CS 5600/6600: F20: Lecture 2
Artificial Neural Networks (ANNs): Part 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

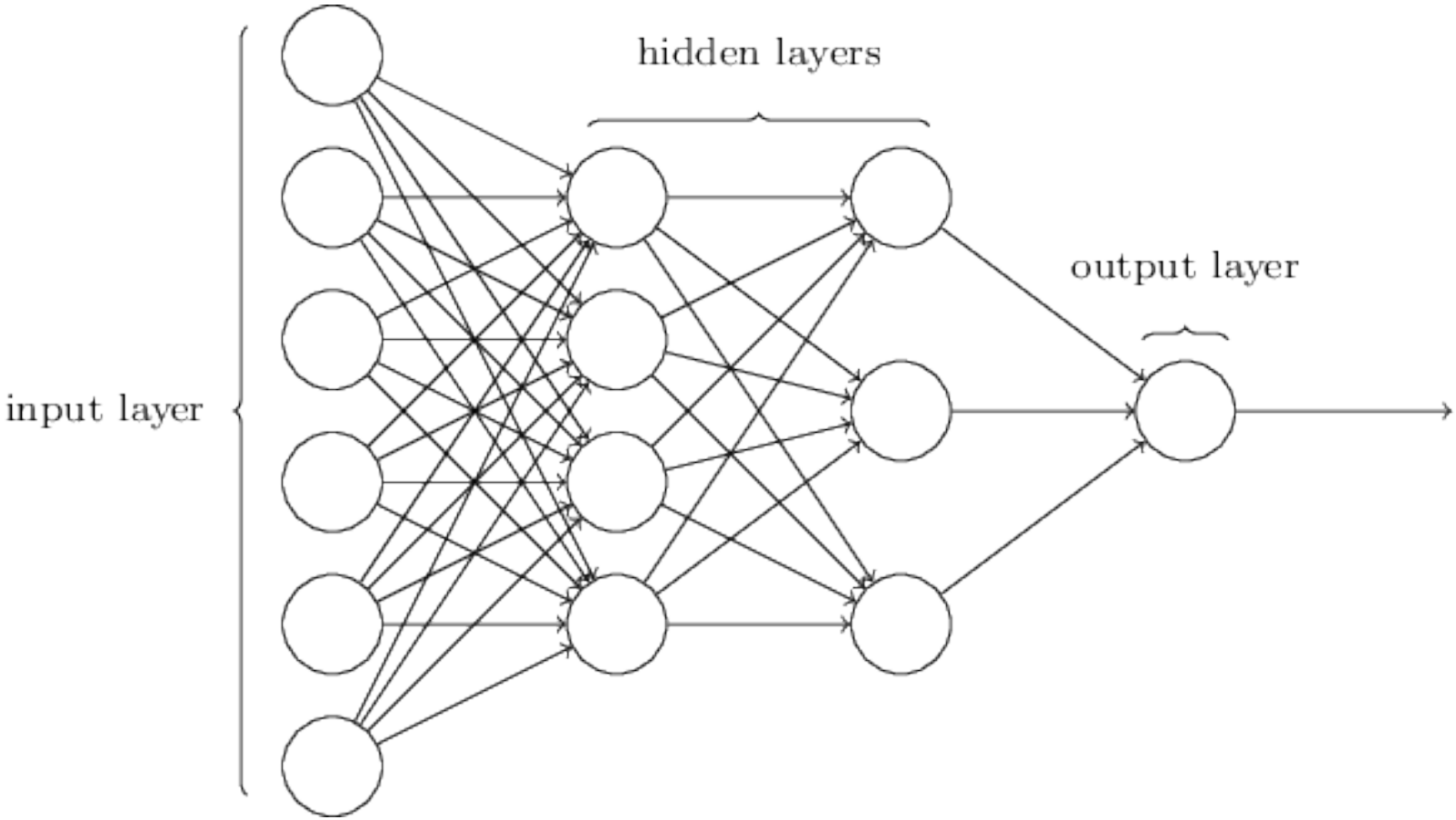
Layers

Neural networks consists of layers. Each layer consists of neurons (either perceptrons or sigmoids).

