AN INTRODUCTION TO SUPERVISED MACHINE LEARNING

Alexandre St-Aubin
McGill University
alexandre.st-aubin2@mail.mcgill.ca

ABSTRACT

We discuss neural networks (NN), the classical approach to supervised learning. We first cover the components of a NN, such as its neurons, weights, edges, and activation functions. We explain how a NN can learn information and infer about new, unseen data. We then discuss the problem of overfitting a model, as well as techniques to avoid it. [1]

Keywords: Machine Learning, Supervised Learning, Neural Networks, Back-propagation, Overfitting.

[1] Work in progress.

CONTENTS

1 INTRODUCTION

Machine learning has surged in popularity due to its ability to tackle complex problems by discerning patterns in data, making accurate predictions, and informing decisions. Unlike traditional approaches, machine learning focuses not only on understanding data but also on inferring from it, allowing models to generalize knowledge and make predictions on unseen data. This emphasis on inference enables machine learning to transcend comprehension, driving innovation and progress across various domains. There are several types of Machine Learning algorithms. The main categories are divided into *Supervised learning, Unsupervised learning and Reinforcement learning.* Figure 1 depicts the main classes of Machine Learning and some popular models for each.

SUPERVISED LEARNING allows the machine to learn 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 target outputs in the form (x, t), where x is the input to the model, and t the target output, i.e. the *label*. The algorithm finds patterns in the data and adjusts its parameters to predict more accurately future inputs. This kind of machine learning algorithm needs a developer to accurately label the training data.

The two main problems requiring Supervised learning are **Classification** and **Regression**. The former consists of classifying objects into their appropriate classes. A well known example operates on the MNIST data set, which is a set of images representing hand drawn numbers from 0 to 9. A classification ML algorithm will analyse each point (x, t) of the data set during the *training phase*, where x is the image, and t is the number it represents. Then, given an unseen image as input, the algorithm should predict which number it represents.

Regression is the other common use of Supervised learning. It aims to predict a continuous output variable, rather than a discrete one (classification). An example would be to predict the cost of a house, given a set of training data consisting of characteristics of houses and their corresponding price.

UNSUPERVISED LEARNING algorithms discern patterns within data without predefined labels. The algorithm identifies correlations and structures within datasets. Key steps in this process include clustering, which groups similar data sets, and dimension reduction, which simplifies variables to extract essential information.

SEMI-SUPERVISED LEARNING acts on a data set that contains both labelled and unlabelled data. Normally, labelled data is rare, while unlabelled data makes up most of the set. The goal of semi-supervised learning algorithms would be to classify the unlabelledby inferring from the labelled data.

REINFORCEMENT LEARNING involves a structured approach where algorithms are provided with actions, parameters, and goals. These algorithms explore various options, learn from trial and error, and adapt their strategies based on past experiences to optimize outcomes [Wakefield].

In this paper, we will focus on *Supervised learning*, in particular, the *Neural Network* approach. However, most of the basic concepts we'll cover, such as Neurons, Backpropagation, and Overfitting, are topics that apply to all spheres of machine learning.

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 biases*. Activated nodes transmit data to the next layer through outgoing edges.

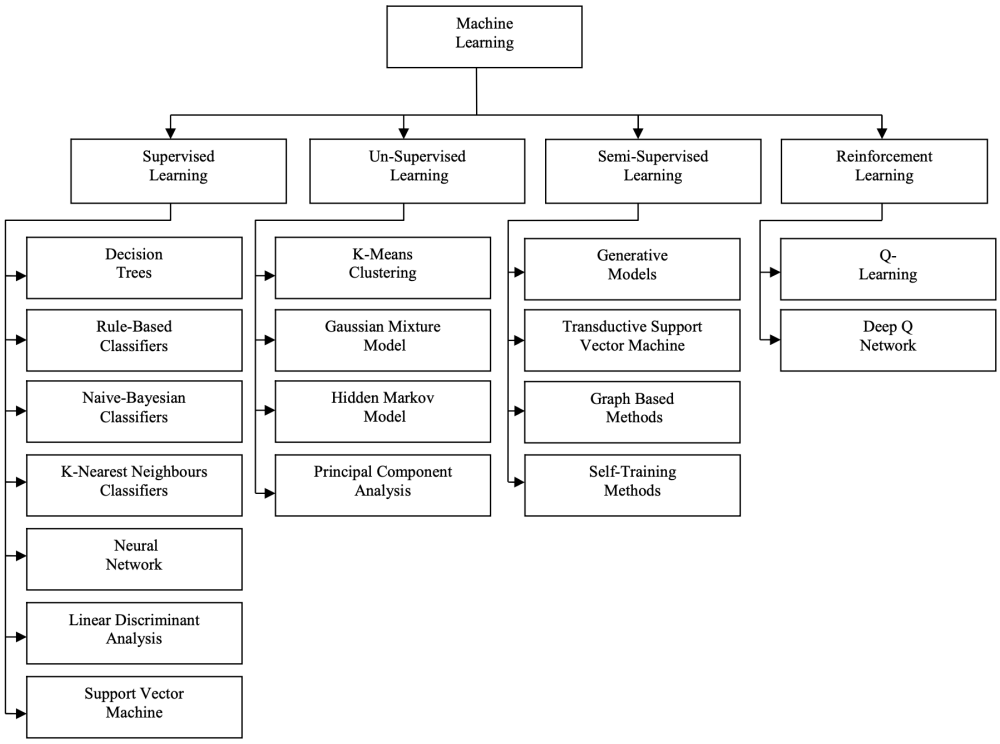

Figure 1: Classification of Machine Learning Algorithms from N and Gupta [2020]

