Module C: Linear Models and Neural Networks

Abdulmalek Al-Gahmi, PhD

October 2, 2023

Useful mathematical results

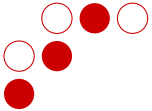
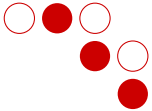
- **The chain rule**: if $F = (f \circ g)$ then $F' = (f \circ g)' = (f' \circ g) \cdot g'$. Or:

$$\frac{df}{dx} = \frac{df}{du} . \frac{du}{dx}$$

- If $f = c$ where c is constant, then $f' = 0$
- If $f = x^p$ then $f' = p\, x^{p-1}$
- If $f = ln(x)$ then $f' = \frac{1}{x}$
- If $f = log_b(x)$ then $f' = \frac{1}{x\, ln(b)}$
- If $f = e^x$ then $f' = f = e^x$
- If $f = g.h$ then $f' = g.h' + g'.h$
- $log(\prod_{i=1}^{n} f(x_i)) = \sum_{i=1}^{n} log(f(x_i))$
- **The normal (Gaussian) distribution**: $N(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$

Outline

- Linear regression
- Regularized linear models
- Logistic regression
- The Perceptron
- Neural networks
- Introduction to deep learning

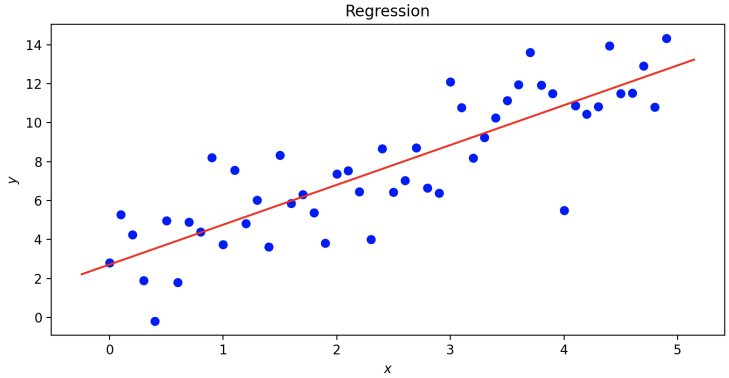
Linear Regression

Simple linear regression

When there is a single feature x, the linear regression model is written as:

$$h(x) = w_0 + w_1 x_1$$

where w_0 and w_1 are the weight parameters we are trying to learn.



This gives us a regression line whose intercept is w_0 and whose slope is w_1.

Multiple linear regression

The linear regression model is written as:

$$h(x) = b + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$$

where b is the bias and $w_1, \cdots, w_m$ are the weight parameters we are trying to learn. Notice that $b = w_0$. We can express this model using the matrix notation as:

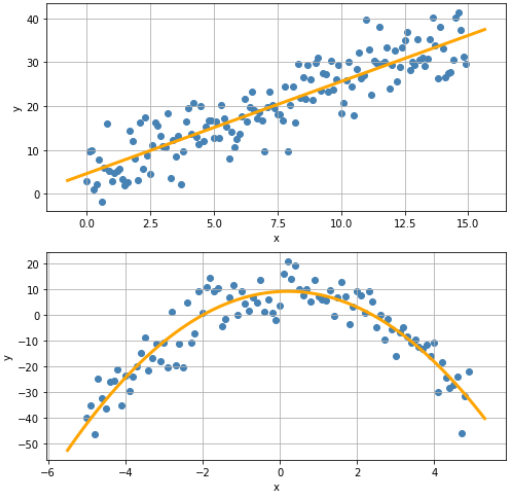
$$h(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$$

where

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

What can **x** be?

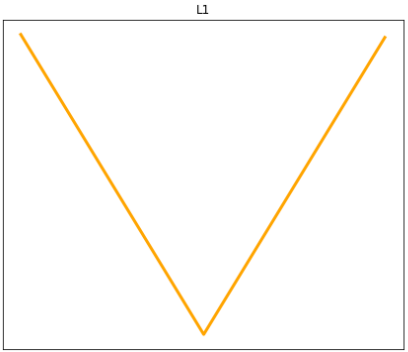
- Quantitative inputs
- Transformations of quantitative inputs. For example: $log(x_i)$ or $\sqrt{x_j}$
- Non-linear expansions of **x**. For example: $x_i^2, x_i^3, \cdots$
- Input interactions. For example: $x_i \times x_j$.
- Numeric "dummy" coding of qualitative inputs. For example: **one-hot encoding**.

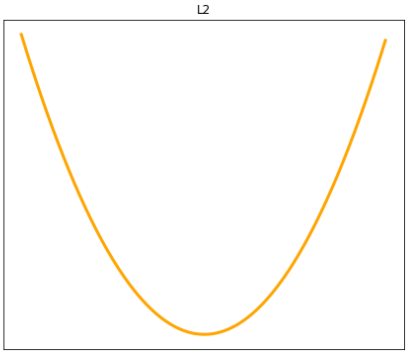


Fitting the data: L1 and L2 losses

Finding the "best" line/hyperplane requires a **loss** or **cost** function. Two popular loss functions are:

- The sum of absolute errors: $L_1 = \sum_{i=1}^{n} |y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)}|$
- The sum of squared errors: $L_2 = \frac{1}{2}\sum_{i=1}^{n} (y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})^2$





The L_2 loss function

- The most common loss function for regression.
- Has nice mathematical properties:
  - It's differentiable at all points.
  - It's convex; it has a global minimum.
- The form

$$L_2 = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - h(\mathbf{x}^{(i)}))^2 = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})^2$$

  can be re-written in matrix notation as:

$$L_2 = \frac{1}{2}(\mathbf{y} - h(\mathbf{X}))^T(\mathbf{y} - h(\mathbf{X})) = \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})$$

The probabilistic interpretation of L_2

We start by making a few assumptions:

- $\mathbf{y} = h(\mathbf{X}) + \epsilon = \mathbf{X}\mathbf{w} + \epsilon$ where ϵ captures any unmodled effects.
- Examples are I.I.D.
- For a single example *i*, $\epsilon^{(i)} = N(0, \sigma^2)$ Thus:

$$p(y^{(i)}|x^{(i)}; \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{(y^{(i)} - w^T\mathbf{x}^{(i)})^2}{2\sigma^2})$$

- It can be shown that by minimizing the $L2$ loss function, we are maximizing the likelihood of $\mathbf{y}$ being the predicted values/classes given the input $\mathbf{X}$.

