Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Informatica

CDB Theory

Professore:
Danilo Montesi

Presentata da:
Massimo Rondelli

Anno Accademico 2023/2024

Contents

Chapter 1

Introduction to Deep Learning

It seems logical to look at the brain's architecture for inspiration to build an intelligent machine. This is the logic that sparked artificial neural networks (ANN$_s$). Machine learning models are inspired by the networks of biological neurons in our brains. ANN$_s$ are at the very core of deep learning. The first part of this chapter introduces artificial neural networks, starting with a quick visit to the very first ANN architectures and leading up to multilayer perceptrons, which are very used today.

1.1 The Perceptron

The perceptron is one of the simplest ANN architectures. It is based on an artificial neuron called **threshold logic unit** (TLU), Figure 1.1. The inputs and outputs are numbers, instead of binary on/off values, and each input connection is associated with a weight. The TLU computes a linear function of its inputs:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n + b = w^T \cdot x + b$$

After that, it applies a step function to the result:

$$h_w(x) = f(z)$$

The model parameters are the input weights $\mathbf{w}$ and the bias term b. A single TLU, can be used for simple linear binary classification. It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class.

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. It is called *fully connected layer.* The inputs

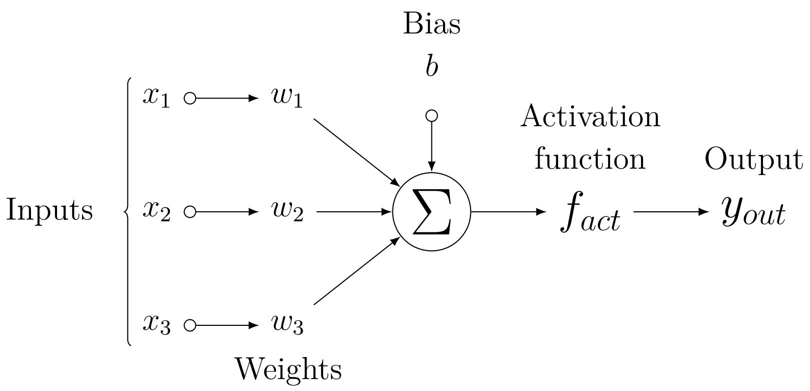

Figure 1.1: The Threshold Logic Unit (TLU) - A fundamental building block of artificial neural networks, implementing a linear decision boundary to classify input data.

constitute the *input layer* and the final output is called *output layer*. It's possible to compute the outputs of a layer for more instance at once, just by applying the following formula:

$$h_{W,b}(X) = \phi(XW + b)$$

The parameters are:

- X represents the matrix of input features.
- The weight matrix W contains all the connection weights.
- The bias vector b contains all the bias terms (one per neuron).
- The function ϕ is called the *activation function*. When the artificial neuron is a TLU, it is called a step function (the activation function will be discussed in the following chapter).

1.1.1 How to train a perceptron

Donald Hebb, considered the "father of neuropsychology", in his 1949 book *The Organization of Behavior*, suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. The connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule. Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction. The perceptron learning rule reinforces connections that help reduce the error. The rule is shown in the following equation:

$$w_{i,j}^{\text{next step}} = w_{i,j} + \eta(y_i - \hat{y}_j) \cdot x_i$$

where:

- $w_{i,j}$ is the connection weight between the i^{th} input and the j^{th} neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

This perceptron training algorithm was proposed by Frank Rosenblatt who he got his inspiration from Hebb's rule. This algorithm is also called *perceptron convergence theorem.*

1.2 Deep Neural Network

A Multilayer Perceptron (MLP), Figure 1.2, is composed of one input layer, one or more layers of TLUs *hidden layers*, and one final layer of TLUs called the *output layer.* When an ANN contains a deep stack of hidden layers, it is called a *deep neural network* (DNN). For many years researchers struggled to find a way to train MLPs. In the 1960s, some researchers discussed the possibility of using gradient descent to train neural networks but just in 1970, a researcher named Seppo Linnainmaa introduced a technique to compute all the gradients automatically and efficiently. This algorithm is called *reverse-mode automatic differentiation.* In two passed through the network (one forward, one backward), it can compute the gradients of the neural network's error about every single model parameter. It can find out how each connection weight and each bias should be tweaked to reduce the neural network's error. If you repeat this process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum. This combination of reverse-mode automatic differentiation is called *backpropagation.*

Backpropagation can be applied to all sorts of computational graphs, not just neural networks. In 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper [RHW86] analyzing how backpropagation allowed neural networks to learn useful internal representations. Today, it is the most popular way to train neural nets.