2.1 Single Layer Perceptron

The simplest neural network is referred to as the **Signle Layer Perceptron** (see Figure 2). 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. 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:

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

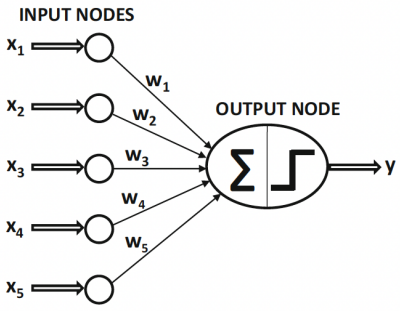

Figure 2: A single layer perceptron, from Aggarwal [2018a].

Here, the sign function serves the role of an activation function (subsection 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 (subsubsection 3.4.1). We shall learn more about loss functions in a following section, but an exam-

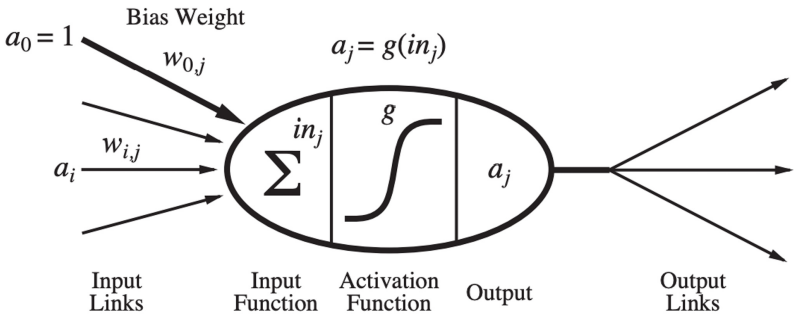

Figure 3: A neuron and its inputs and outputs.

ple 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.

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, see Figure 3 for a picture.

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

1. *The inputs, seen as nodes, consist of either the input layer (subsection 2.4), or the outputs of the previous layer of the neural network.*
2. *The weights (Definition 2.2) and biases (Definition 2.3), seen as edges will act on the outputs of previous neurons before they reach the input function of the neuron.*
3. *The input function takes a sum of the weighed inputs.*
4. *The activation function (subsection 2.5) acts on the input function and outputs the result to the following layers.*

A simple mathematical model for a neuron's output activation is [Russel and Norvig, 2010]

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

where f is the activation function, 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.

Definition 2.2 (Weights)**.** *A weight $w_{i,j} \in \mathbb{R}$ will multiply the output of neuron i before giving it as input to neuron j. Weights are the moving parts of the neural network, as they get changed during each iteration of the learning step in order to optimize the output. They can be initialized to any random value, and after running the network on a pair (x, t) from the training data, the weight update rule (subsection 3.2) will alter them to minimize the error between the output $\hat{y}$ and the target t.*

Definition 2.3 (Bias)**.** *Biases are constant, they are an additional input into the next layer that will always have the value of 1. Bias units are not influenced by the previous layer, they guarantee that even when all the inputs are zeros there will still be an input in the neuron [AI-Wiki].*

When talking about neural networks and ML models in general, the term "**parameters**" often comes up. For example, OpenAI loves to brag about how many parameters their latest language models have (**GPT-4** has 1.76 trillion). Parameters are the elements of a neural networks refer to the variables that the model learns from the training data. These parameters are exactly the weights and biases defined above.

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 neural network** (FFN) 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 [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 (subsection 2.4), W is the weight matrix, and f is the activation function (subsection 2.5).

Definition 2.4 (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** (MLPs) for one simple reason: you can think of a neural network as the composition of multiple perceptrons. In this case, the perceptron would be a synonym for 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

FFNs 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. Layers are a general term given to a grouping of neurons that act together in the neural network, as seen in Figure 4.

The first layer in a neural network is the **Input Layer**, it receives the initial data for the network from the outside world. The "Entry point" of the neural network. Then, come the **hidden layer(s)**, which 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]. 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.

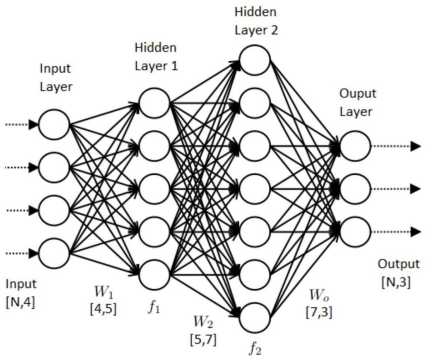

Figure 4: A graph representation of a Neural Network, from Aggarwal [2018a].

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 non-linearity 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 non-linearities 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

1. **Nonlinearity** is one of the most essential characteristics of an activation function. The non-linearity significantly improves the network's ability to learn and model complex, non-linear relationships in the data. If only linear activation functions were used, the entire network would be equivalent to a single linear transformation of the input, regardless of the number of layers. This would severely limit the network's ability to solve complex problems.
2. The activation function must be **computationally cheap** in order to reduce training costs.
3. It must be **bounded**, as gradient-based training approaches are more stable when the range of the activation function is finite.
4. 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 inputs and outputs of certain activation functions, like the logistic function (Sigmoid), can vary greatly. These functions compress a large input space into a smaller output range between [0,1]. This may cause the back-propagation algorithm to have almost no gradients to propagate backward through the network, and any remaining gradients diminish as they move down through the layers. This leaves the initial hidden layers with little to no gradient information. Using non-saturating activation functions, such as ReLU, is one way to solve this problem.*

We present some commonly used activation functions discussed in Jagtap and Karniadakis [2022].

