23 October, 2017

Neural Networks

Course 4: Making the neural network more efficient

Overview

- The Problem With Quadratic Cost
- Cross Entropy
- Softmax
- Weight initialization
- How to adjust hyper-parameters
- Conclusions

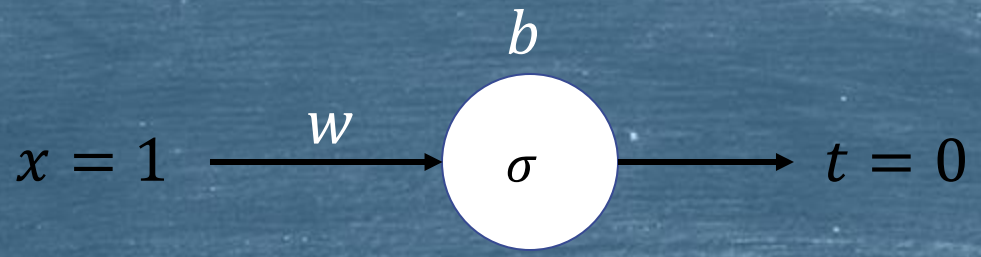
The Problem With Quadratic Cost

The Problem With Quadratic Cost

- On the last course we have used the Mean Square Error as our cost function
- Even though we have achieved a good accuracy using this cost function, this is not the best one since learning can be slow
- An important feature that we want from a neuron (and from a neural network) is to learn fast. For this to work, the weights must be lowered in direct proportion to how big the error is.
  - If the error is big, then big adjustment must be made in order to drive the cost down
  - If the error is small, then we want to make small adjustments to not overshoot our target

The Problem With Quadratic Cost

- A small experiment.
  - We will take a neuron with only one input (one weight) and one bias. The input will always be 1.
  - The role of the neuron is to find the weights that make the output zero. So drive the 1 to zero



  - We will test the network in two variants:
    - $w = 0.6\ b = 0.9\ .z = 1.5.\sigma(1.5) = 0.81$
    - $w = 2\ b = 2\ .z = 4.\sigma(4) = 0.98$
  - Observe how fast the value drops to 0 for each case

The Problem With Quadratic Cost

- As it can be observed, when the error is big (second case) the learning is slower. This is the opposite as what we want.

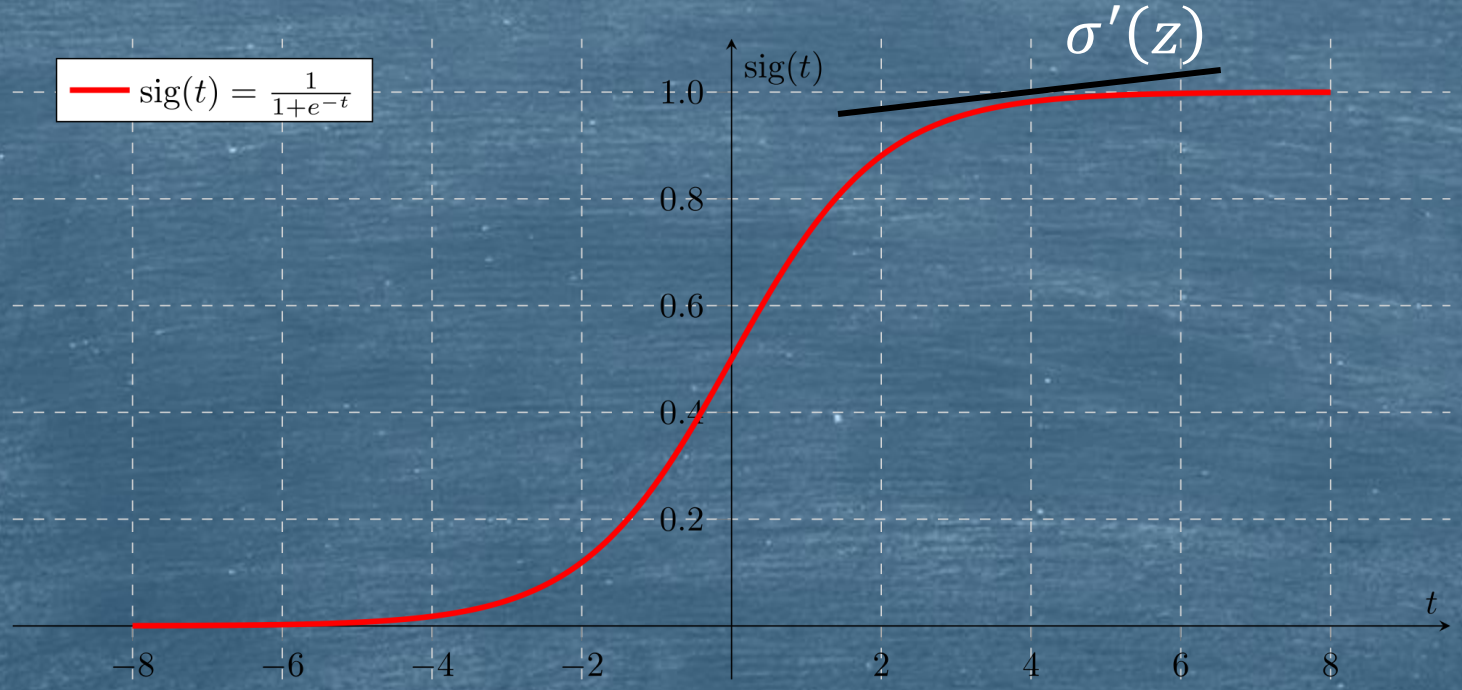
- Why is this happening?
  - The cost $C = \frac{(t-y)^2}{2}$
  - The weight and bias are adjusted according to the formula :

$$w = w - \eta \frac{\partial C}{\partial w} \qquad \frac{\partial C}{\partial w} = \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial w} = \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w} = (y - t)\sigma'(z)x = a\sigma'(z)(for\ our\ case)$$

$$b = b - \eta \frac{\partial C}{\partial b} \qquad \frac{\partial C}{\partial b} = \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial b} = \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial b} = (y - t)\sigma'(z) = a\sigma'(z)(for\ our\ case)$$