There can be arbitrarily many layers stacked on each other. The leftmost layer is called the *input* layer.

The righmost layer is called the *output* layer. The layers in between the input layer and the output layer, if there are any, are called the *hidden* layers.

Layers



Terminological Note

Multilayer ANNs are sometimes called *multilayer perceptrons* (MLPs) although they are made up of sigmoid neurons, not perceptrons. The term MLP can be confusing.

Synapses and Neurons

The links between neurons, called *synapses*, indicate which neurons input their outputs to which other neurons. These links have weights.

The job of a neuron is to take the inputs from the synapses and apply its activation function to them.

The job of a synapse is to take the value from the input neuron, multiply the value by its weight, and output the result into the output neuron.

Parameters vs. Hyperparameters

The neural networks differ from each other in terms of their layers, types of neurons, synapses, and activation functions. These are called *hyperparameters*. Hyperparameters cannot be changed.

The parameters of a neural network are weights and biases that can be manipulated and/or learned.

The neural networks also differ from each other by their learning algorithms.

Designing Neural Networks: Input and Output Layers

The design of input and output layers is often straightforward. For example, if you're classifying 32×32 grayscale images, then it makes sense to have $32 \times 32 = 1024$ input neurons, each of which is a number between 0 and 255. If you're classifying 64×64 grayscale images, you should have $64 \times 64 = 4,096$ neurons.

If you're recognizing digit images, it makes sense to have 10 output nodes - one for each digit.

If you're recognizing the presence/absence of a car in an image - 1 or 2 output neurons are enough.

Designing Neural Networks: Hidden Layers

There are no accepted rules for designing hidden layers. There are some heuristics that NN researchers use but none of them are universally accepted.

The term *deep learning* refers to neural networks with many hidden layers.

Outline

Building an ANN

Suppose we want to predict how age and job experience correlate with salaries.

How many input nodes, output nodes, and hidden nodes should we have? What kinds of nodes? How are they connected? How are the synapse weights initialized?

Building an ANN

We'll use sigmoid nodes, not perceptrons, and we'll have full connectivity between consecutive layers.

We need two input nodes (age and years of job experience), and one output node (salary).

Let's have 3 hidden nodes. Why 3? Why not? There's no mathematically justifiable reason for 3 hidden nodes. I've chosen it for simplicity.

Building an ANN: Training Data

What data do we need for this ANN?

We need lots of 3-tuples (x_i, x_j, x_k), where x_i is a person's age, x_j is the person's years of experience, and x_k is the person's salary (i.e., this is the ground truth). Let call it Y.

Building an ANN

The input to the neural network will be 2-tuples (x_i, x_j) extracted from 3-tuples (x_i, x_j, x_k). Let's call it X.

The output to the neural network is $\widehat{Y}$. Note that $\widehat{Y}$ is our network's approximation to Y.

For each $X = (x_i, x_j)$, we want to compare how far the output of our network, called $\widehat{Y}$, is from $Y = x_k$, i.e., the ground truth.

Building an ANN: Initializing Weights

OK. We've figured out what data we need for our ANN.

The next question question is – how do we initialize weights?

Building an ANN: Initializing Weights

If we know from the literature that there's an existing ANN that solves a similar problem, we can use it as is or use some weights from it. This is called *transfer learning*.

If we don't know or don't want to do any research, we can simply initialize our weights randomly, which is what we'll do.

