Lecture 9
Artificial Neural Networks, Part 2

Alice Gao

November 2, 2021

Contents

1 Learning Goals

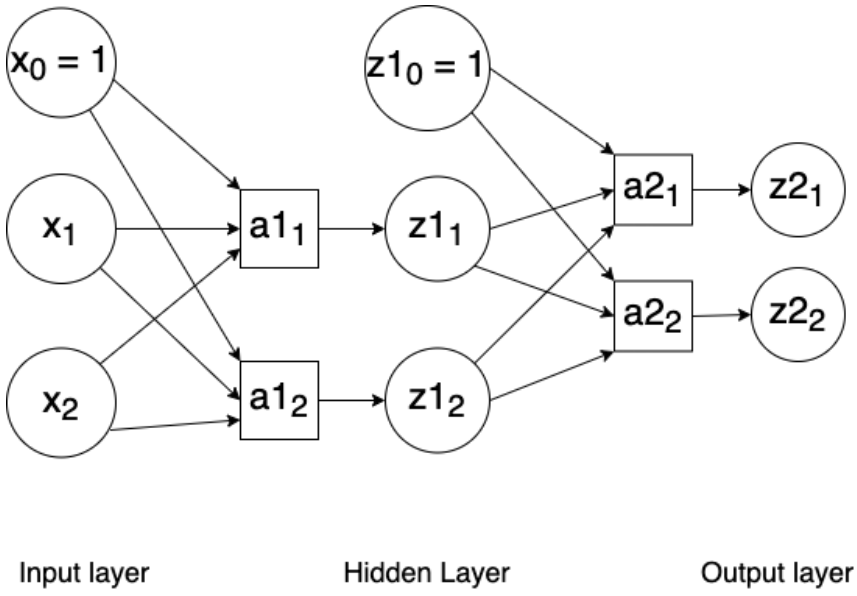
By the end of the lecture, you should be able to

- Explain the steps of the gradient descent algorithm.
- Explain how we can modify gradient descent to speed up learning and ensure convergence.
- Describe the backpropagation algorithm including the forward and backward passes.
- Compute the gradient for a weight in a multi-layer feed-forward neural network.
- Describe situations in which it is appropriate to use a neural network or a decision tree.

2 Gradient Descent

2.1 A motivating example

Throughout this lecture, we will work with this simple two-layer feed-forward neural network. The input layer has three nodes, one of which is a dummy. The hidden layer also has three nodes, one of which is a dummy. The output layer has two nodes.



Our goal is to learn the weights Wk_{ij} such that the actual outputs $z2_1$ and $z2_2$ are close to the expected values given by the training data.

2.2 Introduction

Gradient descent is a well-known optimization algorithm; you may have learned about it in another math class.

One way to think about gradient descent is as a local search algorithm to find the minimum of a function. Consider a search space whose dimension is given by the number of weights in the neural network. We want to find a combination of weights which minimizes the total error made on the training examples.

Gradient descent is very similar to greedy descent, but differs in that we are now dealing with continuous rather than discrete values. Accordingly, we will use the gradient, or partial derivative, to determine the direction and size of each step we want to take. The function in this case will be the error function: the difference between the expected outputs based on the training data and the actual outputs based on the training data and the network.

This quotation might give you an intuitive way to understand the algorithm:

"Walking downhill and always taking a step in the direction that goes down the most."

We descend to the minimum by following the steepest direction at each step.

2.3 Steps of the algorithm

- Initialize the weights randomly.
- For each training example, change each weight in proportion to the negative of the partial derivative of the error with respect to that weight. Overall, we will update a weight w by
$$w := w - \sum_{\text{examples}} \eta \frac{\partial \text{error}}{\partial w}.$$
- Here, η is a constant called the learning rate.
- Terminate after some number of steps when the error is small enough or when the changes made get small enough.

2.4 Deriving some intuition for gradient descent

You may find gradient descent to be unintuitive at first, but here are some of the intuition that may help.

Consider a simple function: $y = x^2$. We want to minimize this function given some starting point x_0; the minimum is at $x = 0$. To do so, we need to consider direction and magnitude.

First suppose we start with x_0 to the right of the minimum. Then $\frac{dy}{dx} > 0$, but to decrease our value, we should move in the direction of the negative of the gradient.

