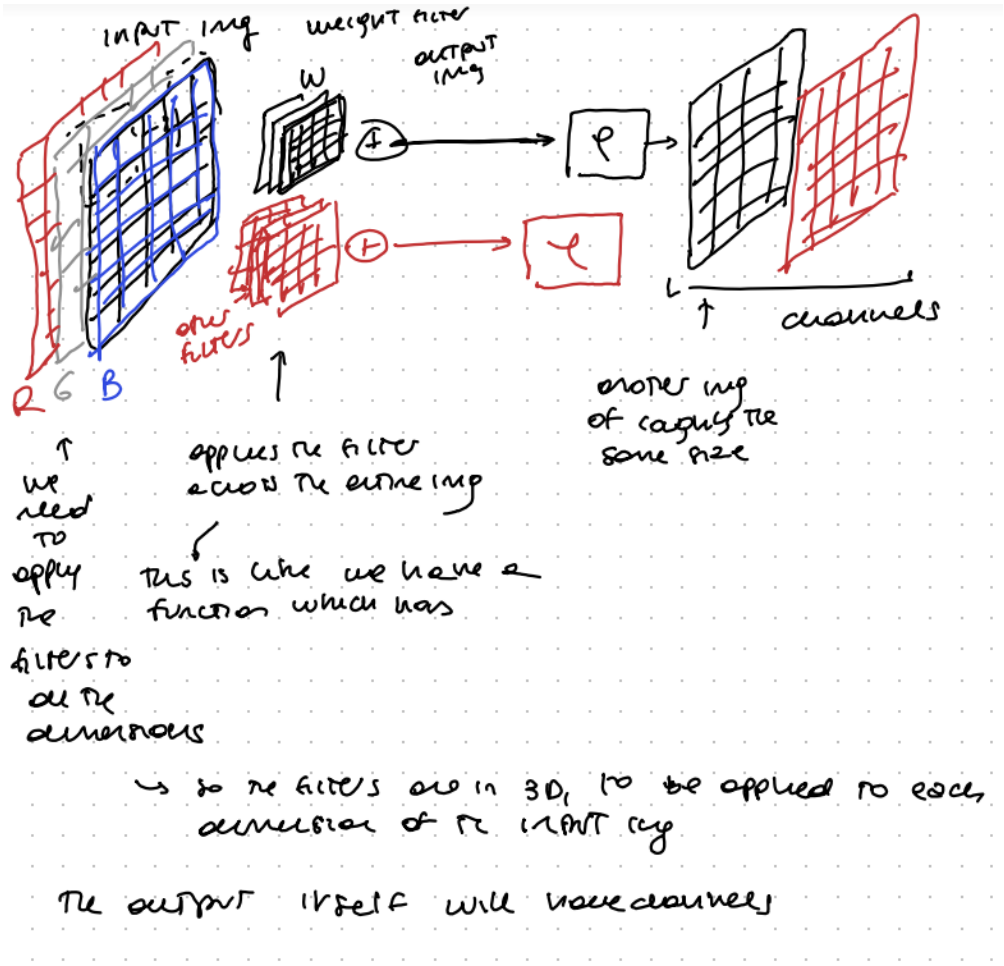
smoothing Filter It takes the pixel in the center, add $\frac{1}{2}$ of the surroundings and the $\frac{1}{4}$ of the 4 others. Now the pixel output does not depend only on itself, but also on the surrounding. It is the equivalent of a low pass filter.

$$\begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix} \tag{70}$$

Sharpening Filter

$$\begin{bmatrix} 0 & \frac{-1}{4} & 0 \\ \frac{-1}{4} & 1 & \frac{-1}{4} \\ 0 & \frac{-1}{4} & 0 \end{bmatrix} \tag{71}$$

Images are stored as multi-dimensional arrays, they feature one o more axes for which ordering matters (e.g. width and height axes); one axis, called the channel axis, is used to access different views of the data (e.g. the red, green and blue channelsof a color image). A discrete convolution is a linear transformation that preserves this notion of ordering: it is **sparse** (only a few input units contribute to a given output unit) and **reuses parameters** (the same weights are applied to multiple locations in the input). Convolutional neural networks act locally and have the shift-invariance property. CNN's are very useful for vision. CNN's pay attention in a block by block way, while transformers have the flexibility of looking around.


INPUT IMG
WEIGHT FILTER
W
OUTPUT IMG
other filters
R G B
channels
another img of roughly the same size
applies the filter across the entire img
we need to apply the filters to all the dimensions
This is like we have a function which has
↳ so the filters are in 3D, to be applied to each dimension of the INPUT img
The output itself will have channels