Improving Deep Neural Networks:
Hyperparameter tuning, Regularization and Optimization

Week-3

In week 2, we have learnt about so many Hyperparameters.
In week 3, we start with a systematic tuning process which will be helpful to converge on a good setting.

Hyperparameters:

Type-1

- α -> Learning Rate

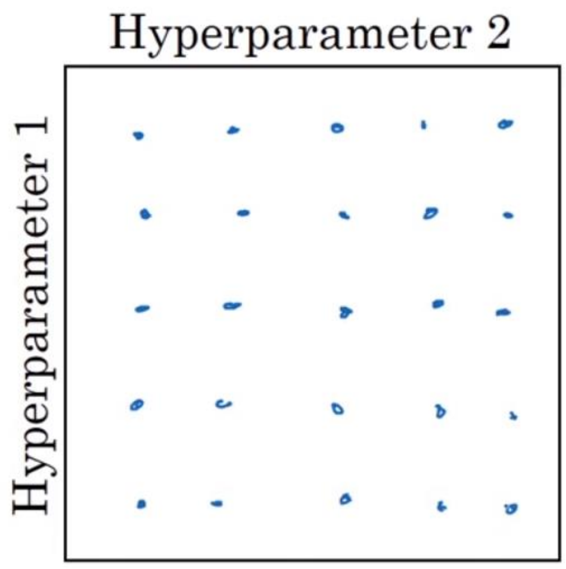
Type-2

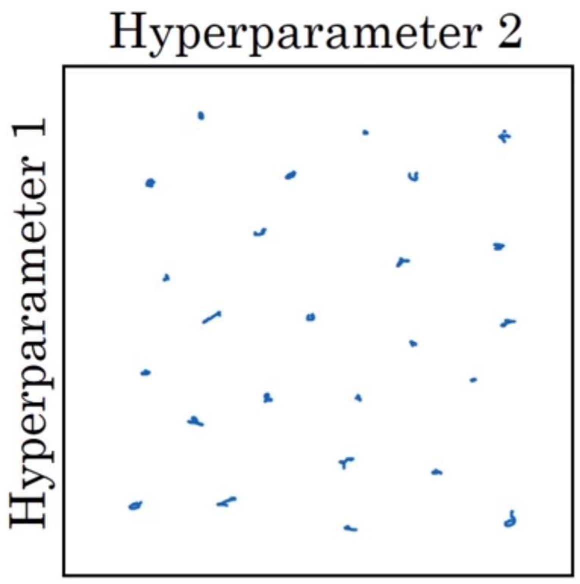
- β -> RMS Prop (Momentum)
- # Hidden Units for different layers
- Mini-Batch Size

Type-3

- # Layers
- β_1, β_2, ε -> Adam optimisation
- Learning rate decay

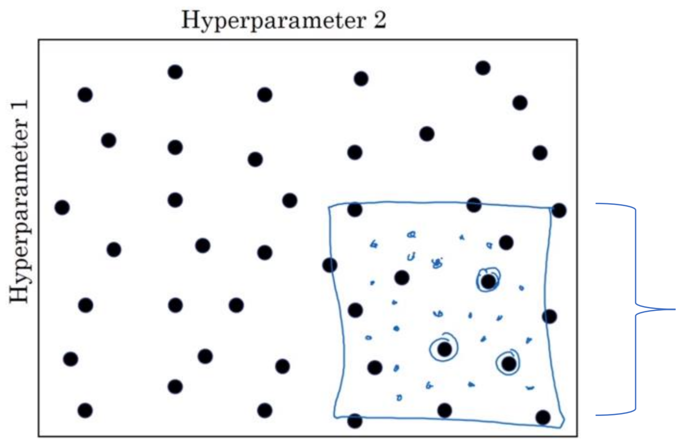
So, to try a set of values for hyperparameters what values should we see, we generally make a grid similar to the image below. We pick 1/25 points, which ever works best. This is ok, if the number of hyperparameters are relatively small. In DL, we choose the points at random, reason being it is difficult to know in advance which hyperparameter will be important for your problem.





