Chapter 4

Artificial Neural Network Background

The world of *Artificial Intelligence* (AI) has gone through multiple intense phases of development and decline since its early years in the 1950s. A period of heavily reduced interest and funding after times of success in the field of AI was later coined with the term *AI winter*. So far, there have been plenty of AI winters, with the two major ones taking their toll between 1973–1979 and 1988–1994. As of now, however, the current wave of interest in deep learning has provided radical solutions to numerous convoluted tasks that no previous AI advancement period ever did. It is eminently possible that since AI's latest take-off, the scientific community may not go through an upcoming AI winter any time soon [115].

Perhaps the most important pillar of modern AI, is the *Artificial Neural Network* (ANN), or simply *Neural Network* (NN), which is a collection of connected units called *artificial neurons* or simply *neurons*. Throughout the document, the word *"network"* will also be used to mean an artificial neural network. Their name is attributed to many people believing that they loosely model biological neurons inside a brain. Each connection of neurons forms an edge, and like the synapses in a brain, it is responsible for transmitting signals to other neurons, which in our case are real numbers. Edges typically have a *weight* associated with them, and the stronger it is, the stronger their transmitted signal becomes, and vice versa. In section 4.1, neurons are shown to be aggregated into *layers*.

The purpose of a neural network is to adjust its trainable parameters (mostly weights) accordingly in order to find a sound relationship between the input and the desired target data. In other words, it is to discover the underlying function that connects those two sets as closely as possible, given that such a function exists in the first place. The procedure bound for doing so is called *training* (section 4.6), in which the neural network is being "taught" of this relationship by processing given *ground truth*[1] input and output example pairs.

[1]The term "ground truth" refers to information that is known to be real or true, which in this case will be direct real-world measurements.

The training of a neural network is usually achieved by the act of continually trying to minimize the difference between its estimated outputs and the ground truth targets. This difference is also known as the *error* or the *loss*, and its exact numeric values are determined by the definition of the neural network's *loss function* (section 4.3). Minimizing the loss function is attained via adjusting its trainable parameters accordingly, as dictated by an algorithm called the *optimizer* (section 4.4) of the neural network. After a sufficient number of these adjustments, the training session terminates based upon certain convergence criteria, i.e., when no further significant beneficial adjustment is expected to occur. This training methodology is widely known as *supervised learning*.

A neural network should be employed when dealing with problems that all other methods, such as deterministic, statistical, optimization, etc., of discovering a qualitative enough input-output relationship have been either exhausted or deemed ineffective due to the problem's profound complexity. Examples of such problems include recognition or classification of images, videos, speech, natural language processing, fraudulent transaction identification et al., including the one tackled by this thesis being speech synthesis.

Due to their distinguished power in dealing with formidable tasks like these, neural networks have received much attention in recent decades. Recently, they have been tasked with attempting projects one could have thought of being unworkable to be carried out algorithmically/computationally some decades ago, such as driving an actual automobile [16, 83, 101]. In this chapter, some fundamental concepts of neural networks are explored that are essential for understanding the final proposed model.

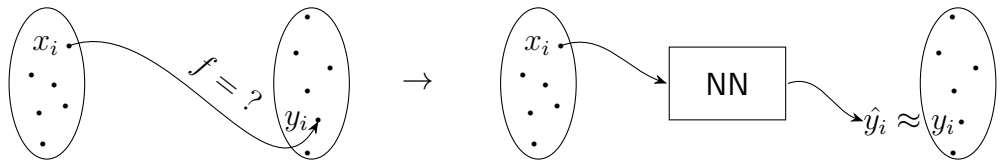


Figure 4.1: The inputs of our problem are x_i and the corresponding ground truth output y_i. Assume a difficult enough problem where coming up with a function f that maps the input to the output set seems intractable with all other available methods. An adequately designed neural network may find a solid enough relationship between the two sets, such that given an input x_i, its estimated output $\hat{y}_i$ is adequately close enough to the ground truth y_i.

4.1 Layers

Layers are made of neurons and are fundamental blocks of neural networks, while their uses vary depending on the goal in pursuit. Over the years, many different types of layers have been developed because of the need to perform a different operation/transformation of a layer's input to get the output. A neural network may be comprised of multiple layers of the same kind, or a collection of different ones in almost any order or succession wanted. A *sequential architecture*, for example, is one where all layers of the model are linearly stacked, i.e., each layer has only one layer from which it receives its input and only one layer that receives its output. The very first and last layers are usually referred to as *input* and *output* layers of the neural network, respectively, while all the in-between layers are also referred to as *hidden layers*.

Depending on its size and type, any kind of trainable layer will introduce several different parameters to be tuned during training. Typically those are the *weights* (noted by W, w, or U, u) and the *biases* (noted by b). Depending on the layer's defined operation, these weights are applied to its input x, and by adding the biases b, we get its resulting output y. This output will then be fed as input into the next layer and so forth until the output layer delivers the final estimation of the model. The collection of all weights and biases of the network basically forms the parameters θ of the function f discussed above (fig. 4.1), which maps the input to the output set of a given problem.

There are also layers that are simply used to apply an operation (e.g., multiplication, averaging, upsampling et al.) without introducing any trainable parameters. Usually, they are added in order to prepare the data accordingly for the upcoming layers of the network but may have other uses as well, such as taking place after the network's output layer to formulate the final prediction/estimation of the overall model.

Aside from the architecture of a neural network model, including its amount of layers, size, type, order of succession, etc., there can also be many other types of parameters to choose from. Within each specific layer, those can be weight initializers, strides, dilation rate, padding, et al., and are all considered *hyperparameters* of the layer. To successfully train a neural network, choosing a right set of all the different hyperparameters that exist within the network's layers is deeply important.

Assembling a new and decent neural network after figuring out all the various parameters and choices briefly described above is a markedly demanding task. It requires familiarity with the specific problem, creativity, competence, and general knowledge within the sphere of artificial neural networks, plus being well-informed on the latest studies conducted in the field. Even so, plenty of parameters are still so tough to come up with, to the point where they are, inevitably, to this day, a product of guessing, countless trial-and-error procedures, or following other ideas that have been shown experimentally to work in practice by past research studies.

4.1.1 Fully Connected Layer

A *Fully Connected* (FC), *Dense*, or *Linear* layer is one of the most popular types of layers, and is basically an $\mathbb{R}^m \to \mathbb{R}^n$ function. It is one where each of its neurons receives input from all neurons of its previous layer (see fig. 4.2). In linear algebra terms, a fully connected layer's output vector y is given by multiplying the input vector x by the weight matrix W before adding the bias vector b as so:

$$y = Wx + b, \text{ with} \tag{4.1}$$

$$y_i = \sum_{j=1}^{N_{\text{in}}} [w_{i,j} \cdot x_j] + b_i, \ i \in [\![1, N_{\text{out}}]\!]. \tag{4.2}$$

For historical reasons, an artificial neural network based solely on fully connected layers is known as a *Multilayer Perceptron* (MLP), but this term can be ambiguous sometimes because it is often used loosely to mean any artificial neural network. Since they condition every single value of their output to all of their input values, dense layers have great power in their representation capabilities. Despite their simple concept, however, they also come with the downside being a large number of trainable parameters (i.e., weights and biases) they introduce due to this "brute-force" configuration.

Let N_{out} be the number of output neurons of a dense layer and N_{in} be its input number of neurons. The total number of trainable parameters $\mathrm{P_W}$ for a fully connected layer is, therefore, the product between N_{out} and N_{in} plus the bias, which is one per weight:

$$\mathrm{P_W} = N_{\text{out}}(N_{\text{in}} + 1). \tag{4.3}$$

There are not many hyperparameters to tune in a dense layer:

Name	**Symbol**	**Description**
Size	N_{out}	Total number of output neurons.
Bias flag	$\mathbb{1}\{b\}$	Whether to use biases or not.
Weight initializer	p_W	Weight initialization distribution.
Bias initializer	p_b	Bias initialization distribution.