1.2.1 How does backpropagation work?

It handles one mini-batch at a time, and it goes through the full training set multiple times. Each pass is called an *epoch*. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*.

The algorithm measures the network's output error. To calculate the error, it uses a loss function that compares the desired output and the actual output of the network and returns some measure of the error. Then, it computes how much each output bias and each connection to the output layer contributed to the error. The algorithm then measures how much of these error contribution came from each connection in the layer below, working backward until it reaches the input layer. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients just computed. It is important to initialize all the hidden layer's connection weights randomly, or else training will fail. If you initialize all weights and biases to zero,

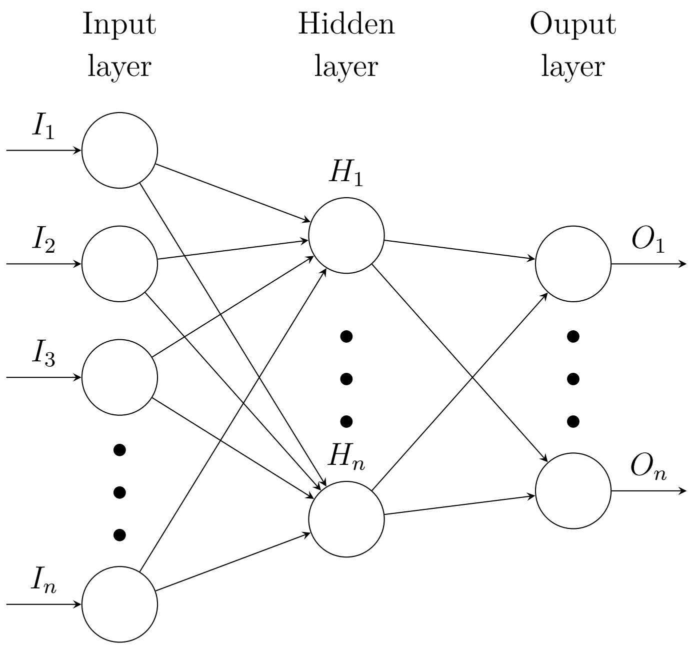

Figure 1.2: Architecture of a Multilayer Perceptron - A type of feedforward neural network that consists of multiple layers of interconnected neurons, with each neuron in one layer connected to every neuron in the next layer.

then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly same way, so they will remain identical. If instead, you randomly initialize the weights, you break the symmetry and allow backpropagation to train the neurons. Summing up, we can say that this technique makes predictions for a mini-batch, the forward pass, measures the error, then goes through each layer in reverse to measure the error contribution from each parameter, backward pass, and finally tweaks the weights and biases of the connections to reduce the error, gradient descent step.

1.2.2 The Vanishing/Exploding Gradients Problems

The second stage of the backpropagation algorithm propagates the error gradient while moving from the output layer to the input layer. The algorithm uses these gradients to update each parameter with a gradient descent step after computing the gradient of the cost function for each network parameter. Unfortunately, when the algorithm descends to the lower layers, the gradient frequently gets smaller and smaller. As a result, training never converges to a good solution and the gradient descent update essentially leaves the connection weights of the lower layers unchanged. The vanishing gradient problem is what's happening here. The inverse can also happen in some situations, causing the gradients to get larger until the layers receive massive weight updates and the algorithm diverges. This is the exploding gradients problem, which recurrent neural networks encounter most frequently. Deep neural networks more typically experience unstable gradients.

A paper [GB10] by Xavier Glorot and Yoshua Bengio published in 2010 identified a few suspects, including the popular weight initialization technique and the combination of the sigmoid activation function. They demonstrated that the variation of each layer's outputs is significantly higher than the variance of its inputs when using this activation function and initialization technique. The activation function reaches saturation at the top layers as the network advances, with the variance increasing after each layer. The sigmoid function's mean is 0.5 rather than 0, which makes this saturation worse. When inputs are high (positive or negative), as can be seen by looking at the sigmoid activation function, Figure 1.3, the function saturates at 0 or 1, with a derivative that is very close to 0. As a result, when backpropagation begins, no gradient is left for it to propagate back through the network.

Using weight initialization, where the weights are small random values, can help to prevent the exploding gradient problem. As presented in the paper, one popular initialization technique is called *"Xavier"* or *"Glorot"*, which adjusts the scale of the weights based on the number of input and output neurons in the layer. Another way to solve these problems is by using non-linear activation functions. Using non-linear activation functions such as ReLU, leaky ReLU, or Maxout instead of sigmoid or tanh can help mitigate the vanishing gradient problem (we are going to see activation functions in the following section). The last two techniques we can use to avoid this problem are *batch normalization* and *regularization*, which will be shown in the following sections.

