Advanced Machine Learning-Lecture 6: Neural Networks

Neural Networks — Theory and Training (Full Conversion)

Up until now, we have mainly studied linear models. Linear models are powerful and simple, but they come with a major drawback: they suffer from what is known as the curse of dimensionality. This means that as the number of input features grows, the data becomes sparse in high-dimensional space. In such cases, linear models struggle to capture the underlying complexity of the data, because a single linear relationship is often too rigid.

To address this, there are two broad solutions that have emerged in machine learning:

- Basis functions centered on training data points. The idea here is to enrich the input space by defining new features based on the training data. This is essentially what Support Vector Machines (SVMs) do. They use kernels to project the input data into a higher-dimensional feature space, where the data may be more easily separated by a linear model.
- Fix the basis functions in advance but make them adaptive. Instead of defining basis functions around each training point, we can predefine a set of basis functions and then allow them to adapt their shape during training. This is the idea behind neural networks (NNs). Each neuron applies a transformation (through its activation function), and by stacking many of them together, we obtain a flexible system that can approximate very complex functions.

So, the motivation for neural networks is that they can be seen as a way to overcome the limitations of linear models by using adaptive basis functions that learn directly from the data.

Building a Neural Network Step by Step

So far, our models have been linear combinations of the input variables. Let's formalize that.

A typical linear model looks like this:

$$y(x, w) = f\left(\sum_{j=1}^{M} w_j \, \phi_j(x)\right),$$

where the $\phi_j(x)$ are basis functions of the input x, and the w_j are weights.
- In linear regression, these basis functions are just the input variables themselves.
- But neural networks extend this idea by *learning new basis functions automatically.*

Step 1: First linear combination

We start with the inputs $x_1, \dots, x_D$. Each neuron in the first hidden layer computes a **linear combination** of these inputs:

$$z_j = \sum_{i=1}^{D} w_{ji}^{(1)} \, x_i + w_{j0}^{(1)}.$$

Here:
- $w_{ji}^{(1)}$ are the weights from input i to hidden unit j.
- $w_{j0}^{(1)}$ is the bias for unit j.
- z_j is sometimes called the **pre-activation** value.

Step 2: Apply a nonlinear transformation (activation)

Each hidden unit then applies a nonlinear function $h(\cdot)$ to its pre-activation:

$$a_j = h(z_j).$$

This nonlinearity is crucial — if we only had linear functions, stacking layers would collapse back into a single linear model.

Step 3: Combine the hidden units

Now, the outputs of the hidden units — the a_j — are themselves linearly combined again, to form the inputs to the next layer (or directly the outputs):

$$z_k = \sum_{j=1}^{M} w_{kj}^{(2)} \, a_j + w_{k0}^{(2)}.$$

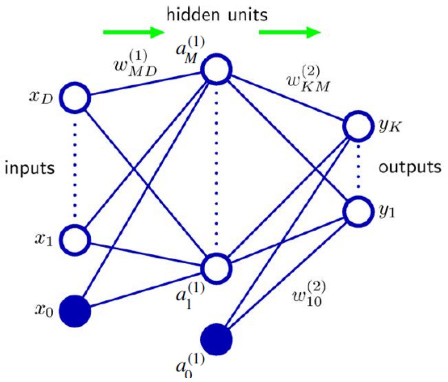
Here we've introduced a second set of weights, $w_{kj}^{(2)}$, which connect hidden unit j to output unit k.

Step 4: Apply the output activation

Finally, we pass these through an **output activation function** $\sigma(\cdot)$:

$$y_k(x, w) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} \, h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \Big) + w_{k0}^{(2)} \right).$$

This is the **general form of a single-hidden-layer neural network**:
1. Input → linear combination → hidden units
2. Hidden units → nonlinearity → transformed features
3. Hidden features → linear combination → output
4. Output → possibly another nonlinearity



Choosing the output activation σ and hidden activation h

The choice of activation depends on the problem:
- For **regression**, we usually use the identity function: $\sigma(z) = z$.
- For **binary classification**, we use the sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$.
- For **multi-class classification**, we use the softmax function, which ensures that the outputs are positive and sum to 1.