The Problem With Quadratic Cost

- So, how the error changes in respect to the weight or the bias, depends on $\sigma'(z)$



- For large values of z, the function is almost flat. So the derivate is very small. Thus, learning is slow. In this case, we say that the neuron has saturated on the wrong value

Cross Entropy

Cross Entropy

- One of the way to solve the slow learning problem is to change the cost function.
- We want a function that its derivate does not contain the $\sigma'(z)$
- For a neuron with multiple inputs (vector x) and an output (y), the cross entropy is defined as:

$$C = -\frac{1}{n}\sum_x [tlny + (1-t)\ln(1-y)]$$

where:

n =number of training items
x = a training item
y = activation for item x
t = expected output for item x

The sum is over all training items.

Cross Entropy

- Does this function fix our problem?

$$C = -\frac{1}{n}\sum_{x}[tlny + (1-t)\ln(1-y)]$$

First, observe that is behaves like a cost function:

- since y∈ [0,1] that means that lny and $\ln(1-y)$ are negative and the output is multiplied by a negative number. So it results a positive number
- When $t = 0\ and\ y \approx 0$, then the sum is 0 (or very close to 0)
- When $t = 1\ and\ y \approx 1$, then the sum is 0 (or very close to 0)
- When $t = 1$ the function depends on ln(y). ln is a monotonic function

So this seems to work like a cost function, but how do $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ look like?

Cross Entropy

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_i} = -\frac{1}{n}\sum_x \frac{\partial [t ln y + (1-t)\ln(1-y)]}{\partial y} \cdot \sigma'(z) \cdot x_i$$

$$\frac{\partial [t ln y + (1-t)\ln(1-y)]}{\partial y} = \frac{t}{y} + \frac{1-t}{1-y} \cdot (1-y)' = \frac{t}{y} - \frac{1-t}{1-y}$$

$$\sigma'(z) = (1-y)y$$

$$\frac{\partial C}{\partial w_i} = -\frac{1}{n}\sum_x (\frac{t}{y} - \frac{1-t}{1-y}) \cdot (1-y) \cdot y \cdot x_i = -\frac{1}{n}\sum_x (t(1-y) - (1-t)y)x$$

- $\frac{\partial C}{\partial w_i} = -\frac{1}{n}\sum_x (t-y)x$
- $\frac{\partial C}{\partial b} = -\frac{1}{n}\sum_x (t-y)$

Cross Entropy

- We will try to repeat the previous experiment, but this time with the cross entropy function.
- A thing that we must also change is the learning rate. Learning rate is dependent on the cost function. Changing the learning rate is not cheating since we are interested in how the learning speed changes and not how fast it is learning.

Cross Entropy

- By now we have been using a cost function for only one output. Of course, this can be generalized:

$$C = -\frac{1}{n}\sum_x \sum_j [t_j ln y_j^L + (1 - t_j)\ln(1 - y_j^L)]$$

- The error in the final layer, $\frac{\partial C}{\partial z_j^L}$ becomes

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial y_i^L} \cdot \frac{\partial y_i^L}{\partial z_i^L} = -\frac{1}{n}\sum_x \left(\frac{t_j}{y_j^L} - \frac{1 - t_j}{1 - y_j^L}\right)\left(1 - y_j^L\right)y_j^L = -\frac{1}{n}\sum_x (t_j^L - y_j^L)$$

$$\frac{\partial C}{\partial z_j^L} = \frac{1}{n}\sum_x (y_j - t_j)$$

Cross Entropy

- Of course, since we will be using Stochastic Gradient Descent, we will not divide the cost of each element by the total dataset (n), but by the length of the mini batch(m)

- So, what we actually need to change in the backpropagation algorithm to make this work, is just how the error is computed

$$\nabla_a C = \frac{\partial C}{\partial z^L} = y^L - t$$

(more on)Cross Entropy

- Where did the function come from?! (it looks very complicated at first sight)
- Consider that we have a system (neural network) that has some configuration and must classify the inputs to several m classes (ex. Digits).
- The probability of classifying an input to each class is y_j, where $\sum_j y_j = 1$
- Let's suppose that for our dataset we have k_j elements for each j class. According, to the model, the likelihood of this happening is:

$$\mathrm{P}(\mathrm{data}|\mathrm{model}) = y_1^{k_1} y_2^{k_2} \dots y_m^{k_m}$$

(more on)Cross Entropy

If we apply the logarithm function, then

$$\ln(\text{P}(\text{data}|\text{model})) = \ln(y_1^{k_1} y_2^{k_2} \dots y_m^{k_m}) = \ln(y_1^{k_1}) + \ln(y_2^{k_2}) + \cdots + \ln(y_m^{k_m}) =$$
$$= \sum_j k_j \ln(y_j)$$

Using the logarithm function, we have some advantages:

- It's a monotonic function.
- Transforms the product into a sum
- Logarithm of a very small number is a negative number

(more on)Cross Entropy

Obviously, we want to increase this probability, but since we're used to minimizing a cost function, we'll minimize the same function but with the opposite sign (-)

If we divide by the number of elements in the dataset (n), then $\frac{k_j}{n}$ becomes the true probability of the elements of each class.