1.2.3 Activation functions

Rumelhart revised the structure of MLP to make backprop function properly: they replaced out the step function for the logistic function, better known as the *sigmoid function*. This was crucial because there is no gradient to work with in the step function because the gradient descent cannot move on a flat surface, allowing it to advance somewhat at each step. The step function only contains flat segments. In actuality, aside from the sigmoid function, the backpropagation algorithm performs well with a wide variety of alternative activation functions. Here are two other popular choices. In the Figure 1.3, it's possible to see how they work.

- *The hyperbolic tangent function:* $tanh(z) = 2\sigma(2z) - 1$

  This activation function is S-shaped, continuous, and differentiable just like the sigmoid function, however its output value spans from -1 to 1, as opposed to 0 to 1, in the case of the sigmoid function. Because of this range, the output of each layer is more or less centered at zero at the start of training.

- *The rectified linear unit function:* $ReLU(z) = max(0, z)$

  Although continuous, the ReLU function is unfortunately not differentiable at $z = 0$ and its derivative is 0 for $z < 0$. But since it performs so well in practice and offers the benefit of being quick to compute, it has taken over as the standard. Since biological neurons appear to implement an activation function that is roughly sigmoid (S-shaped), researchers have focused on sigmoid functions for a very long time. ReLU, however, really performs better in ANNs in general. The biological analogy may have been misleading in this instance.

Why do we need activation functions? All you get when you combine many linear transformations is another linear transformation. Therefore, if there is no nonlinearity between the levels, even a deep stack of layers is equivalent to one layer, making it impossible to handle extremely complicated issues.

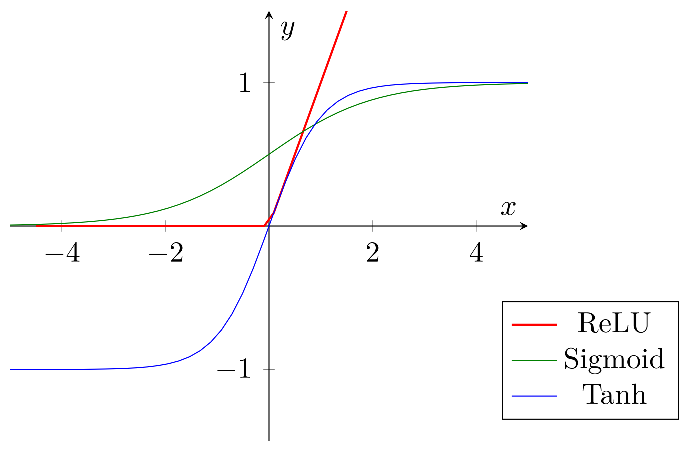

Figure 1.3: Activation Functions (ReLU, Sigmoid, Tanh) - Activation functions determine the output of a neuron in a neural network, and ReLU, Sigmoid, and Tanh are some commonly used activation functions that enable non-linear transformations of input data.

1.2.4 Hidden Layers

Many issues can be solved by starting with a single hidden layer and producing acceptable results. Even the most complex functions can theoretically be modeled by an MLP with just one hidden layer. Deep networks, however, outperform shallow ones in terms of *parameter efficiency* for complicated situations. Deep neural networks benefit from the fact that real-world data is frequently hierarchically structured as follows: The output layer and the highest hidden layers combine these intermediate structures to model high-level structures. Lower hidden layers model low-level structures, intermediate hidden layers combine these low-level structures to model intermediate-level structures, and the highest hidden layers model high-level structures like faces.

In conclusion, the neural network will function properly in many situations if you start with just one or two hidden layers. Increase the number of hidden layers for more complicated problems until the training set starts to become overfit. Large picture classification or speech recognition are two examples of extremely hard jobs that generally demand networks with dozens of layers and a huge amount of training data.

1.2.5 Learning Rate and Optimizer

There are more hyperparameters in an MLP than the number of neurons and hidden layers that can be changed. Some of the most important are listed below:

Learning rate

The learning rate is the most important hyperparameter. The ideal learning rate is often equal to half the maximum learning rate. Training the model for a few hundred iterations with a very low learning rate, like 10^{-5}, and progressively raising it to a very high number, like 10, is one method of determining a good learning rate. If you plot the loss as a function of the learning rate, you should notice that it first decreases. However, after a while, the learning rate will become excessive, causing the loss to quickly increase again. The optimal learning rate will be slightly lower than the point at which the loss begins to increase.