Instead, suppose we start with x_0 to the left of the minimum. Then $\frac{dy}{dx} < 0$, but to increase our value, we should move in the direction in the negative of the gradient.

As for direction, first note that the gradient is flat (zero) at the minimum. Then consider a region where the gradient is large in magnitude. In such a region, it is certain that we must be far from the minimum (since the function is continuous), so we can afford to take a larger step. On the other hand, in a region where the gradient is small in magnitude, we are likely

to be close to the minimum. Thus, we should take a smaller step to avoid overshooting the minimum.

This discussion is not rigorous by any means, but should provide some intuition behind the ideas and derivation of gradient descent.

2.5 Alternative ways to update the weights

In the steps given in a previous section, we said that gradient descent updates the weights after sweeping through all the training examples. But when the training data set is large, wouldn't the weight updates be very infrequent?

To speed up learning, we can instead update weights after each example. This variant is called **incremental gradient descent**. A related variant is **stochastic gradient descent**, where we randomly choose an example at each step instead of proceeding sequentially.

The advantage of incremental gradient descent is increased learning speed, since we update the weights much more frequently. The disadvantage is that we may not converge to the local minimum—a single example may take us away from rather than towards it.

To move the trade-off in our favour, we have another variant called **batched gradient descent**. As the name suggests, we will update the weights after a batch of examples.

We can choose the batch size. On one extreme, a batch size of one is identical to incremental gradient descent. On the other extreme, putting all examples into the batch is identical to the original gradient descent. By moving between the extremes, we can work in our favour. Often, people will start with small batches to learn quickly, then gradually increase the batch size until the weights eventually converge.

3 The Back-propagation Algorithm

There's a huge hype around neural networks right now. If you haven't learned the back-propagation algorithm already, you are probably pretty excited about this video.

What is the back-propagation algorithm? It is essentially an algorithm to learn the weights in a neural network by using the gradient descent optimization algorithm.

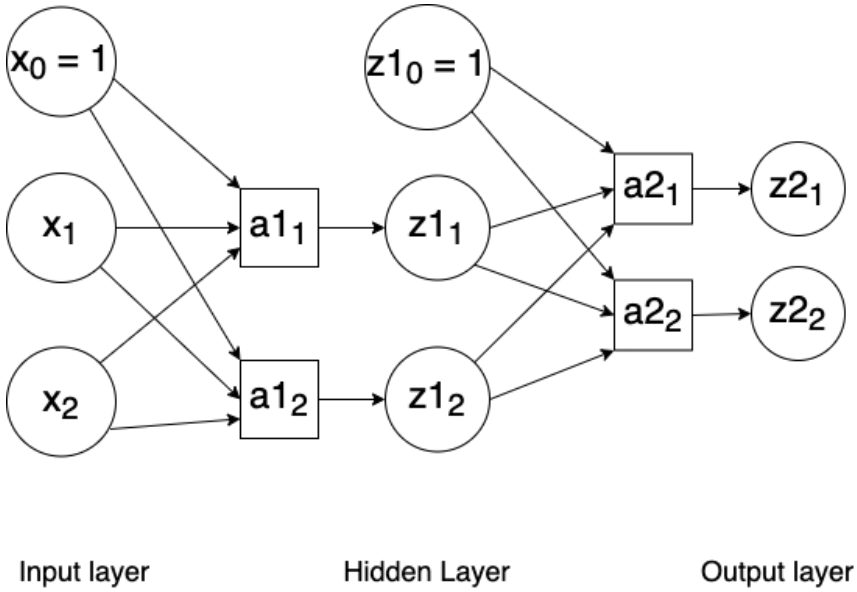


Figure 1: 2-layer feed-forward neural network

I will use a 2-layer feed-forward neural network as an example. Let me explain some notation. Let's start from the inputs x_i. $W1_{ij}$ denotes the weights between the input layer and the hidden layer. $a1_j$ denotes the j-th weighted sum of the input units. $z1_j$ denotes the j-th hidden unit, and it's a result of applying the activation function g to $a1_j$.

The notation for the next layer is very similar. $z1_j$ is the value of the hidden unit. $W2_{jk}$ denotes the weights between the hidden layer and the output layer. $a2_k$ denotes the k-th weighted sum of the hidden units. Finally, $z2_k$ denotes the k-th output unit, and it's a result of applying the activation function g to $a2_j$.