1. *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.
2. *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.
3. *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.

Activation Function ($\Phi(x)$)	Derivatives ($\Phi'(x)$)	Range	Continuity
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 5: Common activation functions, their derivatives, range, and order of continuity, from Jagtap and Karniadakis [2022].

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

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

5. *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.

Figure 5 summarizes the information described above.

3 Learning

At the core of machine learning lies the art of learning itself. In this section, we delve into the essential components that drive this process forward: backpropagation, weight updates, gradient descent, and loss functions.

Backpropagation serves as the engine of learning in neural networks, refining model parameters through the propagation of error starting from the output, all the way to the first hidden layer. Accompanying this is the concept of loss functions, which quantify the disparity between predicted and actual outcomes, guiding the learning trajectory towards convergence.

In the subsections ahead, we dissect each facet of learning, providing insights into backpropagation, weight update rules, and unraveling the nuances of loss functions.

3.1 BackPropagation

Backpropagation is an algorithm for automatic differentiation that calculates the gradients of parameters in neural networks. The gradient estimate is used by optimization algorithms such as *Stochastic Gradient Descent* (subsubsection 3.3.2) 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** (subsection 3.2). 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.

Example 3.1. *Consider the network in Figure 6, 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*

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

where $w_{0,3}$ is the bias weight (Definition 2.3) at node 3. 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.

An even bigger problem than in the above example arises when we think of how we would compute the loss function in hidden layers. Whereas the error $y - h_w$ 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. Note that here, y is the target output, and h_w the value computed by the network.

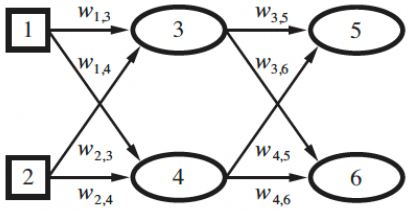

Figure 6: A simple network with 2 inputs, one hidden layer, and two outputs.

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].

Theorem 3.2 (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, illustrated in Figure 7. For the sake of simplicity, we view the neural network as a *Directed Acyclic Graph* G, where

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 $w_{i,j}$.

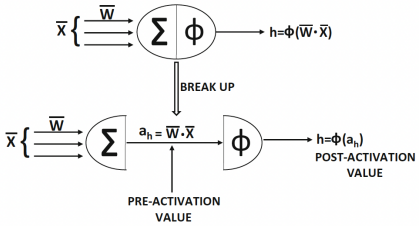

Figure 7: Pre and Post activation values of a neuron.

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

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

3.2 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.

□

3.3 Gradient Descent

Gradient descent is a method for minimizing an **objective function** $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$, namely, the weights and biases. This is achieved by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters. The learning rate η (see subsection 3.2) determines the size of the steps taken towards a (potentially local) minimum. Essentially, we move in the direction of the slope of the surface defined by the objective function, descending until we reach a valley [Ruder, 2017].

In the context of neural networks, the objective function would be the *loss function*, and in order to find the gradient, one would use *backpropagation*. It is common for developers to say they trained their model using back-propagation, but technically, this is incorrect. Back-propagation is not an optimization algorithm and cannot be used to train a model by itself. The term back-propagation is often misunderstood as the entire learning algorithm for multi-layer neural networks. In reality, back-propagation only refers to the gradient computation method, while another algorithm, like SGD, performs the learning using these gradients [Goodfellow et al., 2016]. Before exploring the different gradient descent algorithms, we define some terminology from Brownlee [2022].

Definition 3.3 (Sample)**.** *A sample is a single row of data containing inputs for the algorithm and an output for error calculation. A training dataset consists of many samples, also known as instances, observations, input vectors, or feature vectors.*

Definition 3.4 (Batch)**.** *The batch size is a hyperparameter defining the number of samples processed before updating the model parameters. A batch iterates over samples, makes predictions, calculates errors, and updates the model.*

Common mini-batch sizes are 32, 64, and 128. If the dataset size is not divisible by the batch size, the final batch will have fewer samples.

Definition 3.5 (Epoch)**.** *An epoch is a hyperparameter defining the number of times the entire training dataset is processed. Each epoch ensures all samples have an opportunity to update the model parameters. An epoch consists of one or more batches.*

The number of epochs is typically large, often in the hundreds or thousands, to minimize the model error sufficiently. Running too many epochs may lead to overfitting, as we'll see in subsection 4.1.

3.3.1 Vanilla Gradient Descent

Vanilla gradient descent computes the gradient of the loss function with respect to the parameters (i.e. the weights and biases) for the whole training set. In other words, the *batch size* is the size of the entire training set. So for each parameter θ, the update would be,

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Since calculating the gradients for the entire dataset to perform a single update can be very slow, vanilla gradient descent is impractical for large data sets. The code for vanilla gradient descent might look like this [Ruder, 2017]:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

For a pre-defined number of *epochs*, we first compute the gradient vector `params_grad` of the loss function for the entire dataset with respect to our parameter vector `params` using, say, *backpropagation*.

We then update our parameters in the direction of the gradients, with the learning rate determining the size of each update. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces [Ruder, 2017].

