The Hundred-Page Machine Learning Book

Andriy Burkov

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

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

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 will 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} will look like this:

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

In the above equation, $\boldsymbol{f}_2$, $\boldsymbol{f}_3$ are vector functions of the following form:

$$\boldsymbol{f}_l(\mathbf{z}) \stackrel{\text{def}}{=} \boldsymbol{g}_l(\mathbf{W}_l\mathbf{z} + \mathbf{b}_l), \tag{1}$$

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 certain cost function (such as MSE). Compare eq. 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_1 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 $\mathbf{w}_l$. This is because $\boldsymbol{g}_l$ 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 **vanilla neural network**.

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

6.1.1 Multilayer Perceptron Example

We will have a closer look at one particular configuration of neural networks called **feed-forward neural networks** (FFNN) and more specifically the architecture called a **multi-layer perceptron** (MLP). As an illustration, we consider an MLP with three layers. Our network will take a two-dimensional feature vector as input and output 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 fig. 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 circle or a 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 fig. 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 strictly necessary. Each layer can have a different number of units. Each unit has its own 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 $[x^{(1)}, \ldots, x^{(D)}]$.

As you can see in fig. 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 preceding 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.

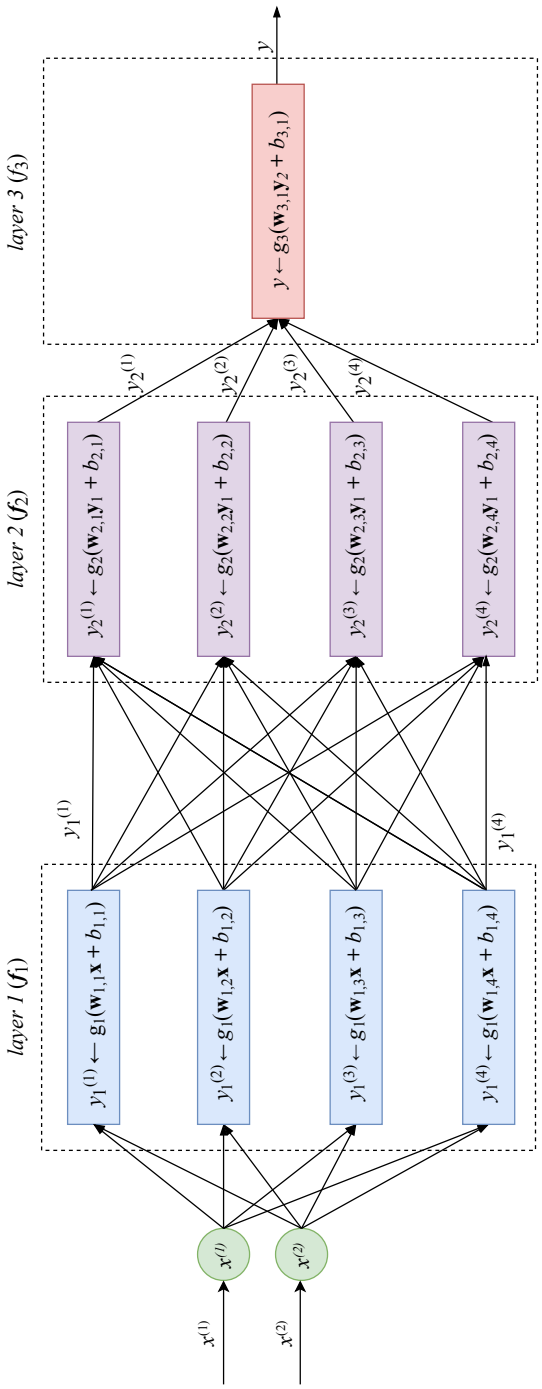


Figure 1: A multilayer perceptron with two-dimensional input, two layers with four units and one output layer with one unit.

The data analyst is free to choose any mathematical function as $g_{l,u}$, assuming it's differentiable[2]. The latter property is important for gradient descent, which is used to find the values of the parameters $\mathbf{w}_{l,u}$ and $b_{l,u}$ for all l and u. The main 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. This is because $\mathbf{W}_l\mathbf{z} + \mathbf{b}_l$ is a linear function and a linear function of a linear function is also a linear function.

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 we 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. This 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.

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 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

[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.

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 we mentioned above, have gradients in the range $(0, 1)$, and backpropagation computes gradients by the chain rule. This 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. This 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). The ReLU activation function suffers much less from the problem of vanishing gradient. Also, **long short-term memory** (LSTM) networks, which we will consider below, as well as such techniques as **skip connections** used in **residual neural networks** allow you to train even deeper neural networks, with thousands of layers.