You can also think of the network structure as a chain as shown below.

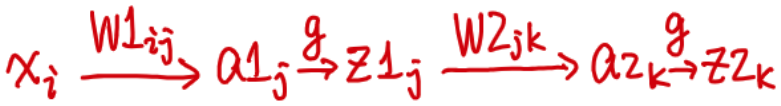


Figure 2: Chain to represent a neural network

3.1 Introducing the Back-Propagation Algorithm

Let's discuss the back-propagation algorithm. The most challenging step in the gradient descent algorithm is to calculate the gradient for each weight. We can write down an expression for the gradient for each weight and calculate it using the expression. Unfortunately, this approach is quite in-efficient. Nowadays, neural networks tend to be quite large, and there could be thousands or more weights that we need to learn. The back-propagation algorithm is an efficient way of calculating the gradients for the weights.

Consider a setting where we have n training examples. For each training example, x consists of the input feature values, and y is the label. In other words, x is the input to our neural network and y is the expected output.

To evaluate how close the actual output values z2 are to the expected output values y, we will use an error/loss function E. When we execute gradient descent, our goal is to minimize the error/loss E by adjusting the weights in the neural network.

Given the training examples, we will calculate the gradients by performing 2 passes in the network: a forward pass and a backward pass.

The forward pass takes the input values x, the current weights W1 and W2, and calculates $E(z2, y)$ the error/loss E between the actual output values z2 and the expected output values y.

The backward pass computes the gradients $\frac{\partial E}{\partial W2_{jk}}$ and $\frac{\partial E}{\partial W1_{ij}}$, which are the partial derivatives of the error function with respect to W2 and W1. For our network, the forward pass flows from left to right, and the backward pass flows from right to left.

For each training example, we will calculate one gradient for each weight. Then, to update each weight, we need to add the gradients for this weight for all the training examples. We will update the weight proportional to this sum of the gradients.

3.2 An Intuitive Description of the Back-Propagation Algorithm

Let me give you an intuitive description of the algorithm. Our goal is to set the weights in the network to minimize the error/loss. How do we do that?

Using each training example, we will compute the gradient for each weight. The gradient tells us how much the error/loss changes if I change the weight by a tiny bit. If the gradient is positive, we should decrease the weight and vice versa. The gradient guides us about how we should change the weight locally to minimize the error/loss.

Why do we need to add up the gradients for all the training examples? The reason is that, different training examples may want to change each weight differently. One example may suggest that we should increase the weight, whereas another example may suggest that we decrease the weight. We do not want to only predict one example well. We would like to achieve a high prediction accuracy for all the examples. Therefore, we need to sum up the gradients for all the examples and use the sum to adjust each weight.

3.3 The Forward Pass

The forward pass starts from the input values on the left.

First, calculate a1 as the weighted sum of the input values using the weights W1. The hidden unit values z1 is the activation function g applied to the weighted sum a1.

$$a1_j = \sum_i x_i W1_{ij} \qquad z1_j = g(a1_j) \tag{1}$$

We will calculate the output values in the same way. a2 is the weighted sum of the hidden unit values using the weights W2. The output value z2 is the result of applying g to the weighted sum a2.

$$a2_k = \sum_j z1_j W2_{jk} \qquad z2_k = g(a2_k) \tag{2}$$

Finally, the error/loss $E(z2, y)$ is a function of the actual output values z2 and the expected output values y.

3.4 The Backward Pass

The backward pass is the core of the back-propagation algorithm.

Our goal is to calculate the gradients for the weights — the partial derivative of E with respect to $W1$ and $W2$. We will calculate the gradients by going backwards in the network, from the outputs on the right to the inputs on the left.

Starting with the outputs on the right, we will first calculate the gradients for the weights $W2$. I've written the expression as the product of two terms: the partial derivative of E w.r.t. to $a2$ and $z1$. Let me define the first term to be $\delta 2$. The second term $z1$ is the input going into the edge for the weight $W2$.

$$\frac{\partial E}{\partial W2_{jk}} = \frac{\partial E}{\partial a2_k} z1_j = \delta 2_k z1_j, \qquad \delta 2_k = \frac{\partial E}{\partial z2_k} g'(a2_k) \tag{3}$$

At this point, it seems unnecessary to define $\delta 2$. However, the deltas will be extremely useful for us to understand the back-propagation process. I will define one set of δ values for each layer. You will see shortly that the δ values for different layers form a recursive relationship, which will allow us to calculate the gradients efficiently.