Linear regression via minimizing L_2

A linear regression model can be trained in two ways:

- Using the normal equation

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

  where the $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ is called the pseudo-inverse of $\mathbf{X}$

- Using gradient descent:
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent

The normal equation

Given the $L_2 = \frac{1}{2}(\mathbf{y} - h(\mathbf{X}))^T(\mathbf{y} - h(\mathbf{X})) = \frac{1}{2}(\mathbf{y} - \mathbf{Xw})^T(\mathbf{y} - \mathbf{Xw})$, we can get the weights $\mathbf{w}$ for the L_2 error by solving the equation:

$$\frac{\delta L_2}{\delta \mathbf{w}} = 0$$

That means:

$$\frac{1}{2}[-(\mathbf{y} - \mathbf{Xw})^T\mathbf{X} - (\mathbf{y} - \mathbf{Xw})^T\mathbf{X}] = 0$$

$$\frac{1}{2}[-2(\mathbf{y} - \mathbf{Xw})^T\mathbf{X}] = 0$$

$$\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{Xw} = 0$$

$$\mathbf{X}^T\mathbf{y} = \mathbf{X}^T\mathbf{Xw}$$

$$\mathbf{X}^T\mathbf{Xw} = \mathbf{X}^T\mathbf{y}$$

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

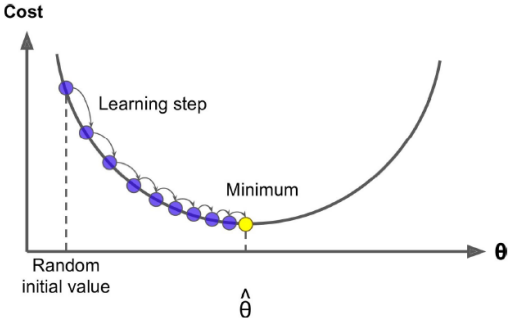
where the $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ is called the pseudo-inverse of $\mathbf{X}$

Gradient descent

Allows us to find the values of $\mathbf{w}$ that minimizes a loss function $L(\mathbf{w})$:

$$\mathbf{w} = \mathbf{w} - \eta \nabla L(\mathbf{w})$$

where η is the learning rate and $\nabla L(\mathbf{w})$ is the gradient (a vector of partial derivatives: one for each weight parameter w_i).



Adopted from [1]

Gradient descent with L2 loss

Given a loss function $L(\mathbf{w})$, we can try to find $\mathbf{w}$ that minimizes the loss function using gradient descent:

$$\mathbf{w} = \mathbf{w} - \eta \nabla L(\mathbf{w})$$

where η is the learning rate and $\nabla L(\mathbf{w})$ is the gradient (a vector of partial derivatives one for each w_i). For a single weight w_i given a single example $(\mathbf{x}, y)$.

$$w_i = w_i - \eta \frac{\delta L(\mathbf{w})}{\delta w_i}$$

If L is L_2 then:

$$w_i = w_i - \eta \frac{\delta \ \frac{1}{2}(y - \mathbf{w}^T\mathbf{x})^2}{\delta w_i}$$

$$w_i = w_i - \eta[-\frac{1}{2}2(y - \mathbf{w}^T\mathbf{x})x_i]$$

$$w_i = w_i + \eta(y - \mathbf{w}^T\mathbf{x})x_i$$

Gradient descent (GD): A five-step implementation process

STEP 1: Initialize the weight parameters **w**.
while not converged: `# the training loop`
STEP 2: Calculate the predicted output $\hat{y}$
STEP 3: Calculate the loss L as a function of the actual output y and the predicted output $\hat{y}$.
STEP 4: Calculate the gradients of the loss function with respect to the weights **w** parameters.
STEP 5: Update the values of the weights **w** parameters:

$$\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} L(y, \hat{y})$$

where η is the learning rate.

Exercise: What is the running time of this algorithm?

Batch gradient descent

Require: Training dataset $\mathbf{D}$ consisting of n examples with inputs $\mathbf{X}$ and targets $\mathbf{y}$.
Require: Learning rate η and initial parameters θ
  while stopping criterion not met **do**
    $\mathbf{g} \leftarrow \frac{1}{n} \nabla_\theta \sum_{i=1}^{n} L(y^{(i)}, \hat{y}^{(i)})$
    $\theta \leftarrow \theta - \eta \mathbf{g}$
  end while

Batch gradient descent goes through the entire training dataset before a single weight update takes place. It is expensive but it is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces. A complete pass over the whole training set is called an **epoch**.

Stochastic gradient descent (SGD)

Require: Training dataset $\mathbf{D}$ with inputs $\mathbf{X}$ and targets $\mathbf{y}$.
Require: Learning rate η and initial parameter θ
  while stopping criterion not met **do**
    for **do**$(\mathbf{x}, y)$ in $\mathbf{D}$
      $\mathbf{g} \leftarrow \nabla_\theta \sum_{i=1}^{m} L(y, \hat{y})$
      $\theta \leftarrow \theta - \eta \mathbf{g}$
    end for
  end while

As SGD goes through the entire training set, it updates the weight θ vector after each example. It is much faster but it fluctuates and can be unstable. Its fluctuation, however, enables it to jump to new and potentially better local minima.

Mini-batch gradient descent

Combines the best of both batch and Stochastic gradient descent. It breaks the dataset **D** into batches of size B (32, 50, 64, etc), an performs a weight update after each batch.