Optimizer

Equally important is selecting a better optimizer than just an odd mini-batch gradient descent. There are several optimizers that you can use to speed up the training but we are going to see just one, *Adam*. Adam [KB14], which stands for *adaptive moment estimation*, combines the concepts of momentum optimization and RMSProp: it tracks an exponentially decaying average of previous gradients, just like momentum optimization, and just like RMSProp, it tracks an exponentially decaying average of previously squared gradients. These are estimates of the gradients' mean and variance. The algorithm's name comes from the fact that the mean is sometimes referred to as the first instant and the variance as the second. In other words, we can say the algorithm keeps track of two moving averages: the mean and the variance of the gradients; these moving averages are updated at each training step. This allows the algorithm to adapt to changes in the distribution of the gradients, which can be very beneficial for training deep neural networks:

$$t = t + 1 \tag{1.1}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_\theta J(\theta) \tag{1.2}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla_\theta J(\theta))^2 \tag{1.3}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{1.4}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{1.5}$$

$$\theta = \theta - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{1.6}$$

where θ represents the parameters of the model, $\nabla_\theta J(\theta)$ is the gradient of the cost function J with respect to the parameters θ, β_1 and β_2 are the decay rates of the moving averages, α is the learning rate, and ϵis a small constant added to the denominator to prevent division by zero.

At each time step t, the mean m_t and the variance v_t of the gradients are updated using the equations (1.2) and (1.3). Here, m_{t-1} and v_{t-1} represent the moving averages of the gradients and their squares from the previous time step. β_1 and $beta_2$ are hyperparameters that control the decay rate of the moving averages. They determine the amount of weight given to the past values of the gradients and their squares and the amount of weight given to the current gradient and its square. After updating m_t and v_t, the moving averages are corrected for bias using the equations (1.4) and (1.5). The bias correction ensures that the moving averages are a better estimate of the true mean and variance of the gradients, especially at the start of the optimization process when t is small. Finally, the parameters θ are updated using the equation (1.6). Here, α is the learning rate, which determines the step size at which the parameters are updated. The term $\frac{\hat{m}_t}{\sqrt{\hat{v}_t+\epsilon}}$ is an approximation of the gradient of the cost function J with respect to the parameters θ. The addition of ϵ in the denominator is used to prevent division by zero.

This update rule balances the magnitude of the steps taken in the direction of the gradient with the magnitude of the gradient itself, allowing the optimization algorithm to make large steps in the directions with a high gradient and small steps in the directions with a low gradient. This makes the optimization process more efficient and helps the algorithm avoid getting stuck in local minima. There are three variants of Adam: AdaMax, Nadam, and Adam W.

1.2.6 Batch Normalization

The danger of the vanishing/exploding gradients problems can be considerably reduced at the beginning of training by employing Glorot initialization in conjunction with ReLU, but this does not ensure that they won't reappear later on. In a 2015 paper [IS15], Sergey Ioffe and Christopher Szegedy suggested a method to solve these issues called *batch normalization* (BN). The procedure includes inserting an operation into the model just before or after each hidden layer's activation function. The technique helps the model discover the ideal mean and scale for each input layer. In many circumstances, standardizing your training set is unnecessary if you include a BN layer as the initial layer of your neural network.

Let's say that we have a mini-batch of N examples, and the activations for a particular layer for one example is given by $X = [x_1, x_2, ..., x_d]$, where d is the number of neurons in the layer. The batch normalization algorithm consists of the following steps:

1. Calculate the mean and variance of the activations for the mini-batch:

$$\text{mean} = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{1.7}$$

$$\text{variance} = \frac{1}{N}\sum_{i=1}^{N} (X_i - \text{mean})^2 \tag{1.8}$$

2. Normalize the activations:

$$\hat{X} = \frac{X - \text{mean}}{\sqrt{\text{variance} + \epsilon}} \tag{1.9}$$

   where ϵ is a small constant added for numerical stability.

3. Scale and shift the normalized activations:

$$X_{\text{bn}} = \gamma \cdot \hat{X} + \beta \tag{1.10}$$

   where γ and β are learnable parameters.