3.3.2 Stochastic Gradient Descent

The volume of data has grown so much in recent years that the processing power available is insufficient to train neural networks by performing gradient descent on the entire training set at each epoch. As a result, neural networks employ **stochastic gradient descent** (SGD), which reduces machine computation time.

SGD is a method that optimizes gradient descent, making it less costly. SGD alters the batch gradient descent algorithm by computing the gradient for just one training sample in each iteration before updating the parameters. Equivalently, the batch size for SGD is equal to 1. We define an example as a pair $(x^{(i)}, y^{(i)})$ where $x^{(i)}$ is the training input and $y^{(i)}$ the label. The update is then [Ruder, 2017],

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta : (x^{(i)}, y^{(i)}))$$

The steps for performing SGD are as follows:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Notice that the data is shuffled before each iteration. This is done to prevent bias that would occur if the data were trained in a specific order. A downside of SGD is that it doesn't converge as surely as vanilla gradient descent to the minimum of the "basin" where the parameters were initialized. Indeed, since SGD updates the parameters after every gradient computation on a single sample, it fluctuates a lot, as seen in Figure 8. This may cause it to jump to a new local minima. Research indicates that by gradually decreasing the learning rate with a **learning rate schedule**, SGD exhibits similar convergence behavior to vanilla gradient descent, nearly always converging to a local minimum for non-convex functions and to the global minimum for convex functions [Ruder, 2017].

3.3.3 Mini-Batch Gradient Descent

Finally, Mini-batch is the middle ground between SGD and Vanilla gradient descent. Instead of computing the gradient on one sample, or on the entire data set, we compute it on a fixed number n of samples,

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta : (x^{(i:i+n)}, y^{(i:i+n)}))$$

This approach reduces the variance of parameter updates, leading to more stable convergence, and leverages highly optimized matrix operations in modern deep learning libraries, making gradient computation with mini-batches very efficient. Common mini-batch sizes range from 50 to 256, though they can vary depending on the application. Mini-batch gradient descent is typically the preferred algorithm for training neural networks, and the term SGD is often used even when mini-batches are employed [Ruder, 2017]. The code for mini-batch gradient descent with a batch size of 50 would be,

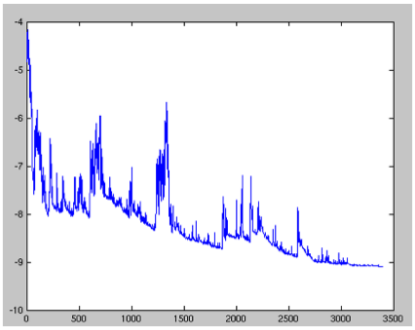

Figure 8: SGD fluctuation, from [Ruder, 2017]

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Now, there exist many more gradient descent algorithms, such as *Nesterov Accelerated Gradient, Adagrad, Adadelta, Adam,* and many more. For more information on the subject, we refer the reader to Ruder [2017].

3.4 Loss Functions

The attentive reader will have noticed that supervised machine learning revolves around optimizing the outputs of a loss function. This also applies to a wide range of other machine learning strategies. Thus, defining a good loss function is essential. We offer an overview of the most widely utilized loss functions for a variety of uses, starting with **Regression Losses**.

3.4.1 Regression Loss Functions

Regression losses are loss functions used to solve the regression problems of *supervised ML*. Recall that regressions model predict the output of a continuous output variable. The regression losses are all based on residuals, namely the difference between the predicted and expected outputs. In the following, let x_i be the i^{th} element of input x, $f(x_i)$ the i^{th} element of the predicted output, and y_i the i^{th} element of the expected output. The following represent commonly used regression losses,

1. *Mean Bias Error Loss [Continuous, Differentiable].* The Mean Bias Error loss is the most basic loss function, it is given by,

$$\mathcal{L}_{MBE}(y_i, f(x_i)) = \frac{1}{n}\sum_{i=1}^{n}[y_i - f(x_i)] \tag{3}$$

   It measures the average bias in the prediction, but because positive errors have the ability to cancel out negative ones and create an incorrect parameter estimate, it is rarely used as the loss function to train regression models. However, it serves as the foundation for the ensuing loss functions and is frequently employed to assess the models' performances [Ciampiconi et al., 2023].

2. *Mean Absolute Error Loss (L1) [Lipschitz-Continuous, Convex].* The Mean Absolute Error loss is one of the most fundamental loss functions for regression; it measures the average of

the absolute bias in the prediction. The absolute value overcomes the problem of the MBE by ensuring that positive errors do not cancel the negative ones. It is defined as,

$$\mathcal{L}_{MAE}(y_i, f(x_i)) = \frac{1}{n}\sum_{i=1}^{n}|y_i - f(x_i)| \tag{4}$$

Notice that the contribution of the errors follows a linear behaviour, implying that many small errors have as much impact as a big one. This implies that the gradient magnitude is not dependent on the error size, thus leading to convergence problems when the error is small. A model trained to minimize the MAE performs well when the target data conditioned on the input is symmetric [Ciampiconi et al., 2023].

3. *Mean Squared Error Loss (L2) [Continuous, Differentiable, Convex].* The Mean Squared Error loss is a well-known and simple loss function for regression. It is given by

$$\mathcal{L}_{MSE}(y_i, f(x_i)) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{5}$$

The squared term makes all the biases positive and magnifies the contribution made by outliers, making it more suitable for problems where noise in the observations follows a normal distribution. The sensitivity to the outliers is the primary disadvantage [Ciampiconi et al., 2023].

