9 Neural networks and automatic differentiation

9.1 Neural networks

An artificial neural network are made up of simple building blocks, that are put together to form a complex model. To understand how neural networks work, we will start by examining the most simple, basic building block that is used to construct these networks. Such a building block called a *neuron* in analogy to how the human brain consists of a network of connected neuronal cells.

9.1.1 The neuron

Each neuron in an artificial neural network is a linear model that is passed through a non-linear activation function. The weights in the linear model inside the neuron are adjustable parameters of the neuron. Similar to the weights in linear regression, the weights of the neuron can be adjusted by to minimize a cost function that measures the discrepancy between the output of the neuron and the desired output.

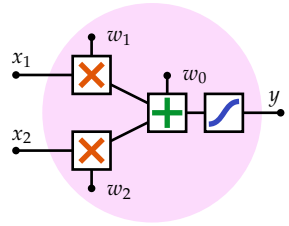

Figure 9.1: Sketch of a single neuron in an artificial neural network. Inside the neuron, each input feature is first multiplied by a weight, and the weighted inputs are then summed together with a bias weight. Finally, the sum is passed through a non-linear activation function to produce the output.

> Example 9.1 Neuron with two inputs
>
> Let us consider a neuron that takes in two input features x_1 and x_2 (see Figure 9.1.) Mathematically, we can the define the output of the neuron as
>
> $$y = f(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2)$$
>
> where w_0, w_1 and w_2 are the weights (parameters inside

the neuron) and f is the non-linear activation function. An example of an often used activation function is the hyperbolic tangent, $f(x) = \tanh(x)$.

If we have the parameters $w_0 = 0$ and $w_1 = w_2 = 1$ and get the inputs $x_1 = 0$ and $x_2 = 2$, our neuron will produce the ouput

$$y = \tanh(0 + 1 \cdot 0 + 1 \cdot 2) = \tanh(2) \approx 0.964.$$

The output of the neuron for different inputs with three different settings of the parameters are illustrated in Figure 9.2. Examining the figure, can you approximately read off the output value we just computed?

In general, we can write the output of a single neuron with a K-dimensional input as

$$y = f\left(w_0 + \sum_{k=1}^{K} w_k x_k\right),$$

where f is a non-linear activation function and w_0, w_1, ..., w_K are the weights in the neuron. If we collect the inputs in a vector $\mathbf{x}$ and collect the weights except w_0 in a vector $\mathbf{w}$, we can write this more compactly using a dot product

$$y = f(w_0 + \mathbf{w}^\top \mathbf{x}).$$

Example 9.2 Implementing a neuron

With the inputs given as a vector `x` and the weights w_1, w_2,...,w_K stored in a vector `w` and the bias weight w_0 stored as a scalar variable `w0`, we can implement a neuron with a hyperbolic tangent activation using the following computer code

```
y = np.tanh(np.dot(x, w) + w0)
```

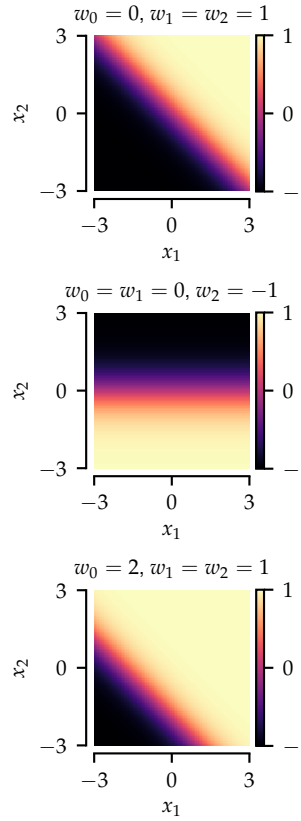

Figure 9.2: Image plots of the output of a single neuron with a two-dimensional input for different values of the weights.

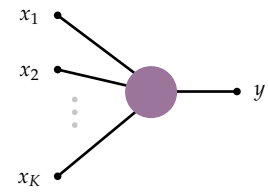

Figure 9.3: In general, a neuron takes K inputs and produces a single output.

Interpretation of the weights In linear regression, we have a nice intuitive interpretation of the weights: 1) The sign of each weight determines if the output tends to increase or decrease with the input and 2) the magnitude of each weight determines how much the output will change with one unit change of the input. This interpretation carries over to the neural network

to some extent: Usually the non-linear function is chosen as an increasing function, which means that the first intuition above holds for the neural network as well. But because of the non-linearity, the second intuition does not hold.