Building an ANN: Initializing Weights

Let's use `numpy.random.rand(x, y)` that creates x, y numpy arrays of normally distributed random numbers with a mean of 0 and a variance of 1.

```
 >>> import numpy as np
 >>> np.random.randn(2, 2)
array([[ 0.35120796,  0.19578908],
       [-0.20131598,  0.06917832]])
 >>> np.random.rand(2, 3)
array([[ 0.92847923,  0.10751104,  0.64730707],
       [ 0.81078793,  0.52196304,  0.59270607]])
```

Building an ANN: Initializing 2 x 3 x 1 ANN

Here's a Py function that builds our ANN.

```
import numpy as np

def build_my_nn():
    # 1. seed random number generator
    np.random.seed(1)
    # 2. initialize 1st weight matrix
    W1 = np.random.randn(2, 3)
    # 3. initialize 2nd weight matrix
    W2 = np.random.randn(3, 1)
    # 4. return 2-tuple of weight matrices
    return W1, W2
```

Building an ANN: Feedfoward From Input to Output

How do we compute $\widehat{Y}$ from $X = (x_i, x_j)$? This process of pushing the input through the network from the input to the output layer is called *feedforward*.

Feedforward in our $2 \times 3 \times 1$ network is defined by the following four equations:

1. $Z^{(2)} = XW^{(1)}$; $Z^{(2)}$ is the input to layer 2;
2. $a^{(2)} = f(Z^{(2)})$; $a^{(2)}$ is the output of layer 2;
3. $Z^{(3)} = a^{(2)}W^{(2)}$; $Z^{(2)}$ is the input to output layer (layer 3);
4. $\widehat{Y} = f(Z^{(3)})$; $\widehat{Y}$ is the output of output layer (layer 3),

where $W^{(1)}$ is the matrix of weights on the synapses connecting layer 1 (input) to layer 2 (hidden), $f(z)$ is the activation function, and $W^{(3)}$ is the matrix of weight on the synapses connecting layer 2 (hidden) to layer 3 (output).

Building an ANN: Feedforward: Equations 1 and 2

Equations 1 and 2:

$$Z^{(2)} = XW^{(1)}.$$

$$a^{(2)} = f(Z^{(2)}).$$

Here is Python (sigmoid is a function that we'll need to implement or use from its implementation from a 3rd-party library):

```
Z2 = np.dot(X, W1)
a2 = sigmoid(Z2)
```

Building an ANN: Feedforward: Equations 3 and 4

Equations 3 and 4:

$$Z^{(3)} = a^{(2)} W^{(2)}.$$

$$\widehat{Y} = f(Z^{(3)}).$$

Here is Python:

```
Z3 = np.dot(a2, W2)
yHat = sigmoid(Z3)
```

Building an ANN: Complete Feedforward in 2 x 3 x 1 ANN

```
## equation 1
Z2 = np.dot(X, W1)
## equation 2
a2 = sigmoid(Z2)
## equation 3
Z3 = np.dot(a2, W2)
## equation 4
yHat = sigmoid(Z3)
```

Outline

Error/Cost Function

We have to decide how far our output $\widehat{y}$ is from the ground truth y. We can use this simple formula.

$$\widehat{y}_{err} = y - \widehat{y}.$$

If negative weights cause numerical instability, we can use

$$\widehat{y}_{err} = |y - \widehat{y}|$$

or

$$\widehat{y}_{err} = (y - \widehat{y})^2.$$

Quadratic Cost Function or Mean Squared Error

A more common way to measure how far the actual output of the network a is from the desired output $y(x)$ is to use the quadratic cost function, aka mean squared error (MSE):

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2.$$

The aim of ANN training is to minimize the cost $C(w, b)$.

Changing Weights One at a Time

We can change the weights of our NN by adjustring one weight at a time. Is this a good idea?

Curse of Dimensionality

Changing weights one at a time is too expensive. This is known as the **curse of dimensionality**. We have to be smarter.

1st Handshake with Backpropagation

Backpropagation is an algorithm of adjusting synapse weights on the basis of the output error, i.e., $\widehat{y}_{err}$.

In our network, we have to compute the adjustments for 2 layers, the output layer and the hidden layer. No reason to tinker with the input layer, because it is what it is.

The adjustment for the output layer is used to adjust $W^{(2)}$. The adjustment for the hidden layer is used to adjust $W^{(1)}$.

