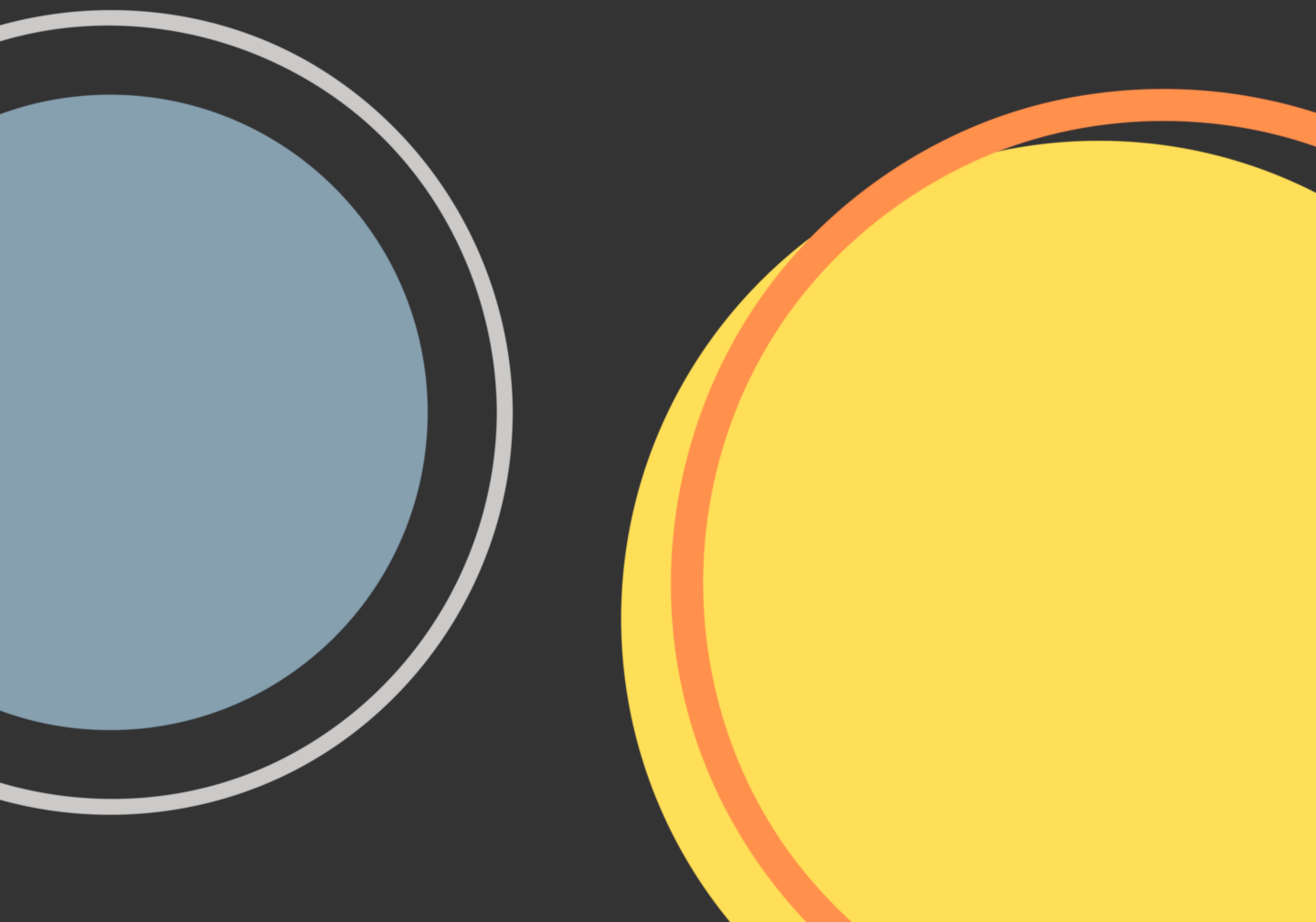
Andriy Burkov

THE HUNDRED-PAGE MACHINE LEARNING BOOK



"A great introduction to machine learning from a world-class practitioner."
— **Karolis Urbonas**, Head of Data Science at Amazon

"I wish such a book existed when I was a statistics graduate student trying to learn about machine learning."
— **Chao Han**, VP, Head of R&D at Lucidworks

"Andriy's book does a fantastic job of cutting the noise and hitting the tracks and full speed from the first page."
— **Sujeet Varakhedi**, Head of Engineering at eBay

"A wonderful book for engineers who want to incorporate ML in their day-to-day work without necessarily spending an enormous amount of time."
— **Deepak Agarwal**, VP of Artificial Intelligence at LinkedIn

"Excellent read to get started with Machine Learning."
— **Vincent Pollet**, Head of Research at Nuance

with back cover text from **Peter Norvig** and **Aurélien Geron**

The Hundred-Page Machine Learning Book

Andriy Burkov

ISBN 978-1-9995795-0-0

Publisher: Andriy Burkov

To my parents:
Tatiana and Valeriy

and to my family:
daughters Catherine and Eva,
and brother Dmitriy

“All models are wrong, but some are useful.”
— George Box

“If I had more time, I would have written a shorter letter.”
— Blaise Pascal

The book is distributed on the “read first, buy later” principle.

Contents

Chapter 6

Neural Networks and Deep Learning

First of all, you already know what a neural network is, and you already know how to build such a model. Yes, it's logistic regression! As a matter of fact, the logistic regression model, or rather its generalization for multiclass classification, called the softmax regression model, is a standard unit in a neural network.

6.1 Neural Networks

If you understood linear regression, logistic regression, and gradient descent, understanding neural networks should not be a problem.

A **neural network** (NN), just like a regression or an SVM model, is a mathematical function:

$$y = f_{NN}(\mathbf{x}).$$

The function f_{NN} has a particular form: it's a **nested function**. You have probably already heard of neural network **layers**. So, for a 3-layer neural network that returns a scalar, f_{NN} looks like this:

$$y = f_{NN}(\mathbf{x}) = f_3(\boldsymbol{f}_2(\boldsymbol{f}_1(\mathbf{x}))).$$

In the above equation, $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are vector functions of the following form:

$$\boldsymbol{f}_l(\mathbf{z}) \stackrel{\text{def}}{=} \boldsymbol{g}_l(\mathbf{W}_l\mathbf{z} + \mathbf{b}_l), \tag{6.1}$$

where l is called the layer index and can span from 1 to any number of layers. The function $\boldsymbol{g}_l$ is called an **activation function**. It is a fixed, usually nonlinear function chosen by the data analyst before the learning is started. The parameters $\mathbf{W}_l$ (a matrix) and $\mathbf{b}_l$ (a vector) for each layer are learned using the familiar gradient descent by optimizing, depending on the task, a particular cost function (such as MSE). Compare eq. 6.1 with the equation for logistic regression, where you replace $\boldsymbol{g}_l$ by the sigmoid function, and you will not see any difference. The function f_3 is a scalar function for the regression task, but can also be a vector function depending on your problem.

You may probably wonder why a matrix $\mathbf{W}_l$ is used and not a vector $\mathbf{w}_l$. The reason is that $\boldsymbol{g}_l$ is a vector function. Each row $\mathbf{w}_{l,u}$ (u for unit) of the matrix $\mathbf{W}_l$ is a vector of the same dimensionality as $\mathbf{z}$. Let $a_{l,u} = \mathbf{w}_{l,u}\mathbf{z} + b_{l,u}$. The output of $\boldsymbol{f}_l(\mathbf{z})$ is a vector $[g_l(a_{l,1}), g_l(a_{l,2}), \ldots, g_l(a_{l,size_l})]$, where g_l is some scalar function[1], and $size_l$ is the number of units in layer l. To make it more concrete, let's consider one architecture of neural networks called **multilayer perceptron** and often referred to as a **vanilla neural network**.

6.1.1 Multilayer Perceptron Example

We have a closer look at one particular configuration of neural networks called **feed-forward neural networks** (FFNN), and more specifically the architecture called a **multilayer perceptron** (MLP). As an illustration, let's consider an MLP with three layers. Our network takes a two-dimensional feature vector as input and outputs a number. This FFNN can be a regression or a classification model, depending on the activation function used in the third, output layer.

Our MLP is depicted in Figure 6.1. The neural network is represented graphically as a connected combination of **units** logically organized into one or more **layers**. Each unit is represented by either a circle or a rectangle. The inbound arrow represents an input of a unit and indicates where this input came from. The outbound arrow indicates the output of a unit.

The output of each unit is the result of the mathematical operation written inside the rectangle. Circle units don't do anything with the input; they just send their input directly to the output.

The following happens in each rectangle unit. Firstly, all inputs of the unit are joined together to form an input vector. Then the unit applies a linear transformation to the input vector, exactly like linear regression model does with its input feature vector. Finally, the unit applies an activation function g to the result of the linear transformation and obtains the output value, a real number. In a vanilla FFNN, the output value of a unit of some layer becomes an input value of each of the units of the subsequent layer.

In Figure 6.1, the activation function g_l has one index: l, the index of the layer the unit belongs to. Usually, all units of a layer use the same activation function, but it's not a rule. Each layer can have a different number of units. Each unit has its parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$, where u is the index of the unit, and l is the index of the layer. The vector $\mathbf{y}_{l-1}$ in each unit is defined as $[y_{l-1}^{(1)}, y_{l-1}^{(2)}, y_{l-1}^{(3)}, y_{l-1}^{(4)}]$. The vector $\mathbf{x}$ in the first layer is defined as $[x^{(1)}, \ldots, x^{(D)}]$.

[1] A scalar function outputs a scalar, that is a simple number and not a vector.

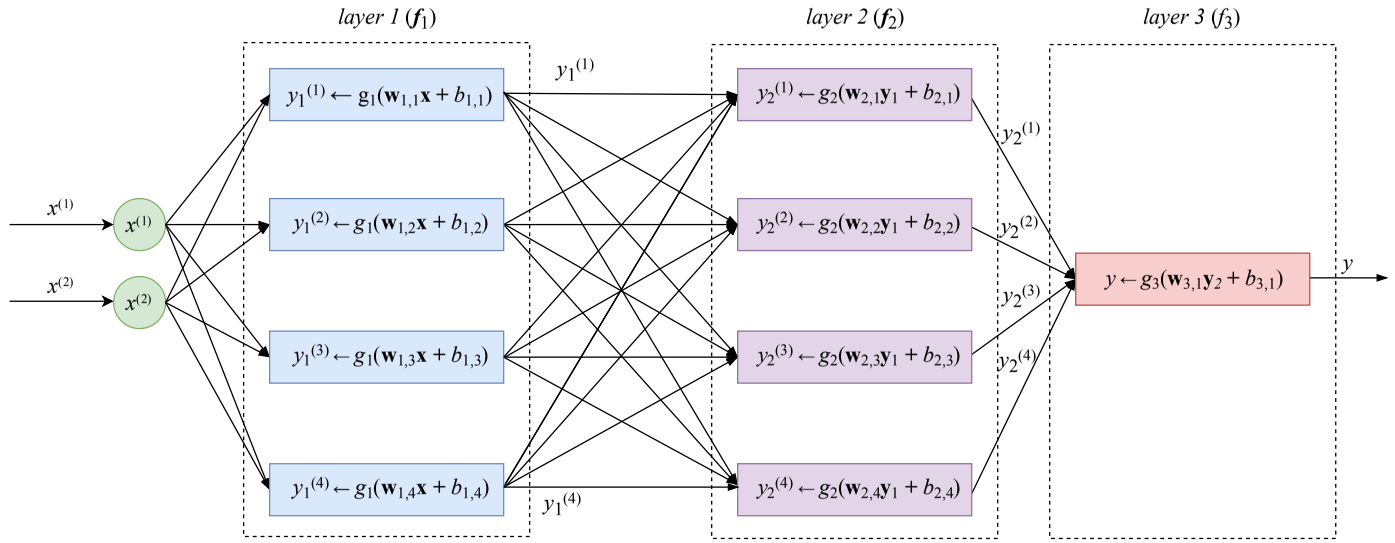


Figure 6.1: A multilayer perceptron with two-dimensional input, two layers with four units and one output layer with one unit.