For the hidden units $h(z)$, we want nonlinear functions such as:
- Sigmoid or tanh, which were historically common.
- ReLU or leaky ReLU, which are widely used today because they avoid saturation problems and train faster.

We have seen so far, how a neural network is built: it is a sequence of **linear combinations + nonlinear transformations**, stacked to create flexible, adaptive models.

Neural Networks as Universal Approximators

One of the most important theoretical results about neural networks is that they are **universal approximators**. What does this mean?

It means that even with just a **single hidden layer**, if we include enough hidden units, a neural network can approximate *any* continuous function on a compact domain, to arbitrary accuracy. This is not just intuition — it is a mathematical theorem.

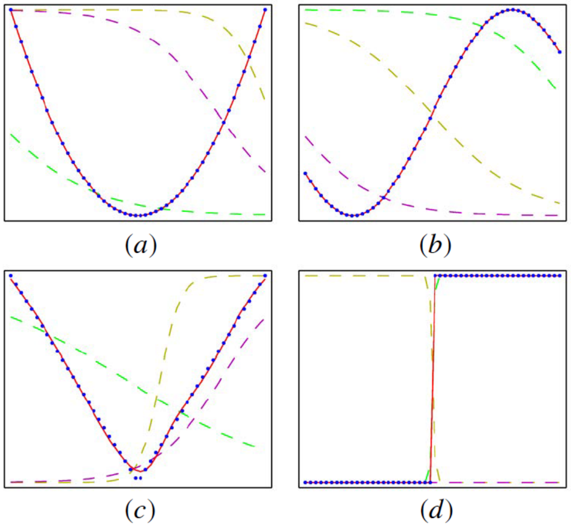
Examples of approximation

Let's look at some examples of functions that are very different in nature:

- a) $f(x) = x^2$: a simple quadratic.
- b) $f(x) = \sin(x)$: a smooth periodic function.
- c) $f(x) = |x|$: a piecewise linear, non-smooth function.
- d) $f(x) = H(x)$: the Heaviside step function, which is discontinuous.

With just **two layers** (an input layer, one hidden layer, and an output), and only **three hidden units** using the tanh activation function, a neural network with $N = 50$ data points can approximate all of these functions quite well.

The point is: even very simple neural networks already capture nonlinearity and can mimic quite complicated shapes.



(*a*) (*b*) (*c*) (*d*)

Intuition

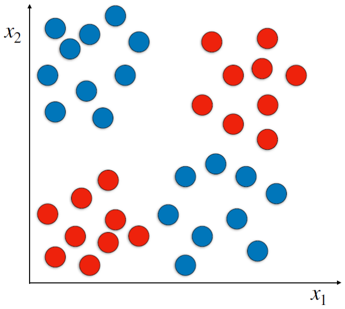
To build intuition, let's think about what each hidden unit does.

- A hidden neuron takes a **linear combination of inputs** and then passes it through a nonlinearity.

- Geometrically, this means each hidden neuron can carve out a region in the input space where it is active.

By combining many such neurons, the network can create arbitrarily complex decision boundaries or function approximations.

So the universal approximation theorem tells us: we don't necessarily need infinitely deep networks — a shallow network is already enough in theory. In practice, though, deeper networks can achieve the same accuracy with fewer neurons and can be trained more efficiently.



These ideas establish the foundation: neural networks are not just another model, but a **general framework for approximating functions**.

Neural Networks and Logic Gates: Intuition

A useful way to build intuition about neural networks is to see how they can replicate simple logical operations. Remember, each neuron computes a weighted sum of its inputs, applies a bias, and then passes the result through an activation function — often a step-like function in these examples.

Note: AND, OR, and XOR are not random names but logical operations from Boolean algebra.
- **AND** outputs 1 only if both inputs are 1.
- **OR** outputs 1 if at least one input is 1.
- **XOR** (exclusive OR) outputs 1 if exactly one input is 1.

These basic logic gates are the foundation of digital circuits, and showing that neural networks can reproduce them illustrates how networks can perform fundamental computations.

AND Gate

For an **AND gate**, the output should only be 1 when both inputs are 1.