In a classification task, fully connected layers usually form the last few layers in a network, mapping the features extracted from the data by, e.g., convolutional (explained in the upcoming section 4.1.2) or other layers to the predicted classes. In cases where the input or output size is relatively small enough so that not a sizable number of parameters is introduced, they can also be used for extracting features.

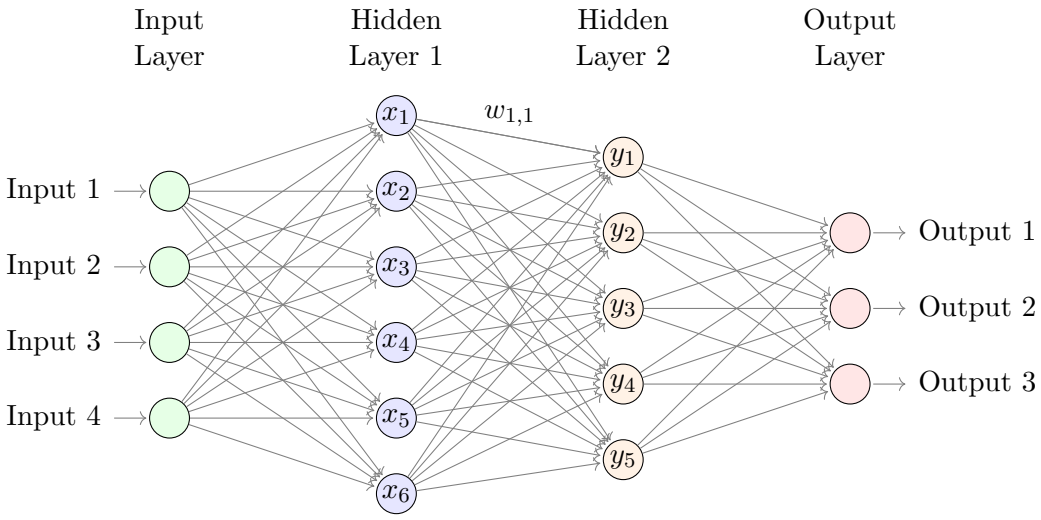


Figure 4.2: A pictorial illustration of a toy network with three dense layers. The linear operation is represented from the first to the second hidden layer, where each edge connecting two neurons represents a different weight. Using eq. (4.3), the total number of trainable parameters for the second hidden layer are $5 \cdot (6 + 1) = 35$.

4.1.2 Convolutional Layer

As its name signifies, a *convolutional layer* (commonly abbreviated as *conv*) seems to be based on the convolution operation, as previously defined in eq. (2.19) for the DSP chapter, but is actually more accurately described by the cross-correlation operation (eq. (2.20)). Therefore, a Convolutional Neural Network (CNN) is a type of neural network based predominantly on convolutional layers. Early on, their success was mostly found in problems involving images, such as image classification, but over the years, convolutional layers have proven helpful in many different domains of study, including speech processing.

Convolutional layers are commonly used for feature extraction based on local information, and they can detect both high and low-level features. The operation can be generalized for any number of dimensions, but regarding the scope of this thesis, it is not needed to go further than a 1D convolutional layer. The equation describing convolution in 1D discrete space between an input sequence x and an output sequence y using one kernel w is:

$$y_j = \sum_{k=1}^{K} \left[w_k \cdot x_{j'} \right] + b, \;\; j' = j - 1 + k. \tag{4.4}$$

A convolutional layer allows us to have many different input sequences, i.e., *input channels*. One can also train multiple kernels over the input, thus resulting in a different output sequence per set of kernels, i.e., *output channel*. The entire set

of kernels is usually called the *filter* of the convolutional layer, although the two terms are sometimes used interchangeably.

By default, a kernel shifts by one value, but there is the choice of shifting it by more values; this number of shifts S is the *strides*. With strides $S = 1$, the kernel shift is by one value each time giving us the default convolution. Taking these into account, a more generalized version of the convolutional layer with C_{in} input channels, S being the strides, and K the kernel size, is:

$$y_{i,j} = \sum_{i'=1}^{C_{\text{in}}} \sum_{k=1}^{K} \left[w_{i',k}^{(i)} \cdot x_{i',j'} \right] + b_i, \; j' = S(j-1) + k. \tag{4.5}$$

As it is apparent, convolution reduces the dimensionality of the output with respect to the input unless the kernel and strides are of size one, in which case both dimensions will be the same. However, there is, most of the time, the need to preserve the input-output dimensionality without being restricted to using filters of size one, and this can be done by *padding* the input. Padding occurs when the input sequence is extended by constant values (typically zeros) appended both before and after the sequence. A padding of two, i.e., $P = 2$, for example, appends two extra zeros at the beginning and end of the input. This technique also allows for the corner/edge values of the input to be fully convolved with the filter, contrary to using no padding where the first and last K values are not fully convolved by the filter, as it would exceed the input size.

Another option is to use *dilated* (also known as *atrous* from the French "à trous" meaning "with holes") convolution, which expands a filter by introducing "holes", i.e., zeros, between the consecutive elements of each kernel. This can also be interpreted as skipping values, and the dilation factor d indicates the number of skips. With dilation factor $d = 1$, we get the default convolution.

Given input channels of length D_{in}, kernel sizes K, strides S, dilation factor d, padding P, then considering C_{out} output channels, the size D_{out} of each will be:

$$D_{\text{out}} = \left\lfloor \frac{D_{\text{in}} + 2P - d(K-1) - 1}{S} + 1 \right\rfloor, \; D_{\text{in}} \geq K. \tag{4.6}$$

The trainable parameters in a convolutional layer are essentially the values of the filters and the biases. A different set of kernels (i.e., filter) is needed per output channel, and each set of kernels will be of size equal to the number of input channels since one filter can take care of one input channel. So, in total, there will be C_{out} sets of C_{in} kernels per set, where each kernel is of the same size K and has one bias term. Therefore, the total number of trainable parameters P_{W} in a 1D convolutional layer is given by the formula:

$$\text{P}_{\text{W}} = C_{\text{out}}(C_{\text{in}}K + 1). \tag{4.7}$$

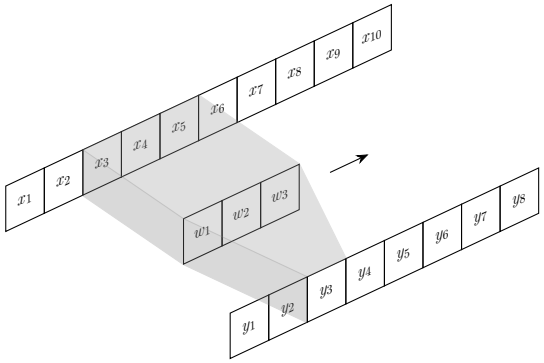


Figure 4.3: One can think of convolving as sliding a window over the input. Here, an input sequence of length $D_{\text{in}} = 10$ is convolved with a kernel of size $K = 3$. By setting strides $S = 1$, no dilation $d = 1$, no padding $P = 0$, the output sequence will be of length $D_{\text{out}} = \lfloor (10 + 2 \cdot 0 - 1 \cdot (3 - 1) - 1)/1 + 1 \rfloor = 8$, using eq. (4.6).