9.1.2 *The activation function*

If each neuron did not contain a non-linear function, it would not make much sense to put together many neurons in a big neural network: Since if each neuron were linear, the total network of neurons would also just be a linear model. The non-linear activation function in each neuron ensures that when we put together many neurons in a big neural network, the full network will be a (possibly very complex) non-linear function.

Many different non-linear functions can be used as activation functions in neural networks. Two of the most popular functions are the hyperbolic tangent (tanh) and the rectified linear unit (ReLU). These two functions are illustrated in Figure 9.4. The hyperbolic tangent can be defined as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$

It is a smooth function that maps the real numbers (from minus infinity to plus infinity) onto the range from -1 to 1. The rectified linear unit is a function that truncates negative inputs to zero and lets positive inputs pass directly through,

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Many other activation function have been used, and each have their different pros and cons, but here we restrict our discussion to the hyperbolic tangent and rectified linear unit. Some of the considerations that go in to selecting the activation function include

Computational speed The ReLU requires almost no computation, whereas the tanh requires the computation of the exponential function.

Dead neurons When the signal just before the activation function is negative, the ReLU simply provides zero output. This means that many of the neurons in the neural network could possibly be inactive. With the tanh this problem does not occur.

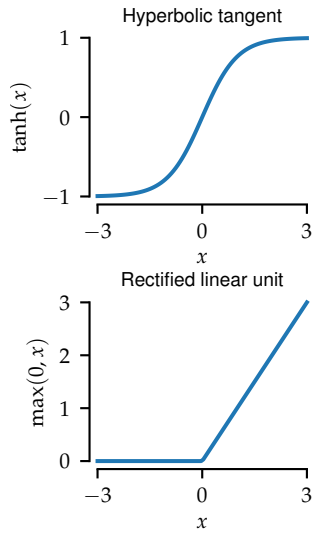

Figure 9.4: Two popular activation functions: The hyperbolic tangent and the rectified linear unit.

Exploding or vanishing gradients When training a deep neural network using gradient descent, we need the gradients to be well behaved. When using the tanh activation the gradients can sometimes tend to be smaller and smaller (vanishing gradients) as we move through the layers. With the ReLU the gradients sometimes tend to get bigger and bigger (exploding gradients).

Unfortunately it is not easy to give any general recommendations for how to choose the best activation function. The best recommendation is usually to try different ones and see what works best in practice.

9.1.3 *Connecting neurons*

To create a neural network, we can simple connect a number of neurons together. One way to do this is to first create a layer of neurons that each takes the features x as their input and produce some outputs. Then we can create a second layer of neurons that take the outputs from the first layer as inputs to produce new outputs. We can continue this process until we finally at some stage create a single neuron that takes all the outputs at the penultimate layer and produces the final output. This approach is called a multilayer perceptron and is illustrated in Figure 9.5.

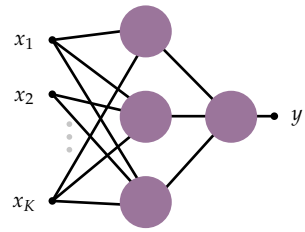

Figure 9.5: An example of a multilayer perceptron neural network with two layers. The first layer contains three neurons and the second layer contains a single neuron, that uses the output of the three neurons on the previous layer as input.

Example 9.3 Implementing multiple neurons

We have seen that a single neuron can be implemented using a dot product between the inputs and the weight vector. When we want to implement multiple neurons that all take the same input but have different weights, we can use a matrix product. Let us again say that we have the inputs in a vector `x` and now we store the weights of each neuron as rows in a matrix `W`. The bias weights w_0 for each neuron are stored in a vector `w0`. Then we can implement all the neurons in a single line of code:

```
y = np.tanh(np.dot(x, W) + w0)
```

Notice that this code is basically identical to the code for the single neuron, as the `np.dot` function can both be used for dot products and matrix multiplication.