Another Formula for Cross Entropy: $-\sum_j p_j \ln(y_j)$

(more on)Cross Entropy

If we divide by the number of elements in the dataset (n), we'll have:

$$-\frac{1}{\mathrm{n}}\sum_j k_j \ln(y_j)$$

Since the output vector (t) is a one-hot element (only one of its elements has value 1, the others have value 0. Ex, for digits: 0, 0, 0, 1, 0 , 0, 0, 0, 0, 0)

$$k_j = \sum_x t_j$$

The above equation becomes:

$$-\frac{1}{n}\sum_j(\sum_x t_j)\ln(y_j)$$

(more on)Cross Entropy

Standard formula for Cross Entropy :

$$C = -\frac{1}{n}\sum_x \sum_j t_j \ln(y_j)$$

If the number of possible classes is just 2, we can really use just one output. The above formula becomes

$$\mathrm{C} = -\frac{1}{n}\sum_x [tlny + (1-t)\ln(1-y)]$$

(more on)Cross Entropy

Another often used cost function, when doing online training is to use

$$C = -\ln(y_j)$$

Of course, this is still the cross entropy, but in a more simplified version that takes account for the fact that t_j=1 for the right label and 0 for the rest. (one hot)

Softmax

Softmax

When we've classified the MNIST digits we didn't consider the outputs as probabilities, yet we've used cross entropy which works with probabilities.

The only thing that must be changed is the output layer.

Instead of outputting $\mathrm{y}_\mathrm{j}^L = \sigma(z), where\ z = \sum_k w_{jk}^L y_k^{L-1} + b_j^L$ we'll compute a probability using z.

More exactly,

$$y_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

Softmax

How does $\frac{\partial C}{\partial z_j^L}$ looks like?

$$y_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

$$C = -\frac{1}{n}\sum_x \sum_j t_j \ln(y_j)$$

Softmax

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial y_i^L}\frac{\partial y_i^L}{\partial z_j^L} = -\frac{1}{n}\sum_x \sum_i \frac{\partial t_i}{\partial y_i^L}\frac{\partial y_i^L}{\partial z_j^L}$$

$$\frac{\partial y_i^L}{\partial z_j^L} = \frac{\partial \frac{e^{z_i^L}}{\sum_k e^{z_k^L}}}{\partial z_j^L} = \frac{\left(e^{z_i^L}\right)' \sum_k e^{z_k^L} - e^{z_i^L}\left(\sum_k e^{z_k^L}\right)'}{\left(\sum_k e^{z_k^L}\right)^2}$$

$$if\ i = j, \frac{\partial y_i^L}{\partial z_j^L} = \frac{\left(e^{z_j^L}\sum_k e^{z_k^L} - \left(e^{z_j^L}\right)^2\right)}{\left(\sum_k e^{z_k^L}\right)^2} = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} - \left(\frac{e^{z_j^L}}{\sum_k e^{z_k^L}}\right)^2 = y_j - y_j^2 = y_j(1 - y_j)$$

$$if\ i! = j, \frac{\partial y_i^L}{\partial z_j^L} = \frac{-e^{z_i^L}e^{z_j^L}}{\left(\sum_k e^{z_k^L}\right)^2} = -\frac{e^{z_i^L}}{\sum_k e^{z_k^L}}\frac{e^{z_j^L}}{\sum_k e^{z_k^L}} = -y_i y_j$$

Softmax

$$\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial y_i^L}\frac{\partial y_i^L}{\partial z_j^L} = -\frac{1}{n}\sum_x\sum_i \frac{t}{\mathrm{y_i}}\frac{\partial y_i^L}{\partial z_j^L} = -\frac{1}{n}\sum_x\left(\frac{t_j}{y_j}y_j(1-y_j) + \sum_{i!=j}\frac{t_i}{y_i}(-y_i y_j)\right) =$$

$$= -\frac{1}{n}\sum_x\left(t_j - t_j y_j - \sum_{i!=j} t_i y_j\right) = -\frac{1}{n}\sum_x\left(t_j - y_j\left(\underbrace{t_j + \sum_{i!=j} t_i}_{\sum_i t_i = 1}\right)\right) =$$