4. Use the normalized and scaled activations as inputs to the next layer in the network.

It is important to note that during training, the mean and variance of the activations are calculated on each mini-batch. During inference, the mean and variance are estimated using a running average that is updated during training. This allows the batch normalization layer to normalize the activations in a way that is consistent with the training data. In summary, the batch normalization algorithm normalizes the activations of a layer in a deep neural network by subtracting the mean and dividing by the standard deviation, scaling and shifting the result using learnable parameters, and using the normalized activations as inputs to the next layer in the network.

In conclusion, batch normalization is a widely used and effective technique for improving the training and performance of deep neural networks. It can help speed up training, prevent overfitting, and make the training process more robust to changes in the scale of the inputs and weights.

1.2.7 Dropout

Dropout, Figure 1.4, is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper [Hin+12] by Geoffry Hinton et al. in 2012. It is a relatively straightforward algorithm: at each training step, each neuron (including input neurons but never output neurons) has a probability p of being temporarily "dropped out," which means it will be completely ignored during this training phase but may be active during the next. This means that their activations are set to zero, and their incoming and outgoing connections are ignored. This has the effect of reducing the number of parameters in the network and making it more difficult for the network to memorize the training data. The dropout rate, also known as the hyperparameter p, is normally set between 10% and 50%; in recurrent neural networks, it is more likely to be between 20% and 30%.

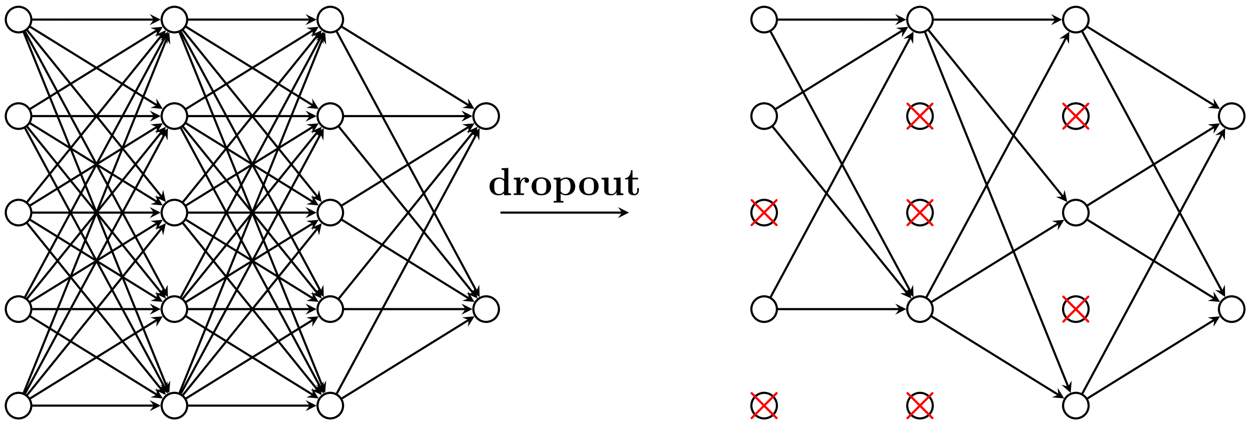

Figure 1.4: Dropout Scheme - A regularization technique used in neural networks to prevent overfitting by randomly dropping out some of the neurons in the network during training.

For example, if the dropout rate is 0.5, then during each training iteration, on average, half of the units in the network will be dropped out. This results in a different, randomly perturbed network architecture at each training iteration, which helps prevent overfitting. The dropout rate can be tuned through experiments to find the optimal value for a given problem and network architecture. In general, a dropout rate of 0.5 is a good starting point, but the optimal value will depend on the specifics of the problem and the network architecture.

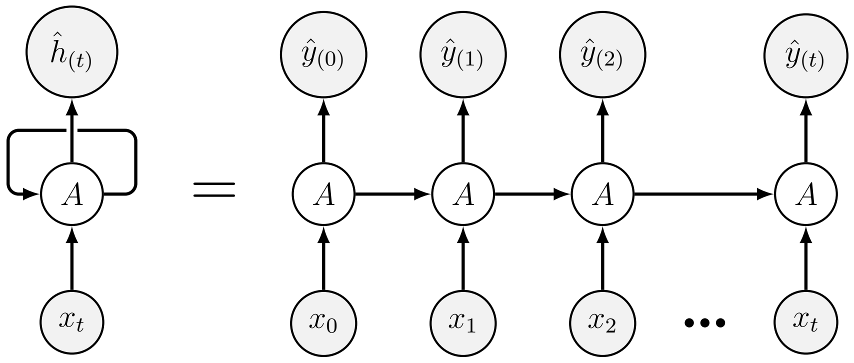

Figure 1.5: A Recurrent Neuron (left) Unrolled through Time (right) - Recurrent neural networks (RNNs) are used to model sequences of data, with each neuron in the network receiving input not only from the current time step but also from the previous time step.