Say Hyperparameter 1 -> α and Hyperparameter 2 -> ε.
If we consider the grid, the cost function would be the same for almost any value of ε i.e. we only get to see 5 values of α (as the value of ε does not have huge impact) but if you use random points we would have tested 25 values of α.

We also use Coarse to fine sampling scheme to find out the values of the Hyperparameters.



This area is good, so we will zoom into this area and try random points in this as we did in the above-mentioned method.

After understanding how sampling at random over the range of hyperparameters can allow you to search over the space of hyperparameters more efficiently.
But it turns out that sampling at random doesn't mean sampling uniformly at random, over the range of valid values. Instead, it's important to pick the appropriate scale on which to explore the hyperparameters

Let’s say there is a hyperparameter α which varies from 0.0001 - 1. Now if you draw the number line from 0.0001 to 1, and sample values uniformly at random over this number line. Well about 90% of the values you sample would be between 0.1 and 1. So you're using 90% of the resources to search between 0.1 and 1, and only 10% of the resources to search between 0.0001 and 0.1. So that doesn't seem right.
Instead, it seems more reasonable to search for ***hyperparameters on a log scale***. Where instead of using a linear scale, you'd have 0.0001 here, and then 0.001, 0.01, 0.1, and then 1 and you instead sample uniformly, at random, on this type of logarithmic scale. Now you have more resources dedicated to searching between 0.0001 and 0.001, and between 0.001 and 0.01, and so on.

A similar concept is used to select Hyperparameters which are computed as exponentially weighted averages such as β, which vary from 0.9-0.999.
Here, we perform the same log operation on 1- β.

Next, we saw the importance of re-testing the hyperparameters occasionally i.e. at least once in few months, so we can fine tune any of the parameters until we are happy with the results.

There are generally two major different ways in which people go about it: (Pandas vs Caviar)

- Babysitting -> Pandas
- Parallel Computation -> Caviar (Fish) over 100 million fish during a mating season

Babysitting

- This is usually done if you have maybe a huge data set but not a lot of computational resources.
- So, for example, on Day 0 you might initialize your parameter as random and then start training. You gradually watch your learning curve, maybe the cost function J each day and fine tune the parameters.
- You are kind of babysitting the model one day at a time even as it's training over a course of many days or over the course of several different weeks.

Parallel Computation

- As the name suggests, you parallelly run different set of configurations over a course of time and see which model best fits your purpose.

So, to make an analogy:

- The 1st method is **panda approach**.
  When pandas have children, they have very few children, usually one child at a time, and then they really put a lot of effort into making sure that the baby panda survives. So that's really babysitting.
- The 2nd method is **Caviar approach**.
  This approach is more like what fish do. There's some fish that lay over 100 million eggs in one mating season. But the way fish reproduce is they lay a lot of eggs and don't pay too much attention to any one of them but just see that hopefully one of them, or maybe a bunch of them, will do well.

Next, we learnt about how **Batch Normalisation** can speed up learning.
We have already seen how normalisation helps in making contours from ellipses to circles, which is easier to optimise.
We now try to extend this by solving the following problem:

- *Can we normalise $a^{[l-1]}$, so as to train $w^{[l]}, b^{[l]}$ faster?*

To solve this, we normalise $Z^{[l]}$now we will try to extend this idea little further by implementing the same throughout the network.

Implementing Batch Norm:
Suppose for a layer l, we have $Z^{[l](1)}$, $Z^{[l](2)}$, $Z^{[l](3)}$,……… $Z^{[l](m)}$. ($Z^{[l](i)}$ = $Z^{(i)}$)

$$\mu = \frac{1}{m}\sum_{i=1}^{m} Z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (Z^{(i)} - \mu)^2$$

$$Z_{norm}^{(i)} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \left.\begin{matrix} \text{Mean -> 0} \\ \text{Variance -> 1} \end{matrix}\right\}$$

But we don't want our hidden units to always have mean '0' and variance '1'.

$$\check{z}^{(i)} = \gamma Z_{norm}^{(i)} + \beta$$

γ & β are learnable parameters of your model, with this you can achieve whatever mean and variance you want.
if $\gamma = \sqrt{\sigma^2 + \varepsilon}$ and $\beta = \mu$, then $\check{z}^{(i)} = Z_{norm}^{(i)}$

We want our mean ≠ 0 and Variance to be large, so that we can take advantage of the non-linearity of the activation functions.