Require: Training dataset **D** with inputs **X** and targets **y**.
Require: Learning rate η and initial parameter θ
  while stopping criterion not met **do**
    Sample a minibatch of m examples from **D**
    $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)})$
    $\theta \leftarrow \theta - \eta \mathbf{g}$
  end while

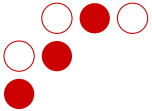
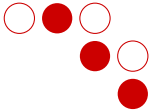
As expected, it is more efficient than SGD, more stable, and fluctuates less.

Model evaluation: Coefficient of determination R^2

- A large value for R^2 indicates a strong linear corelation between independent (input) and dependent (output) variables.
- It indicates how much of the variability of the output variable is explained or accounted for by the model.
- Ideally we like its values to be between 0 and 1, but they can also be negative.
- It is defined as:

$$R^2 = 1 - \frac{\text{Residual Square Sum}}{\text{Total Square Sum}} = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{n} (y^{(i)} - \bar{y})^2}$$

where $\bar{y}$ is the mean of y.

Regularized Linear Models

Regularization and overfitting

- Regularization is a technique used to prevent overfitting in machine learning models.
- Overfitting occurs when a model learns the training data too well, capturing noise or random fluctuations rather than the underlying patterns.
- Regularization adds a penalty term to the model's loss function, discouraging the model from assigning excessive importance to any one feature or having excessively large weights.
- This makes the model less flexible and helps it generalize better to unseen data. The fewer the degrees of freedoms it has , the harder it will be for it to overfit.
- The term "degrees of freedom", here, refers to the number of parameters (weights) that the model can adjust during training.

Regularization in linear models

- In linear models, regularization is achieved by adding a penalty term to the loss function that depends on the model's weights.
- The goal is to make the weights smaller, effectively constraining their values.
- There are three common types of regularization techniques in linear models:
  - **Ridge regression** adds an $L2$ regularization term to the loss function, which penalizes the sum of squared weights.
  - **Lasso regression** adds an $L1$ regularization term to the cost function, penalizing the absolute values of weights.
  - **Elastic net regression** combines both $L1$ (Lasso) and $L2$ (Ridge) regularization terms in the loss function.

Ridge (L2) regression

- Uses an $L2$ regularization term to the loss function, which penalizes the sum of squared weights.
- Adds the following term to the loss function:

$$\frac{\alpha}{m}\sum_{j=1}^{m} w_j^2$$

  where m is the number of input features.
- Does not apply regularization the intercept weight w_0.
- Encourages the weights to be small but doesn't force them to be exactly zero. This called **weight shrinkage**.
- Is particularly useful when there are many correlated features.

Lasso (L1) regression

- Uses an $L1$ regularization term to the loss function, which penalizes the absolute values of weights.
- Adds the following term to the loss function:

$$2\alpha \sum_{j=1}^{m} |w_j|$$

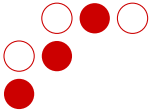
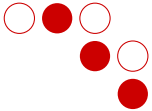
  where m is the number of input features.
- Can drive some weights to exactly zero, effectively selecting a subset of the most important features. This called **feature selection**.
- Is particularly useful when there are many features some of which may not be important.

Elastic net regression

- Combines both $L1$ (Lasso) and $L2$ (Ridge) regularization terms in the loss function.
- Adds the following terms to the loss function:

$$r(2\alpha \sum_{j=1}^{m} |w_j|) + (1 - r)\frac{\alpha}{m} \sum_{j=1}^{m} w_j^2$$

  where m is the number of input features and r is the l1 ratio.
- Provides a balance between feature selection and weight shrinkage.
- Is useful when there are many features, and some of them are correlated.

Logistic Regression

Odds, logit (or log-odds), and propabilities

- The odds in favor of a particular event is defined as:

$$odds = \frac{p}{1-p}$$

  where p is the probability of the positive event.
- The **logit** function, which is also called the **log-odds** is defined as:

$$logit(p) = log(\frac{p}{1-p})$$

- The inverse of the logit function is the **logistic** or **sigmoid** function:

$$p = \sigma(z) = \frac{1}{1+e^{-z}}$$

  where z is the log-odds.
- The logistic regression model assumes there is a linear relationship between the weighted inputs and log-odds. In other words:

$$\mathbf{z} = \mathbf{Xw}$$

Logistic regression

We can generalize the linear model $h(\mathbf{X})$ by thinking of it as:

$$h(\mathbf{X}) = g(\mathbf{X}\mathbf{w}) = g(z)$$

For linear regression:

$$g(z) = z$$

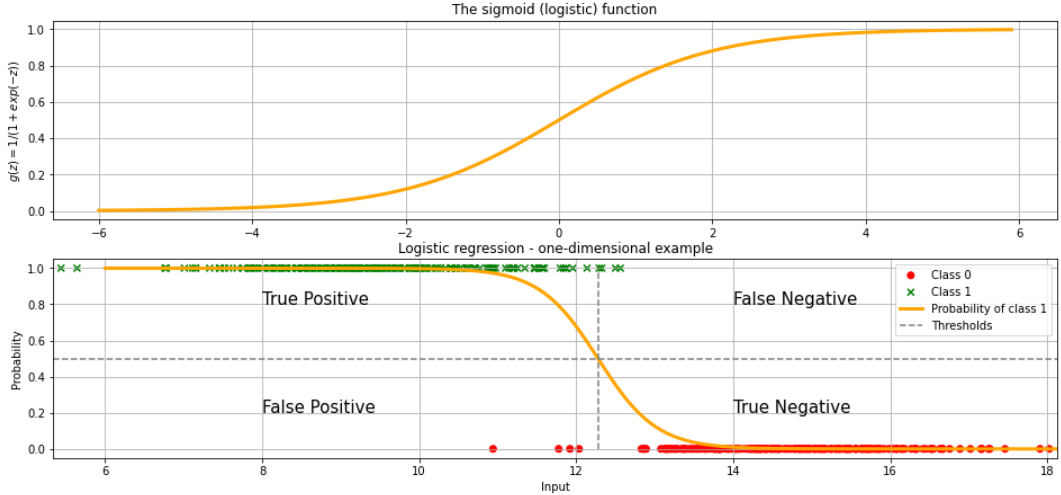
For logistic regression:

$$g(z) = \frac{1}{1 + e^{-z}}$$

where g is the **logistic** or **sigmoid** function.