1.3 Recurrent Neural Networks

A class of nets called recurrent neural networks (RNNs) is capable of foreseeing the future. RNNs are capable of analyzing a variety of time series data, like the number of daily visitors to your website, the local hourly temperature, and more. An RNN can forecast the future using its knowledge of past patterns in the data, presuming of course that those patterns will continue to exist. Similar in appearance to a feedforward neural network, a recurrent neural network also includes connections pointing backward.

Let's discuss the simplest possible RNN, which consists of a single neuron taking inputs, producing output, and sending that output back to the neuron that received it, Figure 1.5 (left). This recurrent neuron receives its own output from the previous time step $\hat{y}_{(t-1)}$ and the inputs $x_{(t)}$ at each time step t. At the first time step, the output is set at 0, since there is no output at the previous time step. This little network can be shown in relation to the time axis, Figure 1.5 (right). "Unrolling the network through time" is what is meant by this. Each neuron receives the output vector from the previous time step, $\hat{y}_{(t-1)}$, as well as the input vector $x_{(t)}$, at each time step t. As you can see, now inputs and outputs are vectors. One set of weights is for the inputs $x_{(t)}$, and the other is for the outputs of the previous time step $\hat{y}_{(t-1)}$ for each recurrent neuron. These weight vectors will be abbreviated w_x and $w_{\hat{y}}$. We can organize all the weight vectors into two weight matrices, W_x and $W_{\hat{y}}$, if we think about the entire recurrent layer rather than just one recurrent neuron. The output vector of the entire recurrent layer can then be calculated in a similar way to what one might anticipate.

$$\hat{y}_{(t)} = \sigma\left(W_x^T x_{(t)} + w_{\hat{y}}^T \hat{y}_{(t-1)} + b\right) \tag{1.11}$$

Like feedforward neural networks, by putting all the inputs at time step t into an input matrix X, we can compute the output of a recurrent layer in one single step for an entire mini-batch.

$$\begin{aligned}\hat{Y}_{(t)} &= \sigma\left(X_{(t)} W_x + \hat{Y}_{(t-1)} W_{\hat{y}} + b\right)\\ &= \sigma\left(\begin{bmatrix} X_{(t)} & \hat{Y}_{(t-1)} \end{bmatrix} W + b\right) \text{ with } W = \begin{bmatrix} W_x \\ W_{\hat{y}} \end{bmatrix}\end{aligned} \tag{1.12}$$

In above equation, (1.12), we can see:

- $\hat{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's output at time step t for each instance in the mini batch.
- $X_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances.
- W_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $W_{\hat{y}}$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- b is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices W_x and $W_{\hat{y}}$ are concatenated vertically into a single weight matrix W.
- The notation $\begin{bmatrix} X_{(t)} & \hat{Y}_{(t-1)} \end{bmatrix}$ represents the horizontal concatenation of the matrices $X_{(t)}$ and $\hat{Y}_{(t-1)}$.

Notice that $\hat{Y}_{(t)}$ is a function of $X_{(t)}$ and $\hat{Y}_{(t-1)}$, which is a function of $X_{(t-1)}$ and $\hat{Y}_{(t-2)}$, which is a function of $X_{(t-2)}$ and $\hat{Y}_{(t-3)}$, and so on. This makes $\hat{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is $X_{(0)}, X_{(1)}, X_{(2)}, ..., X_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are assumed to be all zeros.

1.3.1 How to train RNNs

You could say that a recurrent neuron has a form of memory because its output at a given time step t is a function of all its inputs from earlier time steps. A *memory cell* is a component of a neural network that keeps a certain state over successive time steps. The state of a cell at time step t, represented by the symbol $h_{(t)}$, is a function of some inputs at that time step and its state at the previous time step. So, we can say $h_{(t)} = f\left(x_{(t)}, h_{(t-1)}\right)$. The previous state and the current inputs are functions of the output at time step t, indicated as $\hat{y}_{(t)}$.

An RNN can accept a series of inputs and generate different sequences:

- *Sequence-to-sequence network:* it takes a sequence of inputs and produces a sequence of outputs at each time step t.
- *Sequence-to-vector network:* it takes a sequence of inputs, and you can consider only some outputs. For example, if you have 5 inputs, you might want only the last output, so you can ignore all the previous outputs.
- *Vector-to-sequence network:* The input sequence is a vector that you pass into the network at each time step and let it output a sequence.
- *Encoder-decoder network:* This network is mostly used for translations. You pass a sentence in one language, and the output will be translated in another language.