*We use batch norm in between computing $Z^{[l]}$ and not $a^{[l]}$.

We then saw how to apply batch norm with mini-batches and also realised that the term $b^{[l]}$ gets eliminated in batch norm as:

$$Z^{[l]} = W^{[l]} * a^{[l-1]} + b^{[l]}$$

But later we perform,

$$\mu = \frac{1}{m}\sum_{i=1}^{m} Z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (Z^{(i)} - \mu)^2$$

which makes all constants to '0'. $\boxed{\therefore b^{[l]} = 0}$

Now, our parameters which were initially $W^{[l]}, b^{[l]}$ have changed to $W^{[l]}$, $\beta^{[l]}$ and $\gamma^{[l]}$.

Implementing Gradient Descent:

for t=1: # Mini Batches:

- Compute forward prop on $X^{\{t\}}$
  In each hidden layer, use Batch Norm to replace $Z^{[l[}$ with $\check{z}^{[l]}$.
- Use Backprop & compute $dW^{[l]}, d\beta^{[l]}$ *and* $d\gamma^{[l]}$
- Update Parameters:
  $W^{[l]} = W^{[l]} - \alpha * dW^{[l]}$
  $\beta^{[l]} = \beta^{[l]} - \alpha * d\beta^{[l]}$
  $\gamma^{[l]} = \gamma^{[l]} - \alpha * d\gamma^{[l]}$

We can also use RMS Prop, Momentum and Adam to update parameters.

Problem of Covariance Shift:

Assume you trained your algorithm to identify black cats but you are testing on colored cats, your algorithm might not do well, this problem is known as *"covariance shift"*.

"One problem with machine-learned models is that, in a sense, they're already out-of-date before you even use them. Models are (by necessity) trained and tested on data that already happened. It does us no good to have accurate predictive analytics on data from the past!

The big bet, of course, is that data from the future looks enough like data from the past that the model will perform usefully on it as well. But, given a long enough future, that bet is bound to be a losing one. The world (or, in data science parlance, the distribution generating the data) will not stand still simply because that would be convenient for your model. It will change, and probably only after your model has been performing as expected for some time. You will be caught of guard! You and your model are doomed!"
Source: https://blog.bigml.com/2013/11/01/machine-learning-next/

- This problem of covariance shift is solved with the help of batch norm as it makes weights, later or deeper in your network, say the weight on layer 10, more robust to changes to weights in earlier layers of the neural network, say, in layer one, so your algorithm behaves well even if there is a change in the input as compared to the training data.

- The mean and the variance does not shift even though the data shifts, thus making weights deeper in the network more robust.
- Batch Norm weakens the coupling, by making learning independent and speeds up the learning.

Batch Norm as Regularisation:

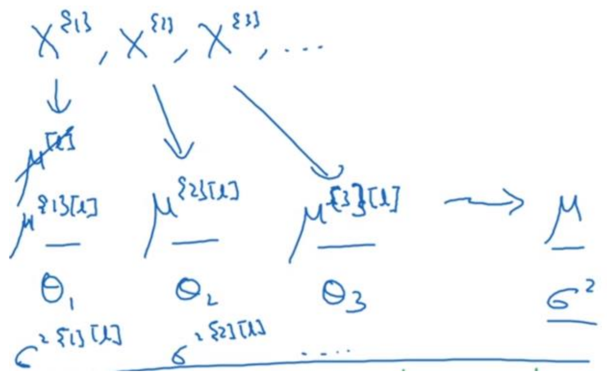
- Each mini-batch is scaled by the mean/variance computed on just that mini -batch.
- This adds some noise to the values $Z^{[l]}$ within that mini-batch, so similar to dropout, it adds some noise to each hidden layer's activations.
- This has a sight Regularisation effect.

Dropout has multiplicative noise as it is multiplying by '0' or '1'.
Batch norm has both multiplicative and additive noise as you are multiplying by γ and adding β.

Batch Norm shouldn't be used for Regularisation

When you are using Batch norm at test time, we can't compute μ and σ^2 for a single example, so we estimate them using exponentially weighted averages.