As you can see in Figure 6.1, in multilayer perceptron all outputs of one layer are connected to each input of the succeeding layer. This architecture is called **fully-connected**. A neural network can contain **fully-connected layers**. Those are the layers whose units receive as inputs the outputs of each of the units of the previous layer.

6.1.2 Feed-Forward Neural Network Architecture

If we want to solve a regression or a classification problem discussed in previous chapters, the last (the rightmost) layer of a neural network usually contains only one unit. If the activation function g_{last} of the last unit is linear, then the neural network is a regression model. If the g_{last} is a logistic function, the neural network is a binary classification model.

The data analyst can choose any mathematical function as $g_{l,u}$, assuming it's differentiable[2]. The latter property is essential for gradient descent used to find the values of the parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$ for all l and u. The primary purpose of having nonlinear components in the function f_{NN} is to allow the neural network to approximate nonlinear functions. Without nonlinearities, f_{NN} would be linear, no matter how many layers it has. The reason is that $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$ is a linear function and a linear function of a linear function is also linear.

Popular choices of activation functions are the logistic function, already known to you, as well as **TanH** and **ReLU**. The former is the hyperbolic tangent function, similar to the logistic function but ranging from -1 to 1 (without reaching them). The latter is the rectified linear unit function, which equals to zero when its input z is negative and to z otherwise:

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

$$relu(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}.$$

As I said above, $\mathbf{W}_l$ in the expression $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$, is a matrix, while $\mathbf{b}_l$ is a vector. That looks different from linear regression's $\mathbf{w}\mathbf{z} + b$. In matrix $\mathbf{W}_l$, each row u corresponds to a vector of parameters $\mathbf{w}_{l,u}$. The dimensionality of the vector $\mathbf{w}_{l,u}$ equals to the number of units in the layer $l-1$. The operation $\mathbf{W}_l\mathbf{z}$ results in a vector $\mathbf{a}_l \stackrel{\text{def}}{=} [\mathbf{w}_{l,1}\mathbf{z}, \mathbf{w}_{l,2}\mathbf{z}, \ldots, \mathbf{w}_{l,size_l}\mathbf{z}]$. Then the sum $\mathbf{a}_l + \mathbf{b}_l$ gives a $size_l$-dimensional vector $\mathbf{c}_l$. Finally, the function $\boldsymbol{g}_l(\mathbf{c}_l)$ produces the vector $\mathbf{y}_l \stackrel{\text{def}}{=} [y_l^{(1)}, y_l^{(2)}, \ldots, y_l^{(size_l)}]$ as output.

[2]The function has to be differentiable across its whole domain or in the majority of the points of its domain. For example, ReLU is not differentiable at 0.

6.2 Deep Learning

Deep learning refers to training neural networks with more than two non-output layers. In the past, it became more difficult to train such networks as the number of layers grew. The two biggest challenges were referred to as the problems of **exploding gradient** and **vanishing gradient** as gradient descent was used to train the network parameters.

While the problem of exploding gradient was easier to deal with by applying simple techniques like **gradient clipping** and L1 or L2 regularization, the problem of vanishing gradient remained intractable for decades.

What is vanishing gradient and why does it arise? To update the values of the parameters in neural networks the algorithm called **backpropagation** is typically used. Backpropagation is an efficient algorithm for computing gradients on neural networks using the chain rule. In Chapter 4, we have already seen how the chain rule is used to calculate partial derivatives of a complex function. During gradient descent, the neural network's parameters receive an update proportional to the partial derivative of the cost function with respect to the current parameter in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing some parameters from changing their value. In the worst case, this may completely stop the neural network from further training.

Traditional activation functions, such as the hyperbolic tangent function I mentioned above, have gradients in the range $(0, 1)$, and backpropagation computes gradients by the chain rule. That has the effect of multiplying n of these small numbers to compute gradients of the earlier (leftmost) layers in an n-layer network, meaning that the gradient decreases exponentially with n. That results in the effect that the earlier layers train very slowly, if at all.

However, the modern implementations of neural network learning algorithms allow you to effectively train very deep neural networks (up to hundreds of layers). This is due to several improvements combined together, including ReLU, LSTM (and other gated units; we consider them below), as well as techniques such as **skip connections** used in **residual neural networks**, as well as advanced modifications of the gradient descent algorithm.

Therefore, today, since the problems of vanishing and exploding gradient are mostly solved (or their effect diminished) to a great extent, the term "deep learning" refers to training neural networks using the modern algorithmic and mathematical toolkit independently of how deep the neural network is. In practice, many business problems can be solved with neural networks having 2-3 layers between the input and output layers. The layers that are neither input nor output are often called **hidden layers**.

6.2.1 Convolutional Neural Network

You may have noticed that the number of parameters an MLP can have grows very fast as you make your network bigger. More specifically, as you add one layer, you add $(size_{l-1}+1) \cdot size_l$ parameters (our matrix $\mathbf{W}_l$ plus the vector $\mathbf{b}_l$). That means that if you add another 1000-unit

layer to an existing neural network, then you add more than 1 million additional parameters to your model. Optimizing such big models is a very computationally intensive problem.

When our training examples are images, the input is very high-dimensional[3]. If you want to learn to classify images using an MLP, the optimization problem is likely to become intractable.

A **convolutional neural network** (CNN) is a special kind of FFNN that significantly reduces the number of parameters in a deep neural network with many units without losing too much in the quality of the model. CNNs have found applications in image and text processing where they beat many previously established benchmarks.

Because CNNs were invented with image processing in mind, I explain them on the image classification example.

You may have noticed that in images, pixels that are close to one another usually represent the same type of information: sky, water, leaves, fur, bricks, and so on. The exception from the rule are the edges: the parts of an image where two different objects "touch" one another.

If we can train the neural network to recognize regions of the same information as well as the edges, this knowledge would allow the neural network to predict the object represented in the image. For example, if the neural network detected multiple skin regions and edges that look like parts of an oval with skin-like tone on the inside and bluish tone on the outside, then it is likely that it's a face on the sky background. If our goal is to detect people on pictures, the neural network will most likely succeed in predicting a person in this picture.

Having in mind that the most important information in the image is local, we can split the image into square patches using a moving window approach[4]. We can then train multiple smaller regression models at once, each small regression model receiving a square patch as input. The goal of each small regression model is to learn to detect a specific kind of pattern in the input patch. For example, one small regression model will learn to detect the sky; another one will detect the grass, the third one will detect edges of a building, and so on.

In CNNs, a small regression model looks like the one in Figure 6.1, but it only has the layer 1 and doesn't have layers 2 and 3. To detect some pattern, a small regression model has to learn the parameters of a matrix $\mathbf{F}$ (for "filter") of size $p \times p$, where p is the size of a patch. Let's assume, for simplicity, that the input image is black and white, with 1 representing black and 0 representing white pixels. Assume also that our patches are 3 by 3 pixels ($p = 3$). Some patch could then look like the following matrix $\mathbf{P}$ (for "patch"):

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

[3]Each pixel of an image is a feature. If our image is 100 by 100 pixels, then there are 10,000 features.

[4]Consider this as if you looked at a dollar bill in a microscope. To see the whole bill you have to gradually move your bill from left to right and from top to bottom. At each moment in time, you see only a part of the bill of fixed dimensions. This approach is called **moving window**.

The above patch represents a pattern that looks like a cross. The small regression model that will detect such patterns (and only them) would need to learn a 3 by 3 parameter matrix $\mathbf{F}$ where parameters at positions corresponding to the 1s in the input patch would be positive numbers, while the parameters in positions corresponding to 0s would be close to zero. If we calculate the **convolution** of matrices $\mathbf{P}$ and $\mathbf{F}$, the value we obtain is higher the more similar $\mathbf{F}$ is to $\mathbf{P}$. To illustrate the convolution of two matrices, assume that $\mathbf{F}$ looks like this:

$$\mathbf{F} = \begin{bmatrix} 0 & 2 & 3 \\ 2 & 4 & 1 \\ 0 & 3 & 0 \end{bmatrix}.$$

Then convolution operator is only defined for matrices that have the same number of rows and columns. For our matrices of $\mathbf{P}$ and $\mathbf{F}$ it's calculated as illustrated below:

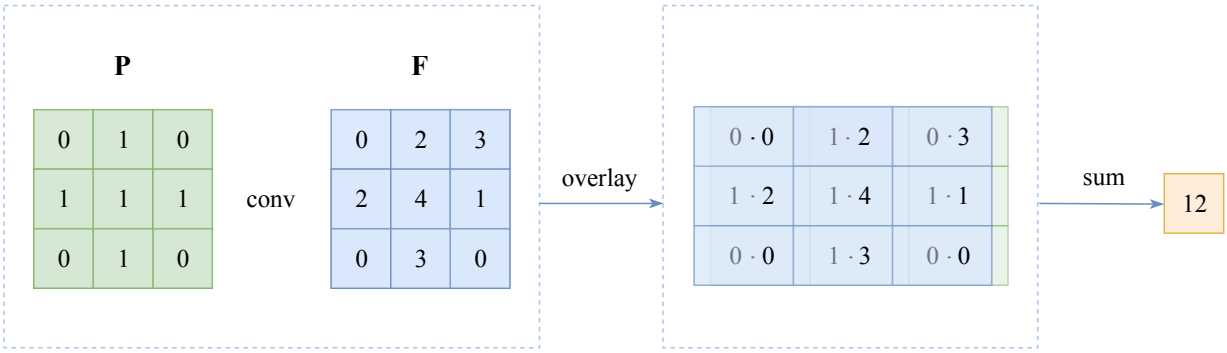


Figure 6.2: A convolution between two matrices.

If our input patch $\mathbf{P}$ had a different patten, for example, that of a letter L,

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

then the convolution with $\mathbf{F}$ would give a lower result: 5. So, you can see the more the patch "looks" like the filter, the higher the value of the convolution operation is. For convenience, there's also a bias parameter b associated with each filter $\mathbf{F}$ which is added to the result of a convolution before applying the nonlinearity (activation function).