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 $size_l(size_{l-1} + 1)$ parameters (our matrix $\mathbf{W}_l$ plus the vector $\mathbf{b}_l$). This 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. 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 will 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, etc. The exception from the rule are the edges: the parts of an image where two different objects "touch" one another.

So, if we can train the neural network to recognize regions of the same information as well as the edges, then this knowledge would allow the neural network to predict the object on the picture. 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 very likely that there'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[3]. 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 sky, another one will detect grass, the third one will detect edges of a building, etc.

In CNNs, a small regression model looks like the one in fig. 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 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 back 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 P (for "patch"):

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

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 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 dot-product between matrices P and F, and then sum all values from the resulting vector, the value we obtain will be higher the more similar F is to P. For instance, assume that F looks like this:

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

Then,

$$P \cdot F = [0 \cdot 0 + 2 \cdot 1 + 0 \cdot 0, 2 \cdot 1 + 4 \cdot 1 + 3 \cdot 1, 3 \cdot 0 + 1 \cdot 1 + 0 \cdot 1] = [2, 9, 1].$$

[3]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*.

Then the sum of all elements of the above vector will be $2 + 9 + 1 = 12$. This operation — the dot product between a patch and a filter and then summing the values — is called **convolution**.

If our input patch P had a different patten, for example that of a letter T,

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

then the convolution would give a lower result: $0 + 9 + 0 = 9$. So, you can see the more the patch "looks" like the filter, the higher the value of the convolution operation will be. For convenience, there's also a bias parameter b associated with each filter F which is added to the result of a convolution before applying the nonlinearity.

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 a convolution is computed at each iteration.

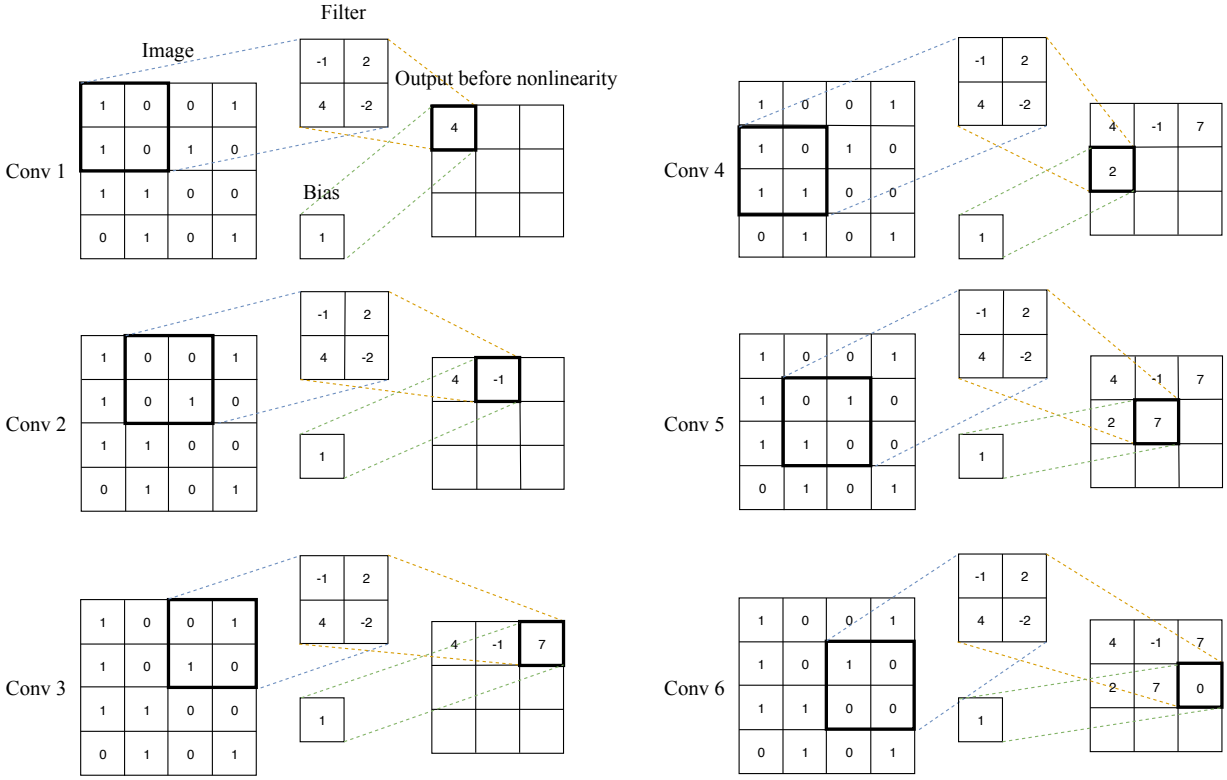


Figure 2: A filter convolving across an image.

