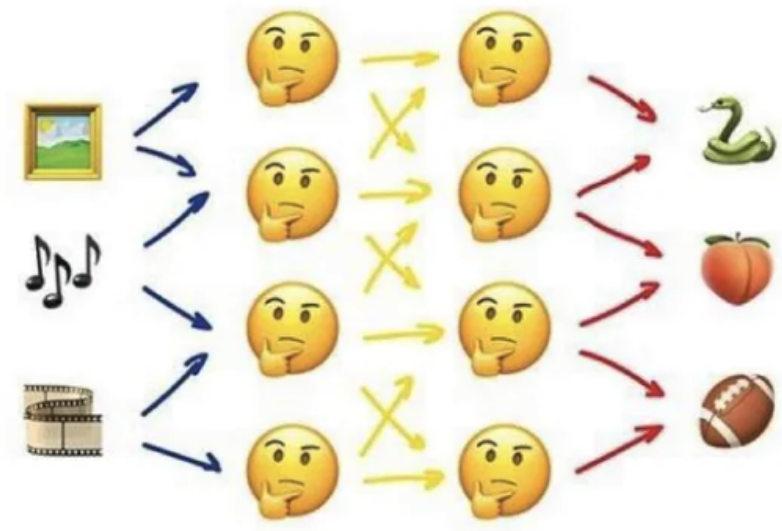
Alexandre St-Aubin

An Introduction to Machine Learning

MAY 2, 2024



McGill University – Directed Reading Program

Contents

1 *Introduction* 3

2 *Neural Networks* 3

2.1 *Single Layer Perceptron* 3

2.2 *Neurons* 4

2.3 *Feed-Forward Neural Networks* 5

2.4 *Layers* 5

2.5 *Activation Function* 6

3 *Learning* 9

3.1 *Loss Function* 9

3.2 *BackPropagation* 9

3.3 *Weight Update Rule* 12

4 *Fitting the model* 13

4.1 *Overfitting* 13

4.2 *Underfitting* 15

4.3 *Mastering the Fit* 15

4.4 *Cross-Validation* 16

5 *Example Using Pytorch* 16

5.1 *Introduction* 16

5.2 *Importing the Data* 17

5.3 *Defining the Model* 17

1 Introduction

There are several types of Machine Learning algorithms. The main categories are divided into *Supervised learning, Semi-supervised learning, Unsupervised learning and Reinforcement learning.* Here, we will focus on *Supervised learning.*

Introduce problems requiring machine learning, why it is so popular, what's the goal. How ML is not about understanding the data but more about being able to infer about new data.

Definition 1.1 (Supervised Learning). In supervised learning, the machine learns through examples. The machine learning algorithm is tasked with developing the strategy for achieving the specified outputs given some input. To do so, a known dataset is supplied that contains a set of inputs and associated outputs. The algorithm finds patterns in the data, learns from observations, and makes predictions. At the end of the learning process, the algorithm can be tested on data that is unknown to it, but that the operator knows the answers to, in order to test its accuracy. Then, the algorithm is adjusted if needed, and the process continues in an iterated manner.

Types of ML problems (classification, regression).
What is a neural network, quickly talk about components of neural network.
Structure of neural network.

2 Neural Networks

Neural networks, a subset of machine learning, form the core of supervised learning and are inspired by the human brain [IBM, 2024]. They are made of input, hidden, and output layers, and consist of interconnected *neurons* with associated *weights and thresholds*. Activated nodes transmit data to the next layer if their output surpasses the threshold.

2.1 Single Layer Perceptron

The simplest neural network is referred to as the perceptron. This neural network contains a single input layer and an output node [Aggarwal, 2018a].

Consider a situation where each *training instance* is of the form $(\overline{X}, y)$, where each $\overline{X} = [x_1, ..., x_d]$ contains d feature variables, and $y \in [-1, +1]$ contains the observed value of the binary class variable. By "observed value" we refer to the fact that it is given to us as a part of the training data, and our goal is to predict the class variable for cases in which it is not observed.

The input layer of the **perceptron** contains d nodes that transmit the d features $\overline{X} = [x_1, ...x_d]$ with **edges** of **weight** $\overline{W} = [w_1, ...w_d]$ to an output node. The input **layer** does not perform any computation in its own right. The linear function $\overline{W} \cdot \overline{X} = \sum_{i=1}^{d} w_i x_i$ is computed at the output node. Subsequently, the sign of this real value is used in order to predict the dependent variable of $\overline{X}$. Therefore, the prediction $\hat{y}$ is computed as follows:

$$\hat{y} = \text{sign}[\overline{X} \cdot \overline{W}] = \text{sign}\left[\sum_{i=1}^{d} w_i x_i\right]$$

Here, the sign function serves the role of an activation function (2.5). The goal of the **perceptron** algorithm with respect to all training instances in a data set $\mathcal{D} = \{(\overline{X_1}, y_1), \cdots, (\overline{X_n}, y_n)\}$ is to minimize the loss function (3.1). We shall learn more about loss functions in a following section, but an example would be to minimize the least-squares function:

$$\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} \left(\text{sign}[\overline{X_i} \cdot \overline{W}] - y_i\right)$$

To optimize the loss function, the perceptron will use a *gradient descent* method, which we will see later.