One layer of a CNN consists of multiple convolution filters (each with its own bias parameter), just like one layer in a vanilla FFNN consists of multiple units. Each filter of the first (leftmost) layer slides — or *convolves* — across the input image, left to right, top to bottom, and convolution is computed at each iteration.

An illustration of the process is given in Figure 6.3 where 6 steps of one filter convolving across an image are shown.

The filter matrix (one for each filter in each layer) and bias values are trainable parameters that are optimized using gradient descent with backpropagation.

A nonlinearity is applied to the sum of the convolution and the bias term. Typically, the ReLU activation function is used in all hidden layers. The activation function of the output layer depends on the task.

Since we can have $size_l$ filters in each layer l, the output of the convolution layer l would consist of $size_l$ matrices, one for each filter.

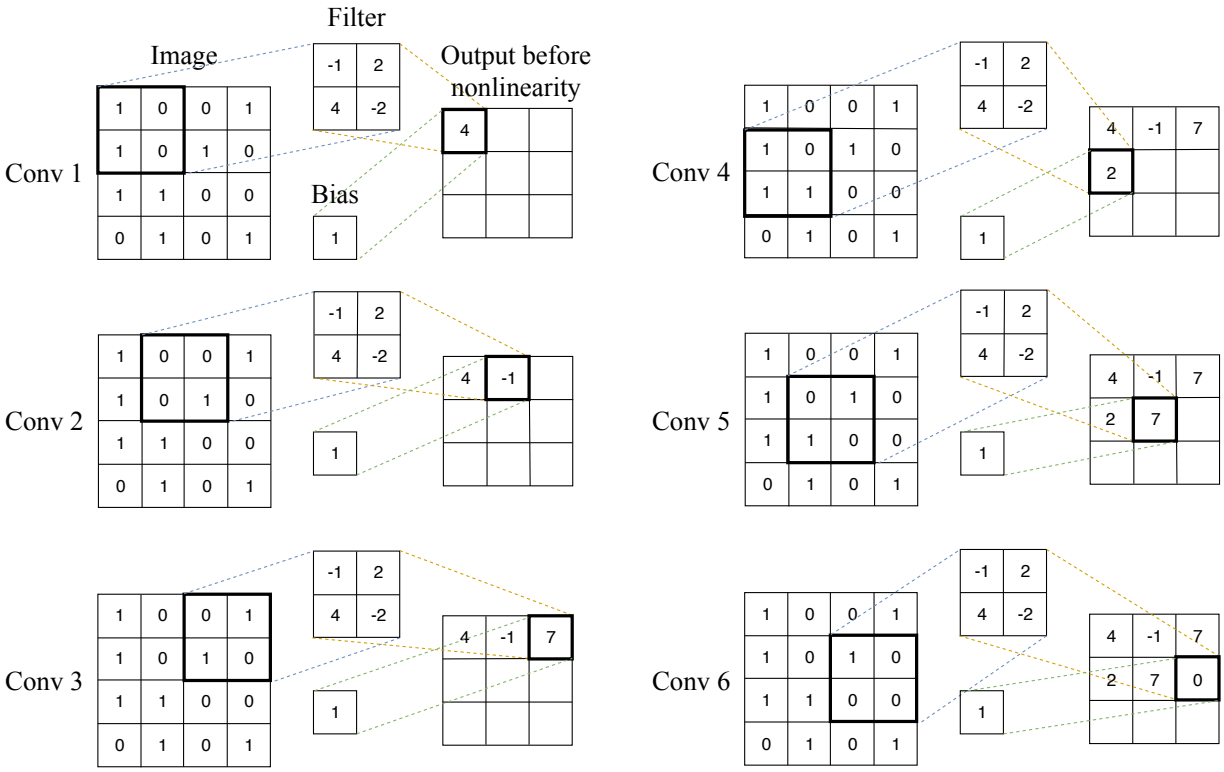


Figure 6.3: A filter convolving across an image.

If the CNN has one convolution layer following another convolution layer, then the subsequent layer $l + 1$ treats the output of the preceding layer l as a collection of $size_l$ image matrices. Such a collection is called a **volume**. The size of that collection is called the volume's depth. Each filter of layer $l + 1$ convolves the whole volume. The convolution of a patch of a volume is simply the sum of convolutions of the corresponding patches of individual matrices the volume consists of.

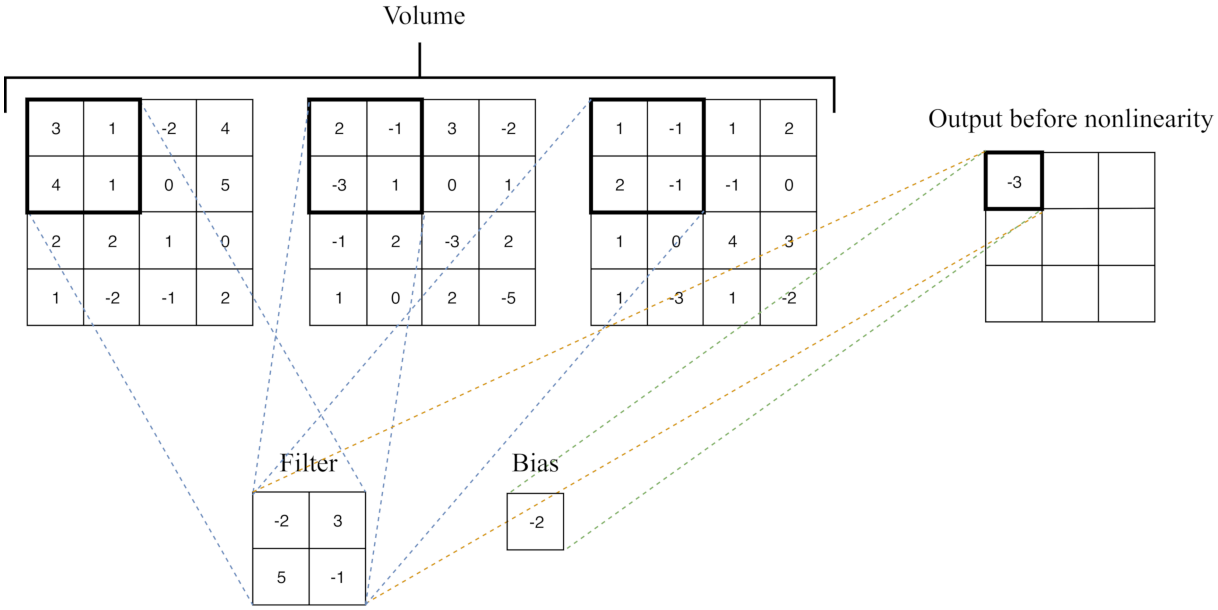


Figure 6.4: Convolution of a volume consisting of three matrices.

An example of a convolution of a patch of a volume consisting of depth 3 is shown in Figure 6.4. The value of the convolution, -3, was obtained as $(-2 \cdot 3 + 3 \cdot 1 + 5 \cdot 4 + -1 \cdot 1) + (-2 \cdot 2 + 3 \cdot (-1) + 5 \cdot (-3) + -1 \cdot 1) + (-2 \cdot 1 + 3 \cdot (-1) + 5 \cdot 2 + -1 \cdot (-1)) + (-2)$.

In computer vision, CNNs often get volumes as input, since an image is usually represented by three channels: R, G, and B, each channel being a monochrome picture.

Two important properties of convolution are **stride** and **padding**. Stride is the step size of the moving window. In Figure 6.3, the stride is 1, that is the filter slides to the right and to the bottom by one cell at a time. In Figure 6.5, you can see a partial example of convolution with stride 2. You can see that the output matrix is smaller when stride is bigger.

Padding allows getting a larger output matrix; it's the width of the square of additional cells with which you surround the image (or volume) before you convolve it with the filter. The cells added by padding usually contain zeroes. In Figure 6.3, the padding is 0, so no additional cells are added to the image. In Figure 6.6, on the other hand, the stride is 2 and padding is 1, so a square of width 1 of additional cells are added to the image. You can see that the output matrix is bigger when padding is bigger[5].

[5]To save space, in Figure 6.6, only the first two of the nine convolutions are shown.

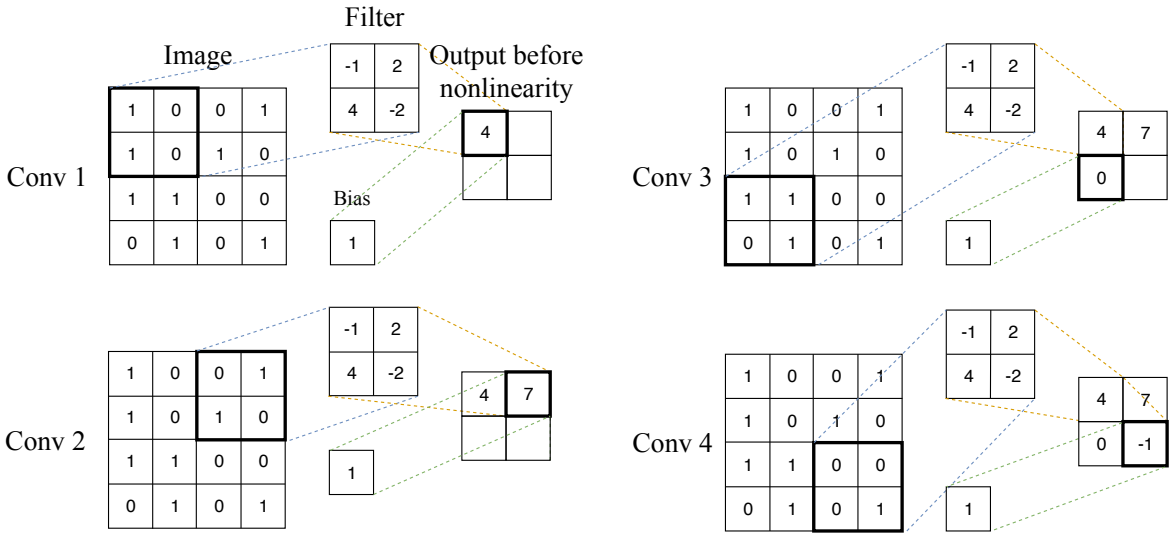


Figure 6.5: Convolution with stride 2.

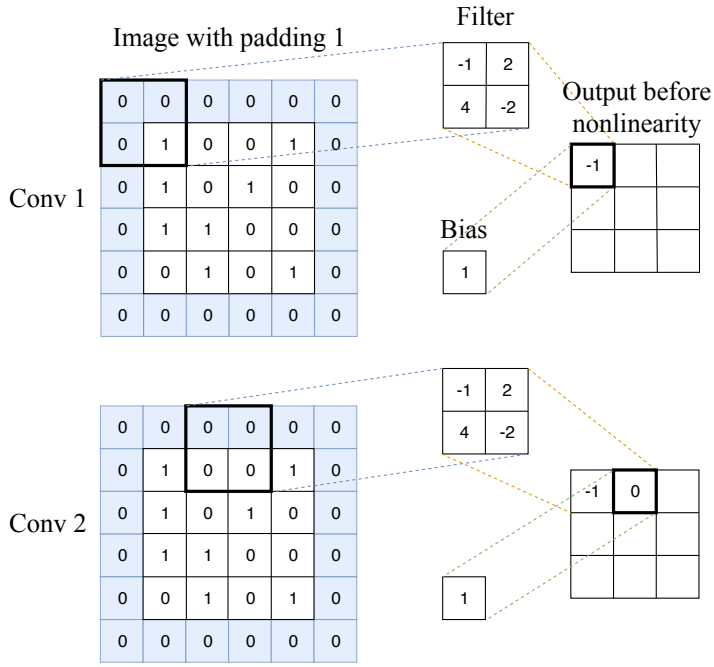


Figure 6.6: Convolution with stride 2 and padding 1.