We can achieve this by choosing the weights and bias carefully. For example:

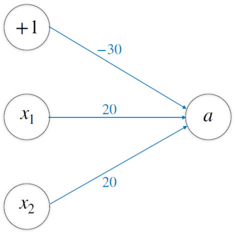
$$a = -30 + 20x_1 + 20x_2,$$

and then applying a step activation $g(a)$ that outputs 1 if $a > 0$, and 0 otherwise.

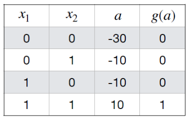
- If $x_1 = 0$ and $x_2 = 0$, then $a = -30$, so output is 0.
- If $x_1 = 1$, $x_2 = 0$, then $a = -10$, still negative, output is 0.
- If $x_1 = 0$, $x_2 = 1$, again $a = -10$, output is 0.

- If $x_1 = 1$, $x_2 = 1$, then $a = 10$, positive, so output is 1.

Thus, this neuron perfectly models the AND function.



Model: AND gate



x_1	x_2	a	$g(a)$
0	0	-30	0
0	1	-10	0
1	0	-10	0
1	1	10	1

OR Gate

For an **OR gate**, the output should be 1 if either input is 1.

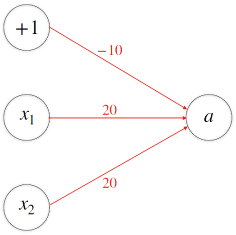
We can again pick weights and bias:

$$a = -10 + 20x_1 + 20x_2,$$

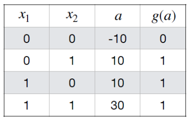
with the same step activation.

- If both inputs are 0, $a = -10$, output is 0.
- If $x_1 = 1$, $x_2 = 0$, then $a = 10$, output 1.
- If $x_1 = 0$, $x_2 = 1$, same: $a = 10$, output 1.
- If both are 1, then $a = 30$, definitely positive, output 1.

This neuron correctly implements the OR logic.



Model: OR gate



x_1	x_2	a	$g(a)$
0	0	-10	0
0	1	10	1
1	0	10	1
1	1	30	1

XOR Gate

The **XOR gate** (exclusive OR) is more challenging. The output should be 1 if exactly one input is 1, and 0 otherwise. A single linear threshold unit cannot capture XOR — this is exactly what Minsky and Papert pointed out in the 1960s.

But with a small neural network, we can do it.

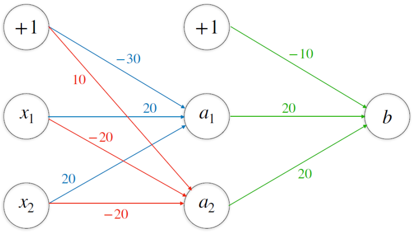
The construction works like this:
1. First, create hidden units that detect the AND and OR combinations in different directions.
2. Then combine them in a second layer so that the output matches XOR.

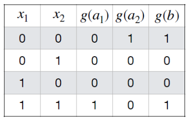
Mathematically, this involves setting weights so that:

- When $(x_1, x_2) = (0, 0)$ or $(1, 1)$, the network cancels to 0.

- When $(x_1, x_2) = (0, 1)$ or $(1, 0)$, the network activates to 1.

This is an early and very powerful demonstration:
- A single neuron can model simple linear decision boundaries (like AND and OR).
- But to model XOR — a nonlinear decision boundary — we need at least **two layers**.





x_1	x_2	$g(a_1)$	$g(a_2)$	$g(b)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

The key lesson: stacking layers of neurons allows us to represent functions that a single linear unit cannot capture. This is the heart of why neural networks are powerful.

Transforming to Outputs by the Activation Function

Up to this point, we introduced hidden units and showed how they transform inputs through linear combinations and nonlinearities. Now, we formalize the **full forward propagation** process for a neural network with one hidden layer.

Step 1: Input to Hidden Layer

We start with the input vector $x = (x_1, \dots, x_D)$.
We add a bias term $a_0^{(0)} = 1$, so we can write inputs compactly as $(a_0^{(0)}, a_1^{(0)}, \dots, a_D^{(0)})$.

Each hidden unit j computes a **pre-activation value**:

$$z_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} \, a_i^{(0)},$$

where $w_{ji}^{(1)}$ are the weights connecting input i to hidden unit j.