Backpropagation: Adjustment for Output Layer

The adjustment for the output layer is used to adjust $W^{(2)}$.

$$\widehat{y}_{err} = y - \widehat{y}$$

$$\widehat{y}_{\Delta} = \widehat{y}_{err} \cdot \sigma'(\widehat{y})$$

Python code is below. The function `sigmoid_deriv()` implements the derivative of $\sigma(x)$, i.e., $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

```
yHat_error = y - yHat
yHat_delta = yHat_error * sigmoid_deriv(yHat)
```

Backpropagation: Adjustment for Hidden Layer

The adjustment for the hidden layer is used to adjust $W^{(1)}$.

$$a_{err}^{(2)} = \widehat{y}_{\Delta} \cdot (W^{(2)})^{T}$$

$$a_{\Delta}^{(2)} = a_{err}^{(2)} \cdot \sigma'(a^{(2)}).$$

In Python:

```
a2_error = yHat_delta.dot(W2.T)
a2_delta = a2_error * sigmoid_deriv(a2)
```

Backpropagation: Adjusting $W^{(1)}$ and $W^{(2)}$

$$W^{(2)} = W^{(2)} + (a^{(2)})^T \cdot \widehat{y}_\Delta$$

$$W^{(1)} = W^{(1)} + X^T \cdot a_\Delta^{(2)}.$$

In Python:

```
W2 += a2.T.dot(yHat_delta)
W1 += X.T.dot(a2_delta)
```

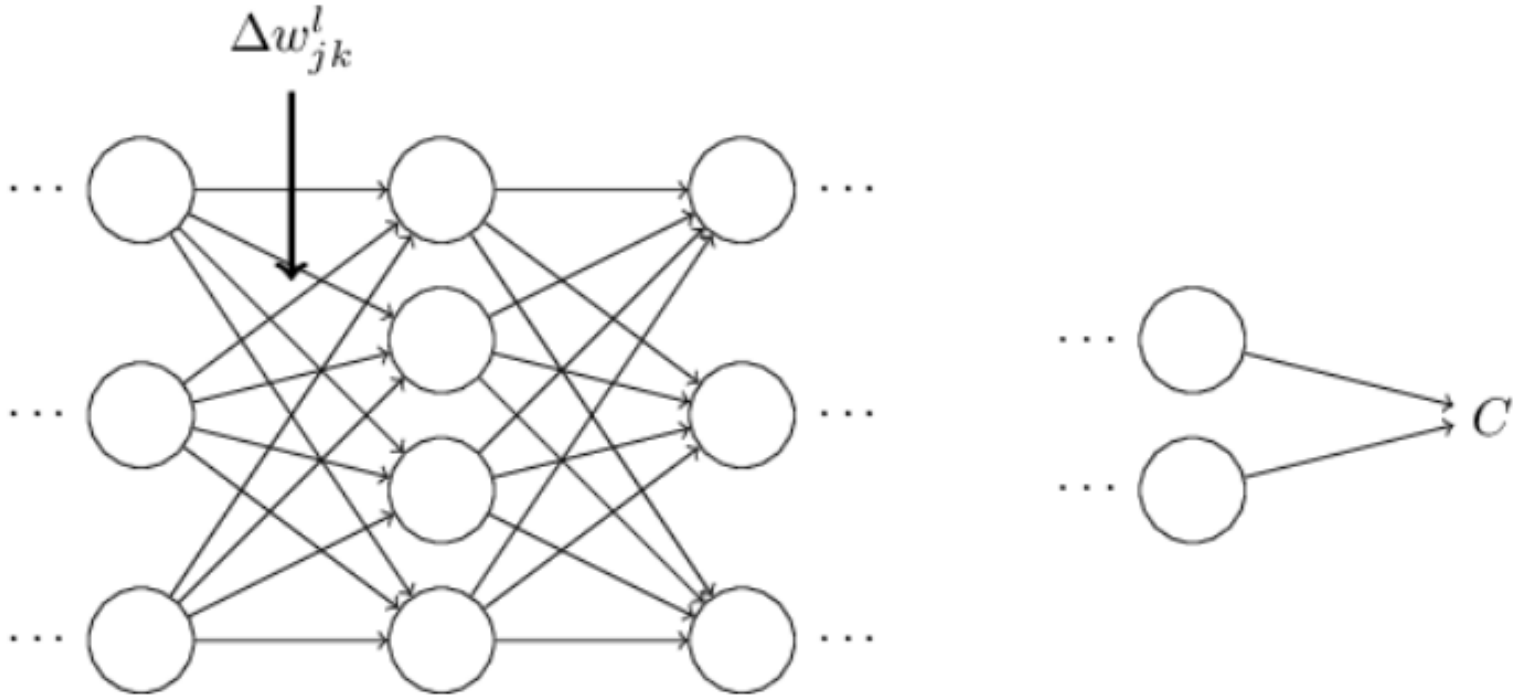
Feedforward and Backprop: Training 3-layer ANN

```
W1, W2 = build_my_nn()
numIters = 1000
for j in range(numIters):
    # Feedforward
    Z2 = np.dot(X, W1)
    a2 = sigmoid(Z2)
    Z3 = np.dot(a2, W2)
    yHat = sigmoid(Z3)
    # Backprop
    yHat_error = y - yHat
    yHat_delta = yHat_error * sigmoid(yHat, deriv=True)
    a2_error = yHat_delta.dot(W2.T)
    a2_delta = a2_error * sigmoid(a2, deriv=True)
    W2 += a2.T.dot(yHat_delta)
    W1 += X.T.dot(a2_delta)
```