At Test time

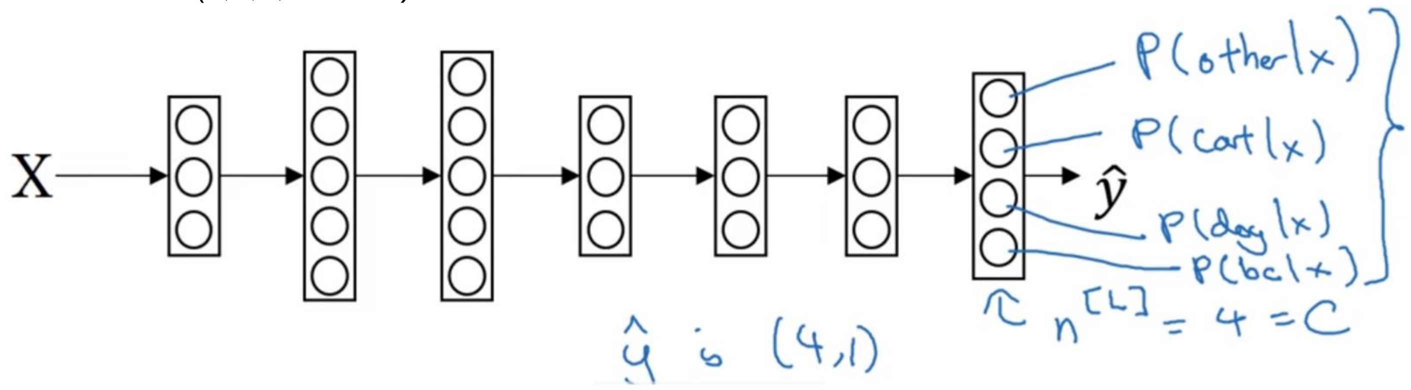


$$Z_{norm} = \frac{Z - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\check{z}^{(i)} = \gamma Z_{norm} + \beta$$

Softmax Regression:

So far, we have done binary classification, we use softmax regression to classify if there are more than 2 types of classes.
C-> # Classes (0,1,2,……C-1)



The final layer of the neural network will be the Softmax layer which gives us the probability of 1 entity over others.

The unusual thing about the Softmax activation function is that it takes an input a vector of dim (n,1) and outputs the vector of the same size unlike other activation functions like sigmoid and relu which take a row value input and outputs a row value.