Next, let's calculate the gradients for the weights $W1$. This expression is similar to that of $W2$. The gradient is the product of two terms: the partial derivative of E w.r.t. to $a1$ and

x. Similarly, I'll define the first term to be $\delta 1$. The second term $x1$ is the input going into the edge for the weight $W1$.

$$\frac{\partial E}{\partial W1_{ij}} = \frac{\partial E}{\partial a1_j} x_i = \delta 1_j x_i, \qquad \delta 1_j = \left(\sum_k \delta 2_k W2_{jk} \right) g'(a1_j) \tag{4}$$

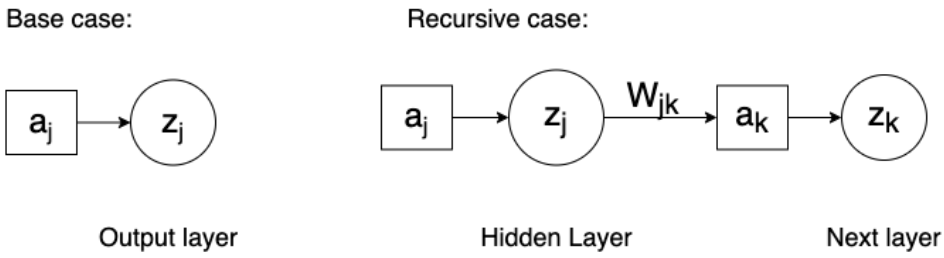
Note that the gradient for the weights in each layer has a similar expression. Each expression is a product of two terms: the delta value and the input going into the weight.

The remaining question is, how do we calculate the delta values? I am showing you the expressions for $\delta 2$ for the output layer and $\delta 1$ for the hidden layer. Note that we need the $\delta 2$ values to calculate $\delta 1$. This gives you a hint of what the recursive relationship looks like.

3.5 The recursive relationship in the delta values

Let's take a closer look at the recursive relationship of the delta values. The recursive relationship allows us to propagate the error backward through the network and calculate the gradients efficiently. This recursive relationship is also how the back-propagation algorithm got its name.

Consider a unit j. This unit may be in the output layer or any hidden layer.



In general, δ_j is the partial derivative of E w.r.t. to a_j, where a_j is the weighted sum of the values from the previous layer.

$$\delta_j = \frac{\partial E}{\partial a_j}.$$

To calculate δ_j, we need to consider two cases.

- When j is an output unit, we have the base case and we will calculate δ_j directly.
- When j is a hidden unit, we have the recursive case. In this case, calculating δ_j requires δ_k, which is the delta value for the next layer — the layer to the right, on the side closer to the output layer.

$$\delta_j = \begin{cases} \dfrac{\partial E}{\partial z_j} \times g'(a_j), & \text{base case, } j \text{ is an output unit} \\ \left(\displaystyle\sum_k \delta_k W_{jk} \right) \times g'(a_j), & \text{recursive case, } j \text{ is a hidden unit} \end{cases} \tag{5}$$

Let's think about the recursive case intuitively. The hidden unit j is connected to multiple units k in the next layer. Therefore, unit j is responsible for some fraction of the error δ_k in each unit k that j connects to. For each unit k, we will weigh each error δ_k by the strength of the connection between j and k — this is the weight W_{jk}. This weighted sum allows us to take the errors δ_k from the next layer and propagate them back to calculate the error δ_j in the current layer.

Also note that, the expressions for the two cases are quite similar. In particular, the second terms are identical — they are both the derivative of the activation function g.

Since our network only has one hidden layer, we only need to use the recursive case once. If a network has multiple hidden layers, we would need to apply the recursive case multiple times.