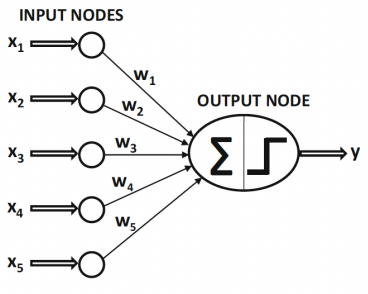


Figure 1: A single layer perceptron.

2.2 *Neurons*

The neuron is the smallest computational unit of a neural network. The single layer perceptron described in the previous section is exactly a neuron.

Definition 2.1 (Neurons). Neurons are made up of 4 elements:

1. The inputs, seen as *nodes*, consist of either the input layer, or the outputs of the previous layer of the neural network.
2. The weights and bias, seen as *edges* will act on the inputs before they reach the input function of the neuron.
3. The input function takes a sum of the weighed inputs.
4. The activation function acts on the input function and outputs the result to the following layers.

receive an input, multiply it by a weight, and output it using an activation function. A simple mathematical model for a neuron's output activation is [Russel and Norvig, 2010]

$$o_j = f\left(\sum_{i=0}^{n} w_{i,j} o_i\right)$$

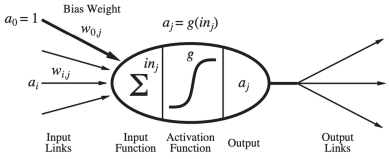


Figure 2: A neuron and its inputs and outputs.

where f is the activation function (2.5), o_i is the output activation of neuron i and $w_{i,j}$ is the weight on the directed edge from neuron i to the neuron j. We will explore activation functions in more depth shortly.

Definition 2.2 (Weights). FIND A DEFINITION (maybe relate it to the edge of a graph)

2.3 *Feed-Forward Neural Networks*

Having defined neurons, the building blocks of neural networks, we may now give a rigorous definition of the latter. A feed-forward network has connections only in one direction—that is, it forms a directed acyclic graph. Every node receives input from "upstream" nodes and delivers output to "downstream" nodes; there are no loops. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves [Russel and Norvig, 2010].

The overall network is a combination of function composition and matrix multiplication:

$$g(x) = f^L(W^L f^{L-1}(W^{L-1} \dots f^1(W^1 x) \dots)),$$

where L is the number of layers (2.4), W is the weight matrix, and f is the activation function (2.5).

Definition 2.3 (Feed-forward Neural Network). A neural network is a *computational directed acyclic graph,* in which a unit of computation is the neuron. Each directed edge in the graph represents a function passing the weighed output of a node in a layer to a node in the next layer.

Artificial neural networks are often refered to as **multi-layer perceptrons** (abbreviated to MLPs) for one simple reason: you can think of such an MLP as the composition of multiple perceptrons, where, in this case, the perceptron would be a synonym for artificial neuron, so the smallest computational unit of a neural network, which performs, for example, a linear combination of its inputs followed by the application of an activation function, which can be the sigmoid, tanh, ReLU, identity, or any other function that is differentiable, if you plan to train the neural network with gradient descent.

2.4 *Layers*

Feed-forward networks are usually arranged in layers, such that each unit receives input only from units in the immediately preceding layer. The layers are divided in 3 groups: the *input, hidden,* and *output* layers. See figure 2.4 for an image of the layers. Layers are a general

term given to a grouping of neurons that act together in the neural network.

Definition 2.4 (Layers).

Input Layer. The first layer in a neural network, it receives the initial data for the network from the outside world. The "Entry point" of the neural network.

Hidden layers. The hidden layer(s) are where the magic happens in neural networks. Each layer is trying to learn different aspects about the data by minimizing the **cost function**. The most intuitive way to understand these layers is in the context of image recognition such as a face. The first layer may learn edge detection, the second may detect eyes, third a nose, etc. [cdeterman].

Output Layer. The final layer in the neural network is the output layer. This layer is responsible for holding the final result or output of the problem. Input, such as raw images, is fed into the input layer, and the output layer produces the corresponding result.