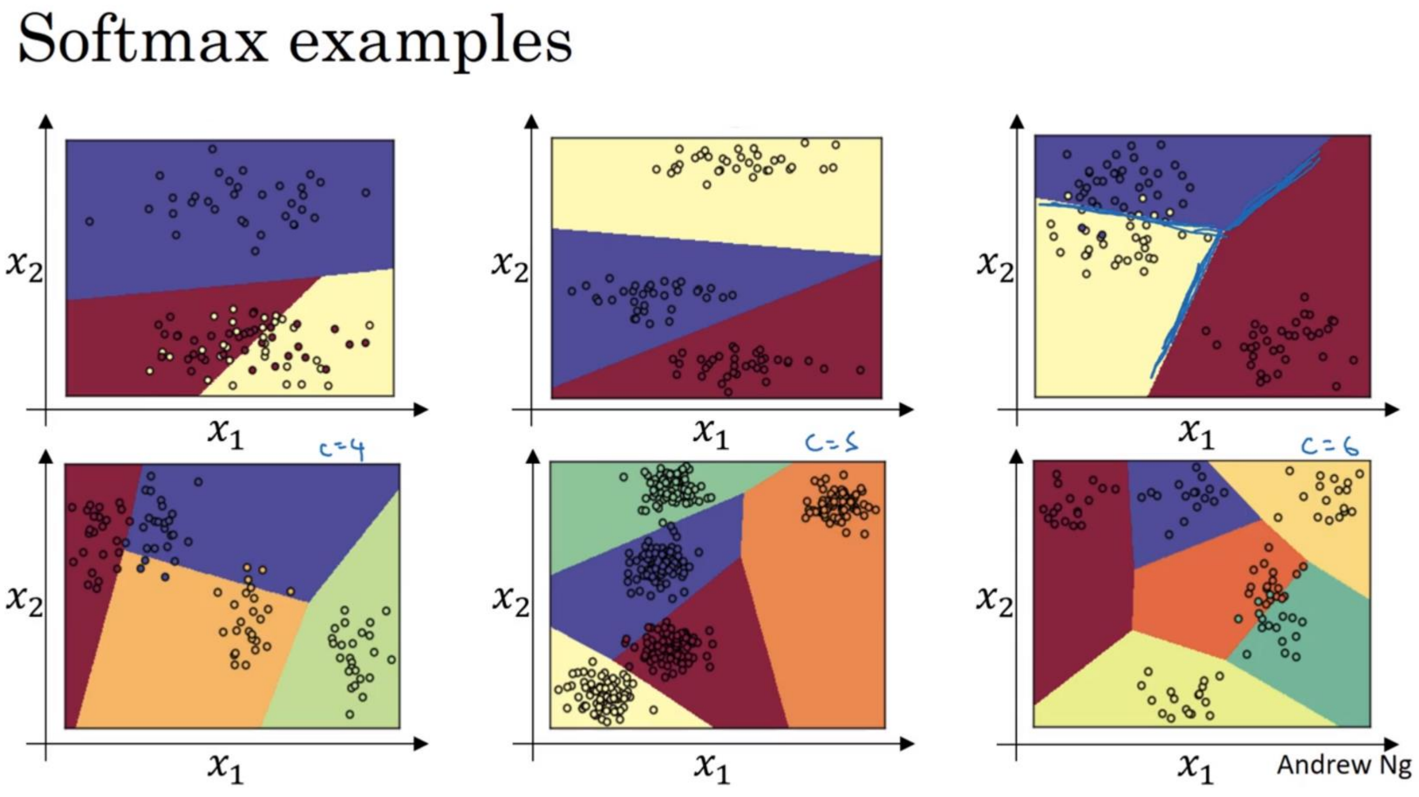
For the last layer L,

$$Z^{[L]} = W^{[L]} * a^{[L-1]} + b^{[L]}$$

Activation Function:

$$t = e^{(Z[L])}$$

$$a^{[L]} = \frac{e^{(Z[L])}}{\sum_{i=1}^{C} t_i}, a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{C} t_i}$$



Training of Softmax classifier:

Assume, $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 1 \end{bmatrix}$, Hard Max-> $= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ $\Rightarrow$

We don't use hard max because this is not differentiable and this will be a blocker for us in Back Prop!!

Softmax Regression generalises logistic regression to C Classes.

Loss Function:

Assume, $y = \begin{bmatrix} 0 \\ 1 -> \text{Cat} \\ 0 \\ 0 \end{bmatrix}$, $a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ -> Not doing very well

L(ŷ,y) = $-\sum_{j=1}^{4}(y_j \log \hat{y}_j)$ = $-y_2 \log \hat{y}_2$ = – log 0.2

If we want to reduce the loss, we have to make $-y_2 \log \hat{y}_2$ small, we have to increase $\hat{y}_2$.

This is similar to “Maximum likelihood” in Statistics

$$J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]} \ldots . W^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{L}(\hat{\mathrm{y}}^{(i)}, \mathrm{y}^{(i)})$$

Backprop using Softmax:

$$\mathrm{J} = -\sum_{i=1}^{m}\left(y_j \log \hat{y}_j\right)$$

$$a^{[L]} = \hat{y}_j = softmax\left(Z_j^{[L]}\right) = \frac{e^{Z_j}}{\sum_j e^{Z_j}}$$

$$Z^{[L]} = W^{[L]} * a^{[L-1]} + b^{[L]}$$

$$\frac{\partial J}{\partial Z^{[L]}} = \frac{\partial J}{\partial a^{[L]}} * \frac{\partial a^{[L]}}{\partial Z^{[L]}}$$