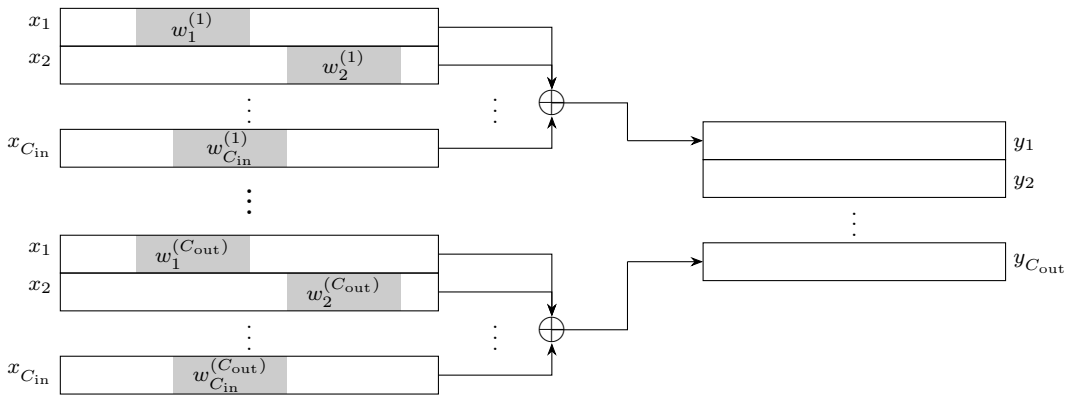


Figure 4.4: An input matrix with C_{in} input channels represented each by a different row is being convolved with C_{out} sets of kernels, equal to the number of output channels. After convolving each input sequence $x_{i'}$ with its corresponding filter $w_{i'}^{(i)}$, all the outputs are added together to form the output sequence y_i.

Another way of thinking about a convolutional layer is that of a restricted fully connected layer. From eq. (4.5), it can be understood that the convolution is a linear operation, so it can be written equivalently as an operation done by a dense layer (eq. (4.1)). The input x of the dense layer in this case is obtained by concatenating the input channels of the convolution $[x_1 | \cdots | x_{C_{\text{in}}}]^\top$, and the weight

matrix for the i^{th} channel is $W^{(i)} = \left[W_1^{(i)} | \cdots | W_{C_{\text{in}}}^{(i)}\right]$, with:

$$W_{i'}^{(i)} = \begin{pmatrix} w_{i',1}^{(i)} & w_{i',2}^{(i)} & \cdots & w_{i',K}^{(i)} & 0 & \cdots & 0 & 0 \\ 0 & w_{i',1}^{(i)} & \cdots & w_{i',K-1}^{(i)} & w_{i',K}^{(i)} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & w_{i',K}^{(i)} & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & w_{i',K-1}^{(i)} & w_{i',K}^{(i)} \end{pmatrix}, \tag{4.8}$$

which is known as a *Toeplitz*, or a *diagonal-constant* matrix.

All the hyperparameter options of a convolutional layer all together are summarized in the following table:

Name	**Symbol**	**Description**
Number of filters	C_{out}	The number of output channels to produce.
Kernel size	K	The length/size of each kernel.
Padding	P, P_v	Apply the P_v constant P times on both sides of a channel.
Strides	S	Number of value shifts $S-1$ of each filter over the input.
Bias flag	$\mathbb{1}\{b\}$	Whether to use biases or not.
Dilation factor	d	Place $d-1$ zeros between the filters' consecutive values.
Groups	g	Split the input into g groups to be independently convolved.
Weight initializer	p_W	Weight initialization distribution.
Bias initializer	p_b	Bias initialization distribution.

4.1.3 Transposed Convolutional Layer

One can think of the transposed convolutional layer as the opposite of a convolutional layer. Instead of reducing the dimensionality to, e.g., extract features or filter out unnecessary information, a transposed convolutional layer acts essentially as a learnable upsampling layer.

The main difference between the two layers is that the transposed convolutional layer introduces "holes"/zeros between the input sequence's consecutive values, and this number of zeros is indicated by the *strides*. Hence, strides have a different meaning compared to the convolutional layer's strides which was the number of shifts of each filter. Here, the number of strides S indicates the upsampling factor of the transposed convolutional layer. After introducing $S-1$ zeros between the input's consecutive values:

$$x'_{i,j} = \begin{cases} x_{i,j'}, & j = S(j'-1)+1 \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

it acts as a normal convolutional layer following eq. (4.5) to calculate its output.

Given those, the input-output dimension relationship in a transposed convolutional layer changes from the one used by the convolutional layer (eq. (4.6)). Instead of dividing by the strides S, the input gets multiplied by S, as upsampling is essentially the goal. With input channels of length D_{in}, kernel sizes K, strides S, dilation factor d, padding P, then considering C_{out} output channels, the size of each output channel D_{out} will be:

$$D_{\text{out}} = S(D_{\text{in}} - 1) + 2P - d(K - 1) + 1. \tag{4.10}$$

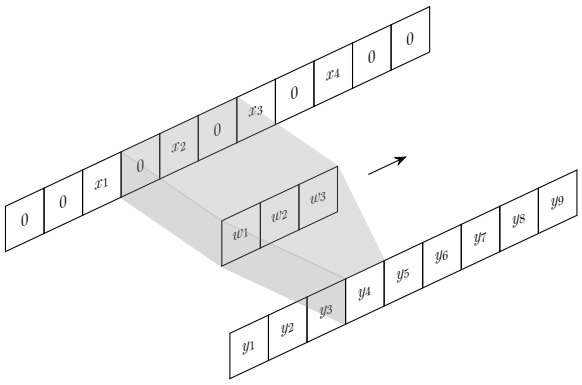


Figure 4.5: Assuming strides $S = 2$, transposed convolution will introduce a zero between all the input's consecutive values, thus upsampling it by a factor of two in this case. So with an input of length $D_{\text{in}} = 4$ padded with $P = 2$, the transposed convolution with a kernel of length $K = 3$ and no dilation $d = 1$, will produce an output of size $D_{\text{out}} = 2 \cdot (4 - 1) + 2 \cdot 2 - 1 \cdot (3 - 1) + 1 = 9$, from eq. (4.10).

The hyperparameter options in the transposed convolutional layer will be the same as in the convolutional layer, with the only difference being the strides hyperparameter S, which instead of indicating how many values a filter will shift, is the upsampling factor:

Name	**Symbol**	**Description**
Number of filters	C_{out}	The number of output channels to produce.
Kernel size	K	The length/size of each kernel.
Padding	P, P_v	Apply the P_v constant P times on both sides of a channel.
Strides	S	Upsample the input by the upsampling factor S.
Bias flag	$\mathbb{1}\{b\}$	Whether to use biases or not.
Dilation factor	d	Place $d - 1$ zeros between the filters' consecutive values.
Groups	g	Split the input into g groups to be independently convolved.
Weight initializer	p_W	Weight initialization distribution.
Bias initializer	p_b	Bias initialization distribution.

4.1.4 Average Pooling Layer

The *Average Pooling* layer calculates the average value for patches of the input and uses it to create a downsampled (pooled) output. It is usually used after a convolutional layer, and with S being the downsampling factor, its operation is:

$$y_j = \frac{1}{K} \sum_{k=1}^{K} x_{j'}, \; j' = S(j-1) + k. \tag{4.11}$$

4.2 Activation Functions

By imagining any amount of successive fully connected or convolutional layers put together, one can observe the fact that they all impose **linear** relationships between their input and output. That implies that the output from a given neural network like this would be linearly dependent on its input, permitting it to solve any problem that does not have a linear solution. As stated in the chapter's introduction, neural networks are meant to deal with exceedingly more complicated representation tasks than those. For that reason, a needful component of any neural network is the imposition of what is called an *activation function*, or simply *activation*, directly to the output of (most times) each trainable layer. These functions should be non-linear, and their purpose is to allow a neural network to learn to solve problems that do not have a linear solution.

Choosing the proper activation functions for a neural network architecture is vital but can also prove to be troublesome. There is a plethora of existing activation functions, but very few can prove to be effective for a given architecture. The study of activation functions is therefore necessary, and a significant amount of research is indeed dedicated to them. For the purposes of this thesis, only a few established activations will be mentioned. An activation function will be symbolized by the letter σ with a subscript to denote which it is.