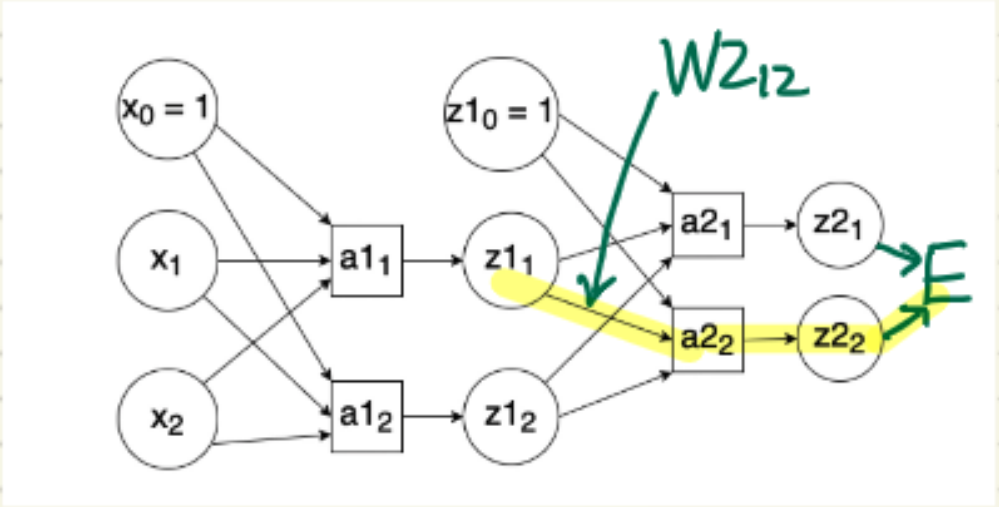
Here is a practice question for you. Construct a 3-layer neural network and write down the delta values for all three layers.

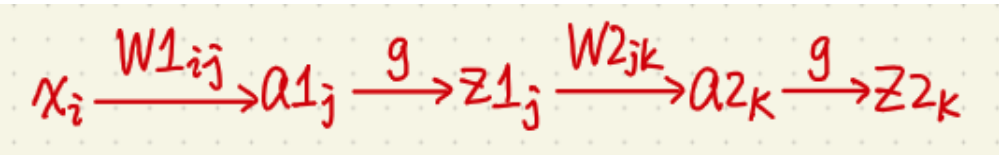
So far, I've shown you all the formulas you need to execute the back-propagation algorithm. If your goal is to implement the algorithm and run it, this is all you need. However, I hope that you are also interested in how the expressions were derived. In the next video, I will derive the partial derivatives by using the chain rule many many times. Once you understand the derivations, you will realize that the recursion with the delta values is nothing mysterious. The back-propagation process emerges directly from the derivation of the gradients.

3.6 Deriving the gradients for W2

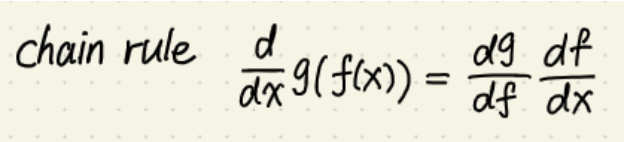
First, let's derive the partial derivatives for the gradients for W2.

To remember what the network looks like, you can look at the picture or the chain. The chain is a simpler illustration of how the values flow through the network from inputs to outputs. In the picture, I have highlighted one path — the one for the weight $W2_{12}$.





Before I start the derivations, let's do a super quick review of the chain rule. Deriving the gradient boils down to repeatedly applying the chain rule. Suppose that we have an expression in which we have applied several functions in sequence. In this case, we applied f first, and then g second. Taking derivative of the expression is equivalent to taking derivatives of the functions in reverse order, from the last one applied to the first one applied. I often think of this process as peeling off the functions one by one. For this example, the derivative is equal to the derivative of g w.r.t. to f multiplied by the derivative of f w.r.t to x.



We are ready to derive our first gradient expression — the partial derivative of E w.r.t $W2_{jk}$. Our expression is basically tracing through the highlighted path from right to left. First, E is a function of z2 and y, so we have $\frac{\partial E}{\partial z2_k}$. Next, z2 is g of a2, so we have $\frac{\partial z2_k}{\partial a2_k}$. Next, a2 is a function of the weights W2 and the hidden unit values z1. Since we want to take derivative w.r.t. to the weights, we have $\frac{\partial a2_k}{\partial W2_{jk}}$.