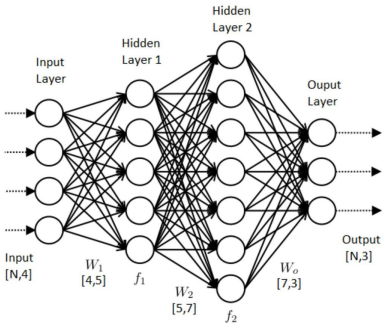


Figure 3: A graph representation of a Neural Network.

2.5 *Activation Function*

An activation function in neural networks is a smooth function applied to the output of each neuron in a layer. It introduces nonlinearity to the network, allowing it to learn complex patterns and relationships in the data.

The activation function determines whether a neuron should be activated or not based on the weighted sum of its inputs. In other words, it defines the output of a neuron given a set of inputs. Without activation functions, neural networks would be limited to linear transformations, and they wouldn't be able to capture the nonlinearities present in many real-world datasets.

Desired Characteristics of Activation Functions [Jagtap and Karniadakis, 2022]
There is no universal rule for choosing the best activation function, but there are some characteristics to look for, namely

Nonlinearity One of the most essential characteristics of an activation function is nonlinearity. In comparison to linear activation functions, the non-linearity of the activation function significantly improves the learning capability of neural networks.

Computationally Cheap The activation function must be easy to evaluate in terms of computation. This has the potential to greatly improve network efficiency.

Finite Range/Boundedness Gradient-based training approaches are more stable when the range of the activation function is finite, because pattern presentations significantly affect only limited weights.

Differentiability The most desirable quality for using gradient-based optimization approaches is continuously differentiable activation functions. This ensures that the back-propagation algorithm works properly.

Remark 2.5. The vanishing and exploding gradient problems: The variation of the inputs and outputs of some activation functions, such as the logistic function (Sigmoid), is extremely large. To put it another way, they reduce and transform a bigger input space into a smaller output space that falls between [0,1]. As a result, the back-propagation algorithm has almost no gradients to propagate backward in the network, and any residual gradients that do exist continue to dilute as the program goes down through the top layers. Due to this, the initial hidden-layers are left with no information about the gradients. For hyperbolic tangent and sigmoid activation functions, it has been observed that the saturation region for large input (both positive and negative) is a major reason behind the vanishing of gradient. One of the important remedies to this problem is the use of non-saturating activation functions, such as ReLU.

We present some commonly used activation functions.

Sigmoid Function. Its range is $[0, 1]$, and is defined as,

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

Advantage: boundedness.
Disadvantages: the vanishing gradient problem, the output not being zerocentered, and the saturation for large input values.

Hyperbolic Tangent Function. It is mostly used for regression problems, has a range of $[-1, 1]$, and is defined as,

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Advantage: zerocentered structure.
Disadvantage: the vanishing gradient problem, i.e. once saturated, it is really challenging for the learning algorithm to adapt the parameters and learn faster.

ReLU Function. ReLU was primarily used to overcome the vanishing gradient problem. ReLU is the most common activation

function used for *classification problems*. Its range is $[0, \infty)$, and is defined as

$$\text{ReLU}(x) = \max(0, x)$$

Advantages: Apart from overcoming the vanishing gradient problem, the implementation of ReLU is very easy and thus cheaper, unlike tanh and sigmoid, where an exponential function is needed. Disadvantages: It has a saturation region, which can prevent the learning of the networks. In particular, ReLU always discards the negative values, which makes the neurons stop responding to the gradient-based optimizer. This problem is known as *dead or dying ReLU problem*, meaning the neurons stop outputting other than zero.

Softplus Function. It approximates the ReLU activation function in a smooth way, with a range of $(0, \infty)$, and it is defined as

$$\text{Softplus}(x) = \ln(1 + e^x)$$

Softmax It is a generalization of logistic function in high dimensions. It normalizes the output and divides it by its sum, which forms a probability distribution. The standard softmax function Softmax: $\mathbb{R}^k \to (0, 1)^k$ is defined as

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \quad \text{for } i = 1, ..., k$$

In other words, it applies the standard exponential function to each element x_i of the input vector x and normalizes these values by dividing them by the sum of all these exponentials, which ensures that the sum of the components of the output vector is 1.

The following is a table summarizing the information above.