4.2.1 Rectifiers

Arguably one of the most frequently used activation functions that has proven to work well in many circumstances is the *Rectified Linear Unit* (ReLU) which is:

$$\sigma_r(x) \triangleq \mathrm{ReLU}(x) \triangleq \begin{cases} x, & x > 0 \\ 0, & \text{otherwise} \end{cases} \quad x \in \mathbb{R}. \tag{4.12}$$

As seen above, using a ReLU activation will map all negative input values to zero. Sometimes, this can cause a problem called "dying ReLU" [74], where during training, a significant amount of neurons learn to output zero for any given input rendering them basically inactive. Once this happens, recovery of these inoperative neurons is hard, so a portion of a neural network with this problem will not contribute to anything. There might also be cases where we just do not want

to restrict the output from a layer to be only positive or zero $[0, +\infty)$ to meet the needs of a given problem.

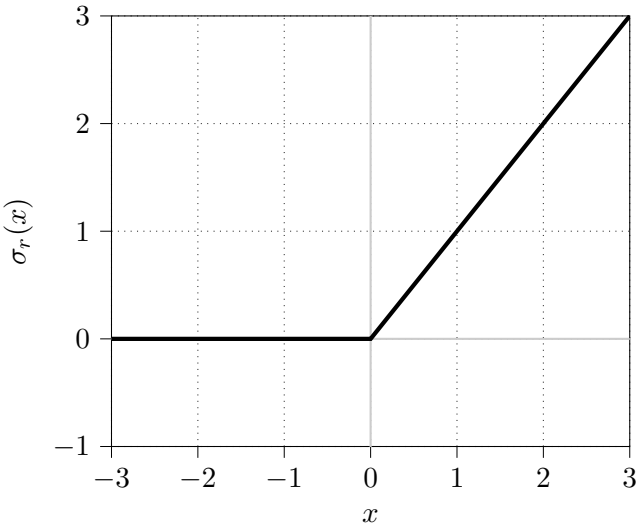


Figure 4.6: Graphical representation of the ReLU activation function, equivalently defined as $\sigma_r(x) = \max(0, x)$.

There are many approaches that counter these obstacles, one of them simply being the use of ReLU variants that do not suppress negative inputs to zero. One way to do this is to introduce a line for the negative values in $(-\infty, 0)$ with some slope $a \in (0, 1)$. An alternative of ReLU that does precisely that is the *Leaky Rectified Linear Unit* (LReLU). There also exists the *Parametric Rectified Linear Unit* (PReLU), which has the same formulation as LReLU (eq. (4.13)), but the slope parameter a is trainable instead of preset.

$$\sigma_\ell(x; a) \triangleq \mathrm{LReLU}(x, a) \triangleq \begin{cases} x, & \text{if } x > 0 \\ ax, & \text{otherwise} \end{cases} \quad x, a \in \mathbb{R}. \tag{4.13}$$

Note that the requirement of non-linearity holds for the rectifiers described above since they are piece-wise linear and hence non-linear throughout their entire domain. That is also the reason why one should not set the slope parameter a of LReLU very close to one.

There are also other rectifier-like activation functions that, for example, introduce slopes instead of lines. Those include the *Exponential Linear Unit* (ELU) [12], the *Gaussian Error Linear Unit* (GELU) [37], et al., but it is unnecessary to elaborate on them for the purposes of this thesis.

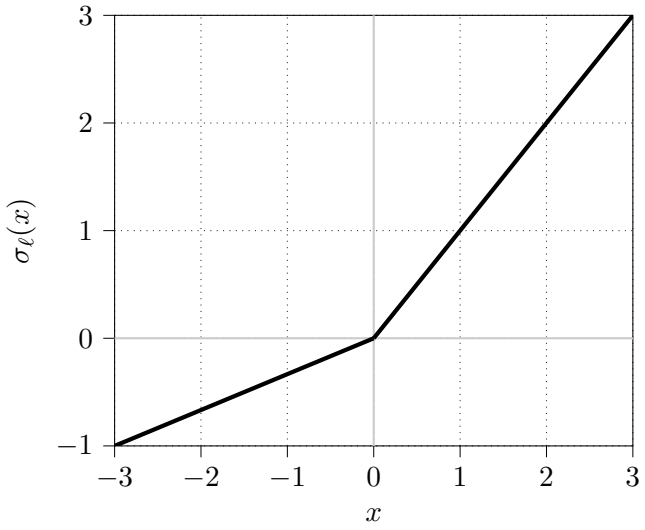


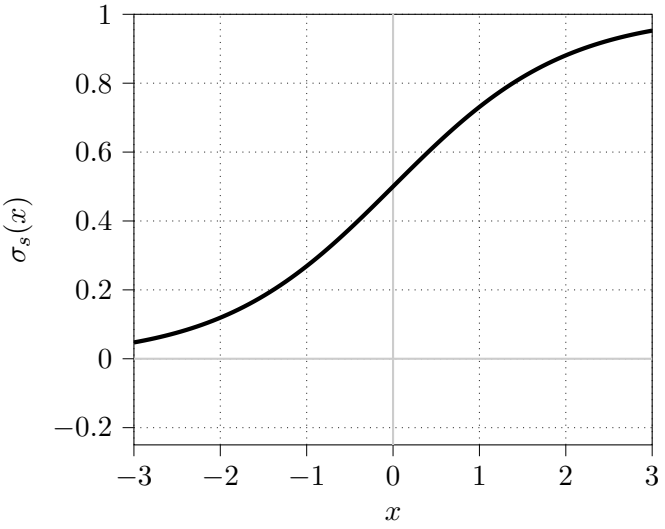
Figure 4.7: Graphical representation of a LReLU with $a = \frac{1}{3}$, equivalently defined as $\sigma_\ell(x; a) = \max(ax, x)$, for $a \in (0, 1)$ as it is commonly set. This could also represent a PReLU plot if a was tuned to that exact value at that point in time.

4.2.2 Sigmoid

Being named after its characteristic "S-like" shape, the *sigmoid* function is:

$$\sigma_s(x) \triangleq \text{sigmoid}(x) \triangleq \frac{1}{1 + e^{-x}}, \ x \in \mathbb{R}. \tag{4.14}$$

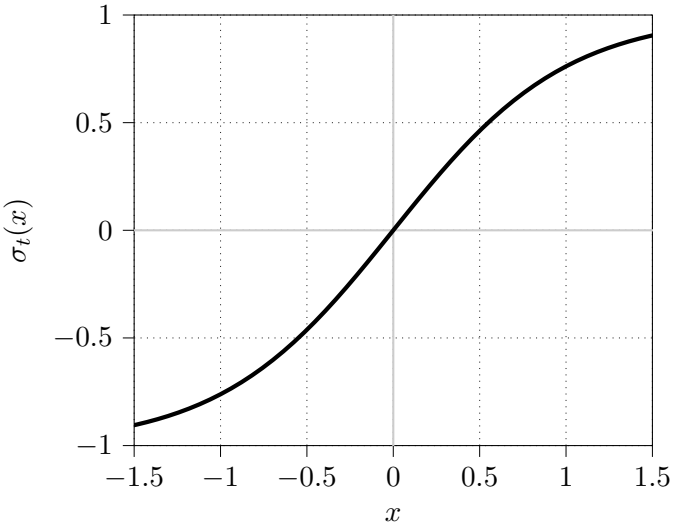
Evidently, sigmoid meets the requirement of non-linearity and maps its input to the range $(0, 1)$, thus being particularly useful for probabilistic models, such as those who deal with various classification tasks, e.g., image classification.



4.2.3 Hyperbolic Tangent

The *Hyperbolic Tangent*, or tanh for short is defined as:

$$\sigma_t(x) \triangleq \tanh(x) \triangleq \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1},\ x \in \mathbb{R}. \tag{4.15}$$



The hyperbolic tangent is also non-linear, and since it maps its input to the range $(-1, 1)$, it is occasionally used for modeling amplitudes of digital signals in many speech-generating models.