IMPORTANT NOTE:: Despite the name, **logistic regression** is used for classification; not regression.

An example


The sigmoid (logistic) function
$g(z) = 1/(1 + exp(-z))$
Logistic regression - one-dimensional example
Probability
Input
True Positive
False Negative
False Positive
True Negative
Class 0
Class 1
Probability of class 1
Thresholds

The logistic (sigmoid) function

- With

$$h(\mathbf{X}) = \frac{1}{1 + e^{-\mathbf{X}\mathbf{w}}}$$

  $h(\mathbf{X}) \in [0, 1]$

- It can be interpreted as a probability:

$$p(y|\mathbf{x}; \mathbf{w}) = \begin{cases} h(\mathbf{x}) & \text{if } y = 1 \\ 1 - h(\mathbf{x}) & \text{if } y = 0 \end{cases}$$

  which can be written as:

$$p(y|\mathbf{x}; \mathbf{w}) = h(\mathbf{x})^y (1 - h(\mathbf{x}))^{1-y}$$

- Its derivative is:

$$g'(z) = \frac{d}{dz} \frac{1}{1 + e^{-z}} = g(z)(1 - g(z))$$

The logistic regression update rule

Given: $p(y|\mathbf{x};\mathbf{w}) = h(\mathbf{x})^y(1-h(\mathbf{x}))^{1-y}$, we can write the likelihood of the weight parameters $\mathbf{w}$ as:

$$\begin{aligned} L(\mathbf{w}) &= p(\mathbf{y}|\mathbf{X};\mathbf{w}) \\ &= \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)};\mathbf{w}) \\ &= \prod_{i=1}^{n} h(\mathbf{x}^{(i)})^{y^{(i)}}(1-h(\mathbf{x}^{(i)}))^{1-y^{(i)}} \end{aligned}$$

It's more convenient mathematically to work with summations than products. So we use $log(L(\mathbf{w}))$.

$$\begin{aligned} log(L(\mathbf{w})) &= \sum_{i=1}^{n} log(h(\mathbf{x}^{(i)})^{y^{(i)}}(1-h(\mathbf{x}^{(i)}))^{1-y^{(i)}}) \\ &= \sum_{i=1}^{n} y^{(i)} log(h(\mathbf{x}^{(i)}) + (1-y^{(i)}) log(1-h(\mathbf{x}^{(i)}))) \end{aligned}$$

The logistic regression update rule (continued)

By maximizing $log(L(\mathbf{w}))$ we are also maximizing $L(\mathbf{w})$. To find the weight parameters $\mathbf{w}$ that maximizes $log(L(\mathbf{w}))$, use the gradient ascent with regard to $\mathbf{w}$. That is:

$$\mathbf{w} = \mathbf{w} + \eta \frac{\delta\, log(L(\mathbf{w}))}{\delta \mathbf{w}}$$

It can be shown for a single example $(\mathbf{x}, y)$ and a single weight w_i that:

$$\frac{\delta\, log(L(\mathbf{w_i}))}{\delta \mathbf{w}} = (y - h(\mathbf{x}))x_i$$

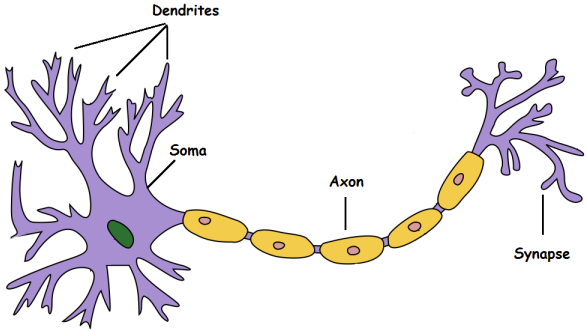
which gives us the update rule:

$$w_i = w_i + \eta(y - h(\mathbf{x}))x_i$$

The Perceptron

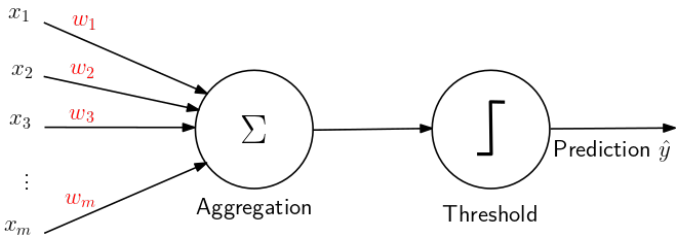
Biological inspiration

- The Perceptron and neural networks in general were biologically inspired. It came as a result of trying to understand how the brain works in order to design artificial intelligence.
- Biological neurons are interconnected nerve cells inside the brain that are involved in processing and transmitting chemical and electrical signals.



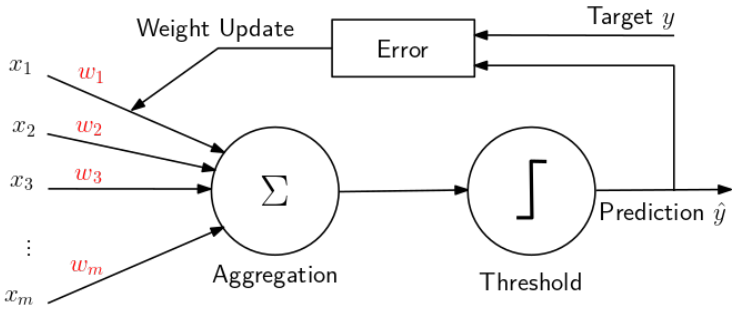
The McCulloch-Pitts (MCP) Neuron

Proposed by McCulloch and Pitts in 1943 [3].