The activation of hidden unit j is then obtained by applying the hidden activation function $h(\cdot)$:

$$a_j^{(1)} = h(z_j^{(1)}).$$

So the hidden layer produces the vector of activations $a^{(1)}$.

Step 2: Hidden to Output Layer

Now, each output unit k again forms a linear combination of the hidden activations:

$$z_k^{(2)} = \sum_{j=0}^{M} w_{kj}^{(2)} \, a_j^{(1)},$$

where $a_0^{(1)} = 1$ is the bias unit for the hidden layer.

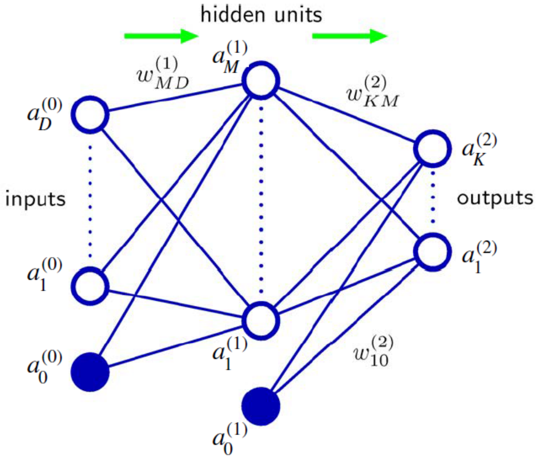
Step 3: Apply Output Activation

The network output is then obtained by applying the output activation function $\sigma(\cdot)$:

$$y_k(x, w) = \sigma(z_k^{(2)}).$$

Putting everything together, we can also expand the formula:

$$y_k(x, w) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)}\, h\left(\sum_{i=0}^{D} w_{ji}^{(1)}\, a_i^{(0)}\right)\right).$$



Step 4: Simplifying the Notation

Notice the repeated structure:
1. Compute a linear combination of inputs → z.
2. Apply an activation function → a.
3. Repeat this for each layer.

By defining:
- $z^{(l)}$ = vector of pre-activations at layer l,
- $a^{(l)}$ = vector of activations at layer l,

we can describe the whole process layer by layer:

$$z^{(l)} = W^{(l)} a^{(l-1)}, \quad a^{(l)} = h(z^{(l)}).$$

At the final layer:

$$y = \sigma(z^{(L)}),$$

where L is the output layer.

This compact matrix notation makes it much easier to work with deeper networks.

In summary, this process takes us from input to hidden layer to output, step by step, and introduces the general notation used for **forward propagation** in neural networks.

Neural Networks – Training

So far, we have seen how to construct the outputs of a neural network by propagating data forward through the layers. But that's only half the story. The real question is: **how do we choose the weights w?**

That's the training problem. We want to set the weights so that the predictions $y(x, w)$ match the true targets t. To do this, we need to:
1. Define an **error (loss) function**, which measures the mismatch between predictions and targets.
2. Adjust the weights to minimize this error.

Weight space symmetries and error functions

There are some subtleties here:

- Neural networks have **symmetries in weight space**. For example, hidden units can be permuted without changing the function. Also, for nonlinearities like tanh, we have identities such as $\tanh(-a) = -\tanh(a)$, which means flipping signs of weights can lead to equivalent solutions.

- This implies there are many equivalent weight configurations that produce the same outputs. So training does not search for a *single* unique solution, but for any weight configuration that minimizes the error.

Now, how do we define the error? It depends on the type of problem:
- **Regression:** use linear outputs and a sum-of-squares error.
- **Binary classification:** use a logistic sigmoid output and cross-entropy error.
- **Multiclass classification:** use a softmax output and multiclass cross-entropy error.

Regression error

For regression, the natural choice is the **sum-of-squares error function**:

$$E(w) = \tfrac{1}{2} \sum_{n=1}^{N} \|y(x_n, w) - t_n\|^2.$$

This measures how far the predicted outputs are from the true targets, squared and summed over all training data points. The factor $\frac{1}{2}$ is conventional — it cancels out the 2 when we take derivatives later.