4. *Root Mean Squared Error Loss [Continuous, Differentiable, Convex].* The Root Mean Squared Error loss is, apart from the square root term, identical to MSE. It main benefit is that the loss has the same units and scale as the relevant variable. The minimization procedure converges to the same optimal value as MSE. However, the RMSE may take different gradient steps depending on the optimization method employed [Ciampiconi et al., 2023]. It is defined as,

$$\mathcal{L}_{RMSE}(y_i, f(x_i)) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2} \tag{6}$$

5. *Huber Loss [Lipschitz-Continuous, Differentiable, Strictly Convex].* Huber loss is a mix of MAE and MSE. When the residuals are sufficiently small, it goes from MAE to MSE. It is parameterized by δ, which indicates the point at which MAE and MSE transition. This enables it to combine the benefits of the MAE and the MSE. When there is a significant discrepancy between the model's output and prediction, the Huber loss becomes less susceptible to outliers since the errors are linear. On the other hand, if the error is tiny, it follows the MSE, which accelerates convergence and makes it differentiable at 0. A crucial decision is which δ to use, which may be modified often throughout the training process depending on what constitutes an outlier [Ciampiconi et al., 2023]. The Huber loss is given by,

$$\mathcal{L}_{Huber}(y_i, f(x_i)) = \sum_{i=1}^{n}\alpha_i(y_i, f(x_i)), \tag{7}$$

where,

$$\alpha_i(y_i, f(x_i)) = \begin{cases} \frac{1}{2}(y_i - f(x_i))^2, & \text{if } |y_i - f(x_i)| \leq \delta \\ \delta\left(|y_i - f(x_i)| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases}$$

6. *Log-cosh Loss [Continuous, Differentiable].* The Log-cosh loss is given by the logarithm of the *hyperbolic cosine* of the residuals between the actual value y and the forecasted value $f(y_i)$. It is more computationally demanding than *Huber loss*, but it provides all the same benefits without the need to establish a hyperparameter. Moreover, the log-cosh loss has the advantage of being twice differentiable, which makes it appropriate for techniques that need to solve the second derivative [Ciampiconi et al., 2023]. It is defined as,

$$\mathcal{L}_{Logcosh}(y_i, f(x_i)) = \frac{1}{n}\sum_{i=1}^{n}\log[\cosh(y_i - f(x_i))] \tag{8}$$

7. *Root Mean Squared Logarithmic Error Loss [Continuous, Differentiable, Convex].* The Root Mean Squared Logarithmic Error (RMSLE) loss is given by,

$$\mathcal{L}_{RMSLE}(y_i, f(x_i)) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}[\log(y_i + 1) - \log(f(x_i) + 1)]^2} \tag{9}$$

When it comes to RMSE, the only distinction is that the logarithm is used on both the observed and the anticipated values. The logarithm's plus one term permits zero values for $f(x_i)$. The RMSLE is more resilient to outliers because of the logarithm, as well as the relative inaccuracy between the expected and actual values. In particular, the RMLSE's size does not increase in proportion to the error's magnitude. Rather, when both the anticipated and actual values are high, data points with large residuals are not penalized as much. Because of this, the RMSLE is a viable option for problems where the targets have an exponential relationship or when penalizing underestimates more heavily than overestimates is desirable. This loss, however, is inappropriate for problems where negative values are permitted [Ciampiconi et al., 2023].

3.4.2 Classification Loss Functions

The second subset of supervised ML, *Classification*, also has its own set of loss functions. The following consists of the most commonly used *Margin-based* Classification losses.

1. *Zero-one Loss.* The Zero-one loss is the most basic classification loss function. It is defined as,

$$\mathcal{L}_{Zero-One}(y, f(x)) = \begin{cases} 1, & \text{if } f(x) \cdot y < 0 \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

In practice, the Zero-one loss can't be used since it is not convex, nor differentiable.

2. *Hinge Loss [Lipschitz-continuous, Convex].* The Hinge loss is among the most famous loss functions for classification. It is given by,

$$\mathcal{L}_{Hinge}(y, f(x)) = \max(0, 1 - f(x) \cdot y) \tag{11}$$

The two main drawbacks of Hinge loss are that it is sensible to outliers, and that its derivative is discontinuous at $f(x) \cdot y = 1$. The latter makes it harder to optimise.

3. *Quadratically Smoothed Hinge Loss [Lipschitz-continuous, Convex, Differentiable].* The Quadratically Smoothed Hinge loss is a smoothed version of the Hinge loss, making it easier to optimize.

$$\mathcal{L}_{QSmoothedHinge}(y, f(x)) = \begin{cases} \frac{1}{2\gamma}\max(0, -(f(x) \cdot y))^2, & \text{if } f(x) \cdot y \geq 1 - \gamma \\ 1 - \frac{\gamma}{2} - f(x) \cdot y, & \text{otherwise.} \end{cases} \tag{12}$$

The hyperparameter γ gives the degree of smoothing. As $\gamma \to 0$, $\mathcal{L}_{QSmoothedHinge}(y, f(x)) \to \mathcal{L}_{Hinge}(y, f(x))$.