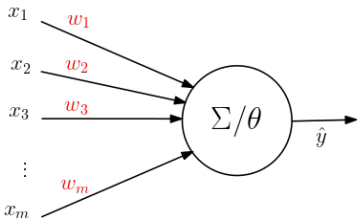


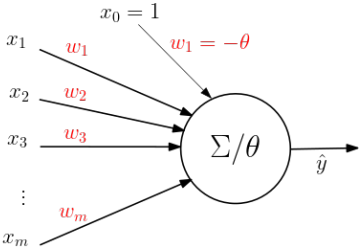
The Perceptron

- Based on the the McCulloch and Pitts (MCP) neuron
- Proposed by Frank Rosenblatt in 1957 [4]



How the Perceptron works





$$y = \begin{cases} +1 \text{ if } \sum_{i=1}^{m} w_i x_i \geq \theta \\ -1 \text{ if } \sum_{i=1}^{m} w_i x_i < \theta \end{cases}$$

which could be re-written as:

$$y = \begin{cases} +1 \text{ if } \sum_{i=1}^{m} w_i x_i - \theta \geq 0 \\ -1 \text{ if } \sum_{i=1}^{m} w_i x_i - \theta < 0 \end{cases}$$

$\Longrightarrow$

$$y = \begin{cases} +1 \text{ if } \sum_{i=0}^{m} w_i x_i \geq 0 \\ -1 \text{ if } \sum_{i=0}^{m} w_i x_i < 0 \end{cases}$$

where

$$x_0 = 1 \text{ and } w_0 = -\theta$$

The Perceptron update rule

For an example $(\mathbf{x}, y)$, we can write:

$$h(\mathbf{x}) = sign(\sum_{i=0}^{m} w_i x_i)$$

which can also be written as:

$$h(\mathbf{x}) = sign(\mathbf{w}^T \mathbf{x})$$

The Perceptron update rule is:

$$w_i = w_i + \eta(y - h(\mathbf{x}))x_i$$

The Perceptron algorithm

Require: Training dataset $\mathbf{D}$ consisting of n examples with inputs $\mathbf{X}$ and targets $\mathbf{y}$.
Require: Learning rate η, initial parameter $\mathbf{w}$
  function fit($\mathbf{X}, \mathbf{y}$)
    Initialize weights $\mathbf{w}$ to small random numbers
    while stopping criterion not met **do**
      for $j = 1$ to n **do**
        $h^{(j)} = sign(\mathbf{w}^T \mathbf{x}^{(\mathbf{j})})$
        $w_i = w_i + \eta(y^{(j)} - h^{(j)})x_i$ for $i = 0$ to m
      end for
    end while
  end function

  function predict($\mathbf{X}, \mathbf{y}$)
    return $sign(\mathbf{w}^T \mathbf{X})$
  end function

Using the Perceptron with logical functions

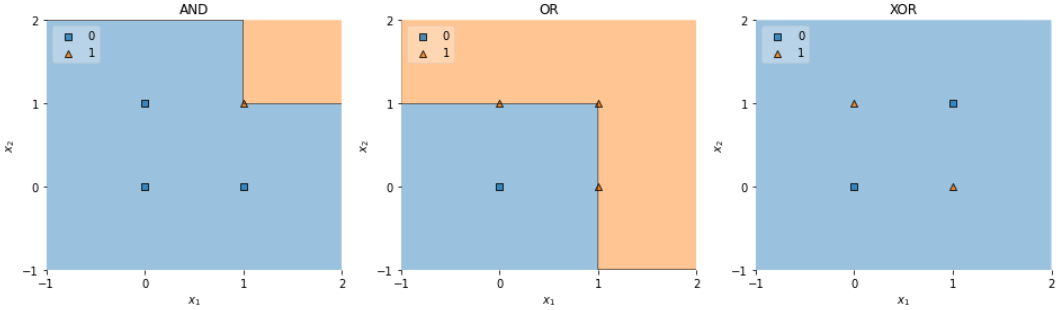
x_1	x_2	AND	OR	XOR
F	*F*	*F*	*F*	*F*
F	*T*	*F*	*T*	*T*
T	*F*	*F*	*T*	*T*
T	*T*	*T*	*T*	*F*

$\Longrightarrow$

x_1	x_2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

What do the Perceptrons that correctly these functions look like?

Using the Perceptron with logical functions



Linear separability

- The Perceptron tries to find a line that is consistent with all the training examples. That is it tries to find a line that the neurons correctly fire on one side and don't on the other.
- It has been shown that the Perceptron only works if the data is linearly separable. That is if there is actually a line that separates the two classes. Examples: OR and AND.
- The Perceptron does not converge when linear separability is not possible. Example: XOR.

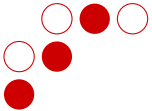
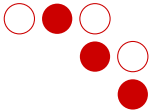
The Perceptron convergence theorem

See the proof by Michael Collins (Link is posted in Canvas)

Binary classification vs multi-class classification

How can a binary classifier by design be used to for multi-class classification problem?

- One vs. rest (OvR)
- One vs. one (OvO)

Artificial Neural Networks (ANNs)

The XOR problem

- **Problem**: Given that the Perceptron only works with linearly separable data, how can we use perceptrons to solve the XOR problem ?

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

- **Solution**: We can write the XOR function as:

$$XOR = \bar{x_1}x_2 + x_1\bar{x_2}$$

which suggests using three perceptrons to correctly solve this problem.

- **Exercise**: How would these perceptrons be arranged to correctly implement the XOR function?

The multilayer Perceptron (MLP)

It consists of multiple perceptrons arranged hierarchically into layers with the output of one layer serving as input to another.

- It must have at least three layers: an input layer, an output layer, and at least one hidden (dense) layer in between.
- It's fully (densely) connected.