Example:

$$\frac{\partial E}{w2_{jk}} = \frac{\partial E}{z2_k}\frac{\partial z2_k}{a2_k}\frac{\partial a2_k}{w2_{jk}} \tag{6}$$

If you have got here and understood how the expression was derived, great job! You got

through the most difficult part —- deriving the most general and abstract expression for this derivative. It only gets easier from here. If you still have questions, no worries. I will show you some more concrete examples right now and they should help clear up some confusion.

Let's make some of the terms more concrete. What is the middle term? $z2$ is $g(a2)$, so the middle term is $g'(a2)$. How about the last term? $a2$ is the sum of $W2$ multiplied by $z1$. So the last term should be $z1_j$, the input corresponding to the weight $W2_{jk}$. This expression is equivalent to the previous one, but it's easier on the eyes.

Example:

$$\frac{\partial E}{w2_{jk}} = \frac{\partial E}{z2_k}\frac{\partial z2_k}{a2_k} \tag{7}$$

$$\frac{\partial a2_k}{w2_{jk}} = \frac{\partial E}{z2_k}g'(a2_k)z1_j \tag{8}$$

$$z2_k = g(a2_k) \Rightarrow \frac{\partial z2_k}{a2_k} = g'(a2_k) \tag{9}$$

$$a2_k = \sum_j w2_{jk}z1_j \Rightarrow \frac{\partial a2_k}{w2_{jk}} = z1_j \tag{10}$$

Let's do an example for a particular weight. Consider the weight going from $z1_1$ to $a2_2$ — the highlighted one in the picture. All we need to do is take the general expression and plug in $j = 1$ and $k = 2$. Here is the result.

Example: An example with $j = 1$ and $k = 2$

$$\frac{\partial E}{w2_{12}} = \frac{\partial E}{z2_2}\frac{\partial z2_2}{a2_2}\frac{\partial a2_2}{w2_{12}} = \frac{\partial E}{z2_2}g'(a2_2)z1_1 \tag{11}$$

If you want to make this expression more concrete, you'd have to pick the error function E and the activation function g. Here are two examples. If the error function is the sum of the squared difference, then the partial derivative looks like this. If the activation function is the sigmoid function, then the partial derivative looks like this. Interestingly, the partial derivative of the sigmoid function can be written as a expression involving two copies of the sigmoid function. If you don't believe it, please verify it yourself.

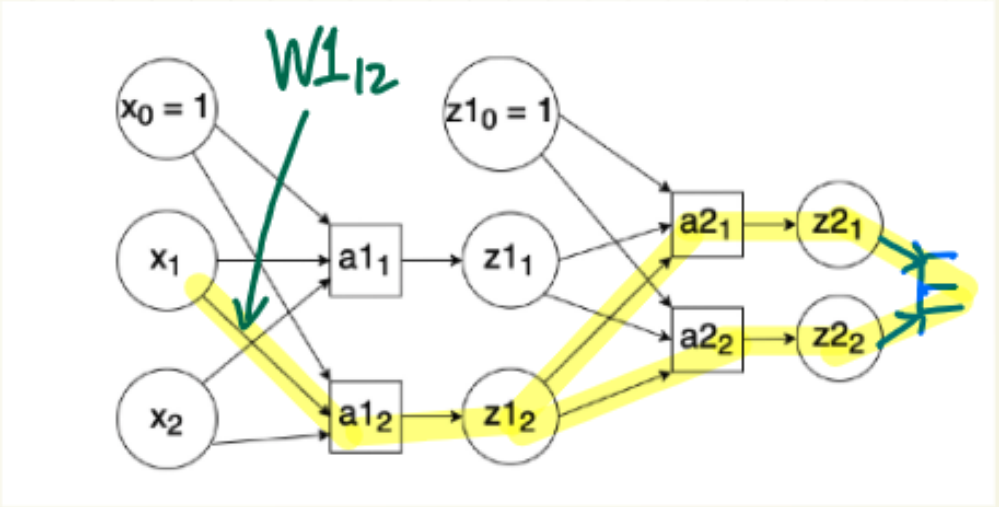
Example:

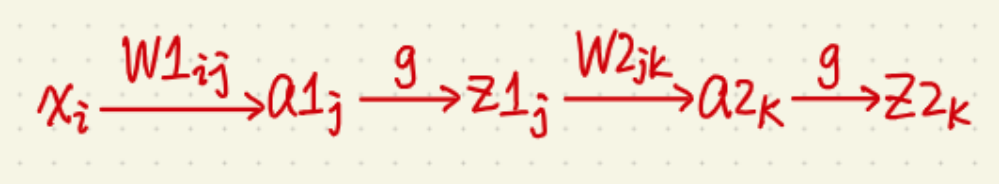
$$E = \frac{1}{2}\sum_k (z2_k - y_k)^2 \Rightarrow \frac{\partial E}{\partial z2_k} = (z2_k - y_k) \tag{12}$$

$$g(x) = \frac{1}{1 + e^{-x}} \Rightarrow g'(x) = g(x)(1 - g(x)) \tag{13}$$

3.7 Deriving the gradients for W1

Next, let's derive the expression for the partial derivative of E w.r.t to W1. These expressions are more complex but also more interesting.





Let's look at a specific example. Consider $W1_{12}$, the weight between x_1 and $z1_2$. I've highlighted the weight in the picture. How does this weight influence the error/loss? This weight affects $a1_2$, which affects $z1_2$. From there, it affects the error/loss E through 2 paths, one through each of the two output units. Because of this, we will have a summation with 2 terms in our expression.

Let's derive our expression now. To apply the chain rule, we'll look at the highlighted path and go from right to the left. We'll start with a summation over k since we need to consider two paths, one for each output unit $z2_k$.

E is a function of both z2 values. So the first term is $\frac{\partial E}{\partial z2_k}$. Next, $z2$ is $g(a2)$, so we have $\frac{\partial z2_k}{\partial a2_k}$. After that, $a2$ is a weighted sum of $z2$, so the next term is $\frac{\partial a2_k}{\partial z1_j}$. At this point, the two paths merge into one, so we can end our summation.

Let's keep following the single path from z1. z1 is g of a1, so the next term is partial z1 over partial a1. a1 is a weighted sum of x using the weights W1, so the final term is partial a1 over partial $W1_{ij}$. That's the entire expression.

Example:

$$\frac{\partial E}{\partial w1_{ij}} = \left(\sum_k \frac{\partial E}{\partial z2_k} * \frac{\partial z2_k}{\partial a2_k} * \frac{\partial a2_k}{\partial z1_j} \right) \frac{\partial z1_j}{\partial a1_j} \frac{\partial a1_j}{\partial w1_{ij}} \tag{14}$$

Similar to the previous derivation, you might find it helpful to make it more concrete. If we simplify some terms, we'll get the second expression.

Example:

$$\frac{\partial E}{\partial w1_{ij}} = \left(\sum_k \frac{\partial E}{\partial z2_k} * \frac{\partial z2_k}{\partial a2_k} * \frac{\partial a2_k}{\partial z1_j} \right) \frac{\partial z1_j}{\partial a1_j} \frac{\partial a1_j}{\partial w1_{ij}} \tag{15}$$

$$= \left(\sum_k \frac{\partial E}{\partial z2_k} g'(a2_k) * w2_{jk} \right) g'(a1_j) x_i \tag{16}$$

Whenever we have a derivative of z with respect to a, the result is the derivative of the activation function g. We have two other similar terms: the derivative of a with respect to z or W. a2 is a sum of z1 weighted by W2. So partial a2 partial z2 is equal to W2. Similarly, a1 is a sum of x weighted by W1. So partial a1 partial W1 is equal to the input value x.

This is all the simplification we can do without knowing the expressions for the error function E and the activation function g. You can also derive the specific expression for $W1_{12}$. Here it is.

Example: An example with $i = 1$ and $j = 2$

$$\frac{\partial E}{\partial w1_{12}} = \left(\sum_k \frac{\partial E}{\partial z2_k} g'(a2_k) * w2_{2k} \right) g'(a1_2) x_1 \tag{17}$$

3.8 Defining the delta values

We have derived some complicated derivative expression so far. Finally, let's connect these to the delta values.

Here are the simplified gradient expressions again.