4. *Modified Huber Loss [Lipschitz-continuous, Differentiable, Strictly convex].* The Modified Huber loss is a version for classification. It is a special case of the Quadratically Smoothed Hinge Loss with $\gamma = 2$. We define it as,

$$\mathcal{L}_{ModHuber}(y, f(x)) = \begin{cases} \frac{1}{4}\max(0, -(f(x) \cdot y))^2, & \text{if } f(x) \cdot y \geq -1 \\ -(f(x) \cdot y), & \text{otherwise.} \end{cases} \tag{13}$$

5. *Ramp Loss [Continuous, Convex].* The Ramp loss is more robust to outliers than the Hinge loss. It is given by,

$$\mathcal{L}_{Ramp}(y, f(x)) = \begin{cases} \mathcal{L}_{Hinge}(y, f(x)), & \text{if } f(x) \cdot y \leq 1 \\ 1, & \text{otherwise.} \end{cases} \tag{14}$$

Other loss functions based on *Information Theory*, such as the *Cross-Entropy loss*, and the *Kullback-Leiber divergence* are widely used in classification problems, and we refer the reader to [Ciampiconi et al., 2023] for more information.

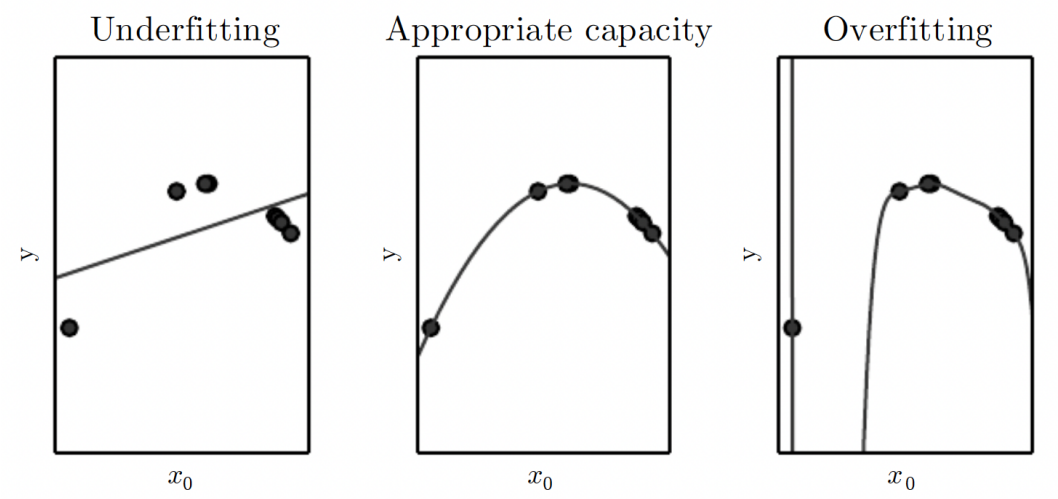

Figure 9: Diagrams representing Underfitting, Appropriate, and Overfitting, from [Goodfellow et al., 2016]

4 Fitting the Model

Amidst the pursuit of learning, lurk the pitfalls of overfitting and underfitting. Overfitting occurs when a model becomes overly tailored to the training data, while underfitting results in oversimplified representations. Finding the right balance between generality and complexity is essential, achieved through techniques like regularization and cross-validation.

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.

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.*

As seen in Figure 10, an overfit model will have low *bias*, but high *variance* on the training data. However, as explained in Burnham and Anderson [2002], 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."

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.*

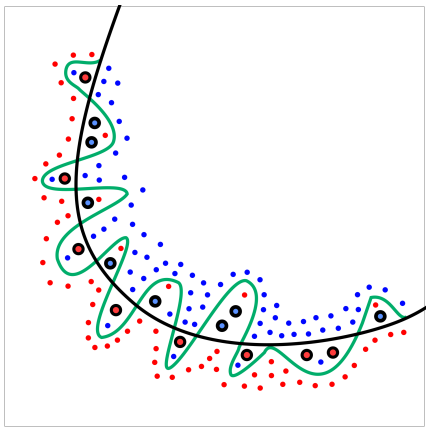

Figure 10: A diagram showing overfitting (green line) of data from [Wikipedia, the free encyclopedia]

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.

The **bias-variance tradeoff** is a fundamental concept in machine learning that deals with the trade-off between a model's bias and its variance. *Bias* refers to the error introduced by approximating a real-world problem with a simplified model, often resulting in *underfitting*. *Variance*, on the other hand, reflects the model's sensitivity to small fluctuations in the training data, potentially leading to *overfitting*. Balancing bias and variance is crucial for achieving optimal predictive performance: reducing bias typically increases variance and vice versa. Finding the right balance involves selecting an appropriate model complexity and regularization techniques to minimize both sources of error. Figure 11 demonstrates the optimal point of capacity.

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 tune the model optimally are part of a subset of ML called *Regularization*, that we'll cover in the following sections.

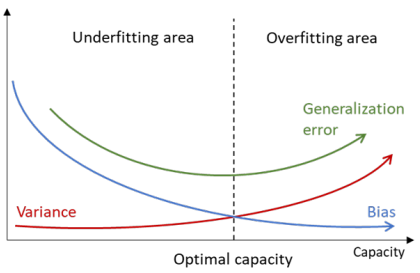

Figure 11: A diagram showing the optimal capacity of a model and its relation with the *bias-variance trade-off*, from [Kowalik, 2023].

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 that doesn't properly 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**, which 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 [Goodfellow et al., 2016]. The following is the pseudo-code for the k-fold cross validation algorithm.