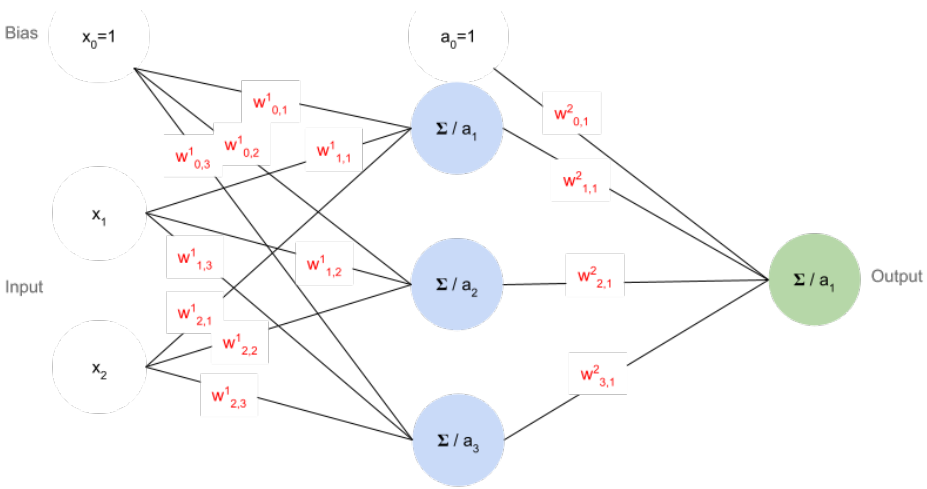
To make this work, a few changes must be made to the Perceptron:

- The threshold will need to be replaced with an activation function. There are many activation functions to pick from, one of which is the sigmoid function:

$$g(z) = \frac{1}{1 + e^{-z}}$$

- A new differentiable loss function such as L_2 will need to be adopted.
- The Perceptron update rule will need to be replaced by gradient descent.

The multilayer Perceptron (MLP)



The multilayer Perceptron (MLP): Learning

- All the learning in MLP happens in the weights. How can we arrive as a weight arrangement that minimizes the loss function?
- There are at least two sets of weights: one between the input and hidden layers and the other between the hidden and output layers.
- We can calculate the error at the output layer but not anywhere else.

The multilayer Perceptron (MLP): Training

Training the MLP involves two steps:

- Calculating the output given the inputs, weights and activations. This requires **moving forward** from input to output.
- Updating the weights according to the loss function. This requires **moving backward** or the **backpropagation of error** (BP or backprop).

Calculating the error(loss) at the output layer is easy; we know both the predicted and actual outputs. At the hidden layer(s), not so much.

The multilayer Perceptron (MLP): Training (continued)

Training the MLP involves two parts:

- Calculating the error(loss) at the output layer is easy; we know both the predicted and actual outputs. At the hidden layer(s), not so much.
- We want to know which weight caused what part of the error.
- We need a good error measure. One such measure is the L_2 loss function:

$$L_2 = \frac{1}{2}\sum_{i=1}^{n}(t^{(i)} - y^{(i)})^2$$

  where y is the predicted output and t is the actual target output.
- While inputs, activations, and weights can generally change, only weights can change during training.

Activation functions

- The Perceptron threshold function will not work here; it's discontinuous and therefore not differentiable. A new activation function will need to be used.
- Common activation functions:
  - The logistic or sigmoid function:

$$g(z) = \frac{1}{1+e^{-z}} \quad \text{and} \quad g'(z) = \frac{d}{dz}\frac{1}{1+e^{-z}} = g(z)(1-g(z))$$

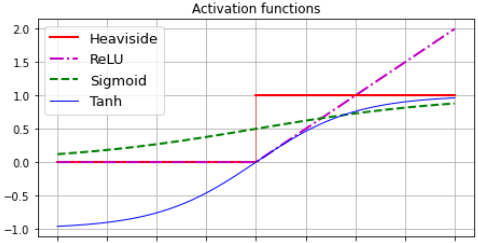
  - Another sigmoid function:

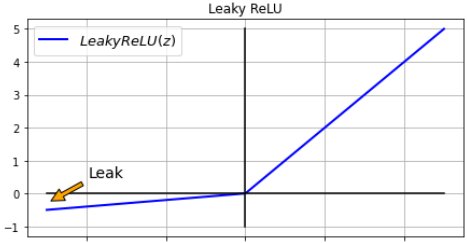
$$tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \quad \text{and} \quad tanh'(z) = \frac{d}{dz}tanh(z) = 1 - tanh^{2}(z)$$

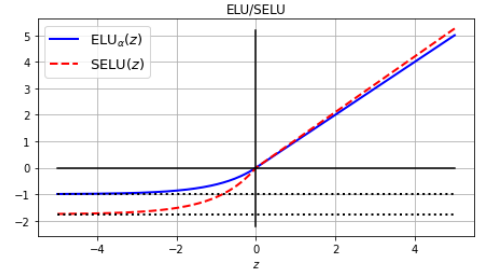
  - The rectified linear unit function (ReLU):

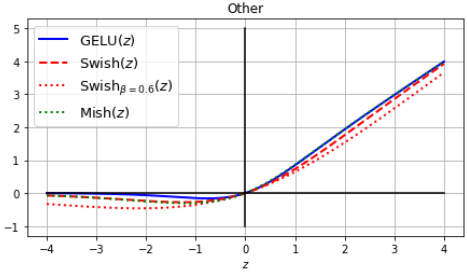
$$ReLU(z) = max(0,z) \quad \text{and} \quad ReLU'(z) = \frac{d}{dz}ReLU(z)\begin{cases}1 & \text{if } z > 0\\ 0 & \text{if } z \leq 0\end{cases}$$

Activation functions









Loss functions

- The L_2 loss function:

$$L_2 = \frac{1}{2}\sum_{i=1}^{n}(t^{(i)} - y^{(i)})^2$$

  where y is the predicted output and t is the actual target output.

- The cross-entropy loss function: Given two probability distributions p and q and a dataset with labels y_i where $i \in \{1, 2, \ldots, L\}$ and L is the number of unique labels, the cross-entropy is defined as:

$$H_c(p, q) = -\sum_{i=1}^{L} p(y_i)\ \log q(y_i)$$

Constructing a neural network

- How many hidden layers?
- How many nodes per hidden layer?
- How many nodes in the output layer?