$$\frac{\partial J}{\partial a^{[L]}} = -\frac{y_j}{a^{[L]}}$$

$$\text{if } i = j : \frac{\partial y_i}{\partial z_i} = \frac{\partial \frac{e^{z_i}}{\Sigma_C}}{\partial z_i} = \frac{e^{z_i}\Sigma_C - e^{z_i}e^{z_i}}{\Sigma_C^2} = \frac{e^{z_i}}{\Sigma_C}\frac{\Sigma_C - e^{z_i}}{\Sigma_C} = \frac{e^{z_i}}{\Sigma_C}\left(1 - \frac{e^{z_i}}{\Sigma_C}\right) = y_i(1 - y_i)$$

$$\text{if } i \neq j : \frac{\partial y_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\Sigma_C}}{\partial z_j} = \frac{0 - e^{z_i}e^{z_j}}{\Sigma_C^2} = -\frac{e^{z_i}}{\Sigma_C}\frac{e^{z_j}}{\Sigma_C} = -y_i y_j$$

$$\frac{\partial a^{[L]}}{\partial Z^{[L]}} = \frac{\partial}{\partial Z_j}\left(\frac{e^{Z_j}}{\sum_j e^{Z_j}}\right) = \frac{e^{Z_j} * \sum_j e^{Z_j} - (e^{Z_j})^2}{(\sum_j e^{Z_j})^2} = \frac{e^{Z_j}}{\sum_j e^{Z_j}} - \frac{(e^{Z_j})^2}{\left(\sum_j e^{Z_j}\right)^2} = a^{[L]} - \left(a^{[L]}\right)^2 = a^{[L]} * (1 - a^{[L]})$$

$$\frac{\partial a^{[L]}}{\partial Z^{[L]}}\frac{\partial J}{\partial Z^{[L]}} = \left(-\frac{y_j}{a^{[L]}} * a^{[L]} * \left(1 - a^{[L]}\right)\right) + () = -\left(y_j\right) * \left(1 - a^{[L]}\right)$$

$$\frac{\partial J}{\partial Z^{[L]}} = -(y_j) * \left(1 - \sum_{j=1}\hat{y}_j\right) = -y_j + y_j * \sum_{j=1}\hat{y}_j = -y_j + \hat{y}_j$$

$$\frac{\partial J}{\partial Z^{[L]}} = \hat{y}_j - y_j$$

Source: https://stats.stackexchange.com/questions/235528/backpropagation-with-softmax-cross-entropy

$$y_c = \varsigma(\mathbf{z})_c = \frac{e^{z_c}}{\sum_{d=1}^{C} e^{z_d}} \quad \text{for } c = 1 \cdots C$$

$$\text{if } i = j : \frac{\partial y_i}{\partial z_i} = \frac{\partial \frac{e^{z_i}}{\Sigma_C}}{\partial z_i} = \frac{e^{z_i}\Sigma_C - e^{z_i}e^{z_i}}{\Sigma_C^2} = \frac{e^{z_i}}{\Sigma_C}\frac{\Sigma_C - e^{z_i}}{\Sigma_C} = \frac{e^{z_i}}{\Sigma_C}(1 - \frac{e^{z_i}}{\Sigma_C}) = y_i(1 - y_i)$$

$$\text{if } i \neq j : \frac{\partial y_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\Sigma_C}}{\partial z_j} = \frac{0 - e^{z_i}e^{z_j}}{\Sigma_C^2} = -\frac{e^{z_i}}{\Sigma_C}\frac{e^{z_j}}{\Sigma_C} = -y_i y_j$$

$$\begin{aligned}
\frac{\partial \xi}{\partial z_i} &= -\sum_{j=1}^{C} \frac{\partial t_j log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{\partial log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{1}{y_j}\frac{\partial y_j}{\partial z_i} \\
&= -\frac{t_i}{y_i}\frac{\partial y_i}{\partial z_i} - \sum_{j \neq i}^{C} \frac{t_j}{y_j}\frac{\partial y_j}{\partial z_i} = -\frac{t_i}{y_i} y_i(1 - y_i) - \sum_{j \neq i}^{C} \frac{t_j}{y_j}(-y_j y_i) \\
&= -t_i + t_i y_i + \sum_{j \neq i}^{C} t_j y_i = -t_i + \sum_{j=1}^{C} t_j y_i = -t_i + y_i \sum_{j=1}^{C} t_j \\
&= y_i - t_i
\end{aligned}$$

Source: http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/

Introduction to Programming Frameworks:

To get a feel of the algorithm, it's a good practice to perform all the steps without the help of any programming frameworks but to perform operations on large convolutional neural networks(CNN) and recurring neural networks(RNN), it is difficult to perform all of them without a framework.