Algorithm 1: k-fold cross-validation

Function `KFoldXV` ($\mathbb{D}, A, L, k$) **:**
  Require: $\mathbb{D}$, the given dataset, with elements $\boldsymbol{z}^{(i)}$
  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 $\boldsymbol{z}^{(i)} \in \mathbb{D}$ to a scalar $\in \mathbb{R}$
  Require: k , the number of folds
  Split $\mathbb{D}$ into k mutually exclusive subsets $\mathbb{D}_i$, whose union is $\mathbb{D}$
  for *i from 1 to k* **do**
    $f_i = A(\mathbb{D}\ \mathbb{D}_i)$;
    for $z^{(j)}$ *in* $\mathbb{D}_i$ **do**
      $e_j = L(f_i, z^{(j)})$;
  return e;

4.5 Regularization Techniques

Regularization is defined as any change we make to a learning algorithm with the goal of lowering its generalization error but not its training error. Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization [Goodfellow et al., 2016]. Finding effective regularization methods is a hot research area in the field.

A significant portion of machine learning involves creating various models and algorithms tailored to suit them. Techniques like cross validation help us empirically determine the most suitable approach for our specific problem. Yet, there isn't one "best" model universally applicable, this concept is often referred to as the "no free lunch theorem". The rationale behind this theorem lies in the fact that assumptions effective in one domain may not perform well in another [Murphy, 2012].

In light of this theorem, it becomes imperative to develop diverse models to accommodate the wide array of real-world data. Additionally, for each model, there exists a multitude of algorithms offering different tradeoffs between speed, accuracy, and complexity [Murphy, 2012].

4.5.1 L2 Parameter Norm Penalty (Weight Decay)

Real-world data possesses intricate attributes, requiring equally intricate models to address them. While reducing parameters represents one approach to mitigate model complexity, it remains a rather restrictive tactic, since the more parameters our model has, the more interconnections within our neural network, thereby greater non-linearities, crucial for representing complex data.

Yet, we must exercise caution to prevent these interconnections from spiraling out of control. Thus, the notion of penalizing complexity emerges as a solution. By employing weight decay, we retain our many parameters while imposing constraints to prevent excessive model complexity.

A method to impose a penalty on complexity involves adding all parameters (weights) into our loss function. However, a straightforward addition isn't viable due to the mix of positive and negative parameters. Thus, we resort to adding the squares of these parameters to the loss function. Although effective, this approach might inflate the loss to an extent where the optimal solution would entail setting all parameters to zero. To avert this scenario, we scale down the sum of squares by a smaller factor, termed *weight decay* [Vasani, 2019]. Weight decay stands out as a widely adopted technique in training numerous cutting-edge deep networks, including prominent models like **GPT-3** [Andriushchenko et al., 2023].

We provide the definition from Andriushchenko et al. [2023]. Let $(x_i, y_i)_{i=1}^n$ be the training inputs and labels where $x_i \in \mathcal{D}$, $y_i \in \mathbb{R}^c$, and c the number of classes. Let $h : \mathbb{R}^p \times \mathcal{D} \to \mathbb{R}^c$ be the hypothesis class of neural network and for any parameter $w \in \mathbb{R}^p$ where the function $h(w, \cdot) : \mathcal{D} \to \mathbb{R}^c$ represents the network predictions. The training loss $\mathcal{L}$ and the ℓ_2-regularized training loss $\mathcal{L}_\lambda$ are given as:

$$\mathcal{L}(w) := \frac{1}{N} \sum_{i=1}^{N} \ell\left(y_i, h(w, x_i)\right), \quad \mathcal{L}_\lambda(w) := \mathcal{L}(w) + \frac{\lambda}{2} w^2, \tag{15}$$

where $\ell(\cdot, \cdot) : \mathbb{R}^c \times \mathbb{R}^c \to \mathbb{R}^p$ denotes the chosen loss function, such as cross-entropy loss. Then, simply use gradient descent with respect to $\mathcal{L}_\lambda$.

We note that choosing an appropriate λ is a difficult problem. No exact method exists as of today. Reasonable values, however, lie between 0 and 0.1 [Kuhn and Johnson, 2013].

4.5.2 Data Augmentation

While weight decay was a regularization method that focussed on altering the model, we now see a technique that alters the data instead, to achieve similar results. Optimizing a machine learning model's ability to generalize is often best achieved by training it on a larger dataset. However, practical constraints limit the amount of available data. To address this limitation, one strategy involves augmenting the training set with "fake" data. This approach is straightforward for classification tasks, by generating new (x, y) pairs through transformations of existing input data in the training set, we can effectively expand the dataset.

Dataset augmentation has demonstrated particular effectiveness in certain classification tasks, notably object recognition. Images, being high-dimensional, encompass a wide array of variation factors, many of which can be simulated easily. Operations such as translating, rotating, or scaling images have proven beneficial, even when the model is designed with partial translation invariance through convolution and pooling techniques.

It's important to be careful when applying transformations to data. For instance, in optical character recognition tasks, correctly identifying distinctions like 'b' versus 'd' and '6' versus '9' is essential.