Training the network (fitting the data)

- Initializing the weights
- Going forward
- Backpropagation of error

Initializing the weights

- What happens when weights are initialized to zero?
- Initialed to small random positive and negative numbers.
- One simple way is to use values uniformly distributed between $-1/\sqrt{n}$ and $+1/\sqrt{n}$, where n is the number of the nodes of the layer preceding the weights.

Going forward

- Apply input to the network and follow the summation/activation calculation as you move from one layer to the next until you reach the output.
- Similar to what we did for the perceptron except we have now more layers.
- Applies to both training and making predictions.

Backpropagation

- An efficient implementation of the chain rule of derivation, arguably the most important rule in Calculus.

$$\frac{df}{dx} = \frac{df}{du} \cdot \frac{du}{dx}$$

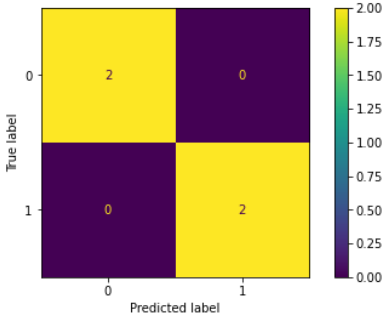
- The combination of both reverse-mode auto-differentiation and gradient descent

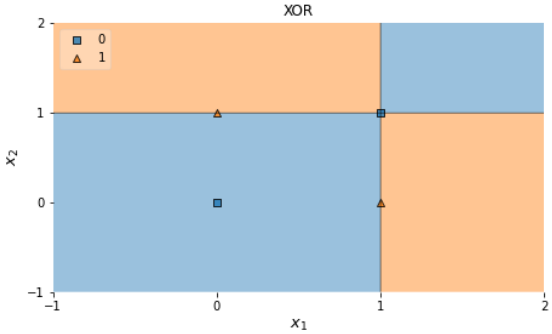
A recipe for working with MLP neural networks

- Select inputs and outputs for your problem
- Normalize or standardize inputs: Feature scaling
- Split the data into training, validation, and test datasets
- Select a network architecture
- Train a network
- Test the network

Fixing the XOR problem

This is a binary classification problem where only a single node in the output layer is needed.



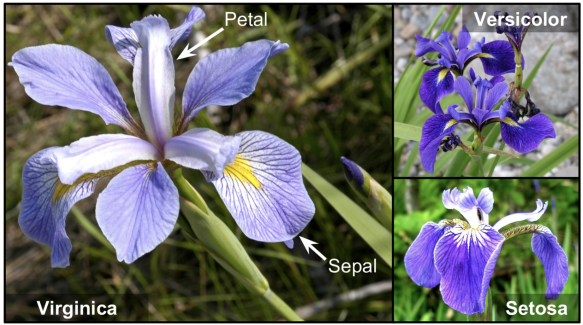


Using MLP for multiclass classification

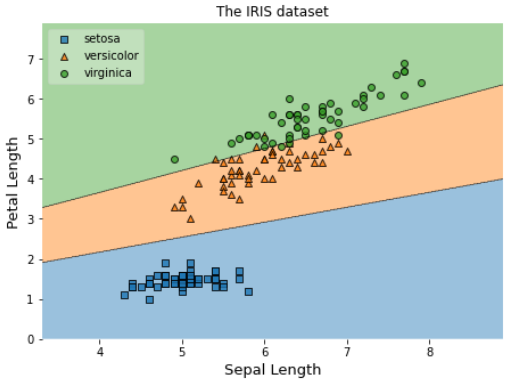
- Use as many nodes in the output layer as the number of classes
- Use one-hot (or 1 of N) encoding
- Use the softmax activation function in the output nodes:

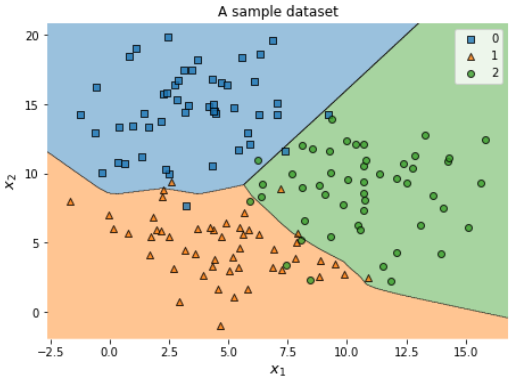
$$g(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}}$$

where C is the number of classes

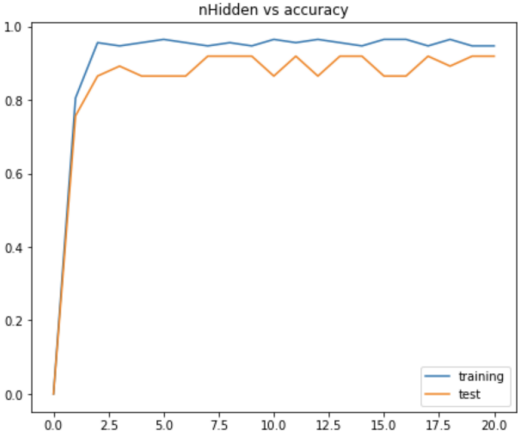


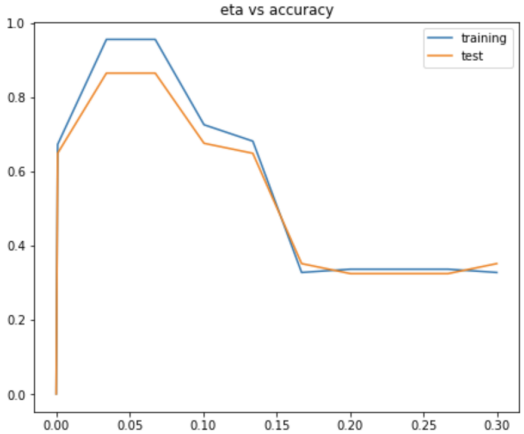
Using MLP for multiclass classification: Examples