4.3 Loss Functions

Since a neural network needs to discover the function mapping the input to the desired output set, another essential practice is measuring how "close" or "far" the neural network's output is from the ground truth. The *loss function* does precisely that, meaning the larger its value is, the more "separate" or "distant" the model's estimation is from the ground truth. As it is later explained, a neural network's true goal is to minimize this loss function. Assuming an estimated sequence $y[n]$ and its corresponding ground truth sequence $\hat{y}[n]$ both of length $N \in \mathbb{N}$, commonly used loss functions are explored below.

4.3.1 Mean Squared Error

The *Mean Squared Error* (MSE), also known as *quadratic* or L_2 *loss*, is commonly used in regression problems and is defined as:

$$\text{MSE}(y[n], \hat{y}[n]) \triangleq \frac{1}{N} \sum_{n=1}^{N} \Big[y[n] - \hat{y}[n] \Big]^2. \tag{4.16}$$

As the name suggests, the MSE measures the average squared difference between the predictions and ground truths. Due to squaring, predictions that are distant from the actual values are heavily penalized compared to less deviated ones. If one wants to measure loss in the spectral domain, the MSE can be used between the magnitude of the DFT (eq. (2.27)) of the signals $y[n]$, $\hat{y}[n]$. By letting $Y[k]$ and $\hat{Y}[k]$ be their respective DFT, then the *Spectral MSE* can be defined as:

$$\text{S-MSE}(Y[k], \hat{Y}[k]) \triangleq \frac{1}{K} \sum_{k=1}^{K} \Big[||Y[k]||_2 - ||\hat{Y}[k]||_2 \Big]^2. \tag{4.17}$$

4.3.2 Mean Absolute Error

The *Mean Absolute Error* (MAE), also known as L_1 *loss*, is defined as:

$$\text{MAE}(y[n], \hat{y}[n]) \triangleq \frac{1}{N} \sum_{n=1}^{N} |y[n] - \hat{y}[n]|. \tag{4.18}$$

Like the MSE, it measures the magnitude of error without considering the direction, but it is more robust to outliers since it does not square the difference.

4.3.3 Cross-Entropy

For comparing probability distributions, say p, $\hat{p}$, a famous loss function used is *Cross-Entropy* defined as:

$$H(p, \hat{p}) \triangleq - \sum_{y=0}^{Y-1} p_y \log \hat{p}_y, \tag{4.19}$$

usually with p being the ground truth probability distribution (one-hot), $\hat{p}$ the predicted one, and y the class index.

4.3.4 Cosine Similarity

The *Cosine Similarity* loss is defined as:

$$\text{COS}(y[n], \hat{y}[n]) \triangleq \frac{\sum_{n=1}^{N} [y[n] \cdot \hat{y}[n]]}{\sqrt{\sum_{n=1}^{N} [y[n]]^2} \cdot \sqrt{\sum_{n=1}^{N} [\hat{y}[n]]^2}}. \tag{4.20}$$

4.3.5 Hinge Loss

The *hinge* loss is used mainly for classification tasks and is defined as:

$$\text{HL}(s, y) \triangleq \max \left(0, s_y - \sum_{y' \neq y} [s_{y'}] + \Delta \right), \ \Delta \in \mathbb{R}^+, \tag{4.21}$$

where y is the ground truth class and $s = [\![s_1, s_Y]\!]$ are the predicted scores.

4.4 Optimizers

At this point, it should be clear that a neural network attempts to solve an *optimization* problem. Broadly speaking, that is to find the best possible values for its trainable parameters over all feasible ones. For a regression task, that is to minimize the expectation $\mathbb{E}$ of the loss function $\mathcal{L}$ over all parameters θ of a given trainable model f. The expectation is taken with respect to the underlying joint distribution p_{XY} that the input-output data are assumed to follow. With x as the input and y the corresponding ground truth, this objective in math terms is:

$$\min_{\theta} \mathbb{E}_{(x,y)\sim p_{XY}} [\mathcal{L}\left(y, f(x,\theta)\right]. \tag{4.22}$$

Once the expectation is computed, all the variables become fixed except for the parameters θ of the model. So, only those are left to determine this function's value, which we want to minimize. An attempt to solve this is by an algorithm that dictates when and how much these parameters θ (i.e., weights and biases) of our given neural network model need to change, such that convergence towards a minimum is achieved. This role is assigned to the neural network's *optimizer*.

There are many optimizers to choose from depending on the implemented architecture and nature of the problem. It may even be the case that only one carefully tuned optimizer can adequately train the weights of a specific network. Nonetheless, some optimizers have proven to work well in most circumstances, such as those explored in this section.

4.4.1 Stochastic Gradient Descent

Algorithm 1 Stochastic Gradient Descent

Input: Number of Iterations: $T \in \mathbb{N}$, Parametric Model: f, with Trainable Weights: w and Biases: b, Input Training Dataset: x, Corresponding Ground Truth: y, Loss Function: $\mathcal{L}$, Learning Rate: $\eta \in \mathbb{R}^+$, Weight Initialization: w_0, Bias Initialization: b_0

Output: Trained parametric model f.

$w \leftarrow w_0$ # Initialize the weights w.
$b \leftarrow b_0$ # Initialize the biases b.
for $i \in [\![1, T]\!]$ **do** # For all number of iterations given:
  $x_i \leftarrow \text{Sample}(x)$ # Fetch a sample x_i from x.
  $\hat{y}_i \leftarrow f(x_i)$ # Get the model's prediction $\hat{y}_i$ for the input x_i.
  $L_i \leftarrow \mathcal{L}(\hat{y}_i, y_i)$ # Compute the loss L_i between $\hat{y}_i$, and y_i.
  $\Delta w \leftarrow -\nabla_w L_i$ # Compute the negative gradient $-\nabla$ of L_i w.r.t. w.
  $\Delta b \leftarrow -\nabla_b L_i$ # Compute the negative gradient $-\nabla$ of L_i w.r.t. b.
  $w \leftarrow w + \eta \Delta w$ # Scale Δw by η, and update the weights.
  $b \leftarrow b + \eta \Delta b$ # Scale Δb by η, and update the biases.
end for
return f

Named "stochastic" due to an inherent element of randomness it possesses, one of the most well-known optimizers is the *Stochastic Gradient Descent* (SGD). Many people consider it one of the most cardinal pillars in all modern artificial intelligence systems due to its core ideas still remaining very relevant.

Depending on the problem as well as the input dataset, many issues can arise if the SGD algorithm described above is used as it is. For example, suppose that the input set x of a given optimization problem has a considerable amount of highly diverse samples x_d. Since in SGD, the gradient updates take place over a **single** drawn sample, those computed for the x_d samples can take a completely opposite direction compared to the overall underlying distribution of x. Such a scenario causes unstable, highly variant gradients, which also implies a slower convergence of the algorithm, and in extreme cases, not even one at all.

To ameliorate these issues, a variant of SGD is most frequently used, known as *Mini-Batch* SGD. Instead of learning with one sample per iteration, Mini-Batch SGD uses a *batch* of samples instead. This way, the gradients and overall convergence of the model are much more stable, as a collection of samples is much more representative of the input data's distribution than just one sample at a time. One can also think of the Mini-Batch SGD as a more generalized version of SGD since the two algorithms are identical when batch size B is equal to one. Nevertheless, it is almost always preferable to use a (non-zero) power of two as batch size[2].

Algorithm 2 Mini-Batch Stochastic Gradient Descent

Input: Number of iterations: $T \in \mathbb{N}$, Parametric Model: f, with Trainable Weights: w, Input Training Dataset: x, Corresponding Ground Truth: y, Loss Function: $\mathcal{L}$, Batch Size: $B \in \mathbb{N}$, Learning Rate: $\eta \in \mathbb{R}^+$, Weight Initialization: w_0

Output: Trained parametric model f.