Activation Function ($\Phi(x)$)	Derivatives ($\Phi'(x)$)	Range	Continutity
Linear : $\Phi(x) = x$	$\Phi'(x) = 1$	$(-\infty, \infty)$	C^∞
Step : $\Phi(x) = \begin{cases} 0 & x \leq 0, \\ 1 & x > 0 \end{cases}$	$\Phi'(x) = \begin{cases} 0 & x \neq 0, \\ \text{Not defined} & x = 0 \end{cases}$	$(0, 1)$	C^{-1}
Sigmoid or Logistic : $\Phi(x) = \frac{1}{1+e^{-x}}$	$\Phi'(x) = \Phi(x)(1 - \Phi(x))$	$(0, 1)$	C^∞
Rectifier Unit (ReLU) : $\Phi(x) = \begin{cases} 0 & x \leq 0, \\ x & x > 0 \end{cases}$	$\Phi'(x) = \begin{cases} 0 & x < 0, \\ 1 & x > 0, \\ \text{Not defined} & x = 0 \end{cases}$	$[0, \infty)$	C^0
Hyperbolic Tangent : $\Phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\Phi'(x) = 1 - \Phi(x)^2$	$(-1, 1)$	C^∞
Softplus : $\Phi(x) = \ln(1 + e^x)$	$\Phi'(x) =$ Sigmoid	$(0, \infty)$	C^∞
Leaky Rectifier Unit (Leaky ReLU) : $\Phi(x) = \begin{cases} 0.01x & x < 0, \\ x & x \geq 0 \end{cases}$	$\Phi'(x) = \begin{cases} 0.01 & x < 0, \\ 1 & x > 0, \\ \text{Not defined} & x = 0 \end{cases}$	$(-\infty, \infty)$	C^0
Exponential Linear Unit (ELU) : $\Phi(x) = \begin{cases} \alpha(e^x - 1) & x \leq 0, \\ x & x > 0 \end{cases}$	$\Phi'(x) = \begin{cases} \alpha e^x & x < 0, \\ 1 & x > 0, \\ \alpha & x = 0 \end{cases}$	$(-\alpha, \infty)$	$\begin{cases} C^1 & \text{If} \alpha = 1 \\ C^0 & \text{Otherwise} \end{cases}$
Gaussian : $\Phi(x) = e^{-x^2}$	$\Phi'(x) = -2x\Phi(x)$	$(0, 1]$	C^∞
Swish ($\beta = 1$) : $\Phi(x) = x \cdot$ Sigmoid	$\Phi'(x) = x \cdot \Phi(x)(1 - \Phi(x)) +$ Sigmoid	$[0, .\infty)$	C^∞
Oscillatory : $\Phi(x) = \sin(x)$	$\Phi'(x) = -\cos(x)$	$[-1, 1]$	C^∞

Figure 4: Common activation functions, their derivatives, range, and order of continuity.

3 *Learning*

3.1 *Loss Function*

Here, we present some common loss functions [Ciampiconi et al., 2023].

3.2 *BackPropagation*

Backpropagation is a gradient estimation method used to train neural network models. The gradient estimate is used by the optimization algorithm to compute the network weight updates. So, when we say a neural network is *learning*, it means that backprop is computing a gradient descent that minimizes the loss function, and updates the weights using a **weight update rule** 3.3. Backpropagation is a way of computing the partial derivatives of a loss function with respect to the weights of a network; we use these derivatives in gradient descent, exactly the way we would with linear regression and logistic regression.

Let us first begin by recalling that a neural network evaluates compositions of functions computed at individual nodes. Think of a neural network as a function $h_w(x)$ of the input, parametrized by the weights. Consider the network in Figure 3.2, let $\{x_1, x_2\}$ be the input vector, f the activation function and a_i denote the activated output at node i. Then, the output at node 3 is given by

$$a_3 = f(w_{0,3} + f(x_1)w_{1,3} + f(x_2)w_{2,3}),$$

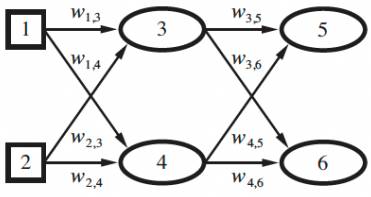


Figure 5: A simple network with 2 inputs, one hidden layer, and two outputs.

where $w_{0,3}$ is the bias weight at node 3 (explain what a bias weight is). Similarly, the output at node 5 is

$$\begin{aligned} a_5 &= f(w_{0,5} + f(a_3)w_{3,5} + f(a_4)w_{4,5}) \\ &= f[w_{0,5} + f(w_{0,3} + f(x_1)w_{1,3} + f(x_2)w_{2,3})w_{3,5} + f(w_{0,4} + f(x_1)w_{1,4} + f(x_2)w_{2,4})w_{4,5}] \end{aligned}$$

And even with such a small network, we can already see how awkward it would be to compute the derivative of a_5 with respect to w. But an even bigger problem arises when we think of how we would compute the loss function in hidden layers. Whereas the error $y - h_w$[1] at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have.