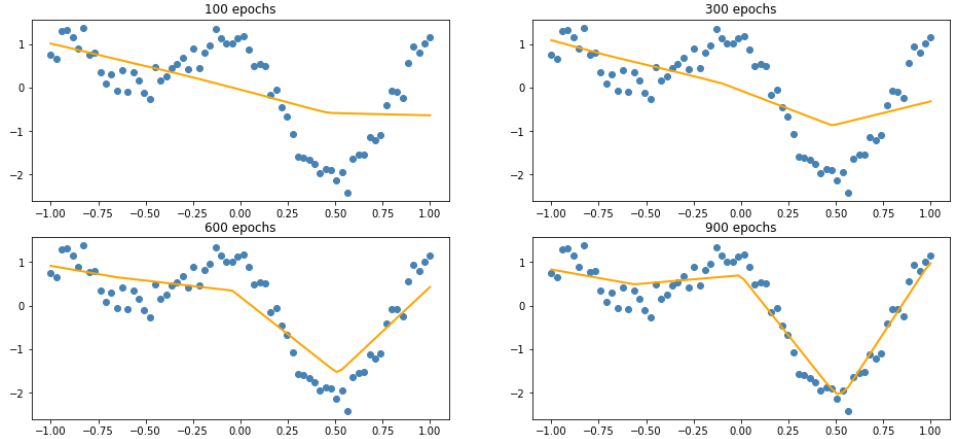
Multiclass classification: Learning curves





Using MLP for regression

Use a single node with a linear activation function ($g(z) = z$) in the output layer.

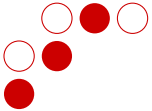
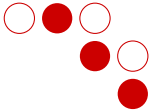


Challenges

- Local minima
- Vanishing gradients
- Overfitting
- Amount of required training data
- Required computation power

Improvements

- Another optimizer: minibatch gradient descent, SGD, or Adam
- Using momentum
- Xavier (Glorot) weight initialization
- Regularization
- Dropout
- Early stopping

Introduction to deep learning

What is deep learning?

- A typical ANN consists of:
  - an input layer
  - one or more hidden layers
  - an output layer
- When an ANN contains a deep stack of hidden layers, it is called a Deep Neural Network (DNN).
- DNNs represent a deep stack of computations. In other words, “Deep” refers to having neural networks (NN) with multiple layers between input and output.
- Modern DNNs involve tens and sometimes hundreds of hidden layers.
- Deep learning is not magic and does not apply to all problems.

Deep learning

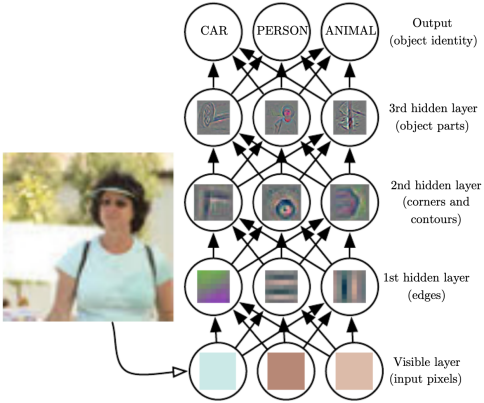
- Deep learning is machine learning involving DNNs.
- A broad family of techniques in which the hypothesis space $\mathcal{H}$ takes the form of complex tunable algebraic expressions.
- DNNs can have more than one input and one output layer.
- DNNs are widely used for applications such as image classification, object recognition, translation, speech recognition and synthesis, and image synthesis.
- It also plays a significant role in reinforcement learning: Deep RL.

Depth vs width

- The universal Approximation theorem states that any continuous function can be approximated by an MLP with at least one hidden layer.
- While this can theoretically be achieved by using a one or two wide enough hidden layers ()as opposed to a deep stack of layers), the number of parameters will be much more. In other words, achieving the same model capacity with shallow networks requires more units and parameters than using deep networks.
- Shallow networks with a large number of units are more prone to overfitting.
- Deep stacks of layers lend themselves better to problems with hierarchical structures; early layers learn low level patterns and later layers learn higher level ones.

Hidden layers

- The number of hidden layers determines the depth of a DNN.
- We can think of the values computed at each hidden layer of the network as a different representation of the input data X.
- Each layer transforms the representation produced by the preceding layer to a new representation.
- One hypothesis for why DL works so well is that the complex end-to-end transformation from input to output is decomposed by the many hidden layers into simple hierarchical transformations each of which can be learned by a local updating process.



Source: [2]

Why deep learning now?

DNNs are not new; they just weren't practical. Here is what is new.

- Hardware advances: GPUs and TPUs; storage hardware.
- Data availability: facilitated by the internet and large storage.
- Algorithmic advances: Better activation functions, weight-initialization schemes, and faster optimizers.

Looking beyond the hype

Deep learning has several properties that show that it is more than a hype or a fad.

- Simplicity: Removes the need for feature engineering; support end-to-end trainable models.
- Scalability: Highly amenable to parallelization on GPUs or TPUs.
- Versatility: Different kinds of problems; online and transfer learning; reproducible results.

Different kinds of networks

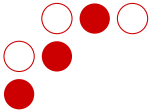
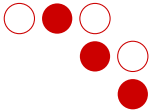
DNNs come in different shapes and topologies. The following three are widely used:

- A sequential fully-connected **feedforward** (dense) network with connections in only one direction: from input to output. No cycles.
- **Convolutional** networks. No cycles here either. Widely used in computer vision.
- A **recurrent network** networks with loops where intermediate output is fed back into the network as input. Widely used with sequences such as time series and text.
- **Transformers** which have the ability to process sequences and has become a cornerstone in natural language processing (NLP) and various other machine learning tasks.

What has deep learning achieved so far?

Deep learning has made an impact:

- Near-human-level image classification
- Near-human-level speech transcription
- Near-human-level handwriting transcription
- Improved text generation
- Improved machine translation
- Improved text-to-speech conversion
- Near-human-level autonomous driving
- Improved search results on the web
- Improved ad targeting (is that good?)
- Better-than-human chess and Go playing

Questions?

References I

[1] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

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

[3] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[4] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. url: `https://books.google.com/books?id=P\_XGPgAACAAJ`.