An illustration of the process is given in fig. 2 where 6 steps of a filter convolving across an image are shown.

The numbers in the filter matrix, for each filter F in each layer, as well as the value of the bias term b, are found by the gradient descent with backpropagation, based on data by minimizing the cost function.

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.

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*. 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.

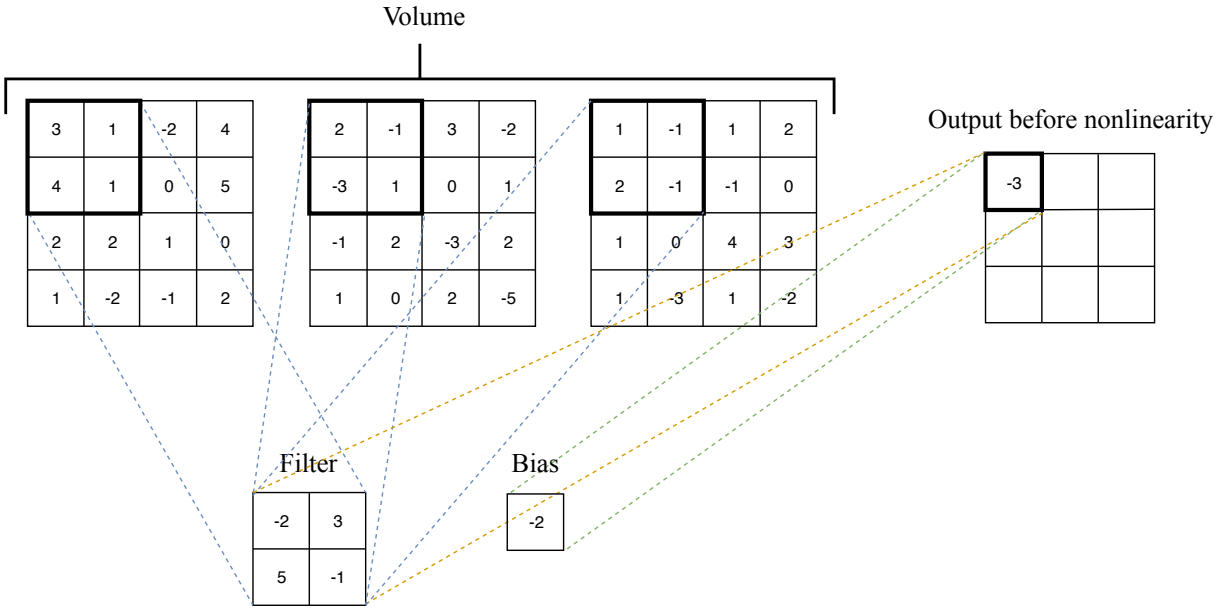
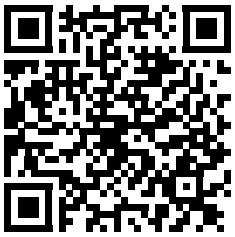


Figure 3: Convolution of a volume consiting of three matrices.

An example of a convolution of a patch of a volume consisting of three matrices is shown in fig. 3. 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.



By now, you should have a good high-level understanding of the CNN architecture. We didn't discuss some important features of CNNs though, such as strides, padding and pooling. Strides and padding are two important hyperparameters of the convolution filter and the sliding window, while pooling is a technique that works very well in practice by reducing the number of parameters of a CNN even more.