[1] y is the target output, and h_w the value computed by the network

Therefore, we need some kind of iterative approach to compute the derivatives, and a way to *back-propagate* the error from the output layer to the hidden layers. The resulting iterative approach uses *dynamic programming*, and the **weight update rule** is the *chain rule* of differential calculus [Aggarwal, 2018b].

continue from p.733 of Russel, p.570 of Murphy, and p.109 of Charu

Theorem 3.1 (Multivariate Chain Rule). Let $z = f(y_1, y_2, \ldots, y_m)$ be a differentiable function of m variables, where each $y_j = g_j(x_1, x_2, \ldots, x_n)$ is a differentiable function of n variables. Then, for each $i = 1, 2, \ldots, n$, the partial derivative of z with respect to x_i is given by

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

We note that backpropagation can be implemented by using either the *pre-activation*, or *post-activation* values at each neuron. Here, we'll focus on the method that acts on the pre activation values neurons. For the sake of simplicity, we view the neural network as a *Directed Acyclic Graph* G, where

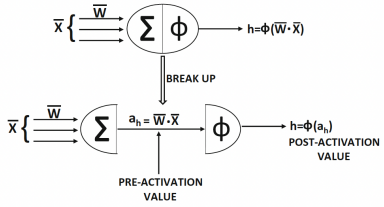


Figure 6: Pre and Post activation values of a neuron.

1. Each node represents a neuron, and is denoted by a number j.
2. The weight on the edge from neuron i to j is denoted $w_{i,j}$.

The algorithm can be divided in two phases: *forward* and *backward*.

Forward Phase. The term "forward phase" refers to this process of computing values of each hidden layer depending on the current weight values using a specific input vector. These computations naturally cascade forward across the layers. The aim of the forward phase is to compute every intermediate hidden and output variable for a given input. The Backward phase will call for these values. The derivative of the loss function L with respect to this

output, as well as the value of the output o, are calculated at the completion of the computation. For the sake of simplicity, we will explore the case of a single output node for the time being. We will then talk about the simple generalization to many outputs.

Backward Phase. In this phase, the gradient of the loss function with respect to different weights is calculated. First, the derivative $\frac{\partial o}{\partial w}$ is computed. This establishes the gradient computation's initialization. The multivariable chain rule is then used to propagate the derivatives in the opposite direction. Since we are focussing on the pre-activation approach, the gradients are computed with respect to the pre-activation values of the hidden variables, which are then propagated backwards.

More precisely, the process can be described as follows. Let (x, t) be the input to our algorithm, where t is the target output of the input vector x. Define the value

$$L = \text{Loss}(y, t),$$

where Loss could be any *loss function*, such as the MSE. At each neuron j, let its post-activation output be

$$o_j = \Phi(\text{net}_j) = \Phi\left(\sum_{i=0}^{n} w_{i,j} o_i\right),$$

where Φ is any *activation function*, $w_{i,j}$ is the weight on the edge between neurons i and j, and o_i is the output from neuron i. Then, the pre-activation value is easily seen to be net_j.

1. The **forward pass** propagates x through the neural network, computing the values of all hidden neurons to reach the output $\hat{y}$ of the neural network, which corresponds to the predicted output. Then, $L = \text{Loss}(\hat{y}, t)$ is computed.

2. The derivative $\frac{\partial L}{\partial \hat{y}}$ at the output can be directly computed. Then, to compute the derivative of L with respect to o_j for an arbitrary neuron j,

3. Consider L as a function of all neurons receiving input from j, and denote this set by $I_j = \{i_1^{(j)}, ..., i_n^{(j)}\}$. Then,

$$\frac{\partial L(o_j)}{\partial o_j} = \frac{\partial L(\text{net}_{i_1^{(j)}}, ..., \text{net}_{i_n^{(j)}})}{\partial o_j}$$

to obtain the following recurrence relation for the derivative of L

with respect to o_j, by the chain rule,

$$\begin{aligned}\frac{\partial L}{\partial o_j} = \sum_{i \in I_j} \left(\frac{\partial L}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial o_j} \right) &= \sum_{i \in I_j} \left(\frac{\partial L}{\partial o_i} \frac{\partial o_i}{\partial \text{net}_i} \frac{\partial \text{net}_i}{\partial o_j} \right) \\ &= \sum_{i \in I_j} \left(\frac{\partial L}{\partial o_i} \frac{\partial o_i}{\partial \text{net}_i} w_{j,i} \right)\end{aligned} \tag{1}$$

We can see that the derivative with respect to o_j can be computed if those of the neurons on the next layer are already known, per the *recurrent* behaviour of this algorithm.