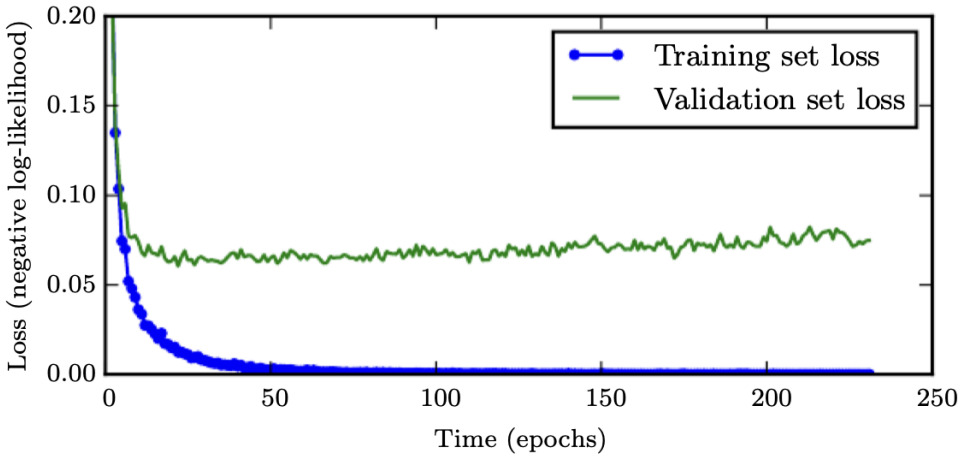

Figure 12: Graph from Goodfellow et al. [2016]. The curves depict the evolution of the negative log-likelihood loss over the number of training iterations or epochs. Initially, the training objective steadily decreases over time as the model learns from the data, yet the validation set loss eventually starts to rise again after reaching a minimum. This pattern indicates that despite improvements in the training loss, the model's performance on unseen data begins to degrade over time, highlighting the phenomenon of overfitting.

Therefore, transformations like horizontal flips are inappropriate for augmenting datasets in these cases, clearly [Goodfellow et al., 2016].

4.5.3 Early Stopping

When training large models with great complexity, it's common to observe a phenomenon where *training error* steadily decreases over time, while *validation set* error starts to rise again, as illustrated in Figure 12. Consequently, it's possible to achieve a model with improved validation set error (and ideally, better test set error) by reverting to the parameter setting at the point when validation set error was lowest. To implement this, we save a copy of the model parameters each time validation set error improves. Upon termination of the training algorithm, we return these parameters instead of the latest ones. The algorithm concludes when no parameters show improvement over the best recorded validation error for a specified number of iterations [Goodfellow et al., 2016].

4.5.4 Dropout

Dropout is yet another technique to address the regularization problem. The core concept is to randomly deactivate units, along with their connections, within the neural network during training. This prevents units from becoming overly reliant on each other. Throughout training, dropout samples from an array of "thinned" networks. At the testing phase, approximating the ensemble effect of these thinned networks can be achieved by utilizing a single unthinned network with reduced weights. This approach significantly mitigates overfitting and surpasses other regularization techniques. Dropout has demonstrated its efficacy in enhancing the performance of neural networks across various supervised learning tasks, including vision, speech recognition, document classification, and computational biology. Notably, it has achieved state-of-the-art results on numerous benchmark data sets [Srivastava et al., 2014].

5 Conclusion

As we've journeyed through the amazing realm of supervised machine learning, we've seen how powerful it can be, from the basic concept of neural networks, all the way to the more advanced method of backpropagation. It becomes evident, however, that the reliance of supervised learning on carefully labeled datasets poses a significant limitation. The need for abundant, high-quality

data, paired with corresponding inputs and desired outputs, can be restrictive and often impractical in real-world scenarios. This limitation becomes particularly apparent when faced with complex tasks where defining the desired outputs may be challenging or even impossible.

This is the basis of the power of reinforcement learning (RL), an approach that learns from interacting directly with the environment, surpassing the limitations of supervised learning. In RL, agents move through a set of options, learning the best methods by making mistakes and getting feedback in the form of rewards. Due to its self-learning mechanism, RL can now address issues for which explicit supervision is impractical, providing access to previously uncharted territory for conventional machine learning techniques.

Consider, for instance, the task of training a model to beat the world's best chess player. In such a scenario, the sheer complexity of the game and the infinite array of possible moves render the notion of a predefined dataset ridiculous to even think about. Supervised learning falls short in this context, as the desired optimal moves to win the game are elusive and ever-changing. Here, RL emerges as the preferred approach, empowering agents to learn from experience, adapt strategies in real-time, and ultimately, achieve superhuman performance.

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.

AI-Wiki. Weights and biases. URL https://machine-learning.paperspace.com/wiki/weights-and-biases. [Online; accessed April 30, 2024].

Maksym Andriushchenko, Francesco D'Angelo, Aditya Varre, and Nicolas Flammarion. Why do we need weight decay in modern deep learning?, 2023.

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. Difference between a batch and an epoch in a neural network, 2022. URL https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/. [Online; accessed May 14, 2024].

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].

Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Springer, 2013. ISBN 978-1-4614-6848-6.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012. URL https://probml.github.io/pml-book/book0.html.

Thomas Rincy N and Roopam Gupta. A survey on machine learning approaches and its techniques:. In *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*, pages 1–6, 2020. doi: 10.1109/SCEECS48394.2020.190.

Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

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

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56): 1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Dipam Vasani. This thing called weight decay, Nov 2019. URL https://towardsdatascience.com/this-thing-called-weight-decay-a7cd4bcfccab.

Katrina Wakefield. A guide to the types of machine learning algorithms and their applications. URL https://www.sas.com/en_gb/insights/articles/analytics/machine-learning-algorithms.html. [Online; accessed May 5, 2024].

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