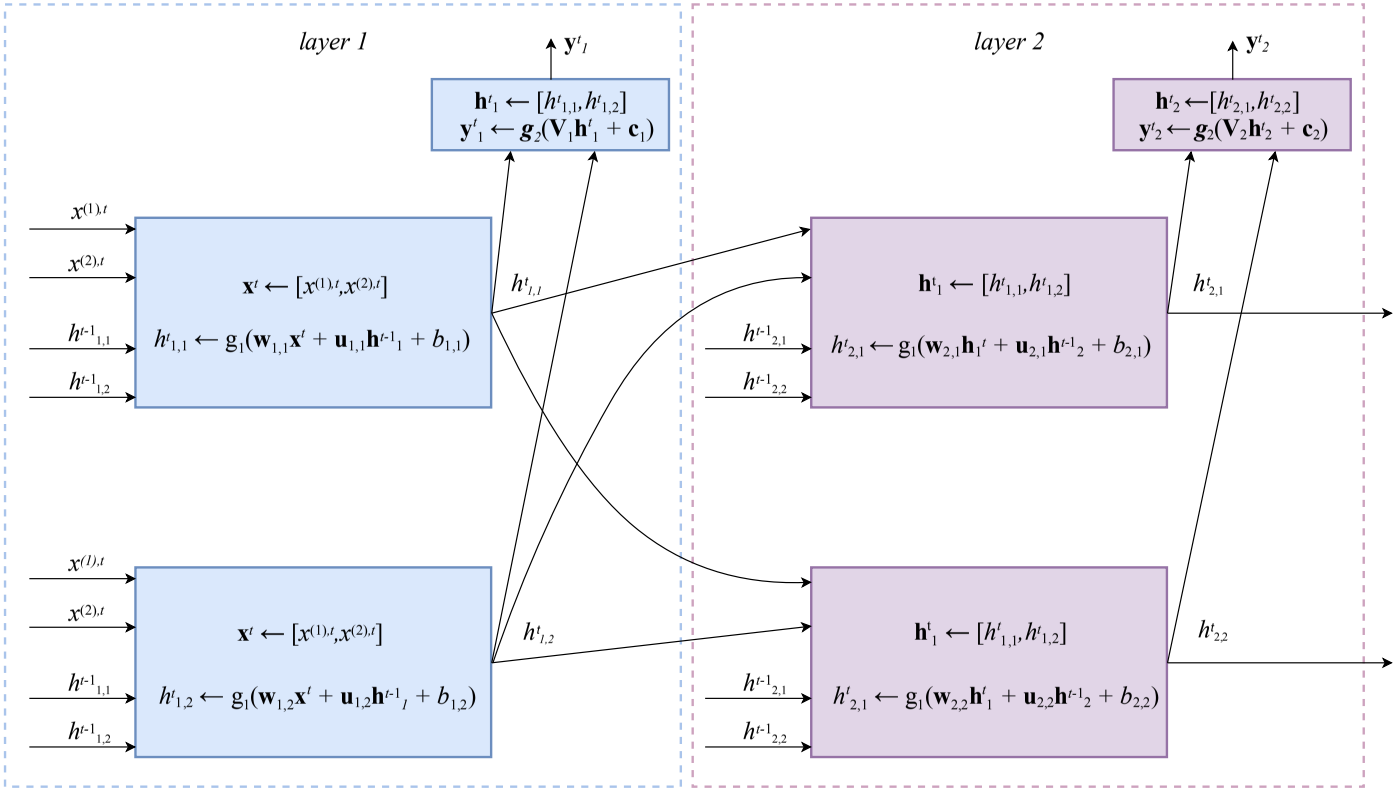


Figure 4: The first two layers of an RNN. The input feature vector is two-dimensional; each layer has two units.

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. Labeling a sequence means predicting a class to each feature vector in a sequence. Classifying a sequence means predicting a class for the entire sequence. Generating a sequence means 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, because it contains loops. Basically 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 recurrent layer l receives two inputs: a vector of outputs 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 fig. 4. As we 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 $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^{t-1}, \mathbf{x}^t, \mathbf{x}^{t+1}, \dots, \mathbf{x}^{length_\mathbf{X}}]$, 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 $t = 1, \dots, length_\mathbf{X}$ represents a word in the sentence at position t.

As depicted in fig. 4, in an RNN, the input example is "read" by the neural network one feature vector at a timestep. A timestep is denoted by the index t. 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 obtined by applying an 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 an 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 $\mathbf{c}_{l,u}$. A typical choice for $\boldsymbol{g}_2$ is the **softmax function**:

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

The softmax function is a generalization of the sigmoid function to multidimensional data. 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 certain 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 *her* seen later in the sentence.

Units make decisions about what information to store, and when to allow reads, writes, and erasures. These decisions are learned from data and implemented through the concept of *gates*. There are several architectures of gated units. The simplest one (working well in practice) is called the **minimal gated GRU** and is composed of a memory cell and a forget gate.

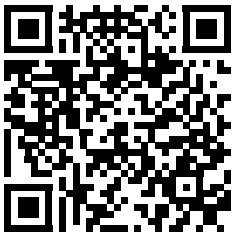
Let's look at the math of a GRU unit on an example of the first layer of the RNN (the one that takes the sequence of feature vectors as input). A minimal gated GRU unit u in layer l takes two inputs: the vector of the memory cell values from all units in the same layer

from the previous timestep, $\mathbf{h}_l^{t-1}$, and a feature vector $\mathbf{x}^t$. It then uses these two vectors like 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. The sigmoid function takes values in the range of $(0, 1)$. 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 $\tilde{h}_{l,u}^t$ (this happens in 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 period of 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. Sequence-to-sequence RNNs in particular are frequently used to build statistical machine translation models and other model for text to text transformations. A generalization of RNNs is a **recursive neural network** model.