4. We now have all the necessary tools to compute the partial derivative of L with respect to the weight $w_{i,j}$. Again, by applying the chain rule, we get

$$\begin{aligned}\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial w_{i,j}} &= \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{i,j}} \\ &= \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \left(\frac{\partial}{\partial w_{ij}} \sum_{k=1}^{n} w_{k,j} o_k \right) \\ &= \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \left(\frac{\partial w_{i,j} o_i}{\partial w_{i,j}} \right) = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} o_i\end{aligned} \tag{2}$$

and so, we may consider the following recursively defined function to simplify notation,

$$\delta_j = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} = \begin{cases} \frac{\partial L(t, o_j)}{\partial o_j} \frac{d\varphi(\text{net}_j)}{d\text{net}_j}, & \text{if } j \text{ is an output neuron,} \\ \left(\sum_{i \in I_j} w_{j,i} \delta_i \right) \frac{d\varphi(\text{net}_j)}{d\text{net}_j}, & \text{otherwise.} \end{cases}$$

to conclude that the partial derivative of L with respect to the weight $w_{i,j}$ is given by

$$\frac{\partial L}{\partial w_{i,j}} = o_i \delta_j$$

3.3 *Weight Update Rule*

After having found the derivatives of the loss function with respect to the weights, one needs a way to update the weights. The goal is to modify the weights of the neural network so that given the input (x, t), where x is the input vector and t the target output, it points more toward t. Then, in the future, it will have a better chance of classifying x correctly [Hagan et al., 2014]. The naive way would be to set the weights so that they point directly to t. Unfortunately, this leads to overfitting, which we'll address in the next chapter.

To update the weight $w_{i,j}$ using backpropagation, we first choose a learning rate $\mu > 0$. We may then choose to update $w_{i,j}$ by adding $\Delta w_{i,j}$ to it, where

$$\Delta w_{i,j} = -\mu \frac{\partial L}{\partial w_{i,j}} = -\mu o_i \delta_j$$

Then, the weight update rule would be defined as,

$$w_{i,j}^{(\text{updated})} = w_{i,j}^{(\text{old})} - \mu o_i \delta_j$$

We claim that this update rule decreases the loss L.

Proof. If $\frac{\partial L}{\partial w_{i,j}} > 0$, an increase in $w_{i,j}$ increases L; conversely, if $\frac{\partial L}{\partial w_{i,j}} < 0$, an increase in $w_{i,j}$ decreases L. The new $\Delta w_{i,j}$ is added to the old weight, and the product of the learning rate and the gradient, multiplied by -1 guarantees that $w_{i,j}$ changes in a way that always decreases L. □

4 *Fitting the model*

The main difficulty with machine learning is that we have to be able to learn from new, unknown inputs as well as those that our model was trained on. The capacity of a model to perform well on unseen data is called **Generalization**. When training a model, we want the error between the expected and predicted outputs to be as low as possible on the training data. This is referred to as the *training error*. We could stop here, and we'd have an *optimization* problem. But this is where the difference with machine learning is, we require that our model also predict new, unseen data with a low error. We call this error the **test error**. When determining the quality of a ML algorithm, we look mainly at the two following attributes,

1. How low is the training error?
2. How small is the gap between the test and training errors?

These correspond, respectively, to the two main challenges in Machine Learning: **Underfitting** and **Overfitting** [Goodfellow et al., 2016].

4.1 *Overfitting*

Overfitting refers to the process of producing an analysis that matches a given set of data too closely, which can lead to problems when trying to fit new data or make reliable predictions about future events. That is, a mathematical model starts overfitting when it begins to memorize the data, rather than learn from it.

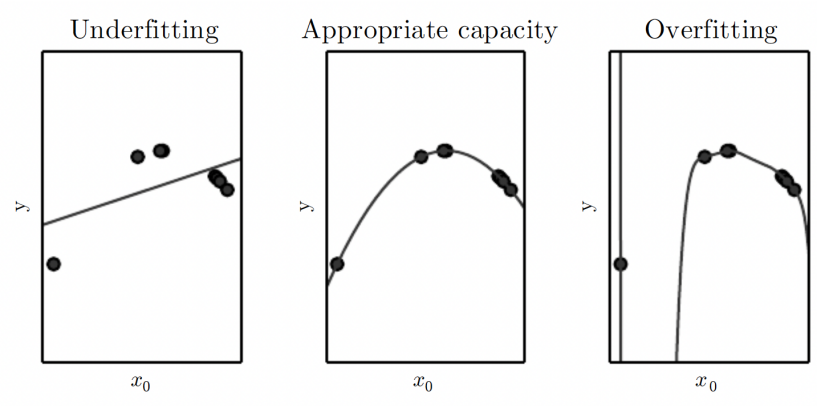


Figure 7: Diagrams representing Underfitting, Appropriate, and Overfitting, from [Goodfellow et al., 2016]

In the topic of machine learning, if we choose too large of a network, it will behave like a lookup table on the data it was trained with, but won't generalize well to new data. Put another way, when a neural network has an excessive number of parameters, it can overfit, just like any other statistical model [Russel and Norvig, 2010].