The idea is to unroll an RNN over time before using traditional backpropagation to train it. The term *backpropagation over time* (BPTT) refers to this technique. The network is initially passed forward after it has been unrolled. After that, a loss function is used to evaluate the output sequence.

$$L(Y_{(0)}, Y_{(1)}, ..., Y_{(T)}; \hat{Y}_{(0)}, \hat{Y}_{(1)}, ..., \hat{Y}_{(T)}) \tag{1.13}$$

where $Y_{(i)}$ is the i^{th} output, $\hat{Y}_{(i)}$ is the i^{th} prediction and T is the max time step. For example, if we think about *sequence-to-vector network*, we want to compute only just the last two outputs of the network, ignoring the first three outputs. It means that the loss function isn't computed on all outputs, but just on the last two.

The unrolled network then propagates the gradients of that loss function backward. The gradients only pass through the outputs $\hat{Y}_{(3)}$ and $\hat{Y}_{(4)}$, since in the example the outputs $\hat{Y}_{(0)}$, $\hat{Y}_{(1)}$ and $\hat{Y}_{(2)}$ are not used to calculate the loss. Thusly, because W and b are identical parameters at every time step, their gradients will be changed numerous times during backprop. The parameters can be updated using a gradient descent step using BPTT when the backward phase is finished, and all the gradients have been computed. This is how RNN training is made.

1.3.2 Long short-term memory (LSTM)

In 1997, Seep Hochreiter and Jürgen Schmidhuber proposed the ”Long Short-Term Memory” (LSTM) cell, which was progressively improved over time by other researchers [HS97] . If the LSTM cell is viewed as a black box, it can be used in a similar way to that of a basic cell but will perform much better. Training will converge more quickly and find longer-term patterns in the data.

How do LSTM cells work? Figure 1.6 represents its architecture. The LSTM cell appears just like a standard cell from the outside, except for the fact that its state has been divided into two vectors, $h_{(t)}$ and $c_{(t)}$. The short-term state is represented by $h_{(t)}$, and the long-term state is represented by $c_{(t)}$.

The main concept is that the network can learn what to read from it, and what to discard or store in the long-term state. You can see that as the long-term state $c_{(t-1)}$ moves from left to right throughout the network, it first uses a forget gate to delete some memories before adding some new ones using an addition operation that includes memories that were chosen by an input gate. Without any further change, the result $c_{(t-1)}$ is sent out directly. Several memories are added, and some are deleted at each time step. The long-term state is copied and then passed via the tanh function following the addition operation. The output gate then filters the outcome. This produces the short-term state $h_{(t)}$.

Now let’s see how gates perform. First, four separate fully connected layers take the current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$. Each one has a specific function:

- The layer that outputs $g_{(t)}$ is the primary layer. Its regular functions include processing the inputs for the present $x_{(t)}$ and the past $h_{(t-1)}$ states. The output of this layer is not sent directly outside. Instead, its most fundamental parts are stored in the long-term state. The rest is dropped.

- Gate controllers are the three additional layers. The outputs are in the sigmoid activation function range, which is 0 to 1. The outputs from the gate controllers are fed into element-wise multiplication processes; if the output is a 0, the gate is closed; if a 1, the gate is opened. Particularly:

– The *forget gate* $f_{(t)}$: determines which elements of the long-term state should be removed.
– The *input gate* $i_{(t)}$: determines which $g_{(t)}$ components go into the long-term state.
– The *output gate* $o_{(t)}$: determines which elements of the long-term state should be read and output at this time step, both to $h_{(t)}$ and $y_{(t)}$,

In conclusion, we can say that an LSTM cell can understand how to identify important input, *"input gate"* role, and it can store in a long-term state, preserve and use it whenever it wants, *"forget gate"* role. There are more variants of the LSTM cell. Let's see now, the most used and important: the *GRU* cell.

1.3.3 Gated Recurrent Unit (GRU)

In a 2014 paper, Kyunghyun Cho et al. made the suggestion for the Gated Recurrent Unit (GRU) cell [Cho+14]. The GRU cell, which is an LSTM cell simplified, seems to work just as well. The main changes are as follows:

- A single vector $h_{(t)}$ is created by combining the two state vectors.