Probabilistic interpretation of regression

We can also interpret regression probabilistically. Suppose the outputs are modeled as Gaussian-distributed around the network prediction:

$$p(t \mid x, w) = \mathcal{N}(t \mid y(x, w), \beta^{-1}),$$

where β is the precision (inverse variance).

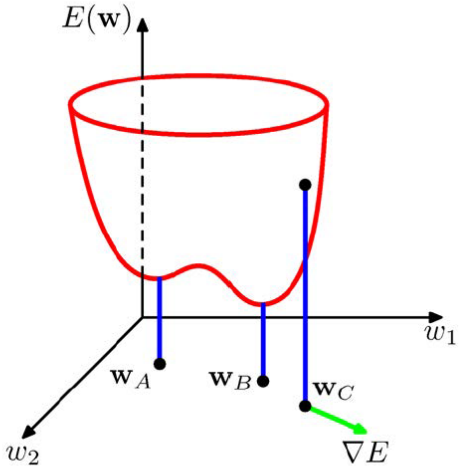
For a whole dataset, the likelihood is:

$$p(t \mid X, w, \beta) = \prod_{n=1}^{N} p(t_n \mid x_n, w, \beta).$$

Taking the negative log-likelihood gives us:

$$-\ln p(t \mid X, w, \beta) = \frac{\beta}{2} \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2 - \frac{N}{2} \ln \beta + \frac{N}{2} \ln(2\pi).$$

Notice: the first term is just proportional to the **sum-of-squares error** we defined earlier. This shows the close connection between maximum likelihood under a Gaussian noise assumption and least-squares regression.



Training a neural network means **minimizing an error function**, and for regression that error naturally arises from both a geometric (least squares) and probabilistic (Gaussian likelihood) perspective.

Neural Networks – Training with Gradient Descent and Backpropagation

So far, we have set up the structure of a neural network and defined error functions. Now the question is: **how do we actually minimize these error functions?**

The standard approach is **gradient descent**.

Gradient descent

The basic rule is: take a step in the opposite direction of the gradient of the error with respect to the weights.

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}),$$

where:
- τ is the iteration number,
- η is the learning rate, and
- $\nabla E(w)$ is the vector of partial derivatives of the error with respect to the weights.

Many error functions in neural networks can be written as sums over the data points:

$$E(w) = \sum_{n=1}^{N} E_n(w),$$

where $E_n(w)$ is the error contribution from data point n.

This allows us to define **stochastic gradient descent (SGD)**: instead of computing the gradient on the entire dataset at once, we approximate it using a single data point (or a small batch).

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E_n(w^{(\tau)}).$$

This is much cheaper computationally when N is large, and it introduces some randomness that can actually help escape local minima.

Simple case: linear model

Suppose we have a linear model:

$$y_k = \sum_i w_{ki} x_i.$$

If we define the squared error for a single data point:

$$E_n = \tfrac{1}{2} \sum_k (y_{nk} - t_{nk})^2,$$

then the derivative of the error with respect to a weight is:

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj})\, x_{ni}.$$

This shows the gradient has a simple structure: the error at the output times the input value.

Generalizing to hidden units

Now let's make it more general.

Consider a hidden unit j that computes:

$$z_j = \sum_i w_{ji} a_i,$$

where a_i are the activations from the previous layer.

If we want to compute the derivative of the error with respect to a weight:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}}.$$

Since $\frac{\partial z_j}{\partial w_{ji}} = a_i$, this gives us:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j a_i,$$

where we define the **error signal** for unit j as:

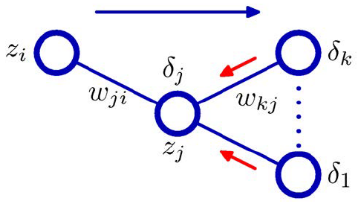
$$\delta_j = \frac{\partial E_n}{\partial z_j}.$$

Hidden unit error signals

If $a_j = h(z_j)$, then by the chain rule:

$$\delta_j = h'(z_j) \sum_k w_{kj} \delta_k.$$

This shows how the error signal at a hidden unit j depends on:
- the derivative of the activation function at its input ($h'(z_j)$), and
- the weighted sum of the error signals from the next layer.



This is the essence of the **backpropagation step**.