$$= -\frac{1}{n}\sum_x (t_j - y_i)$$

Softmax

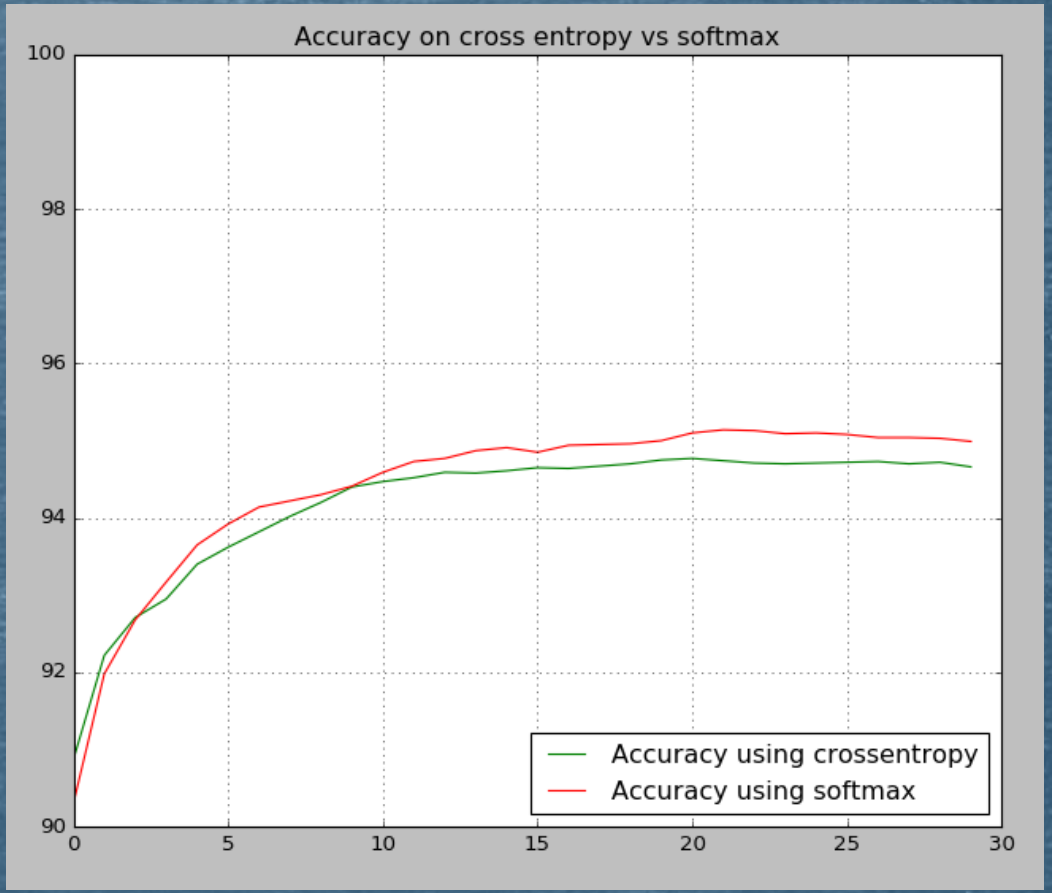
In order to use softmax function, the only thing that must be modified, in addition to using cross entropy, is the activation function in the output layer

$$y_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

In fact, the reason why the previous version of the network works (the one that doesn't use probabilities in the output layer) is because it has the same gradient as the cross entropy + softmax

Softmax

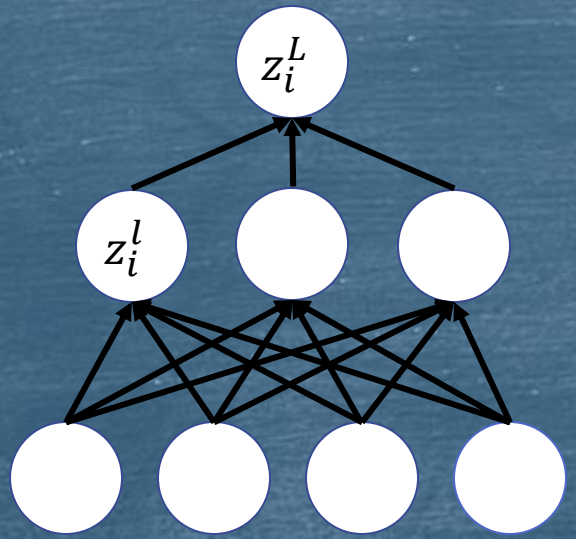
- Usually, the cross entropy function achieves best results when used together with softmax (if the problem allows)



Weight Initialization

Weight Initialization

- What do we need to initialize weights with random values?
- What if we would initialize all of them with 0s?



$$z_i^l = wx + b = 0x + 0 = 0$$

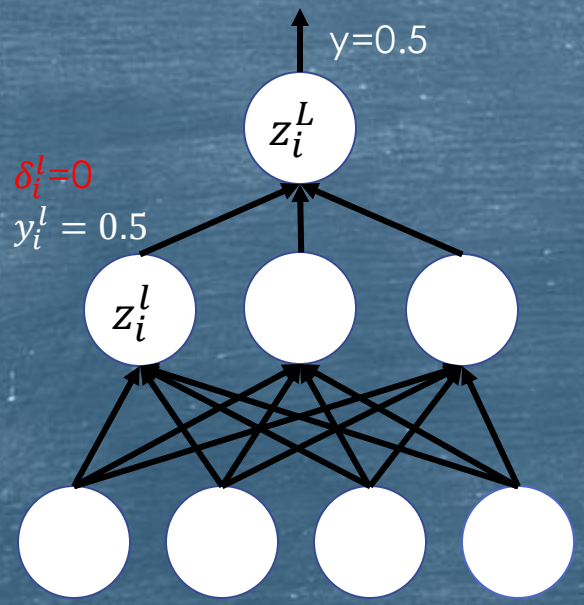
$$\sigma(z_i^l) = \sigma(0) = \frac{1}{1+e^0} = 0.5$$

(similarly), $\sigma(z_i^L) = 0.5$

Whatever the input, the output will be the same

Weight Initialization

- What if we would initialize all of them with 0s?

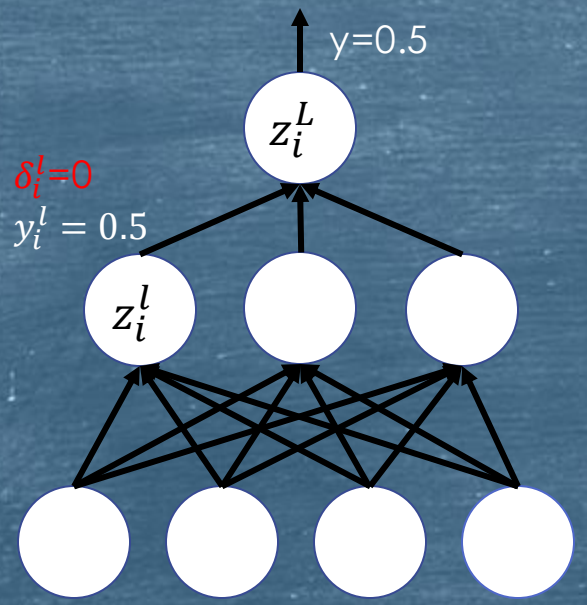


The error (if cross entropy is used $C = -\frac{1}{n}\sum_x [tlny + (1-t)\ln(1-y)]$), will always be for the first iteration $(\ln(0.5))$

The error that will be backpropagated, will be: $\delta_i^l = y_i^l(1-y_i^l)\sum_k \delta_i^{l+1} \cdot w_{ik}^{l+1} =$
$0.5 * (1-0.5) * (\delta^L * 0) = 0$

Weight Initialization

- What if we would initialize all of them with 0s?