To implement a multilayer perceptron, we can just re-

peat the code above for each layer of the network. In each line of code, the output from the previous layer is used as input in the next layer. For example, to implement a three-layer perceptron, we could write

```
h1 = np.tanh(np.dot(x, W1) + b1)
h2 = np.tanh(np.dot(h1, W2) + b2)
y = np.tanh(np.dot(h2, W3) + b3)
```

Here, we have three weight matrices `W1`, `W3`, and `W3` and three bias vectors which we have now named `b1`, `b2`, and `b3`.

9.2 Automatic differentiation

When we use the gradient descent algorithm to optimize the parameters in a model, we of course need to compute the gradient. As you may recall, the gradient is simply the vector of partial derivatives of the objective function with respect to each of the parameters in the model. In a simple model, it might be okay to just derive the formulas required to compute the gradient and implement them manually. But when we work with more complicated models such as deep neural network, it quickly becomes too cumbersome to derive everything by hand. Luckily, we can get the computer to figure out what the gradient is automatically, using an approach called automatic differentiation. In fact, automatic differentiation allows us to compute the gradient of almost anything that we can implement as a computer program.

9.2.1 *Computations as a graph*

To see how this works, let us start by considering how a mathematical expression is evaluated on a computer. Any mathematical expression can be evaluated by computing a sequence of elementary operations.

Example 9.4 Computation of a simple expression

Consider the following mathematical expression

$$z = x \cdot y + y \cdot \cos(\pi x).$$

When we want to evaluate this expression (that is com-

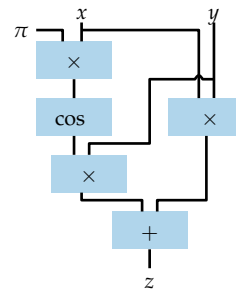

Figure 9.6: Computation graph for the mathematical expression in Example 9.4

pute the value of z for some given values of x and y) we usually work our way from the inside out and from left to right. First, we compute the product of x and y and save the result for later. Then we multipy π with x, take the cosine and save the result. Finaly we add together the two intermediate results we have computed. We can think of this step-by-step computation as a sequence of elementary operations,

$$
\begin{aligned}
a &= x \cdot y \\
b &= \pi \cdot x \\
c &= \cos(b) \\
d &= y \cdot c \\
z &= a + d
\end{aligned}
$$

Alternatively, we can think of the operations as a *computation graph* which links the input values to the output through a sequence of operations as shown in Figure 9.6.

9.2.2 *Computing a partial derivative*

Once we have formulated a computation as a graph, we can compute the partial derivative of the output with respect to any parameter we are interested in, simply by working our way backwards through the sequence of operations. Assuming that we know the derivative of each of the elementary functions that are used, at each step we simply apply this knowledge and chain together each result using the chain rule.

As you may recall, the chain rule says that we can evaluate the derivative of a function composite[1] by multiplying the derivative of the outer function with the derivative of the inner function.

[1] A function composite is when you take the output of a function and use it as input in another function, such as $f(g(x))$.

Definition 9.1 The chain rule

The derivative of a function composite

$$z(x) = f\big(g(x)\big)$$

is the product of the derivative of the outer function f evaluated at $g(x)$ and the derivative of the inner function g evaluated at x. With f' and g' denoting the derivatives of the function, this can be stated as

$$z'(x) = f'\big(g(x)\big) \cdot g'(x).$$

In Leibniz's notation, the chain rule is written as

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g} \cdot \frac{\mathrm{d}g}{\mathrm{d}x}.$$

The chain rule makes it explicit how we can compute the derivative of a sequence of operations, as we have represented in a computation graph. We simply need to take the derivative of each computation and chain the results together by multiplying the derivatives.

When dealing with multivariable functions (such as functions that operate on vectors), an extension of the chain rule tells us that we simply need to apply the chain rule to each variable and sum the results.

Definition 9.2 The multivaiable chain rule

The derivative of a multivariable composite function

$$z(x) = f\big(g_1(x), g_2(x), \ldots, g_K(x)\big)$$

is the sum of the derivative of the outer function with respect to each of its arguments multiplied with the derivative of the respective inner function

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \sum_{k=1}^{K} \frac{\partial f}{\partial g_k} \cdot \frac{\mathrm{d}g_k}{\mathrm{d}x}.$$

Example 9.5 Derivative of a simple expression

Continuing Example 9.4, let us compute the derivative of z with respect to one of our variables. We can compute the

derivative with respect to either x or y, but for now let us not decide which one we are interested in, and just take the derivative with respect to some arbitrary parameter that we call t. We start at the final computation, which is a sum so the derivative is the sum of the individual derivatives

$$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial d}{\partial t}. \tag{9.1}$$