As was hinted above, the degree of overfitting is influenced by the amount of data supplied, as well as the model's *complexity*. The complexity of the model is determined by the number of underlying parameters that a neural network has. Additional degrees of freedom are the consequence of having more parameters, which can be utilized to explain certain training data points without making a strong generalization to new ones [Aggarwal, 2018c].

Example 4.1. Suppose we have 5 training pairs (x, t) available. Then, it can be shown that there exists a degree 4 polynomial that fits the 5 training points exactly with zero error. This does not mean, however, that the polynomial will approximate unseen data with zero error.

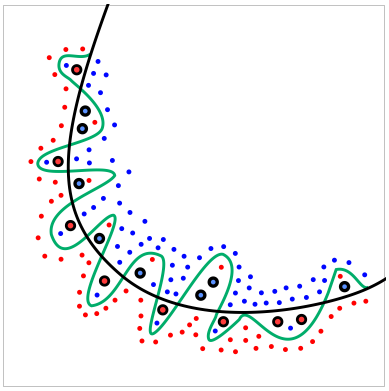


Figure 8: A diagram showing overfitting (green line) of data from [Wikipedia, the free encyclopedia]

As seen in figure 4.1, an overfit model will have low *bias*, but high *variance* on the training data. However, an overfit model will "have estimated (and actual) sampling variances that are needlessly large (the precision of the estimators is poor, relative to what could have been accomplished with a more parsimonious model) [...]. A best approximating model is achieved by properly balancing the errors of underfitting and overfitting." [Burnham and Anderson, 2002].

Remark 4.2. In machine learning models, overfitting is more likely when learning is performed for too long, or when the training data set is small.

4.2 *Underfitting*

Underfitting is essentially the inverse of overfitting. It occurs when a model is too basic, and lacks the complexity, i.e. the number of parameters, to explain the patterns in the data. An underfit model will have high bias and low variance on the training data, the exact opposite of an overfit model. In other words, underfitting is failing to learn enough from the training set. For instance, when fitting a linear model to non-linear data, under-fitting would happen. Such a model's predicting ability would be mediocre both for the training and testing data.

Underfitting is not as widely discussed as overfitting, since it can be detected easily by evaluating how well the model is performing on the training data.

4.3 *Mastering the Fit*

Now that we've shown how bad a Machine Learning algorithm can be, we'll show how to fix it. By changing a model's capacity, we may alter how likely it is to overfit, or underfit.

Definition 4.3 (Capacity). We define the capacity of a model as its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set, while models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set. The capacity of a model is a way to measure its complexity.

Before altering the capacity of a model, one must evaluate whether a model is overfitting or underfitting. This can be done with the help of cross-validation, which we'll discuss in the next chapter.

As long as the training error is low, one way to increase regularization without playing with capacity is to increase the amount of training data. If this is not possible, there are a number of components we can adjust in order to control the capacity of a model, such as [Brownlee, 2020],

1. The number of nodes per layer (**width**)
2. The number of layers (**depth**)

It makes sense that augmenting these would increase the capacity, as they clearly increase the complexity of the model, which we know is closely related its capacity. It should be noted, however, that increasing the number of nodes and layers in a model can also increase its running time and memory usage.

Other ways to increase capacity include tuning the learning rate optimally, decreasing the *weight decay coefficient*, and decreasing the

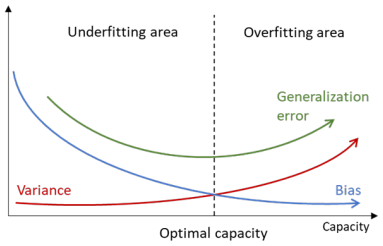


Figure 9: A diagram showing the capacity of a model, from [Kowalik, 2023]

dropout rate. We refer the reader to chapter 11 of Goodfellow et al. [2016] for more information on this topic.

4.4 Cross-Validation

Cross-validation is a technique that aims to test the ML model's testing capabilities on unseen data. This helps identify issues such as *overfitting*, discussed above, or *selection bias*, which is the result of selecting training data in a non-random way which doesn't represent the population. Cross-validation provides insight into how well the model will generalize to a different dataset.

The most common technique used is **k-fold cross-validation.** It allows one to repeatedly train and test the model k times on various randomly chosen subsets of the training data. The average test error across k trials can then be used to estimate the test error. In trial i, the i^{th} subset of the data is used as the test set, and the remaining data is utilized as the training set. One issue is that there are no unbiased estimators of the variance of such average error estimators, but estimations are normally used [Goodfellow et al., 2016].