Example:

$$\frac{\partial E}{\partial w2_{jk}} = \frac{\partial E}{\partial z2_k} g'(a2_k) z1_j \tag{18}$$

$$\frac{\partial E}{\partial w1_{ij}} = \left(\sum_k \frac{\partial E}{\partial z2_k} g'(a2_k) w2_{jk} \right) g'(a1_j) x_i \tag{19}$$

Note that, the last term in each expression is the input value for that layer. Let's disregard the last term in both expressions. Take the rest of the expression and define it to be delta. I've denoted them as $\delta 2_k$ and $\delta 1_j$.

Example:

$$\frac{\partial E}{\partial w2_{jk}} = \frac{\partial E}{\partial z2_k} g'(a2_k) z1_j = \delta 2_k z1_j \tag{20}$$

$$\text{where } \delta 2_k = \frac{\partial E}{\partial z2_k} g'(a2_k) \tag{21}$$

$$\frac{\partial E}{\partial w1_{ij}} = \left(\sum_k \frac{\partial E}{\partial z2_k} g'(a2_k) w2_{jk} \right) g'(a1_j) x_i = \delta 1_j x_i \tag{22}$$

$$\text{where } \delta 1_j = \left(\sum_k \frac{\partial E}{\partial z2_k} g'(a2_k) w2_{jk} \right) g'(a1_j) \tag{23}$$

$$\text{and } \delta 2_k = \sum_k \frac{\partial E}{\partial z2_k} g'(a2_k) \tag{24}$$

$$\text{or } \delta 1_j = \left(\sum_k \delta 2_k w2_{jk} \right) g'(a1_j) \tag{25}$$

Note that, $\delta 2_k$ appears inside the expression for $\delta 1_j$. If we re-write the delta expressions separately, we get the delta expressions that I showed you in the previous section.

Example:

$$\delta 2_k = \frac{\partial E}{\partial z2_k} g'(a2_k) \tag{26}$$

$$\delta 1_j = \left(\sum_k \delta 2_k w2_{jk} \right) g'(a1_j) \tag{27}$$

Here is a final practice question for you. Construct a 3-layer network with 2 hidden layers. Each hidden layer has 2 real nodes and 1 dummy node. The output layer has 2 nodes. Calculate the gradients for all the weights in the 3 layers. Write down the general expressions and then write down the expressions using the delta values. When using the delta values, be sure to write down the expressions for the delta values for every layer. If you understood everything in this video, you should be able to complete this question.

4 Neural Networks vs. Decision Trees

In our short machine learning unit, we discussed two models: decision trees and neural networks. When should you use one versus the other?

4.1 When should we use a neural network?

- Neural networks are well-suited to high-dimensional or real-valued inputs, or noisy (sensor) data. You are likely familiar with the most recent stories of neural networks in applications with such data: images, videos, audio, and so on.
- A second situation is when the form of the target function is unknown (no model). There is a theorem which says that with enough levels and nodes, a neural network can approximate any arbitrary function. In the case that the function is unknown, you may want a very flexible model which allows you to discover the true function.
- The third case for using a neural network is when it is not important for humans to explain the learned function. Once we learn a neural network, it usually has high prediction accuracy but tends not to be interpretable.

4.2 Disadvantages of neural networks

- It is difficult to determine the network structure. There are many layer and many neurons.
- It is difficult to interpret weights, especially in multi-layer networks.
- Neural networks tend to over-fit in practice (predict poorly outside of the training data) since they have very high variance.

4.3 Choosing a decision tree or a neural network

There are many factors to consider when choosing between a decision tree and a neural network. Each model will work better under different circumstances. The following table provides a summary of the differences.

	Decision trees	**Neural networks**
Data types	Good for tabular data (rows and columns).	Good for images, audio, text, etc.
Size of data set	Work well with little data.	Require lots of data, but overfit easily.
Form of target function	Model nested if-then-else statements.	Model arbitrary functions.
Architecture	Only a few parameters.	Lots of parameters, all are critical.
Interpreting the learned function	Easy to understand.	Essentially a black box: difficult to interpret.
Time available for training and classification	Fast to train and classify.	Slow to train and classify.

You may be tempted to opt for a neural network because of how powerful it is, but bear in mind the drawbacks. It is often easier to begin with a simpler model, then increase the complexity as needed. Sometimes, a simpler model might do just as well as a complicated model, with less time required to implement and test.