Outline

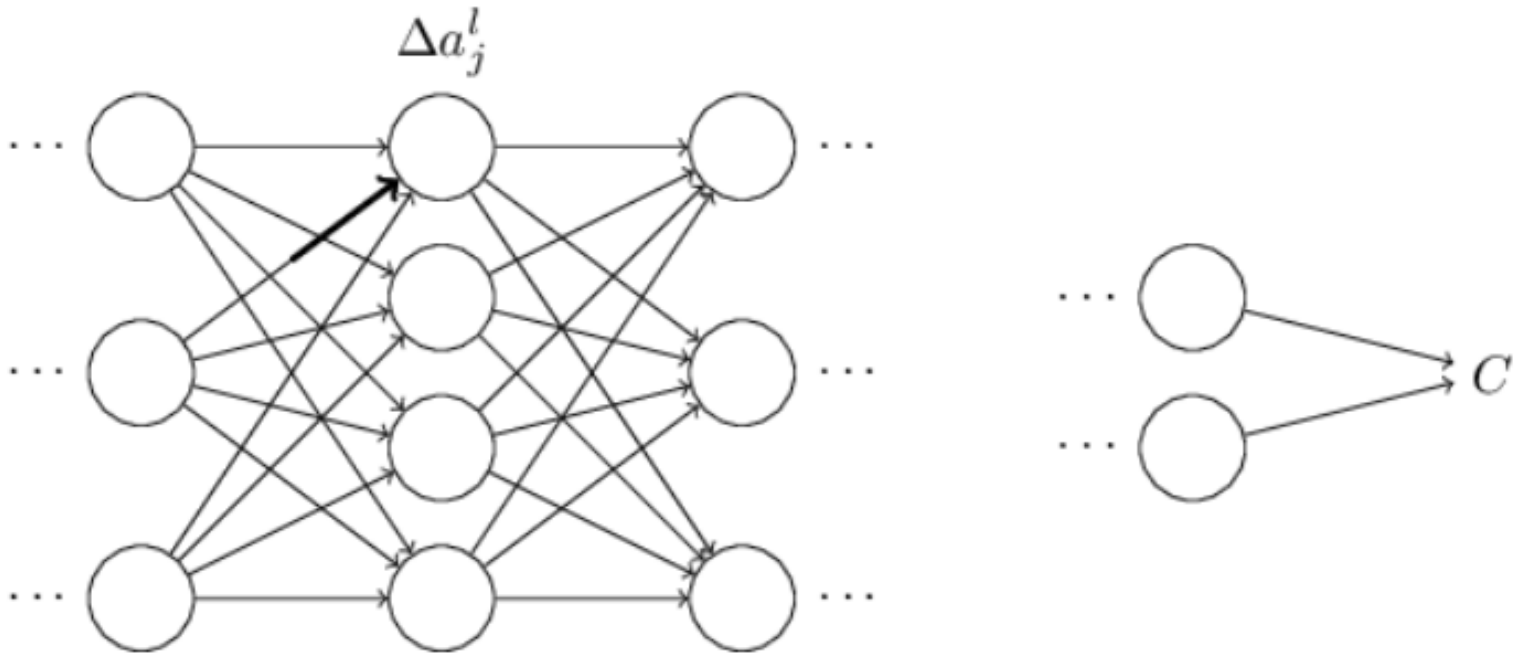
Weight Change: Δw^l_{jk}

Let's assume that we'll make a change Δw^l_{jk} in weight w^l_{jk}:



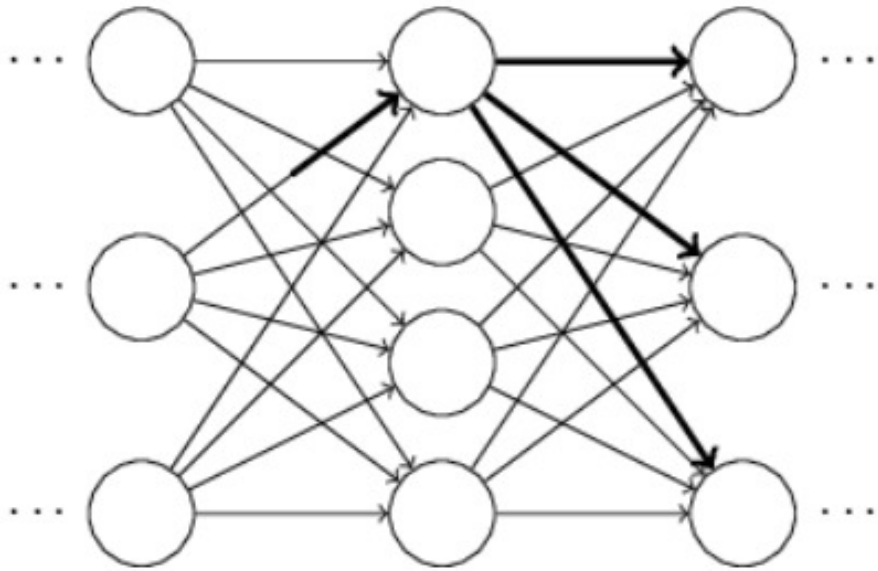
Activation Change: Δa_j^l

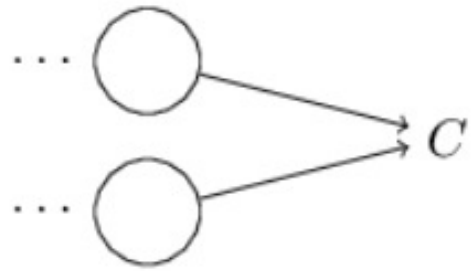
The change in weight Δw_{jk}^l causes a change in the output activation in the corresponding neuron Δa_j^l:



Change in All Activations

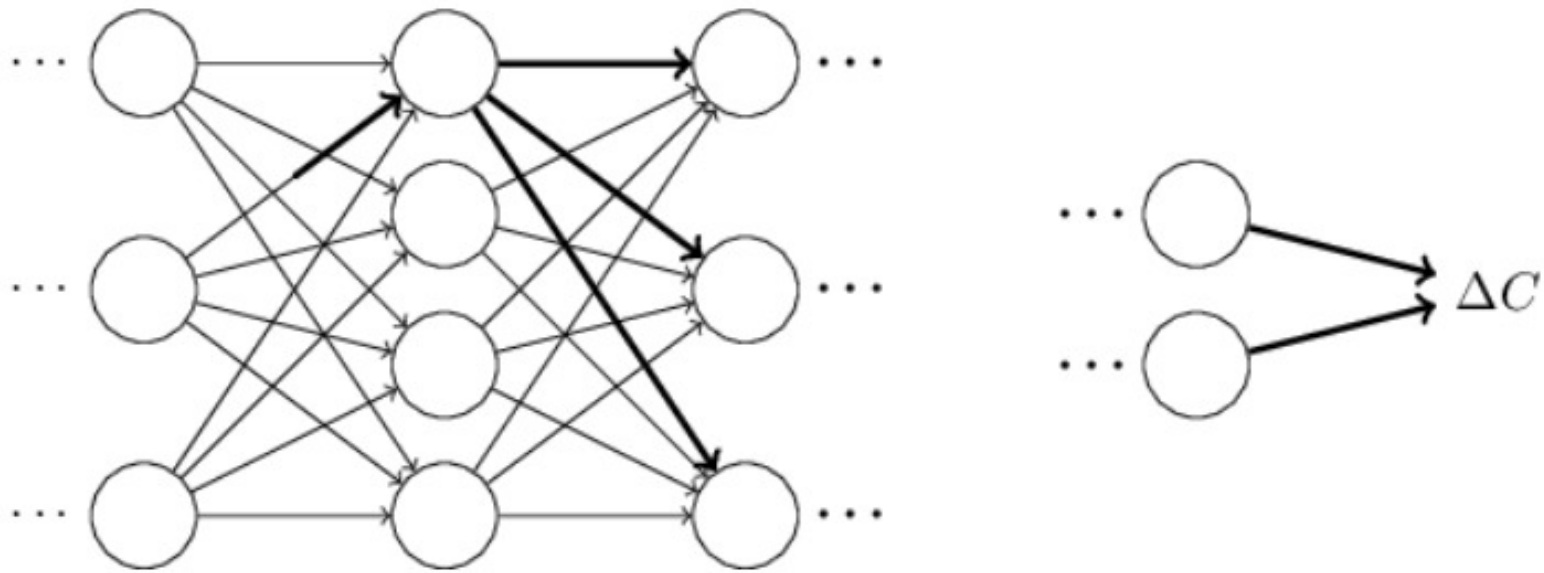
The change in activation in one layer causes a change in all activations in the next layer:





Change in All Activations

The change in all activations in one layer will percolate all the way to the final layer and the cost function:



Change in Cost Function

The change in all activations in one layer will percolate all the way to the final layer and the cost function:

$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}.$$

Change in Activation

The change Δw_{jk}^l causes Δa_j^l. This change is

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

Change in Activations

The change in activation Δa_j^l causes changes in all activations at layer $(l+1)$. In particular, for neuron q in layer $(l+1)$:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l.$$

Let's substitute

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

in the above equation to obtain:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

A Single Activation Path

Let's take a path of activations $a_j^l, a_q^{l+1}, ..., a_n^{L-1}, a_m^L$. Then

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l.$$

Activation Along All Activation Paths

The total change in C must take into account all activation paths. Thus,

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l$$

Derivation of Weight Gradient

We have

$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}$$

and

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \cdots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}.$$

Hence,

$$\frac{\partial C}{\partial w^l_{jk}} \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a^L_m} \frac{\partial a^L_m}{\partial a^{L-1}_n} \frac{\partial a^{L-1}_n}{\partial a^{L-2}_p} \cdots \frac{\partial a^{l+1}_q}{\partial a^l_j} \frac{\partial a^l_j}{\partial w^l_{jk}}.$$

References

1. T.M. Mitchell. *Machine Learning*.
2. M. Neilsen. *Neural Networks and Deep Learning*.