Algorithm 1: k-fold cross-validation

```
Function KFoldXV(𝔻, A, L, k):
    Require: 𝔻 , the given dataset, with elements z^(i)
    Require: A, the learning algorithm, seen as a function that
     takes a dataset as input and outputs a learned function
    Require: L, the loss function, seen as a function from a
     learned function f and an example z^(i) ∈ 𝔻 to a scalar ∈ ℝ
    Require: k , the number of folds
    Split 𝔻 into k mutually exclusive subsets 𝔻_i, whose union
     is 𝔻
    for i from 1 to k do
        f_i = A(𝔻 𝔻_i);
        for z^(j) in 𝔻_i do
            e_j = L(f_i, z^(j));
    return e;
```

5 Example Using Pytorch

5.1 Introduction

In this example, we'll demonstrate how to use PyTorch to implement a Machine Learning model [Brownlee, 2023].

5.2 *Importing the Data*

The first step involves preparing and loading your data. Tabular data, such as CSV files, can be handled using common Python libraries. For instance, Pandas facilitates CSV loading, while tools from scikit-learn assist in encoding categorical data like class labels.

PyTorch offers the `Dataset` class, which permits customization for data loading. Below is a skeletal outline of a custom Dataset class.

```
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # store the inputs and outputs
        self.X = ...
        self.y = ...

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]
```

Listing 1: Python code for a PyTorch Dataset class

Once the data is loaded, PyTorch offers the `DataLoader` class for navigating a `Dataset` instance during model training and evaluation.

The `random_split()` function splits a dataset into *train* and *test* sets. After splitting, a subset of rows from the Dataset can be supplied to a `DataLoader`, along with parameters such as batch size and whether data should be shuffled every *epoch*.

For instance, a `DataLoader` can be defined by providing a chosen sample of rows from the dataset.

```
...
# create the dataset
dataset = CSVDataset(...)
# select rows from the dataset
train, test = random_split(dataset, [[...], [...]])
# create a data loader for train and test sets
train_dl = DataLoader(train, batch_size=32, shuffle=True)
test_dl = DataLoader(test, batch_size=1024, shuffle=False)
```

Listing 2: Python code for creating a DataLoader instance

5.3 *Defining the Model*

To define a model in PyTorch we define a class that extends the PyTorch `Module` class. In the constructor, we will set the layers and the `forward()` function, which defines the behaviour of the *forward* step of the backprop algorithm.

In the following code, we define the layers of the NN using the function `nn.Linear(in, out, bias)`, where `in` and `out` represent the

size of each input and output sample, respectively.

```
 # model definition
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # define the layers
        self.layer = Linear(n_inputs, 1)
        # define the activation function
        self.activation = Sigmoid()

    # forward propagate input
    def forward(self, X):
        X = self.layer(X)
        X = self.activation(X)
        return X

```

References

Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*, chapter 1.2. Springer Publishing Company, Incorporated, 1st edition, 2018a. ISBN 3319944622.

Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*, chapter 3.2. Springer Publishing Company, Incorporated, 1st edition, 2018b. ISBN 3319944622.

Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*, chapter 4.1. Springer Publishing Company, Incorporated, 1st edition, 2018c. ISBN 3319944622.

Jason Brownlee. How to control neural network model capacity with nodes and layers, 2020. URL https://machinelearningmastery.com/how-to-control-neural-network-model-capacity-with-nodes-and-layers/. [Online; accessed April 30, 2024].

Jason Brownlee. Pytorch tutorial: How to develop deep learning models with python - machinelearningmastery.com, Apr 2023. URL https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/.

K.P. Burnham and D.R. Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer Verlag, 2002.

cdeterman. what is a 'layer' in a neural network. Stack Overflow. URL https://stackoverflow.com/questions/35345191/what-is-a-layer-in-a-neural-network/35347548#35347548.

Lorenzo Ciampiconi, Adam Elwood, Marco Leonardi, Ashraf Mohamed, and Alessandro Rozza. A survey and taxonomy of loss functions in machine learning, 2023.

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

Martin T Hagan, Howard B Demuth, Mark H Beale, and Jesús Orlando, De. *Neural Network Design (2nd Edition)*, chapter 4 Perceptron Learning Rule. Martin Hagan, 2014. ISBN 978-0971732117. URL https://hagan.okstate.edu/NNDesign.pdf.

IBM. What are neural networks, 2024. URL https://www.ibm.com/topics/neural-networks. Accessed 7 Febuary 2024.

Ameya D. Jagtap and George Em Karniadakis. How important are activation functions in regression and classification? a survey, performance comparison, and future directions, 2022.

Marek Kowalik. Capacities of quantum neural networks, part 1, 2023. URL https://medium.com/@marekkowalik97/capacities-of-quantum-neural-networks-part-1-1a731f44be0. [Online; accessed May 2, 2024].

Stuart Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach*, chapter 18. Pearson Education, New Jersey, 3rd edition, 2010.

Wikipedia, the free encyclopedia. Overfitting. URL https://en.wikipedia.org/wiki/Overfitting. [Online; accessed April 30, 2024].