It is similar to writing a function to do matrix multiplication instead of using a Linear Algebra Library.

Deep Learning Frameworks:

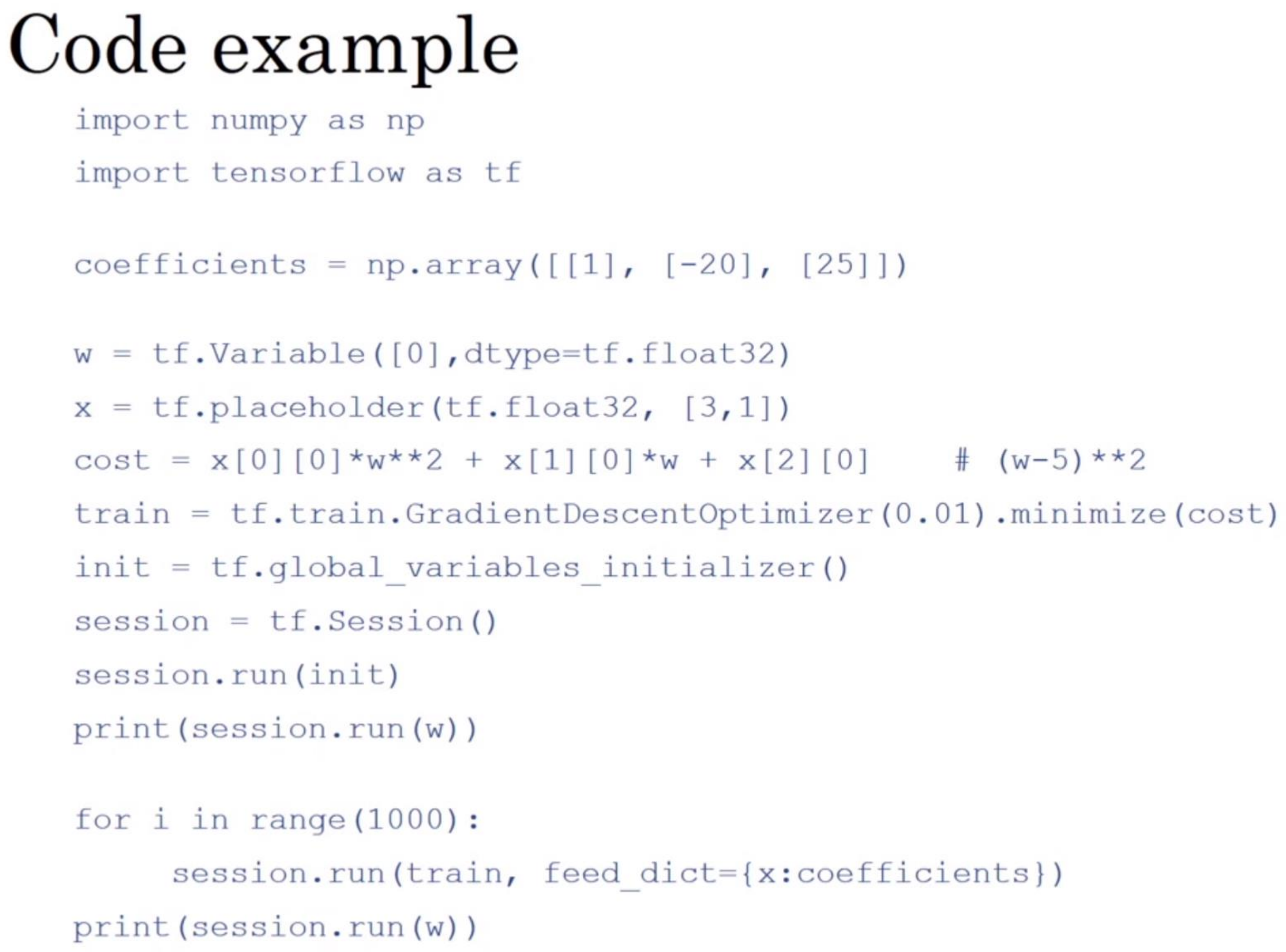
- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- Mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Which Framework to Choose?