An example of an image with padding 2 is shown in Figure 6.7. Padding is helpful with larger filters because it allows them to better "scan" the boundaries of the image.

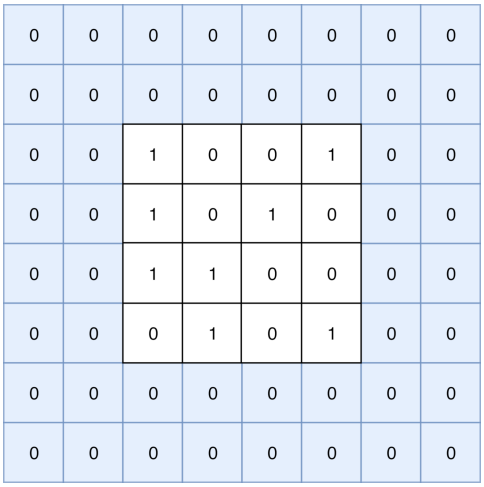


Figure 6.7: Image with padding 2.

This section would not be complete without presenting **pooling**, a technique very often used in CNNs. Pooling works in a way very similar to convolution, as a filter applied using a moving window approach. However, instead of applying a trainable filter to an input matrix or a volume, pooling layer applies a fixed operator, usually either max or average. Similarly to convolution, pooling has hyperparameters: the size of the filter and the stride. An example of max pooling with filter of size 2 and stride 2 is shown in Figure 6.8.

Usually, a pooling layer follows a convolution layer, and it gets the output of convolution as input. When pooling is applied to a volume, each matrix in the volume is processed independently of others. Therefore, the output of the pooling layer applied to a volume is a volume of the same depth as the input.

As you can see, pooling only has hyperparameters and doesn't have parameters to learn. Typically, the filter of size 2 or 3 and stride 2 are used in practice. Max pooling is more popular than average and often gives better results.

Typically pooling contributes to the increased accuracy of the model. It also improves the speed of training by reducing the number of parameters of the neural network. (As you can see in Figure 6.8, with filter size 2 and stride 2 the number of parameters is reduced to 25%, that is to 4 parameters instead of 16.)

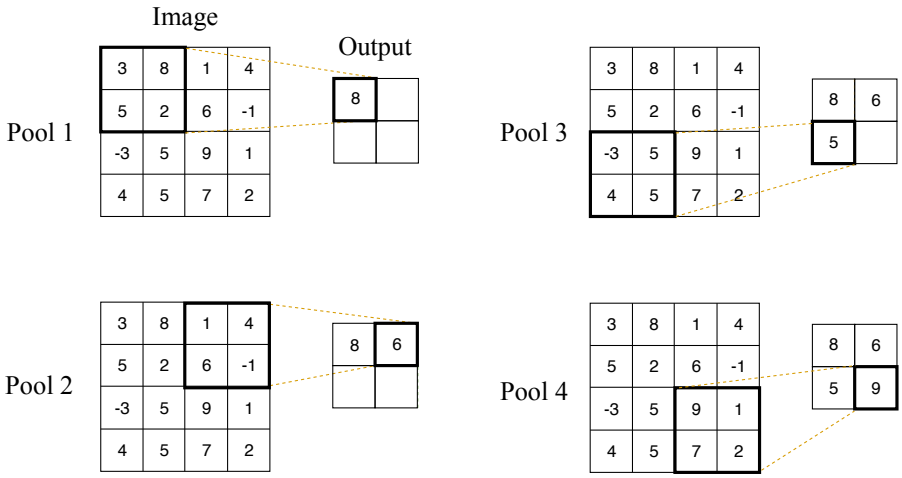


Figure 6.8: Pooling with filter of size 2 and stride 2.

6.2.2 Recurrent Neural Network

Recurrent neural networks (RNNs) are used to label, classify, or generate sequences. A sequence is a matrix, each row of which is a feature vector and the order of rows matters. To label a sequence is to predict a class for each feature vector in a sequence. To classify a sequence is to predict a class for the entire sequence. To generate a sequence is to output another sequence (of a possibly different length) somehow relevant to the input sequence.

RNNs are often used in text processing because sentences and texts are naturally sequences of either words/punctuation marks or sequences of characters. For the same reason, recurrent neural networks are also used in speech processing.

A recurrent neural network is not feed-forward: it contains loops. The idea is that each unit u of recurrent layer l has a real-valued **state** $h_{l,u}$. The state can be seen as the memory of the unit. In RNN, each unit u in each layer l receives two inputs: a vector of states from the previous layer $l-1$ and the vector of states from this same layer l from *the previous time step*.

To illustrate the idea, let's consider the first and the second recurrent layers of an RNN. The first (leftmost) layer receives a feature vector as input. The second layer receives the output of the first layer as input.

This situation is schematically depicted in Figure 6.9. As I said above, each training example is a matrix in which each row is a feature vector. For simplicity, let's illustrate this matrix as a sequence of vectors $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{t-1}, \mathbf{x}^t, \mathbf{x}^{t+1}, \ldots, \mathbf{x}^{length_\mathbf{X}}]$, where $length_\mathbf{X}$ is the length of the input sequence. If our input example $\mathbf{X}$ is a text sentence, then feature vector $\mathbf{x}^t$ for each $t = 1, \ldots, length_\mathbf{X}$ represents a word in the sentence at position t.

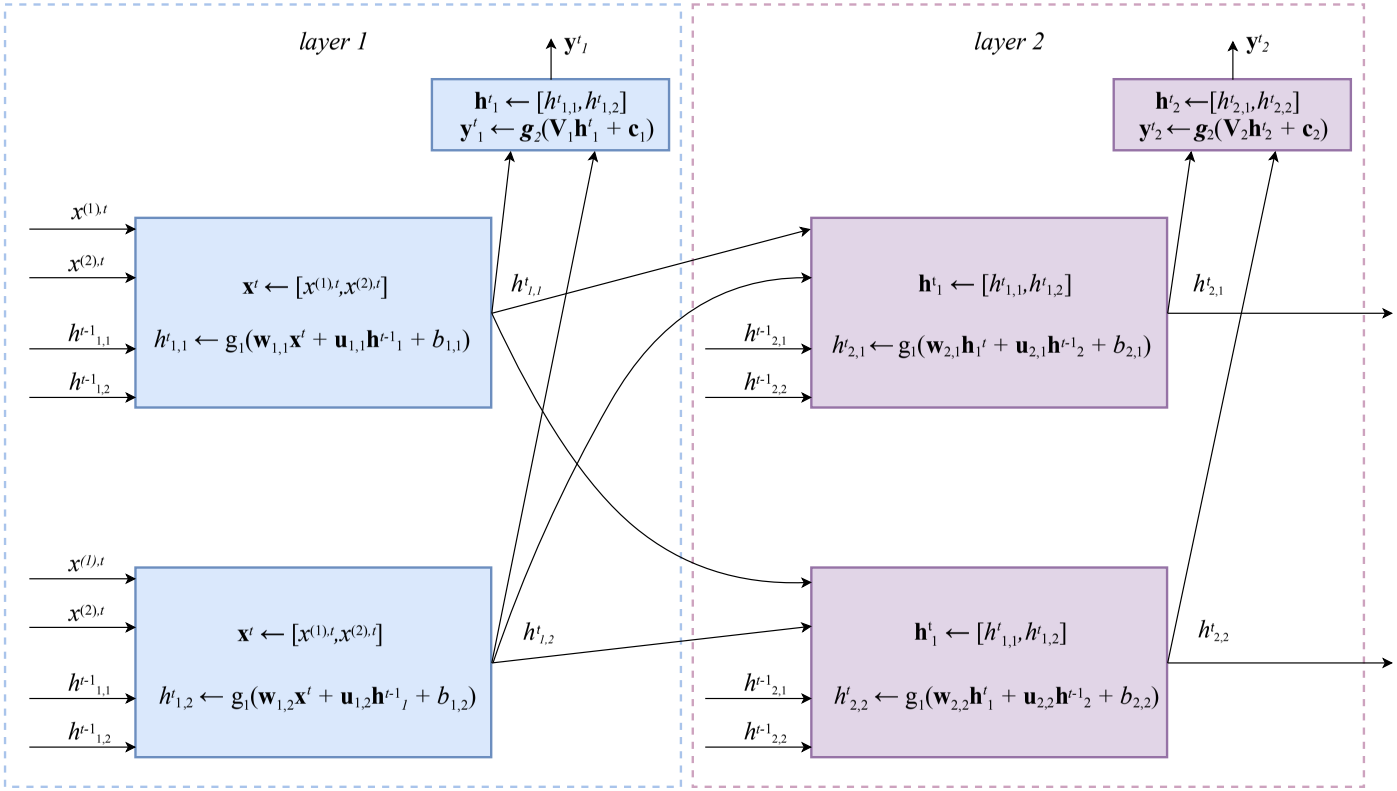


Figure 6.9: The first two layers of an RNN. The input feature vector is two-dimensional; each layer has two units.

As depicted in Figure 6.9, in an RNN, the feature vectors from an input example are "read" by the neural network sequentially in the order of the timesteps. The index t denotes a timestep. To update the state $h_{l,u}^t$ at each timestep t in each unit u of each layer l we first calculate a linear combination of the input feature vector with the state vector $\mathbf{h}_{l,u}^{t-1}$ of this same layer from the previous timestep, $t-1$. The linear combination of two vectors is calculated using two parameter vectors $\mathbf{w}_{l,u}$, $\mathbf{u}_{l,u}$ and a parameter $b_{l,u}$. The value of $h_{l,u}^t$ is then obtained by applying activation function g_1 to the result of the linear combination. A typical choice for function g_1 is $tanh$. The output $\mathbf{y}_l^t$ is typically a vector calculated for the whole layer l at once. To obtain $\mathbf{y}_l^t$, we use activation function $\boldsymbol{g}_2$ that takes a vector as input and returns a different vector of the same dimensionality. The function $\boldsymbol{g}_2$ is applied to a linear combination of the state vector values $\mathbf{h}_{l,u}^t$ calculated using a parameter matrix $\mathbf{V}_l$ and a parameter vector $\mathbf{c}_{l,u}$. In classification, a typical choice for $\boldsymbol{g}_2$ is the **softmax function**:

$$\boldsymbol{\sigma}(\mathbf{z}) \stackrel{\text{def}}{=} [\sigma^{(1)}, \ldots, \sigma^{(D)}], \text{ where } \sigma^{(j)} \stackrel{\text{def}}{=} \frac{\exp\left(z^{(j)}\right)}{\sum_{k=1}^{D} \exp\left(z^{(k)}\right)}.$$

The softmax function is a generalization of the sigmoid function to multidimensional outputs. It has the property that $\sum_{j=1}^{D} \sigma^{(j)} = 1$ and $\sigma^{(j)} > 0$ for all j.