$w \leftarrow w_0$ # Initialize the weights w.
for $i \in [\![1, T]\!]$ **do** # For all number of iterations T given:
  for $j \in [\![1, B]\!]$ **do** # For all number of batch samples B given:
    $x_{i,j} \leftarrow \text{Sample}(x)$ # Fetch a sample $x_{i,j}$ from x.
    $\hat{y}_{i,j} \leftarrow f(x_{i,j})$ # Get the model's prediction $\hat{y}_{i,j}$ for the input $x_{i,j}$.
    $L_{i,j} \leftarrow \mathcal{L}(\hat{y}_{i,j}, y_{i,j})$ # Compute the loss L_i between $\hat{y}_{i,j}$, and $y_{i,j}$.
  end for
  $\bar{L}_i \leftarrow \frac{1}{B} \sum_{j=1}^{B} L_{i,j}$ # Compute the mean loss $\bar{L}_i$ for this batch.
  $\Delta w \leftarrow -\nabla_w \bar{L}_i$ # Compute the negative gradient $-\nabla$ of $\bar{L}_i$ w.r.t. w.
  $w \leftarrow w + \eta \Delta w$ # Scale Δw by η, and update the weights.
end for
return f

[2]Generally speaking, since memory management in all modern operating systems is organized in powers of two, selecting a power of two for the batch size avoids fetching more pages than needed, therefore improving the algorithm's overall speed.

Due to the addition of the batch concept, Mini-Batch SGD proved to be so effective that it ended up replacing the original SGD nearly everywhere. Nowadays, the acronym "SGD" is almost always used to refer to the Mini-Batch SGD algorithm (or its variants) instead of the original SGD.

One can notice that the biases have been omitted from the pseudocode. In contexts like this, we can basically deal with any bias as it being a weight, which in practice is. Since both are just trainable parameters for our cause, there is no real need to update or treat the biases from the weights of a parametric model separately.

The Mini-Batch SGD algorithm can be further improved, especially in terms of stability. The *Adaptive Moment Estimation* (Adam) optimizer [61] does precisely that by introducing two *momentum* variables. Updates from previous iterations are relevant, but, as apparent from Mini-Batch SGD's pseudocode, they are not considered when registering the newer updates. This can lead to fluctuating weights being updated consecutively in very different directions, hindering the algorithm's convergence. A solution is to maintain the momentum gained from previous updates, as done by Adam's *first moment* variable m.

Additionally, Adam uses the *second moment* variable v, responsible for giving each weight in the model its "own" learning rate, in a sense. Contrary to Mini-Batch SGD, when, e.g., in Adam, a weight receives a relatively big update, its learning rate will get smaller for the subsequent iterations of the algorithm. This offers yet another improvement to the algorithm's stability by not allowing consecutive big updates to the weights that can slow the algorithm's convergence.

Adam's effectiveness comes from combining the advantages discussed out of two other extensions of Mini-Batch SGD; the *Adaptive Gradient Algorithm* (AdaGrad) [21], and *Root Mean Square Propagation* (RMSProp)[3] optimizers which we will not dive into. Despite its popularity, strength, and efficacy in most cases, there are still scenarios where Adam can fail to find an adequate minimum or generalize well compared to other algorithms, such as even Mini-Batch SGD [159]. A common cause for this issue can be due to Adam's heightened stability attributes. As an example, it is known that for most convolutional neural networks, Mini-Batch SGD generalizes better, while Adam shines when put into optimizing Generative Adversarial Networks (explained in section 4.7). Hence, research is still looking to invent new optimizers [107, 156], or other ways to improve Adam [157, 161].

[3]First presented in a Coursera class on neural networks taught by Dr. Geoffrey Everest Hinton.

Algorithm 3 Adam

Input: Number of iterations: $T \in \mathbb{N}$, Parametric Model: f, with Trainable Weights: w, Input Training Dataset: x, Corresponding Ground Truth: y, Loss Function: $\mathcal{L}$, Batch Size: $B \in \mathbb{N}$, Learning Rate: $\eta \in \mathbb{R}^+$, Decay Rates: $\beta_1, \beta_2 \in (0, 1)$, Weight Initialization: w_0, Denominator Stabilizer: $\varepsilon \in \mathbb{R}^+$

Output: Trained parametric model f.

$w \leftarrow w_0$ # Initialize the weights w.
$m \leftarrow 0$ # Initialize the first moment to 0.
$v \leftarrow 0$ # Initialize the second moment to 0.
for $i \in [\![1, T]\!]$ **do** # For all number of iterations T given:
  $m_{\text{prev}} \leftarrow m$ # Update the previous first moment.
  $v_{\text{prev}} \leftarrow v$ # Update the previous second moment.
  for $j \in [\![1, B]\!]$ **do** # For all number of batch samples B given:
    $x_{i,j} \leftarrow \text{Sample}(x)$ # Fetch a sample $x_{i,j}$ from x.
    $\hat{y}_{i,j} \leftarrow f(x_{i,j})$ # Get the model's prediction $\hat{y}_{i,j}$ for the input $x_{i,j}$.
    $L_{i,j} \leftarrow \mathcal{L}(\hat{y}_{i,j}, y_{i,j})$ # Compute the loss between $\hat{y}_{i,j}$, and $y_{i,j}$.
  end for
  $\bar{L}_i \leftarrow \frac{1}{B} \sum_{j=1}^{B} L_{i,j}$ # Compute the mean loss $\bar{L}_i$ for this batch.
  $g \leftarrow -\nabla_w \bar{L}_i$ # Compute the negative gradient $-\nabla$ of $\bar{L}_i$ w.r.t. w.
  $m \leftarrow \frac{\beta_1 m_{\text{prev}} + (1-\beta_1) g}{(1-\beta_1^i)}$ # Compute the biased first moment estimate m.
  $v \leftarrow \frac{\beta_2 v_{\text{prev}} + (1-\beta_2) g^2}{(1-\beta_2^i)}$ # Compute the biased second moment estimate v.
  $\Delta w \leftarrow \frac{m}{\sqrt{v}+\varepsilon}$ # Compute the weight update direction Δw.
  $w \leftarrow w + \eta \Delta w$ # Scale Δw by η, and update the weights.
end for
return f

4.5 Backpropagation

An artificial neural network, given some input x, generates its prediction $\hat{y}$ during the *forward propagation*, or *forward pass*, where the information flows forward, passing through the layers of the neural network. Then, the loss L is measured given a loss function $\mathcal{L}$ such as the ones described previously in section 4.3. Given this loss L, the chosen optimizer tunes the parameters θ of the neural network accordingly based on the *gradient* of the loss L with respect to the neural network's parameters θ. This procedure is also apparent by looking at the SGD pseudocodes provided in the section 4.4 above. *Backpropagation* (BP) [109, 108], refers to the algorithm which by this gradient is computed. On the grounds that all modern neural networks are trained using backpropagation, it has become the sine qua non of deep learning. Hence, having an understanding of it is an integral part of knowing about neural networks.