- Ease of programming.
  (Development & Deployment)
- Running Speed
- Truly Open
  (Open Source with good governance)

  Some companies, initially give it as open-source, later they make it their propitiatory.
- Problem Type (RNN, CNN, Computer Vision)

Tensor Flow:



Code example

```
import numpy as np
import tensorflow as tf

coefficients = np.array([[1], [-20], [25]])

w = tf.Variable([0],dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]     # (w-5)**2
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

for i in range(1000):
     session.run(train, feed_dict={x:coefficients})
print(session.run(w))
```

In Tensor Flow you only need to write the equation for forward-prop, back-prop is taken care by the framework itself.

Programming Assignment:

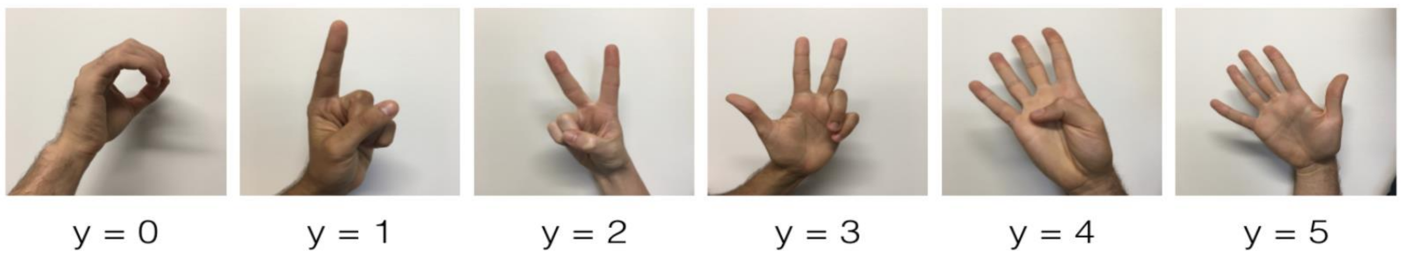
Problem Statement:

One afternoon, with some friends we decided to teach our computers to decipher sign language. We spent a few hours taking pictures in front of a white wall and came up with the following dataset. It's now your job to build an algorithm that would facilitate communications from a speech-impaired person to someone who doesn't understand sign language.

- **Training set**: 1080 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (180 pictures per number).
- **Test set**: 120 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (20 pictures per number).

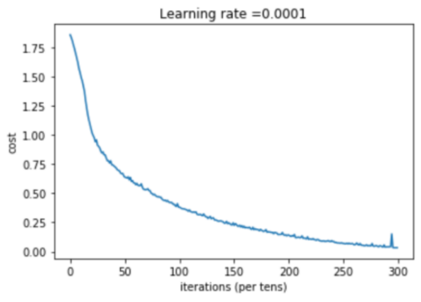
Note that this is a subset of the SIGNS dataset. The complete dataset contains many more signs.

Here are examples for each number, and how an explanation of how we represent the labels. These are the original pictures, before we lowered the image resolutoion to 64 by 64 pixels.



Observations:

Your model seems big enough to fit the training set well. However, given the difference between train and test accuracy, you could try to add L2 or dropout regularization to reduce overfitting.



Think about the session as a block of code to train the model. Each time you run the session on a minibatch, it trains the parameters. In total, you have run the session a large number of times (1500 epochs) until you obtained well trained parameters.

Tensorflow is a programming framework used in deep learning

- The two main object classes in tensorflow are Tensors and Operators.
- When you code in tensorflow you have to take the following steps:
  - Create a graph containing Tensors (Variables, Placeholders ...) and Operations (tf.matmul, tf.add, ...)
  - Create a session
  - Initialize the session
  - Run the session to execute the graph
- You can execute the graph multiple times as you've seen in model()
- The backpropagation and optimization is automatically done when running the session on the "optimizer" object.

Results:

Train Accuracy – 79.7 %
Test Accuracy – 71.6 %