The dimensionality of $\mathbf{V}_l$ is chosen by the data analyst such that multiplication of matrix $\mathbf{V}_l$ by the vector $\mathbf{h}_l^t$ results in a vector of the same dimensionality as that of the vector $\mathbf{c}_l$. This choice depends on the dimensionality for the output label $\mathbf{y}$ in your training data. (Until now we only saw one-dimensional labels, but we will see in the future chapters that labels can be multidimensional as well.)

The values of $\mathbf{w}_{l,u}$, $\mathbf{u}_{l,u}$, $b_{l,u}$, $\mathbf{V}_{l,u}$, and $\mathbf{c}_{l,u}$ are computed from the training data using gradient descent with backpropagation. To train RNN models, a special version of backpropagation is used called **backpropagation through time**.

Both *tanh* and *softmax* suffer from the vanishing gradient problem. Even if our RNN has just one or two recurrent layers, because of the sequential nature of the input, backpropagation has to "unfold" the network over time. From the point of view of the gradient calculation, in practice this means that the longer is the input sequence, the deeper is the unfolded network.

Another problem RNNs have is that of handling long-term dependencies. As the length of the input sequence grows, the feature vectors from the beginning of the sequence tend to be "forgotten," because the state of each unit, which serves as network's memory, becomes significantly affected by the feature vectors read more recently. Therefore, in text or speech processing, the cause-effect link between distant words in a long sentence can be lost.

The most effective recurrent neural network models used in practice are **gated RNNs**. These include the **long short-term memory** (LSTM) networks and networks based on the **gated recurrent unit** (GRU).

The beauty of using gated units in RNNs is that such networks can store information in their units for future use, much like bits in a computer's memory. The difference with the real

memory is that reading, writing, and erasure of information stored in each unit is controlled by activation functions that take values in the range $(0, 1)$. The trained neural network can "read" the input sequence of feature vectors and decide at some early time step t to keep specific information about the feature vectors. That information about the earlier feature vectors can later be used by the model to process the feature vectors from near the end of the input sequence. For example, if the input text starts with the word *she*, a language processing RNN model could decide to store the information about the gender to interpret correctly the word *their* seen later in the sentence.

Units make decisions about what information to store, and when to allow reads, writes, and erasures. Those decisions are learned from data and implemented through the concept of *gates*. There are several architectures of gated units. A simple but effective one is called the **minimal gated GRU** and is composed of a memory cell, and a forget gate.

Let's look at the math of a GRU unit on an example of the first layer of the RNN (the one that takes the sequence of feature vectors as input). A minimal gated GRU unit u in layer l takes two inputs: the vector of the memory cell values from all units in the same layer from the previous timestep, $\mathbf{h}_l^{t-1}$, and a feature vector $\mathbf{x}^t$. It then uses these two vectors as follows (all operations in the below sequence are executed in the unit one after another):

$$
\begin{aligned}
\tilde{h}_{l,u}^t &\leftarrow g_1(\mathbf{w}_{l,u}\mathbf{x}^t + \mathbf{u}_{l,u}\mathbf{h}_l^{t-1} + b_{l,u}),\\
\Gamma_{l,u}^t &\leftarrow g_2(\mathbf{m}_{l,u}\mathbf{x}^t + \mathbf{o}_{l,u}\mathbf{h}^{t-1} + a_{l,u}),\\
h_{l,u}^t &\leftarrow \Gamma_{l,u}^t \tilde{h}_l^t + (1 - \Gamma_{l,u}^t) h_l^{t-1},\\
\mathbf{h}_l^t &\leftarrow [h_{l,1}^t, \ldots, h_{l,size_l}^t]\\
\mathbf{y}_l^t &\leftarrow \boldsymbol{g}_3(\mathbf{V}_l\mathbf{h}_l^t + \mathbf{c}_{l,u}),
\end{aligned}
$$

where g_1 is the *tanh* activation function, g_2 is called the gate function and is implemented as the sigmoid function taking values in the range $(0, 1)$. If the gate $\Gamma_{l,u}$ is close to 0, then the memory cell keeps its value from the previous time step, h_l^{t-1}. On the other hand, if the gate $\Gamma_{l,u}$ is close to 1, the value of the memory cell is overwritten by a new value $\tilde{h}_{l,u}^t$ (see the third assignment from the top). Just like in standard RNNs, $\boldsymbol{g}_3$ is usually softmax.

A gated unit takes an input and stores it for some time. This is equivalent to applying the identity function ($f(x) = x$) to the input. Because the derivative of the identity function is constant, when a network with gated units is trained with backpropagation through time, the gradient does not vanish.

Other important extensions to RNNs include **bi-directional RNNs**, RNNs with **attention** and **sequence-to-sequence RNN** models. The latter, in particular, are frequently used to build neural machine translation models and other models for text to text transformations. A generalization of an RNN is a **recursive neural network**.

Appendices

Appendix A

Backpropagation

Backpropagation is a very simple algorithm, though it's often seen as complex to understand in detail. Intuitively, you feel that the name of the algorithm somehow implies that the information on the prediction error made by the output layer of the neural network propagates back to the input layer by updating weights in all units, but how exactly it works, let's see in this chapter.

My description of backpropagation is inspired by Michael Nielsen's description of that algorithm in his excellent "Neural Networks and Deep Learning" book.

A.1 Prelimiaries

Consider the simple feedforward neural network in Figure A.1. It has one hidden layer with four units, the input is three dimensional and there are two outputs.

For simplicity, I changed the way to depict the network and its parameters. First of all, the outputs of the units are denoted as $a_l^{(u)}$, where l is layer index and u is unit index. The character a stands for "activation". More precisely,

$$\begin{aligned} a_l^{(u)} &\stackrel{\text{def}}{=} \boldsymbol{\sigma}\left(\mathbf{w}_{l,u}\mathbf{a}_{l-1} + b_{l,u}\right) \\ &= \boldsymbol{\sigma}\left(\sum_j w_{l,u}^{(j)} a_{l-1}^{(j)} + b_{l,u}\right) \\ &= \boldsymbol{\sigma}\left(\sum_j a_{l-1}^{(j)} w_{l,u}^{(j)} + b_{l,u}\right), \end{aligned}$$

where $a_l^{(u)}$ is the activation (output) of a neuron u in layer l, and j is the index of units in the layer $l-1$.

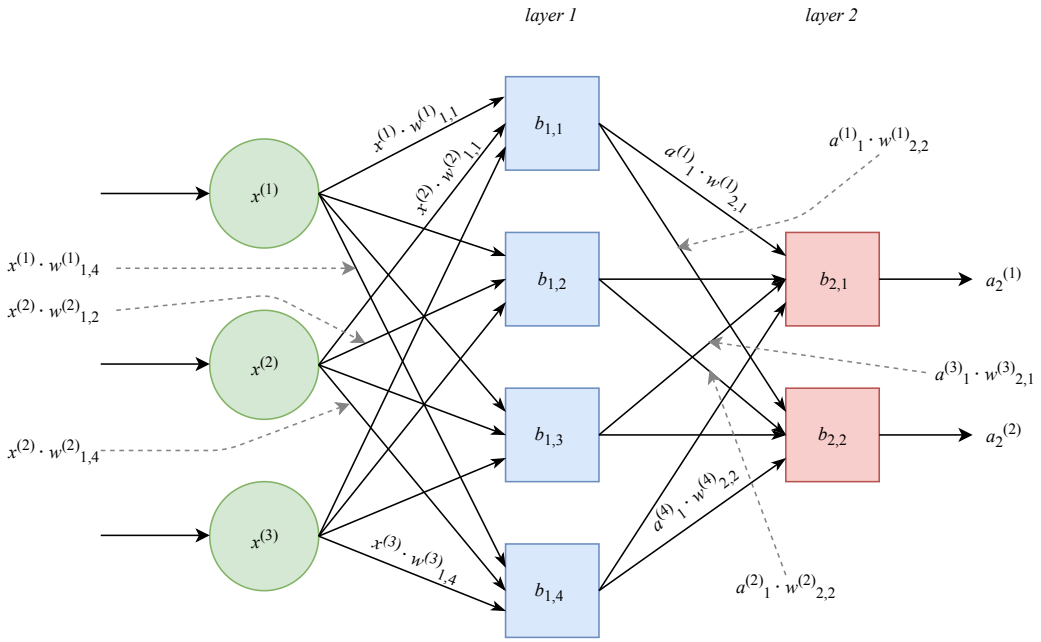


Figure A.1: Simple FFNN with one hidden layer.

Now you see why I labeled the arrows in Figure A.1 with $a_{l-1}^{(j)} w_{l,u}^{(j)}$. The arrow represents two parameters: the output of a unit from the previous layer and the weight applied to this output in a neuron in the current layer.

Let's denote by $z_l^{(u)}$ the term between parentheses:

$$z_l^{(u)} \stackrel{\text{def}}{=} \sum_j w_{l,u}^{(j)} a_{l-1}^{(j)} + b_{l,u}.$$

From now onwards, I will use the form $w_{l,u}^{(j)} a_{l-1}^{(j)}$ and not $a_{l-1}^{(j)} w_{l,u}^{(j)}$ by common convention.

For the backpropagation algorithm to work, two assumptions must be satisfied. The first assumption is that the cost function is an average of the loss functions applied to individual examples.

The second assumption is that the loss function is a differentiable function of the activations.

To illustrate backprapagation using our simple network example, let's define the cost function C as follows:

$$C \stackrel{\text{def}}{=} \frac{1}{N} \sum_{\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}} \frac{1}{2} \sum_{j \in \{1,2\}} \left(y_i^{(j)} - a_L^{(j)} \right)^2, \tag{A.1}$$

where we have $j \in \{1, 2\}$ and $\frac{1}{2}$ because the output layer L of our simple network has two units. The cost C defined as eq. A.1 is a typical cost function based on the squared loss used in regression.

To make it even more simple, let's assume that we only use one example $(\mathbf{x}, \mathbf{y})$ to train the network. (I will show later how using more than one example affects the computation.) The cost function becomes:

$$C \stackrel{\text{def}}{=} \frac{1}{2} \sum_{j \in \{1,2\}} \left(y^{(j)} - a_L^{(j)} \right)^2 = \sum_{j \in \{1,2\}} \frac{1}{2} \left(y^{(j)} - a_L^{(j)} \right)^2. \tag{A.2}$$

Assuming that $\mathbf{x}$ and $\mathbf{y}$ are fixed (we know their values and cannot modify them), the cost C is a function of activations $a_L^{(j)}$.