The network will adjust the weights in the final layer w^2, with the same amount for each hidden unit

$$w = w + \eta * \delta_i^l y^{l-1} = w + \eta * 0 * 0 = w = 0$$

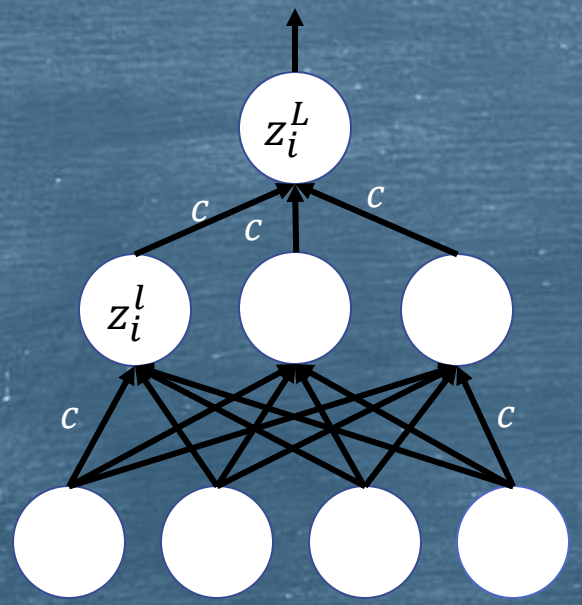
$$b = b + \eta * \delta_i^l = b + \eta * 0 = b = 0$$

In the same way can be shown that $w^1 = 0$

We ended up in the same place as the first iteration. So the network doesn't learn

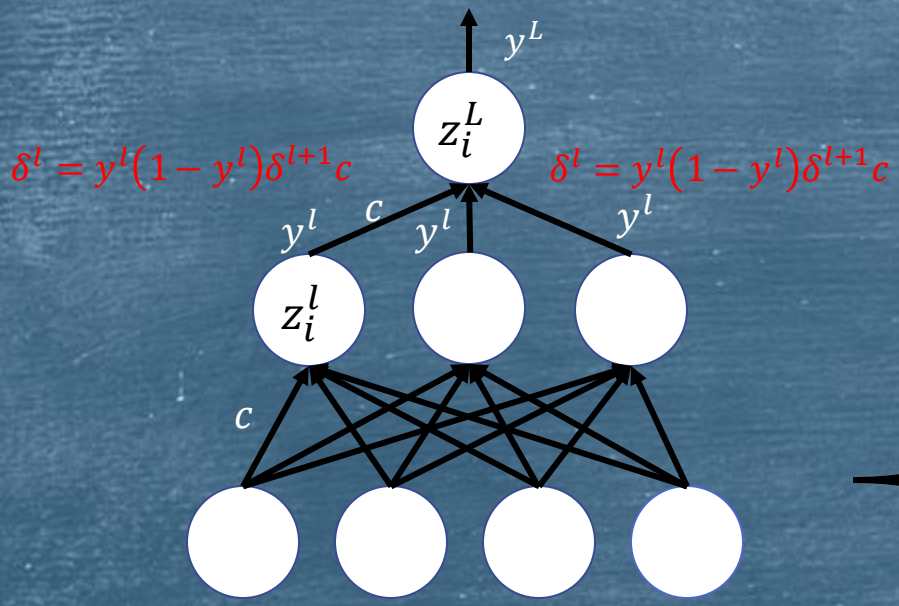
Weight Initialization

- What if we would initialize all of them with a constant (different than 0)?



Weight Initialization

- What if we would initialize all of them with a constant (different than 0)?



Each hidden unit will compute the same activation ($y^l = \sigma(cx + c)$).

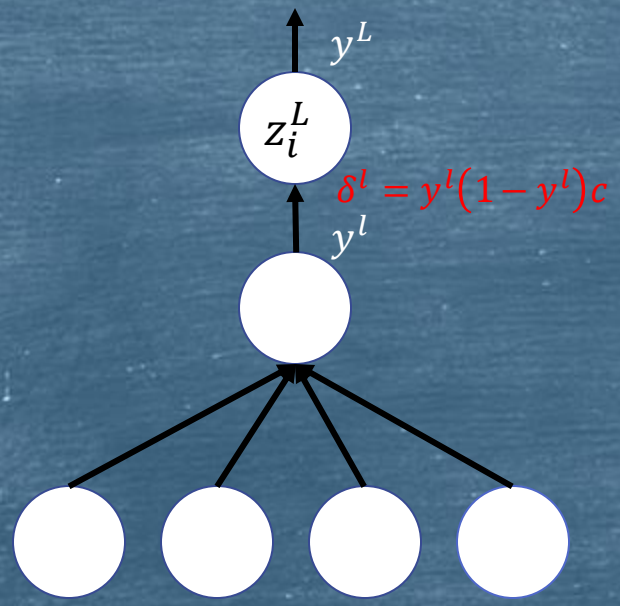
When the error is backpropagated, it will be the same for each hidden unit:

$\delta_i^l = y^l\,(1 - y^l\,)c\delta^{l+1}$

Since all weigths are the same and all weight updates are based on current weight values, and on the error, all weight adjustments will be the same

Weight Initialization

- What if we would initialize all of them with a constant (different than 0)?