Then we go to the second to last computation, which is a product so we need to use the product rule,

$$\frac{\partial d}{\partial t} = y\frac{\partial c}{\partial t} + c\frac{\partial y}{\partial t}. \tag{9.2}$$

Continuing this process, we end up with the following sequence of derivatives, which we show here along side with the original computations

$$\begin{aligned}
a &= x \cdot y & \frac{\partial a}{\partial t} &= x\frac{\partial y}{\partial t} + y\frac{\partial x}{\partial t} \\
b &= \pi \cdot x & \frac{\partial b}{\partial t} &= \pi\frac{\partial x}{\partial t} \\
c &= \cos(b) & \frac{\partial c}{\partial t} &= -\sin(b)\frac{\partial b}{\partial t} \\
d &= y \cdot c & \frac{\partial d}{\partial t} &= y\frac{\partial c}{\partial t} + c\frac{\partial y}{\partial t} \\
z &= a + d & \frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial d}{\partial t}
\end{aligned}$$

The new sequence of computations can then simply be run forward to compute the partial derivative. For example, to compute $\frac{\partial z}{\partial x}$ we can simply insert the initial values $\frac{\partial x}{\partial t} = 1$ and $\frac{\partial y}{\partial t} = 0$.

We could manually find the partial derivative of our expression with respect to x to be

$$\frac{\partial z}{\partial x} = y - \pi y \cdot \sin(\pi x)$$

Evaluating the derivative $\frac{\partial z}{\partial x}$ at $x = 0.5$, $y = 2$ we get

$$\left.\frac{\partial z}{\partial x}\right|_{x=0.5,y=2} = 2 - \pi \cdot 2 \cdot \sin(\pi \cdot 0.5) = \underline{2(1-\pi)}$$

We can get to the same result by inserting numbers in the

sequential computations above, which gives us

$$
\begin{aligned}
a &= 0.5 \cdot 2 = 1 & \frac{\partial a}{\partial t} &= 0.5 \cdot 0 + 2 \cdot 1 = 2 \\
b &= \pi \cdot 0.5 = \tfrac{\pi}{2} & \frac{\partial b}{\partial t} &= \pi \cdot 1 = \pi \\
c &= \cos(\tfrac{\pi}{2}) = 0 & \frac{\partial c}{\partial t} &= -\sin(\tfrac{\pi}{2}) \cdot \pi = -\pi \\
d &= 2 \cdot 0 = 0 & \frac{\partial d}{\partial t} &= 2 \cdot (-\pi) + 0 \cdot 0 = -2\pi \\
z &= 1 + 0 = 1 & \frac{\partial z}{\partial t} &= 2 + (-2\pi) = \underline{2(1 - \pi)}
\end{aligned}
$$

Problems

1. Repeat the calculations at the end of Example 9.5 to evaluate the derivative of z with respect to y at $x = 0.5$, $y = 2$. Check your result by inserting in $\frac{\partial z}{\partial y} = x + \cos(\pi x)$.

Solutions

1. Computation of $\left.\frac{\partial z}{\partial y}\right|_{x=0.5,y=2}$

$$
\begin{aligned}
a &= 0.5 \cdot 2 = 1 & \frac{\partial a}{\partial t} &= 0.5 \cdot 1 + 2 \cdot 0 = 0.5 \\
b &= \pi \cdot 0.5 = \tfrac{\pi}{2} & \frac{\partial b}{\partial t} &= \pi \cdot 0 = 0 \\
c &= \cos(\tfrac{\pi}{2}) = 0 & \frac{\partial c}{\partial t} &= -\sin(\tfrac{\pi}{2}) \cdot 0 = 0 \\
d &= 2 \cdot 0 = 0 & \frac{\partial d}{\partial t} &= 2 \cdot 0 + 0 \cdot 1 = 0 \\
z &= 1 + 0 = 1 & \frac{\partial z}{\partial t} &= 0.5 + 0 = \underline{0.5}
\end{aligned}
$$

Check:

$$\frac{\partial z}{\partial x} = x + \cos(\pi x) = 0.5 + \cos(\pi \cdot 0.5) = \underline{0.5}$$