Just like, in Chapter 4, we looked for partial derivatives of the parameters to solve linear regression using gradient descent, we will find the partial derivatives of C with respect to all parameters of the neural network, that is all $w_{\cdot,\cdot}^{(\cdot)}$ and $b_{\cdot,\cdot}$ (where $\cdot$ replaces any possibble index in our neural network). Then, we will use gradient descent with these partial derivartives to update parameters of the neural network to minimize cost C, just like we did in linear regression in Chapter 4.

A.2 The Four Basic Equations

First of all, define,

$$\delta_{l,j} \stackrel{\text{def}}{=} \frac{\partial C}{\partial z_l^{(j)}} \tag{A.3}$$

as the partrial derivative of the cost function with respect to the function $z_l^{(j)}$.

To some readers it may sound unusual to define a derivative with respect to a function but, if you remember the **chain rule** that we considered in Chapter 2, all will become clear. Recall that if we have function $f(x)$ defined as $f(g(x))$, where g is another function, then, to find the derivative of f with respect to x, according to chain rule we write,

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}.$$

As you can see, $\frac{\partial f}{\partial g}$ is a derivative of a function with respect to another function.

The backpropagation will allow us to compute $\delta_{l,j}$ for every layer l and every j. Once we have them, we we will be able to use them to calculate $\frac{\partial C}{\partial w_{l,u}^{(j)}}$ and $\frac{\partial C}{\partial b_{l,u}}$ for all j, l, u.

Backpropagation starts with the output layer L:

$$\delta_{L,1} \stackrel{\text{def}}{=} \frac{\partial C}{\partial z_L^{(1)}} = \sum_{k \in \{1,2\}} \frac{\partial C}{\partial a_L^{(k)}} \frac{\partial a_L^{(k)}}{\partial z_L^{(k)}} = \frac{\partial C}{\partial a_L^{(1)}} \frac{\partial a_L^{(1)}}{\partial z_L^{(1)}} + \frac{\partial C}{\partial a_L^{(2)}} \frac{\partial a_L^{(2)}}{\partial z_L^{(1)}} = \frac{\partial C}{\partial a_L^{(1)}} \frac{\partial a_L^{(1)}}{\partial z_L^{(1)}}. \tag{A.4}$$

In the above equation, the summation term comes from the cost function equation eq. A.2. (Recall from calculus that the derivative of a sum is a sum of derivatives.) The term $\frac{\partial C}{\partial a_L^{(2)}} \frac{\partial a_L^{(2)}}{\partial z_L^{(1)}}$ disappeared because $\frac{\partial a_L^{(2)}}{\partial z_L^{(1)}} = 0$. Indeed, $a^{(2)}$ doesn't depend on $z_L^{(1)}$ and, thus, the derivative of $a^{(2)}$ with respect to $z_L^{(1)}$ is zero.

The derivation for $\delta_{L,2}$ can be done similarly, and, more generally, we can write,

$$\delta_{L,j} = \frac{\partial C}{\partial a_L^{(j)}} \frac{\partial a_L^{(j)}}{\partial z_L^{(j)}}. \tag{A.5}$$

For example, for our choice of the cost function given by eq. A.2, $\delta_{L,1} = \frac{1}{2} 2 \left(y^{(1)} - a_L^{(1)}\right) \sigma' \left(z_L^{(1)}\right) = \left(y^{(1)} - \sigma\left(z_L^{(1)}\right)\right) \sigma' \left(z_L^{(1)}\right)$. If, for example, σ is the logistic function $\sigma(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}}$, then $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. In this case, $\delta_{L,1} = \left(y^{(1)} - \sigma\left(z_L^{(1)}\right)\right) \sigma\left(z_L^{(1)}\right) \left(1 - \sigma\left(z_L^{(1)}\right)\right)$. The value of $z_L^{(1)}$ can be calculated from the network graph.

From the above, you can see that we can easily calculate $\delta_{L,j}$ for every unit j in the output layer of the network.

Now, let's find the value of $\delta_{l,j}$ where $l \neq L$. Because $z_{l+1}^{(\cdot)}$ is a function of $z_l^{(\cdot)}$, by using the chain rule we can write,

$$\delta_{l,j} \stackrel{\text{def}}{=} \frac{\partial C}{\partial z_l^{(j)}} = \sum_{k=1,\ldots,size_{l+1}} \frac{\partial C}{\partial z_{l+1}^{(k)}} \frac{\partial z_{l+1}^{(k)}}{\partial z_l^{(j)}} = \sum_{k=1,\ldots,size_{l+1}} \delta_{l+1,k} \frac{\partial z_{l+1}^{(k)}}{\partial z_l^{(j)}}, \tag{A.6}$$

where we substituted $\frac{\partial C}{\partial z_{l+1}^{(k)}}$ by $\delta_{l+1,k}$ by definition of $\delta_{l+1,k}$ (eq. A.6).

The expression $\frac{\partial z_{l+1}^{(k)}}{\partial z_l^{(j)}}$ can be found as follows. First of all, by definition,

$$z_{l+1}^{(k)} \stackrel{\text{def}}{=} \sum_{r=1,\ldots,j,\ldots,size_l} \left[w_{l+1,k}^{(r)} a_l^{(r)} + b_{l,k} \right].$$

By differentiating $z_{l+1}^{(k)}$ with respect to $z_l^{(j)}$ we obtain,

$$\begin{aligned}
\frac{\partial z_{l+1}^{(k)}}{\partial z_l^{(j)}} &= w_{l+1,k}^{(1)} \frac{\partial a_l^{(1)}}{\partial z_l^{(j)}} + \frac{\partial b_{l,k}}{\partial z_l^{(j)}} + \ldots + w_{l+1,k}^{(j)} \frac{\partial a_l^{(j)}}{\partial z_l^{(j)}} + \frac{\partial b_{l,k}}{\partial z_l^{(j)}} + \ldots + w_{l,k}^{(size_l)} \frac{\partial a_l^{(j)}}{\partial z_l^{(j)}} + \frac{\partial b_{l,k}}{\partial z_l^{(j)}} \\
&= 0 + 0 + \ldots + w_{l+1,k}^{(j)} \frac{\partial a_l^{(j)}}{\partial z_l^{(j)}} + 0 + \ldots + 0 + 0 \\
&= w_{l+1,k}^{(j)} \frac{\partial a_l^{(j)}}{\partial z_l^{(j)}}
\end{aligned}$$

In the above equation, all terms in the summation but one equal zero for the same reason as it happened in [eq:zero-derivative].

By plugging the last term of the above equation into eq. A.6, we obtain,

$$\delta_{l,j} = \sum_{k=1,\ldots,size_{l+1}} \delta_{l+1,k} \frac{\partial z_{l+1}^{(k)}}{\partial z_l^{(j)}} = \sum_{k=1,\ldots,size_{l+1}} \delta_{l+1,k} w_{l+1,k}^{(j)} \frac{\partial a_l^{(j)}}{\partial z_l^{(j)}}. \tag{A.7}$$

Again, $\delta_{l,j}$ is simple to calculate: $\delta_{l+1,k}$ is known because we already know $\delta_{L,k}$ when we calculate $\delta_{L-1,j}$; we know $w_{l+1,k}^{(j)}$ because it's the current value of the parameter in the gradient descent, and, finally, $\frac{\partial a_l^{(j)}}{\partial z_l^{(j)}}$ is simply $\sigma'(z_l^{(j)})$ and depends on the choice of σ (can be logistic function or some other differentiable function).

Now we know how to compute $\delta_{L,j}$ and $\delta_{l,j}$, but the quantities that really interest us are $\frac{\partial C}{\partial w_{l,k}^{(j)}}$ and $\frac{\partial C}{\partial b_{l,k}}$, because we will use them to update $w_{l,k}^{(j)}$ and $b_{l,k}$ using the **gradient descent** algorithm (or, rather, the **stochastic gradient descent**). Let's derive them.

By chain rule,

$$
\begin{aligned}
\frac{\partial C}{\partial w_{l,k}^{(j)}} &= \frac{\partial C}{\partial z_l^{(k)}} \frac{\partial z_l^{(k)}}{\partial w_{l,k}^{(j)}} \\
&= \delta_{l,k} \frac{\partial z_l^{(k)}}{\partial w_{l,k}^{(j)}} \\
&= \delta_{l,k} \left(\frac{\partial w_{l,k}^{(1)} a_{l-1}^{(1)}}{\partial w_{l,k}^{(j)}} + \ldots + \frac{\partial w_{l,k}^{(j)} a_{l-1}^{(j)}}{\partial w_{l,k}^{(j)}} + \ldots + \frac{\partial w_{l,k}^{(size_{l-1})} a_{l-1}^{(size_{l-1})}}{\partial w_{l,k}^{(j)}} + \frac{\partial b_{l,k}}{\partial w_{l,k}^{(j)}} \right) \\
&= \delta_{l,k} \left(0 + \ldots + \frac{\partial w_{l,k}^{(j)} a_{l-1}^{(j)}}{\partial w_{l,k}^{(j)}} + \ldots + 0 + 0 \right) \\
&= \delta_{l,k} a_{l-1}^{(j)}.
\end{aligned}
\tag{A.8}
$$

Similarly,

$$
\begin{aligned}
\frac{\partial C}{\partial b_{l,k}} &= \frac{\partial C}{\partial z_l^{(k)}} \frac{\partial z_l^{(k)}}{\partial b_{l,k}} \\
&= \delta_{l,k} \frac{\partial z_l^{(k)}}{\partial b_{l,k}} \\
&= \delta_{l,k} \left(\frac{\partial w_{l,k}^{(1)} a_{l-1}^{(1)}}{\partial b_{l,k}} + \ldots + \frac{\partial w_{l,k}^{(j)} a_{l-1}^{(j)}}{\partial b_{l,k}} + \ldots + \frac{\partial w_{l,k}^{(size_{l-1})} a_{l-1}^{(size_{l-1})}}{\partial b_{l,k}} + \frac{\partial b_{l,k}}{\partial b_{l,k}} \right) \\
&= \delta_{l,k} \left(0 + \ldots + 0 + \ldots + 0 + 1 \right) \\
&= \delta_{l,k}.
\end{aligned}
\tag{A.9}
$$

That's it. Now we have all the four equations of the backpropagation algorithm: eq. A.5, eq. A.7, eq. A.8, and eq. A.9. We are now ready to write backpropagation in form of an algorithm.

A.3 The Backpropagation Algorithm

Initialization:

For all l, k, j, set $w_{l,k}^{(j)}$ to small random values close to 0 (typically in the interval $[0.0, 0.1]$). Set $b_{l,k} \leftarrow 0$ for all l, k.