Backpropagation is an automated way of efficiently computing this gradient. Firstly, a graph representing a neural network's entire corresponding output function is formed, called *computational graph*. It is a directed graph where each node can either correspond to a variable or an operation, and each edge shows which variable is fed into which operation. At each iteration, the values of all the variables in the computational graph must be computed with respect to the input batch through the so-called *forward pass*. These are then leveraged with a procedure called *backward pass* to compute the gradient of the loss function by essentially using the *chain rule* over and over:

$$\frac{\partial}{\partial x}\left(f \circ g\right) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}. \tag{4.23}$$

Given that a neural network is in its core a combination of matrix multiplications and differentiable function compositions:

$$f(x) = g^L\left(W^L g^{L-1}\left(W^{L-1}\cdots g^2\left(W^2 g^1\left(W^1 x\right)\cdots\right)\right)\right), \tag{4.24}$$

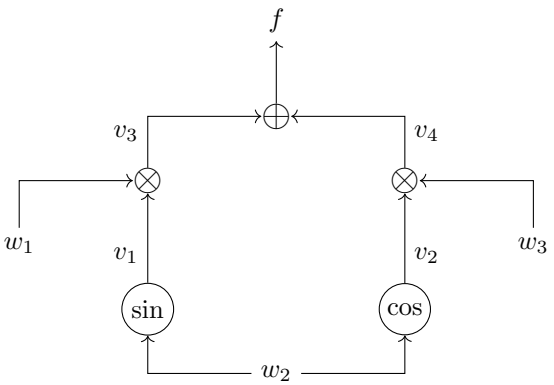
the chain rule can be generalized for any such neural network, no matter its architecture or complexity. This fact makes backpropagation suitable for supervised learning of all parametric models like these. So, given a differentiable loss function $\mathcal{L}(y, f(x,\theta))$, its gradient can always be calculated with respect to the weights via backpropagation. A category of optimization also deals with non-differentiable objectives or models that do not need gradient calculations, such as *Boltzmann Machines* [1], but it is a topic beyond the scope of this document.

Backpropagation is best explained by giving a small example of how backpropagation would work given simpler functions. Assume a simple output function with just three trainable weights like so:

$$f(w_1, w_2, w_3) = w_1 \sin(w_2) + w_3 \cos(w_2). \tag{4.25}$$

As stated earlier, backpropagation would construct the function's corresponding computational graph, get its output during the forward pass, and then calculate its gradient with respect to the weights using eq. (4.23) repeatedly during the backward pass. The calculations are shown in more detail in fig. 4.8.

One has to imagine this procedure taking place on a large computational graph with thousands, maybe millions, or even billions of parameters and operations that constitute this case's entire neural network. Given that there are parts within the computation of the gradients that are parallelizable [42, 93, 116], backpropagation is a core method for making the training procedure of these large parametric models feasible in practice.



Forward Pass

$$
\begin{aligned}
v_1 &= \sin(w_2) \\
v_2 &= \cos(w_2) \\
v_3 &= v_1 w_1 \\
v_4 &= v_2 w_3 \\
f &= v_3 + v_4
\end{aligned}
$$

Backward Pass

$$\bar{v}_3 = \frac{\partial f}{\partial v_3} = 1, \quad \bar{v}_1 = \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \bar{v}_3 w_1 = w_1, \quad \bar{w}_1 = \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial w_1} = \bar{v}_3 v_1 = v_1 = \sin(w_2).$$

$$\bar{v}_4 = \frac{\partial f}{\partial v_4} = 1, \quad \bar{v}_2 = \frac{\partial f}{\partial v_4}\frac{\partial v_4}{\partial v_2} = \bar{v}_4 w_3 = w_3, \quad \bar{w}_3 = \frac{\partial f}{\partial v_4}\frac{\partial v_4}{\partial w_3} = \bar{v}_4 v_2 = v_2 = \cos(w_2).$$

$$\bar{w}_2 = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial w_2} + \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial w_2} = \bar{v}_1 \cos(w_2) - \bar{v}_2 \sin(w_2) = w_1 \cos(w_2) - w_3 \sin(w_2).$$

Figure 4.8: An example of how backpropagation works given a function (eq. (4.25)) that one can imagine represents the sum of a sine and a cosine wave of the same trainable frequency but different amplitude. In this example, the frequency w_2 and the amplitudes w_1, w_3 of the sinusoids are the trainable parameters. In order for an optimizer such as SGD to update these weights, it needs the gradient, as also displayed in the pseudocodes of section 4.4. This gradient is computed via the backpropagation method illustrated above in the figure. Calculating the derivative analytically from eq. (4.25) as a sanity check, reaches us to the same result: $\nabla_w f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3}\right) = \Big(\sin(w_2),\ w_1\cos(w_2) - w_3\sin(w_2),\ \cos(w_2)\Big)$.

4.6 Common Practices, Problems, and Remedies

Before closing the neural network chapter, some standard practices and common problems that frequently turn up while training beg for a brief discussion. When designing an artificial neural network or deciding its hyperparameters, many choices preemptively consider those issues. Firstly, there is a brief talk about how handling the data at our disposal usually takes place, followed by explanations about the most significant problems in deep learning, what commonly causes them, and some notable ways of managing them.

4.6.1 Training, Evaluation, and Testing

Ordinarily, a neural network should be exposed to a large enough portion of data, that being the input-output ground truth pairs. The entire collection of data at our disposal for a given problem is the *dataset* and is commonly split into three separate ones. The biggest of the three is used for the neural network's training, i.e., the *training dataset* (or *train set*). As explained in the previous chapters, this is the set of examples for fitting the model's parameters via minimizing the loss function. To monitor the current performance of a model, tune its hyperparameters correctly, or avoid future unwanted behaviors, another part of the dataset is selected, i.e., the *validation dataset* (or *validation set*), having data that were **not** used for training. Finally, the third one is the *testing dataset* (or *test set*) that is used to test the model's performance on data that has **neither** been trained or evaluated upon. The *split* refers to the percentage of the dataset used for training, evaluation, and testing. A 70–10–20 split, for example, means that 70% of the dataset is used for training, 10% for validation, and 20% for testing. Determining the sizes and strategies for the division of the dataset is very dependent on the problem and the available data themselves.

A crucial goal of a neural network's training is making it *generalize*. This refers to the ability of a model to apply its learning gained from the training dataset to other "unseen" data, e.g., the testing dataset. If a model performs well in both cases, it has generalized well, whereas if it underperforms on the testing data while performing well on the training data, it fails to generalize its gained knowledge. The following sections offer a more detailed explanation of these common problems.

4.6.2 Overfitting and Underfitting

A wide variety of obstacles frequently arise while training a neural network to fit a given dataset, and it is crucial to be aware of them so as to identify and tackle them. The two most notorious of which are the problems of *overfitting* and *underfitting*. A model suffering from either of these problems fails to solve the assigned task and may need relatively minor to considerable modifications and adjustments to overcome them.

Overfitting occurs when a parametric model too closely or exactly fits a specific set of data to the point where it fails to generalize. An overfitted model, for example, performs exceptionally well when inputting data belonging to its training dataset and very poorly on data that it has not "seen" before in the test set. Employing a wrong architecture or an overcomplicated model with many more parameters than needed for a specific problem can result in overfitting too. Another reason why a model overfits can be the result of using either a wrong or small dataset for the particular task. A small dataset means that it is not representative enough of the particular problem, to the degree where it is very likely to lead to overfitting, a problem also known as *selection bias*.

Regularization techniques try to overcome overfitting with various methods by taking effect on some chosen neural network layers. Another advantage of regularization is that it is applicable to almost all architectures. The three most common regularizers that we will talk about are L_1, L_2 and *dropout*.

With an L_1 regularizer, a penalty equal to the absolute magnitude of a chosen weight matrix w is added to the loss function, scaled by a hyperparameter λ_1:

$$L_1(w) = \lambda_1 \sum_{i=0}^{N-1} |w_i|, \ \lambda_1 \in \mathbb{R}^+, \text{ with} \tag{4.26}$$

$$\frac{\partial L_1(w)}{\partial w_i} = \begin{cases} \lambda_1, & w_i > 0 \\ -\lambda_1, & \text{otherwise} \end{cases} \quad i \in [\![0, N-1]\!]. \tag{4.27}$$

By taking the gradient (eq. (4.27)), one can notice that the magnitude of each component, i.e., $\pm\lambda_1$, is constant, meaning that all weights are equally penalized. As an effect, this induces sparsity to the weights, forcing small weights to converge faster to zero. The L_2 or *Tikhonov* regularizer, on the other hand, takes the square instead of the magnitude of the weights and also scales by a hyperparameter λ_2:

$$L_2(w) = \lambda_2 \sum_{i=0}^{N-1} [w_i]^2, \ \lambda_2 \in \mathbb{R}^+, \text{ with} \tag{4.28}$$

$$\frac{\partial L_2(w)}{\partial w_i} = 2\lambda_2 w_i, \quad i \in [\![0, N-1]\!]. \tag{4.29}$$

The gradient components, in this case, are proportional to the magnitude of the weights rather than constant. This implies that less penalty is given to smaller weights and more to larger weights, thus prohibiting the largest from growing very big, which is potentially a sign of overfitting. The higher the value λ_1 or λ_2 is set, the more aggressive the corresponding regularization penalty will be.