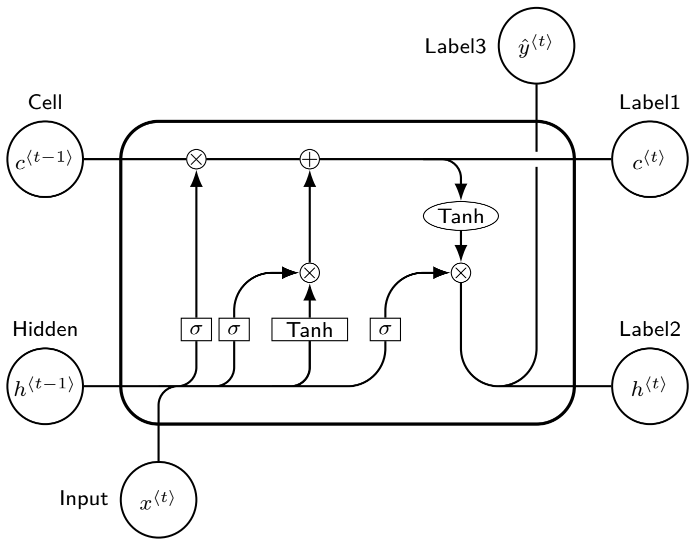

Figure 1.6: An LSTM Cell - A type of recurrent neural network cell that can selectively remember or forget information from previous time steps, making it particularly useful for processing sequential data such as text and speech.

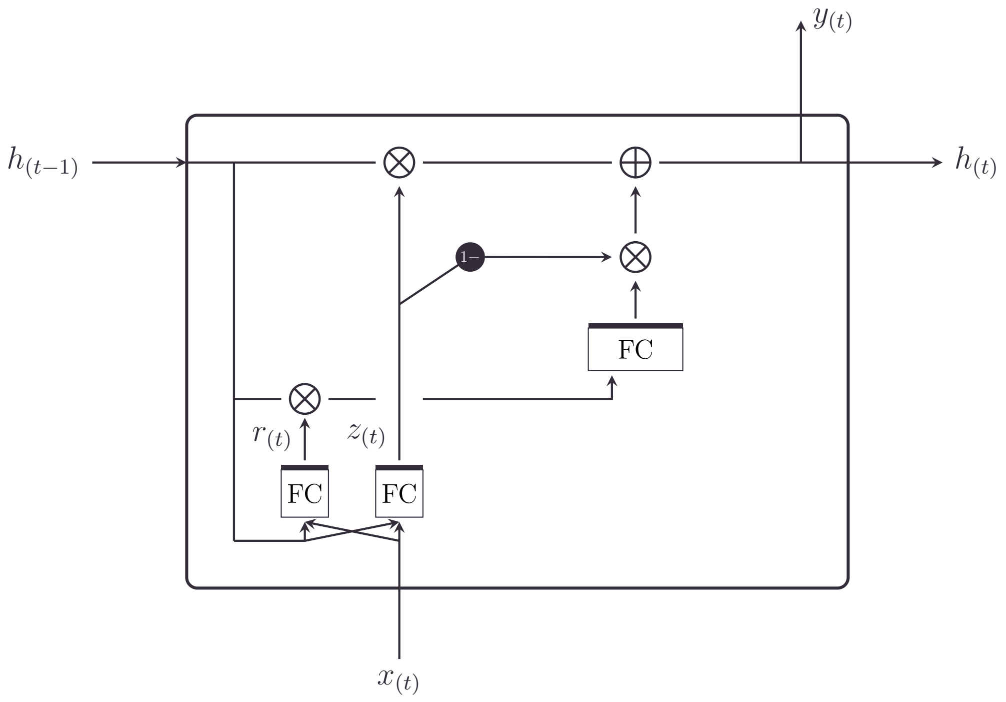

Figure 1.7: An GRU Cell - A type of recurrent neural network cell that uses gating mechanisms to control the flow of information through the cell, allowing it to selectively update or retain information from previous time steps.

- Both the input gate and the forget gate are managed by a single gate controller $z_{(t)}$. The input gate is closed $(1 - 1 = 0)$ and the forget gate is open $(= 1)$ if the gate controller sends a 1. The opposite occurs if the output is a 0. To put it another way, whenever a memory needs to be saved, the area where it will be stored must first be deleted.
- The entire state vector is output at each time step; there is no output gate. The main layer $g_{(t)}$ will only see certain portions of the prior state, thanks to a new gate controller $r_{(t)}$.

One of the key elements in the success of RNNs is the use of LSTM and GRU cells.

Bibliography

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[GB10] 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*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[Hin+12] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Cho+14] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).