Input: $(\mathbf{x}, \mathbf{y})$

Forward pass:

For each unit k in layer 1, set,

$$a_1^{(k)} \leftarrow \sigma\left(\sum_{j=1,\ldots,D} w_{1,k}^{(j)} x^{(j)} + b_{1,k}\right).$$

Let L be the index of the output layer of the neural network. For layers l from 2 to L and for each unit k in l, set,

$$a_l^{(k)} \leftarrow \sigma\left(\sum_{j=1,\ldots,size_{l-1}} w_{l,k}^{(j)} a_{l-1}^{(j)} + b_{l,k}\right).$$

Backward pass:

For each unit j in L, set,

$$\delta_{L,j} \leftarrow \frac{\partial C}{\partial a_L^{(j)}} \frac{\partial a_L^{(j)}}{\partial z_L^{(j)}}.$$

For each layer $l = L-1, \ldots, 1$ and each unit j in l, set,

$$\delta_{l,j} = \sum_{k=1,\ldots,size_{l+1}} \delta_{l+1,k} w_{l+1,k}^{(j)} \frac{\partial a_l^{(j)}}{\partial z_l^{(j)}}.$$

For $l = L, \ldots, 2$, each unit k in l, and each unit j in $l-1$ set,

$$\frac{\partial C}{\partial w_{l,k}^{(j)}} \leftarrow \delta_{l,k} a_{l-1}^{(j)},$$
$$\frac{\partial C}{\partial b_{l,k}} \leftarrow \delta_{l,k}.$$

For layer $l = 1$, each unit k in layer 1, and each feature j in $\mathbf{x}$ set,

$$\frac{\partial C}{\partial w_{1,k}^{(j)}} \leftarrow \delta_{l,k} x^{(j)},$$
$$\frac{\partial C}{\partial b_{1,k}} \leftarrow \delta_{1,k}.$$

Output: $\frac{\partial C}{\partial w_{l,k}^{(j)}}$, $\frac{\partial C}{\partial b_{l,k}}$, for all l, k, j.

The above algorithm returns the partial derivatives of all parameters of the neural network. Now, as we have the partial derivatives for one example, we can use the stochastic gradient descent algorithm to minimize the cost function and find the optimal values of parameters.

It is straightforward to transform the above algorithm for the **minibatch stochastic gradient descent**. Remember the first assumption about the cost function: it's an average of individual losses for each training example. Also remember from calculus that the gradient of an average of functions is the average of gradients of individual functions.

Let our minibatch be of size n and denote as $\left(\frac{\partial C}{\partial w_{l,k}^{(j)}}\right)_i$ and $\left(\frac{\partial C}{\partial b_{l,k}}\right)_i$ the partial derivatives computed for the training example $(\mathbf{x}_i, \mathbf{y}_i)$, for $i = 1, \ldots, n$. Then, an update during one iteration of the minibatch stochastic gradient descent will look as follows:

$$w_{l,k}^{(j)} \leftarrow w_{l,k}^{(j)} - \frac{\alpha}{n} \sum_{i=1,\ldots,n} \left(\frac{\partial C}{\partial w_{l,k}^{(j)}}\right)_i,$$
$$b_{l,k} \leftarrow b_{l,k} - \frac{\alpha}{n} \sum_{i=1,\ldots,n} \left(\frac{\partial C}{\partial b_{l,k}}\right)_i,$$

for all l, k, j, where α is the learning rate.

Appendix B

RNN Unfolding

As I mentioned in Chapter 6, even if our RNN has just one or two recurrent layers, because of the sequential nature of the input, backpropagation has to "unfold" the network over time. From the point of view of the gradient calculation, in practice this means that the longer is the input sequence, the deeper is the unfolded network.

Let's see how it happens. To begin, let's repeat below the two-layer RNN from Figure 6.9 in a simplified form.

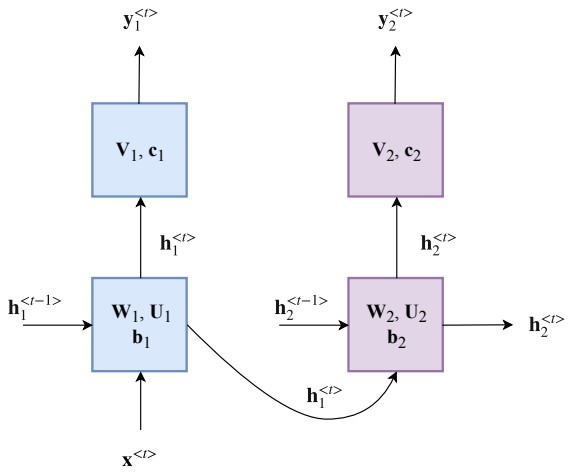


Figure B.1: The RNN from Figure 6.9 in a simplified form.

In the above figure, we see the two-layer RNN from Figure 6.9, in which, for simplicity we collapsed the two units of each layer into one box and removed equations. However, it's

exactly the same two-layer network, without modifications.

Now, assume that this network recieves as input a sequence $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3]$ of length 3. The **unfolding** consists of transforming the original two-layer network into a deeper network, by "copying" layers of the original network threee times and connecting copies so that the new network can process the input sequence of length 3.

In Figure B.2, you can see the unfolded network from Figure B.1 for input of length 3.

As you could notice, if the input sequences are of different length, the unfolded network for each sequence length is different. However, each "copy" of the original layers in the unfolded network shares parameters with other "copies". So, the network becomes deeper but the number of parameters in the unfolded network remains the same as in the original network, indpendently of the length of the input.

In practice, however, we often fix the maximum length of the input sequence in advance. We then either cut the input sequence at this length, or add padding in cases when the input sequence is shorter than the maximum length.

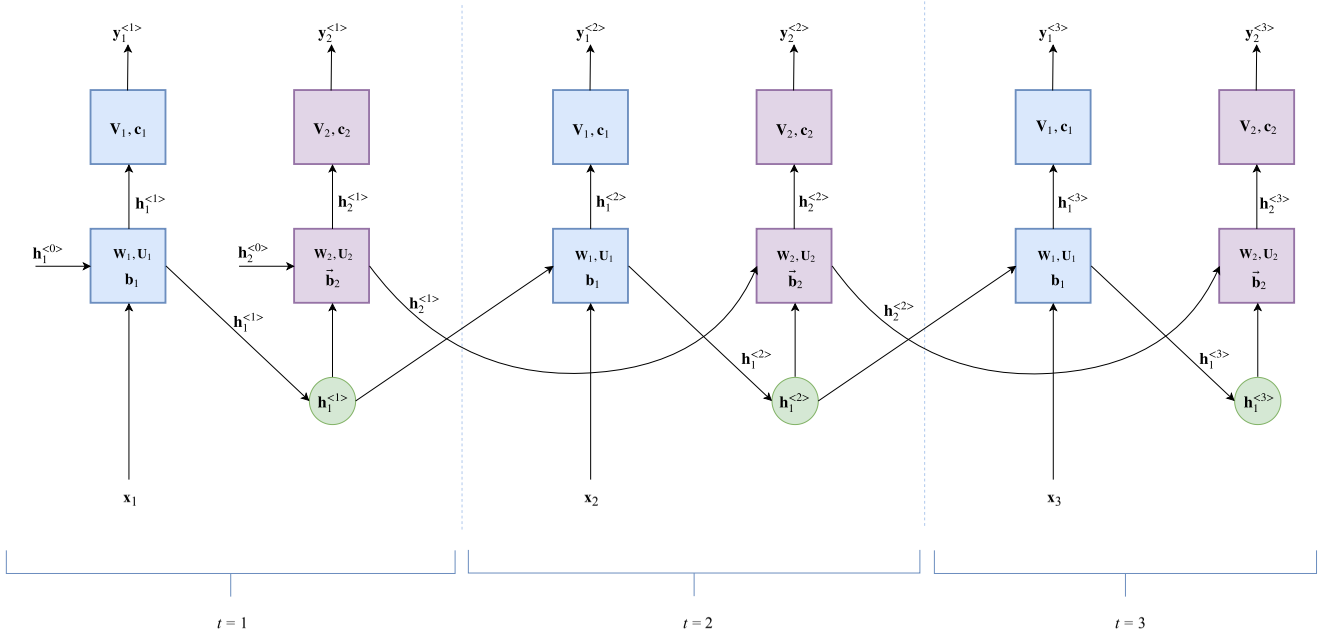


Figure B.2: The unfolded network from Figure B.1 for input of length 3.

Appendix C

Bidirectional RNN

In a bidirectional RNN the input is read twice. First from left to right and then from right to left. The outputs are generated on the second pass (from right to left).

The process is shown in C.1 where a two-layer bidirectional RNN is shown unfolded for an input of length 3. As you can see, during the first pass (left to right) only $\vec{\mathbf{h}}_t^l$ are generated for each input $\mathbf{x}^t$ in the sequence. To generate the outputs $\mathbf{y}_l^t$, both left-to-right pass states $\vec{\mathbf{h}}_t^l$ and right-to-left pass states $\overleftarrow{\mathbf{h}}_t^l$ are needed.

Usually, the value of $\overleftarrow{\mathbf{h}}_2^3$ is simply a copy of $\vec{\mathbf{h}}_2^3$. The prediction $\mathbf{y}_l^t$ is typically obtained as $\vec{\mathbf{V}}_l \left[\vec{\mathbf{h}}_l^t, \overleftarrow{\mathbf{h}}_l^t\right] + \vec{\mathbf{c}}_l$, where $\left[\vec{\mathbf{h}}_l^t, \overleftarrow{\mathbf{h}}_l^t\right]$ is a **concatenation** of two vectors.

The concatenation of two vectors is simply illustrated on an example. Let's have two vectors $\mathbf{a} \stackrel{\text{def}}{=} \left[a^{(1)}, a^{(2)}, a^{(3)}\right]$ and $\mathbf{b} \stackrel{\text{def}}{=} \left[b^{(1)}, b^{(2)}, b^{(3)}, b^{(4)}\right]$. The concatination $[\mathbf{a}, \mathbf{b}]$ is defined as,

$$[\mathbf{a}, \mathbf{b}] \stackrel{\text{def}}{=} \left[a^{(1)}, a^{(2)}, a^{(3)}, b^{(1)}, b^{(2)}, b^{(3)}, b^{(4)}\right].$$

For concationation operartion to work, vectors don't have to be of the same dimensionality, however usually $\vec{\mathbf{h}}_l^t$ and $\overleftarrow{\mathbf{h}}_l^t$ are of the same dimensionality. It is especially convenient because, as I said above, usually, the value of $\overleftarrow{\mathbf{h}}_2^3$ is simply a copy of $\vec{\mathbf{h}}_2^3$.