A dropout regularizer [120] operates in a different way by dealing with the model's excessive amounts of trainable parameters in a straight manner. To decrease unnecessary complexity, dropout regularization completely deactivates neurons at random[4] during the neural network's training phase, according to an input probability hyperparameter. Similarly, a higher probability will end up turning off more units, increasing the potency of this regularizer and vice versa.

On the other hand, underfitting occurs when a parametric model does not possess the capability to represent the target input-output relationship of a specific set, namely, the training dataset. An inadequate architecture, for instance, with a low number of parameters, can be an apparent cause of underfitting as the model inherently lacks the representation capabilities needed. A wrong choice of learning rate can also be a possible source of this problem: with too small of a learning rate, the weights will likely get stuck in a region around a suboptimal local minimum,

[4]Commonly using samples from a Bernoulli distribution.

while with an exceedingly large one, they will most likely fail to converge to any point. Convergence is also affected by the size of the batch. A much smaller than optimal batch size causes high variance in the gradient updates like a "zigzag" phenomenon which can lead to non-optimal or even no convergence. Using very large batch sizes leads to sharp local minima [58] hindering the generalization capability as well. The following section also deals with two well-known gradient problems that cause a model to overfit.

Another parameter that affects the course of training to a certain degree is the neural network's *initialization*, meaning its initial state, i.e., the weight values before training commences. A different initial weight state can potentially result in a different set of output weights and generally even a different overall behavior of a neural network. Preferably, for an SGD-type optimizer to work, weights must be initialized to small random values from a given distribution. Choosing the distribution for each layer, as well as its hyperparameters (e.g., mean, variance), may depend on many things. These include the nature of the problem, the activation function of the specific layer, the behavior of the gradient, et al., and is often the product of a heuristic, experimental approach.

4.6.3 Vanishing and Exploding Gradient

The *vanishing gradient* problem happens if, during backpropagation, the resulting gradients get too small. This makes weight updates in the neural network also small, to the point of not significantly improving it at all. When this occurs during a model's training session, then at that point, this model gets "stuck" in a usually suboptimal local minimum region, rendering it unable to fit its train set adequately, making for another underfitting scenario. Usually, the more layers an architecture has, the more likely this problem is to arise. There are certain activation functions, such as the sigmoid (eq. (4.14)) or the hyperbolic tangent (eq. (4.15)) whose gradient gets closer to zero as its input x grows very big in magnitude:

$$\lim_{x\to\pm\infty}\left(\frac{d}{dx}\sigma_s(x)\right) = \lim_{x\to\pm\infty}\frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \lim_{x\to\pm\infty}\left(\frac{e^{-x}}{(1+e^{-x})^2}\right) = 0.$$
$$\lim_{x\to\pm\infty}\left(\frac{d}{dx}\sigma_t(x)\right) = \lim_{x\to\pm\infty}\frac{d}{dx}\left(\frac{e^{2x}-1}{e^{2x}+1}\right) = \lim_{x\to\pm\infty}\left(\frac{4e^{2x}}{(e^{2x}+1)^2}\right) = 0.$$

Enough consecutive layers with such activation functions will make the model susceptible to the vanishing gradient issue. Since the derivatives are multiplied together along the way, the gradient decreases exponentially as it gets propagated backward. Usual solutions include using different activation functions that lack this property (e.g., rectifiers section 4.2.1), *batch normalization* [44] which normalizes the input to be within a certain range, *residual layers* that add a layer's input to a subsequent layer output, among others.

The *exploding gradient* is the inverse of the vanishing gradient problem, meaning that the gradients get overly big. Analogously, weight updates also get big to

the point where weights never converge to a satisfactory set of values, thus underfitting the train set. Since the gradients are multiplied during backpropagation, consecutive layers with values greater than one will make the growth exponential. An inappropriate loss function is a possible cause for exploding the gradient; e.g., a logarithm-based loss must be designed carefully, considering that the logarithm's derivative approaches infinity as the input gets closer to zero. Weight regularization such as L_1 (eq. (4.26)), or L_2 (eq. (4.28)), can help prevent a gradient from exploding, *gradient clipping* which basically sets a ceiling for the maximum gradient's allowed magnitude, redesigning the layers of the architecture, et al.

A separate reason for these gradient issues can be attributed to poor parameter initialization. For instance, in order for the backward signal to flow without getting saturated or exploded in the computational graph, the variance of each layer output also plays a role [29]. More specifically, when initializing the weights of a neural network, ensuring for each layer that the variance of its output is equal to the variance of its input has shown to counter gradient vanishing or exploding problems. This is done by normalizing the variance of the initializing distribution by the square root of the sum of input and output connections of a layer. The specific activation function used can itself also determine the weight initialization scheme [36].

References

[1] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[12] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[16] Truong-Dong Do, Minh-Thien Duong, Quoc-Vu Dang, and My-Ha Le. Real-time self-driving car navigation using deep neural network. In *2018 4th International Conference on Green Technology and Sustainable Development (GTSD)*, pages 7–12. IEEE, 2018.

[21] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[29] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[37] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[42] Zhouyuan Huo, Bin Gu, Heng Huang, et al. Decoupled parallel backpropagation with convergence guarantee. In *International Conference on Machine Learning*, pages 2098–2106. PMLR, 2018.

[44] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[58] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[61] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[74] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.

[83] Brilian Tafjira Nugraha, Shun-Feng Su, et al. Towards self-driving car using convolutional neural network and road lane detector. In *2017 2nd international conference on automation, cognitive science, optics, micro electro-mechanical system, and information technology (ICACOMIT)*, pages 65–69. IEEE, 2017.

[93] Mark Pethick, Michael Liddle, Paul Werstein, and Zhiyi Huang. Parallelization of a backpropagation neural network on a cluster computer. In *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[101] Qing Rao and Jelena Frtunikj. Deep learning for self-driving cars: Chances and challenges. In *Proceedings of the 1st international workshop on software engineering for AI in autonomous systems*, pages 35–38, 2018.

[107] Swalpa Kumar Roy, Mercedes Eugenia Paoletti, Juan Mario Haut, Shiv Ram Dubey, Purbayan Kar, Antonio Plaza, and Bidyut B Chaudhuri. Angulargrad: A new optimization technique for angular convergence of convolutional neural networks. *arXiv preprint arXiv:2105.10190*, 2021.

[108] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning internal representations by error propagation, 1985.

[109] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[115] Sebastian Schuchmann. Analyzing the prospect of an approaching ai winter. *Unpublished doctoral dissertation*, 2019.

[116] Xavier Sierra-Canto, Francisco Madera-Ramirez, and Victor Uc-Cetina. Parallel training of a back-propagation neural network using cuda. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 307–312. IEEE, 2010.

[120] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[157] Michael Zhang, James Lucas, Jimmy Ba, and Geoffrey E Hinton. Lookahead optimizer: k steps forward, 1 step back. *Advances in neural information processing systems*, 32, 2019.

[159] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven Chu Hong Hoi, et al. Towards theoretically understanding why sgd generalizes better than adam in deep learning. *Advances in Neural Information Processing Systems*, 33:21285–21296, 2020.

[161] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar C Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *Advances in neural information processing systems*, 33:18795–18806, 2020.