Backpropagation algorithm

We can now describe the full algorithm for a single data point:

1. **Forward pass:**
   - Apply input vector x_n to the network.
   - Compute all activations a_j and pre-activations z_j.
2. **Output errors:**
   - Compute the error signals for the output units:

$$\delta_k = y_k - t_k.$$

3. **Hidden errors:**
   - Backpropagate the errors to hidden units:

$$\delta_j = h'(z_j) \sum_k w_{kj} \delta_k.$$

4. **Weight derivatives:**
   - Compute the derivatives of the error with respect to weights:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j a_i.$$

This completes the chain: **forward propagation** computes outputs, and **backpropagation** computes gradients. With these gradients, we can apply gradient descent to adjust all the weights.

Neural Networks – Example: Regression with $\tanh$ Activation Functions

Now that we've introduced backpropagation, let's see how it works in practice with a concrete example.
We consider a regression problem, so the **output layer is linear**. The hidden units, however, use the **tanh activation function**.

Forward propagation

1. **Inputs with bias:**

$$a^{(0)} = (a_0^{(0)} = 1,\ a_1^{(0)}, \dots, a_D^{(0)}).$$

2. **Hidden pre-activations:**

$$z_j^{(1)} = \sum_{i=0}^{D} w_{ji}^{(1)} a_i^{(0)}.$$

3. **Hidden activations (tanh):**

$$a_j^{(1)} = h(z_j^{(1)}) = \tanh(z_j^{(1)}).$$

Recall:

$$h(z) = \tanh(z), \quad \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad h'(z) = 1 - h(z)^2.$$

4. **Output pre-activations:**

$$z_k^{(2)} = \sum_{j=0}^{M} w_{kj}^{(2)} a_j^{(1)}.$$

5. **Outputs (linear regression):**

$$y_k = z_k^{(2)}.$$

Error function

We use the **sum-of-squares error**:

$$E = \tfrac{1}{2} \sum_{i=1}^{N} (y_i - t_i)^2.$$

$$\boxed{x_k = a_k^{(0)} \xrightarrow{\sum_{i=0}^{D} w_{ji}^{(1)} a_i^{(0)}} z_j^{(1)} \xrightarrow{\tanh(z_j^{(1)})} a_j^{(1)} \xrightarrow{\sum_{j=0}^{M} w_{kj}^{(2)} a_j^{(1)}} z_k^{(2)} = y_k \xrightarrow{\frac{1}{2}\sum_{i=1}^{N} (y_i - t_i)^2} E}$$

Output layer error

To start backpropagation, we compute the error signal at the output:

$$\frac{\partial E}{\partial z_k^{(2)}} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_k^{(2)}}.$$

Since $y_k = z_k^{(2)}$, this simplifies to:

$$\delta_k^{(2)} = \frac{\partial E}{\partial z_k^{(2)}} = y_k - t_k.$$

So the output error is simply the difference between prediction and target.

Propagating back to hidden layer

1. Derivative of the output with respect to a hidden activation:

$$\frac{\partial z_k^{(2)}}{\partial a_l^{(1)}} = w_{kl}^{(2)}.$$

2. Derivative of a hidden activation with respect to its pre-activation:

$$\frac{\partial a_l^{(1)}}{\partial z_l^{(1)}} = 1 - \tanh^2(z_l^{(1)}) = 1 - (a_l^{(1)})^2.$$

3. Hidden error signal:

$$\delta_l^{(1)} = (1 - (a_l^{(1)})^2) \sum_{k=1}^{K} w_{kl}^{(2)} \, \delta_k^{(2)}.$$

Gradients for weights

- For the weights from hidden to output:

$$\frac{\partial E}{\partial w_{kj}^{(2)}} = \delta_k^{(2)} \, a_j^{(1)}.$$

- For the weights from input to hidden:

$$\frac{\partial E}{\partial w_{ji}^{(1)}} = \delta_j^{(1)} \, a_i^{(0)}.$$

Key takeaway

- The forward pass computes activations $a^{(l)}$.
- The backward pass computes error signals $\delta^{(l)}$.
- Gradients of weights are products of activations and error signals.

This example shows how everything we discussed comes together: forward propagation, error calculation, and backpropagation, all in explicit equations for the case of tanh hidden units and linear outputs.