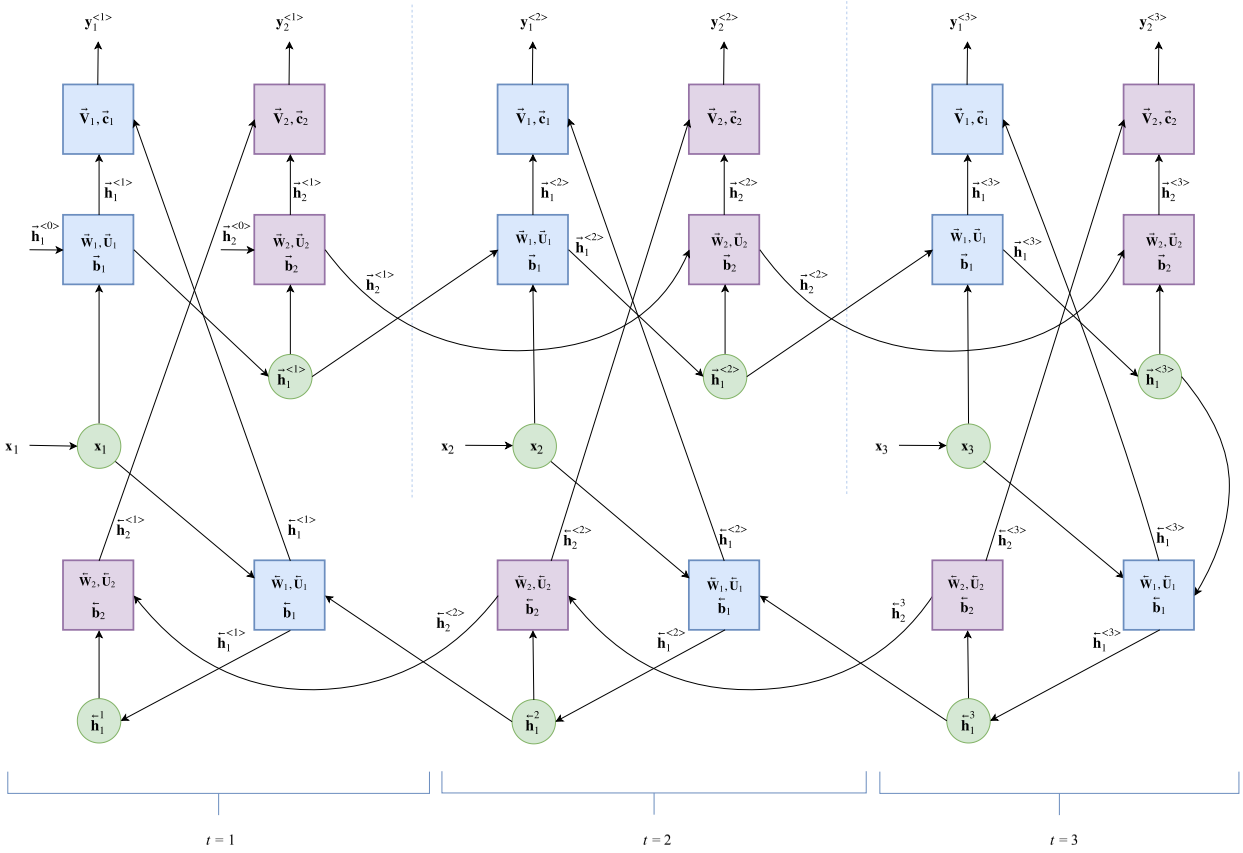


Figure C.1: The unfolded bidirectional network from Figure B.1 for input of length 3.

Appendix D

Attention in RNN

Attention is usually seen as upgrade of sequence to sequence neural network models such as machine translation models, though it can also be used in image-to-text models and other models where some input is embedded first and then a sequential prediction is made. Recall the bidirectional RNN from Figure C.1 and let simplify it by removing some details as well as the units that make predictions. You can see this simplified illustration in Figure D.1.

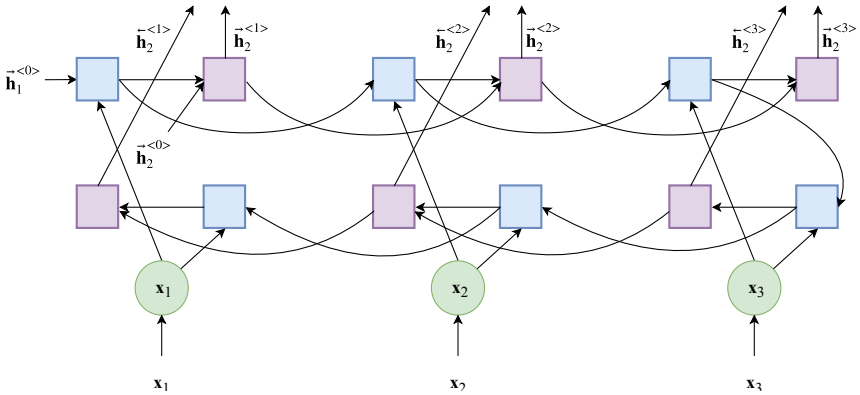


Figure D.1: The simplified representation of the unfolded bidirectional network from Figure C.1. Most parameters are not shown but they are implicite. The output units are removed on purpose: they aren't needed in the encoder.

In our sequence to sequence neural network, the input sequnce $[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3]$ is read by the encoder shown in the D.1 first. To apply attention during the decoding phase, we only need the values of $\overleftarrow{\mathbf{h}}_2^{<\cdot>}, \vec{\mathbf{h}}_2^{<\cdot>}$, that is the states of the right-most layer of the encoder for the forward and backward direction of our bidirectional RNN encoder.

In Figure D.2 I repeat the same encoder as the one shown in Figure D.1, but now it contains the decoder part with attention.

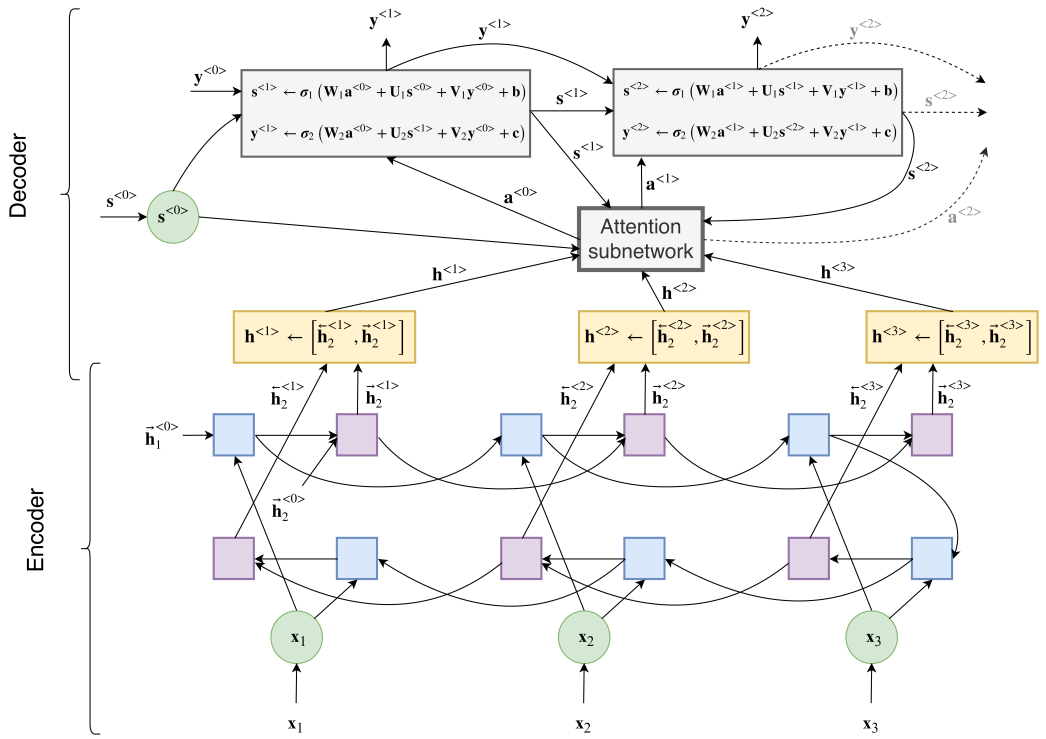


Figure D.2: A sequence to sequence RNN model with attention.

As you can see in Figure D.2, the state vectors $\overleftarrow{\mathbf{h}}_2^{<\cdot>}, \vec{\mathbf{h}}_2^{<\cdot>}$ are concatinated to obtain one state vector $\mathbf{h}_2^{<\cdot>}$ per time step.

On the top of Figure D.2 is a typical decoder: it starts with an empty input vector $\mathbf{y}^{<0>}$ and some intial state $\mathbf{s}^{<0>}$ and iteratively generates a sequence of outputs $\mathbf{y}^{<1>}, \mathbf{y}^{<2>}, \ldots$, and states $\mathbf{s}^{<1>}, \mathbf{s}^{<2>}, \ldots$. In the sequence-to-sequence architecture without attention $\mathbf{s}^{<0>}$ is typically $\mathbf{h}^{<\cdot>}$, the last state of right-most layer of the decoder. When we add attention, then $\mathbf{s}^{<0>}$ can be an empty vector as well (when I say empty, I mean filled with zeroes but it also can be a random vector).

In the heart of model with attention lies the **attention subnetwork**. As you can see in Figure D.2, the attention subnetwork takes all three $\mathbf{h}^{<\cdot>}$ as input. To enable decoding of $\mathbf{y}^{<t>}$, the output at time step t, the attention subnetwork also takes as input $\mathbf{s}^{<t-1>}$, the decoder state vector from the previsous time step. At each decoding time step t, the attention subnetwork

returns $\mathbf{a}^{<t>}$, a vector that aggregates the $\mathbf{h}^{<1>}, \mathbf{h}^{<2>}$ and $\mathbf{h}^{<3>}$ in a specific way. Below I explain how it does that. I will stick to out example with three encoding steps, but it can be generalized to any input sequence length.

To obtain the attention vector $\mathbf{a}^{<t>}$, the attention subnetwork aggregates the $\mathbf{h}^{<1>}, \mathbf{h}^{<2>}$ and $\mathbf{h}^{<3>}$ in as follows:

$$\mathbf{a}^{<t>} \leftarrow \alpha^{<t,1>}\mathbf{h}_1 + \alpha^{<t,2>}\mathbf{h}_1 + \alpha^{<t,3>}\mathbf{h}_3,$$

where $\alpha^{<t,\cdot>}$ can be interpreted as the amount of attention the network has pay to $\mathbf{h}^{(\cdot)}$ when predicting $\mathbf{y}^t$, and $\alpha^{<t,1>} + \alpha^{<t,2>} + \alpha^{<t,3>} = 1$.

The weigths $\alpha^{(\cdot),(\cdot)}$ are trainable. To make them sum to one, the **softmax** function used:

$$\begin{aligned}
\alpha^{t,1} &\leftarrow \frac{\exp(q^{<t,1>})}{\sum_{t'}^{3}\exp(q^{<t,t'>})}, \\
\alpha^{t,2} &\leftarrow \frac{\exp(q^{<t,2>})}{\sum_{t'}^{3}\exp(q^{<t,t'>})}, \\
\alpha^{t,3} &\leftarrow \frac{\exp(q^{<t,2>})}{\sum_{t'}^{3}\exp(q^{<t,t'>})}.
\end{aligned}$$

The values $q^{<t,\cdot>}$ are given by a small subnetwork (usually it's a shallow FFNN with one hidden layer) that takes as input a pair $(\mathbf{s}^{<t-1>}, \mathbf{h}^{<t'>})$ and predicts a real number $q^{<t,t'>}$. This small subnetwork is a part of the bigger network inf Figure D.2 and both small and big network are trained simultaneously by backpropagation. The model learned by that small subnetwork is called an **alignment model** because it learns how well a specific state from the encoder is aligned with the current decoder state.