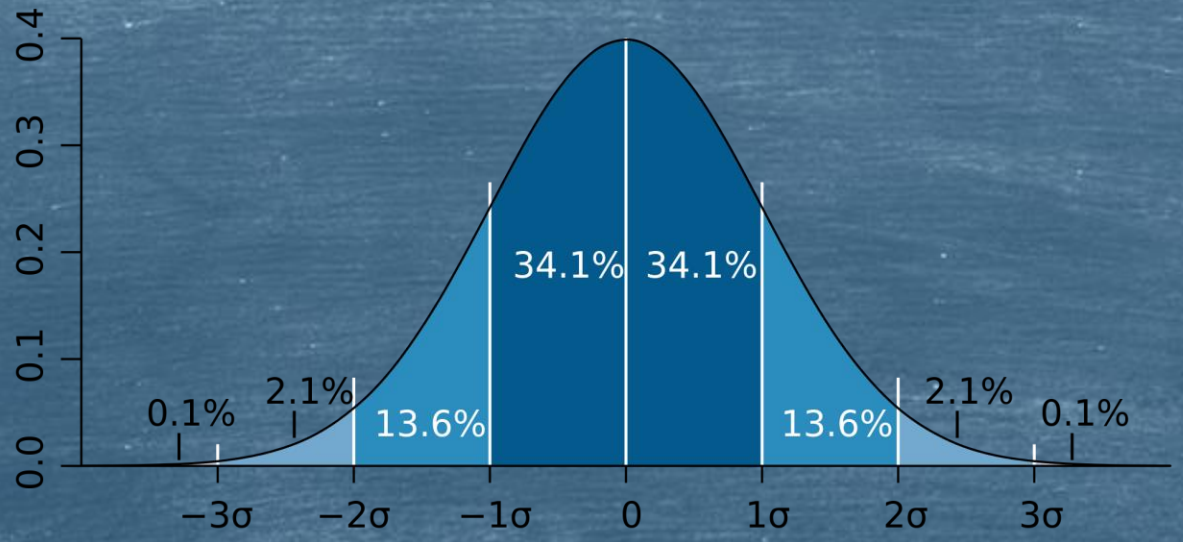
So, even though we have multiple neurons in the hidden layer, they will always have the same weights, thus they'll be the same

Is almost as there is only one neuron in the hidden layer

Making the weights random, achieves a better exploration of the feature space

Weight Initialization

Until now we've been using random weights with normal standard distribution. (normal distribution with $\sigma = 1$)

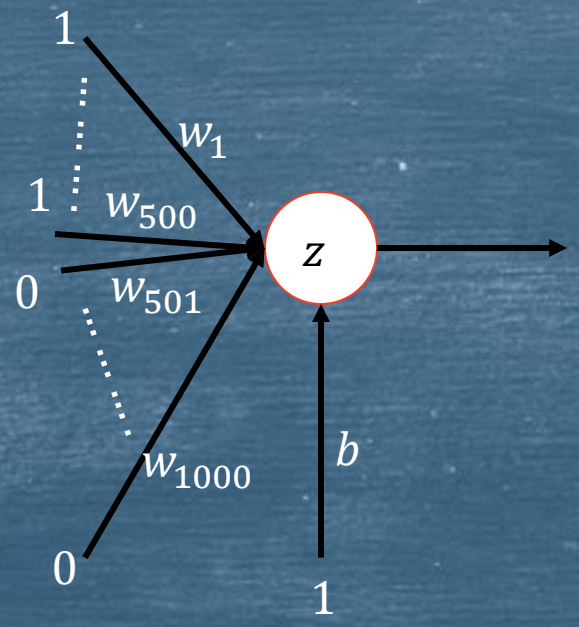


That means that 68% of the weights have values in interval [-1,1], 95% have values in interval [-2,2], 99.3 in [-3,3]

Weight Initialization

The problem with these kind of values is when we compute the net input $z = \sum_x wx + b$

Let's consider a neuron with 1000 inputs. Halve of which are 0. The other ones are 1



Weight Initialization

That means that the net input $z = \sum_{i=1}^{500} w_i$+b
Since w_i and b are normally distributed, that means that:

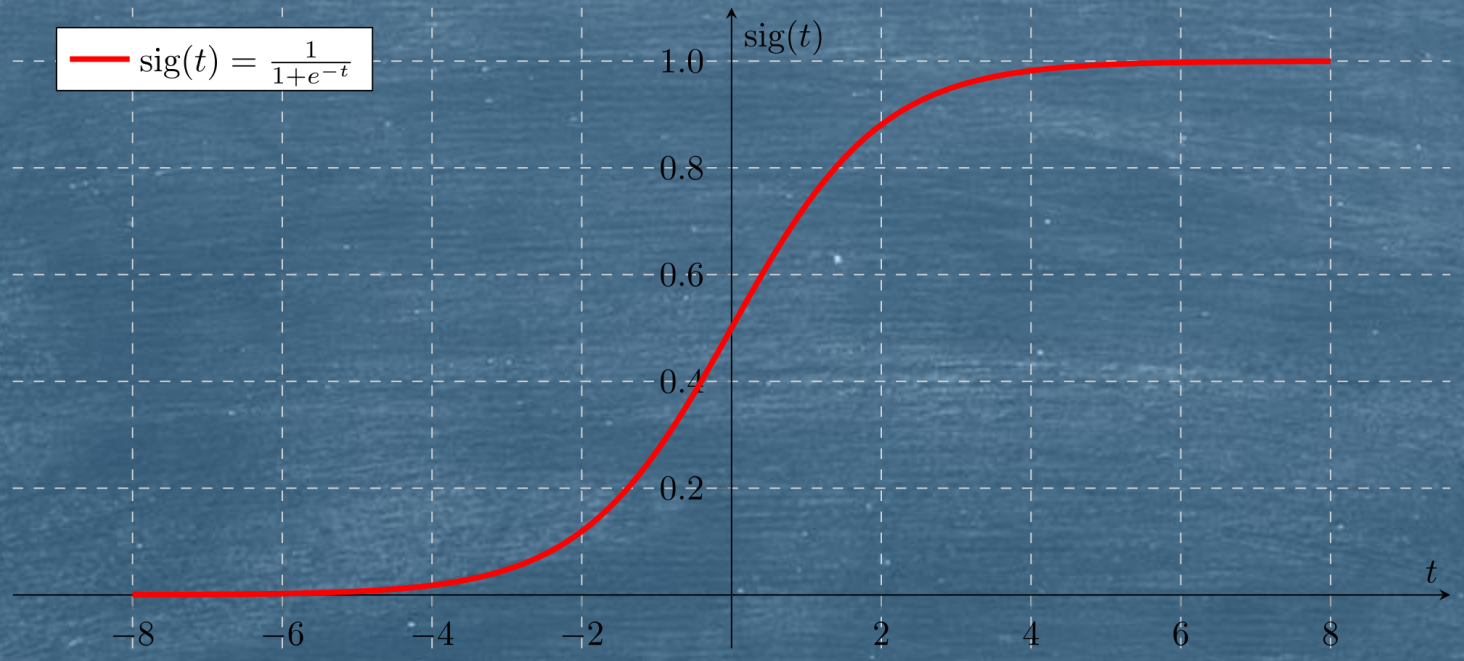
$$\mu_z = \sum_{i=1}^{500} \mu_{wi} + \mu_b = 0$$
$$var(z) = \sum_{i=1}^{500} var(w_i) + var(b) = 501$$

So, in this case, z is a variable that with a normal distribution with mean 0 and standard deviation of $\sqrt{501}$

Weight Initialization

That means that 95% of z values will be in the interval [-1002, 1002].

That is a very big interval, since a neuron usually saturates for values greater than 4.



Weight Initialization

The solution is to initialize the weights with such values that when added, the net input will not saturate the neuron.

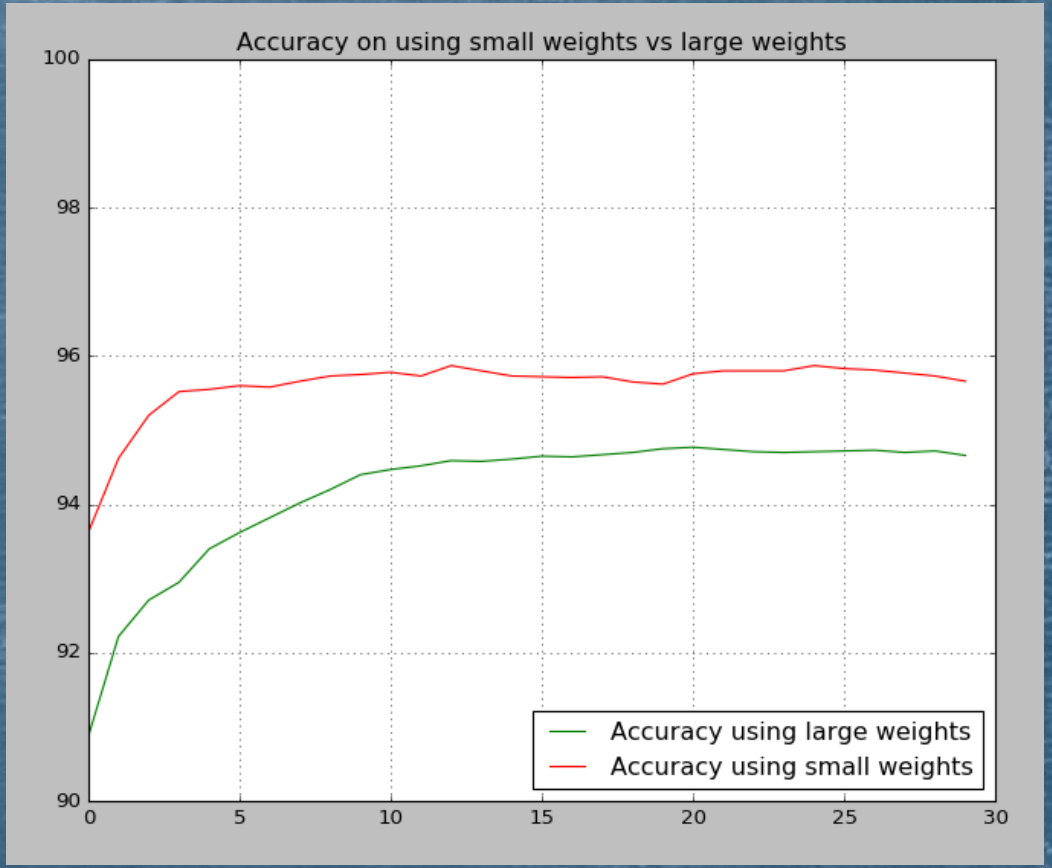
Thus, all values will be initialized with a random value from a normal distribution with mean 0 and a standard deviation of $\frac{1}{\sqrt{n_{in}}}$ where n_{in} is the total number of connection that go into the neuron.

In our case, the standard deviation will be $\frac{1}{\sqrt{1000}}$

The bias can still be generated from a standard normal distribution since it just adds 1 to the variance.

Weight Initialization

- By using small weights, the network accuracy increases
- Also, using small weights, the network arrives faster at the best accuracy



How to adjust hyper-parameters

How to adjust hyper-parameters

Besides the weights, our network has some parameters that control how it learns:

- Learning rate η
- The mini-batch size
- The number of epochs
- The number of hidden neurons

All parameters should be tested on a separate dataset (validation set) in order to avoid fitting the parameters to the test set

How to adjust hyper-parameters

The first, and probably the most difficult is to achieve any non-trivial learning. You must obtain results better than you would obtain by a random selection.

In the case of MNIST digits, this means you should obtain something greater than 10%

- Start with a smaller dataset. This increases the speed
  - In the case of MNIST digits this could mean to work with only two numbers (0 and 1)
- Start with a smaller network.
  - In the case of MNIST digits, that could mean to start with a network of 784x10 neurons. (No hidden layer)

How to adjust hyper-parameters

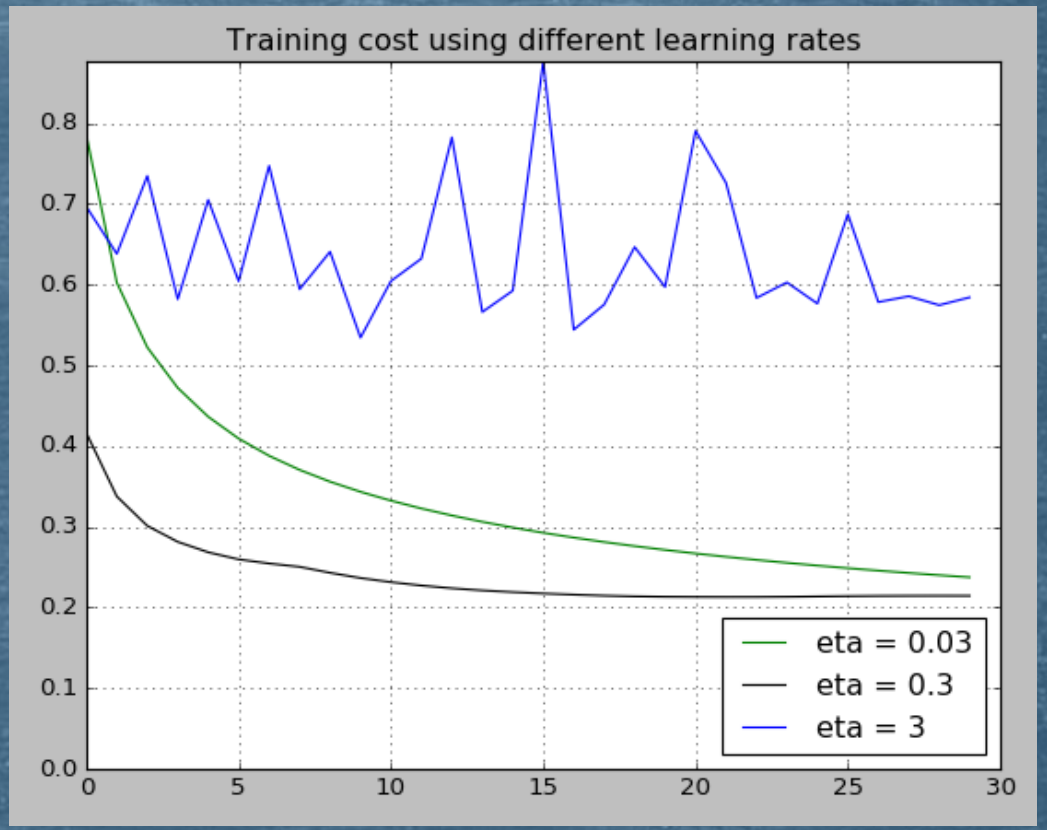
- Increase monitoring frequency
  - Work with just a fraction of the validation set
  - Monitor the accuracy not only on iterations, but also after you have computed some mini batches (for example, 10 minibatches)

All of the above steps are useful to allow you to receive quick feedback from the network. This allows to test many values for the parameters.

Start by adjusting the learning rate until you see some learning happens.

How to adjust hyper-parameters

- You should be increasing or decreasing the learning rate and monitor the cost of the training data. Here is how it looks like for different values of the η
- For large values of η the gradient descent overshoots
- For low variants of η the gradient descent is low



How to adjust hyper-parameters

- You should start with a value for the learning rate where the training cost decreases in the first iterations.
- Increase it by magnitude (10) until the network starts oscillating. This is the threshold
- You can then refine it by slowly increasing it until the costs starts oscillating again (gets close to the threshold). In fact, the value should be a factor, or two below the threshold.

How to adjust hyper-parameters

- Choosing the number of iterations is simple: just use early stopping. That means, after each iterations test the network on the validation set. If after x iterations there is no improvement in the classification accuracy, then stop. X can be for example 10 iterations.

- At the beginning you should let the network learn for a significant number of iterations in order to avoid the situation where it gets to a plateau only to continue learning again

How to adjust hyper-parameters

- Mini Batch Size:
  - Mini Batch should be used since we can make use of modern libraries that can compute the weights of all the elements in the batch, at once.
  - The validation accuracy should be plotted against time (real time, not number of epochs) and choose the one that achieves the highest increase.

Questions & Discussion