**Week 3 – Notes**

**Hyperparameter Tuning**

**Tuning Process**

The most important hyperparameters to tune are:

1. learning rate

2. mini-batch size, beta and the number of hidden units

3. number of layers and learning rate decay

Rarely you optimize beta 1, beta 2 or epsilon (of the optimizer)

Do not use a grid, instead use random values because some hyperparameter may not be that important and by using a random sampling technique, we test more hyperparameters values

The second technique is to go from coarse to fine, so for some hp values to zoom in the hp space

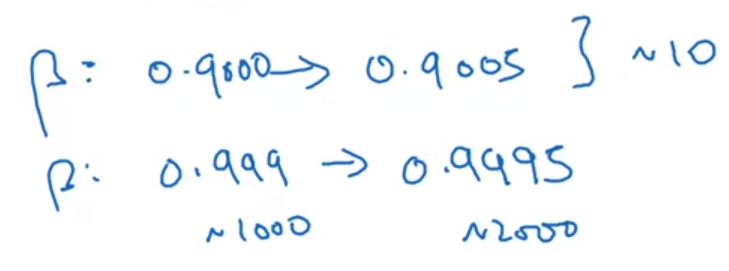
**Using an Appropriate Scale to pick Hyperparameters**

For some hp, the best sampling technique is to pick them by using a uniform distribution; e.g.: the number of neurons in a layer

For the learning rate, is better to pick the values by using a logarithm scale => explore the log space uniformly

Alpha: 0.0001 -> 1 => sample values of the form 10^r, where r is a random variable (picked uniformly) between -4 and 0

For exponentially weighted averages again pick beta = 1 – 10^r, where r is uniformly sampled between a negative value and -1 => in this way we use the same number of resources to sample between 0.9 and 0.99 as we use to sample between 0.99 and 0.999

 -> in this way, these 2 intervals are sampled uniformly, without letting out a lot of values between 0.999 and 0.9995

**Hyperparameters Tuning in Practice: Pandas vs. Caviar**

Intuition do get stale, so over re-evaluate the picked hyperparameters

There are two different approaches:

1. Babysitting one model => you deal with a huge data set / large model and do not have the resources to train several models => babysit one model: train it and over time store various versions of the model and change the hyperparameters manually
2. Caviar => you have a lot of power / deal with small data sets or models => train lots of models and then pick the best one

**Batch Normalization**

**Normalizing Activations in a Network**

Makes the network much more robust to a larger spectrum of hp

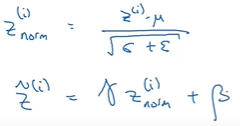
Enables the training of larger neural networks

As you normalize the inputs, you normalize the inputs or the output of the activation functions

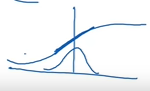
In practice is more common to normalize the inputs of the activation functions

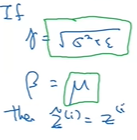
For example, if you apply batch normalization for the layer 2, then the layer 3 will benefit from a faster training

Considering that we work with one layer [l], we take the z value of that layer for each neuron i and compute the mean and the standard deviation



Because we don’t want for each normalized distribution to have the mean 1 and the standard deviation 0 (for example this won’t fit a sigmoid activation function), we use 2 learnable parameters gamma and beta to modify the mean and the standard deviation





**Fitting Batch Norm into a Neural Network**

For each layer we have beta and gamma that are used to normalize a layer

These 2 parameters are learned through backpropagation

A picture containing handwriting, text, diagram, line

Description automatically generated

When working with mini-batches, you apply batch normalization for each batch

If we use batch normalization, we don’t have to use the b parameter, because it will be canceled out by the beta one

A picture containing handwriting, text, font, diagram

Description automatically generated

Beta [l] and gamma [l] have the same dimensions as b [l], so one value for each neuron from that layer

A picture containing handwriting, font, whiteboard, calligraphy

Description automatically generated

The implementation of the gradient descent with mini-batches is straight forward:

Forward pass for the mini-batch and replace z [l] with z tilda [l]

Backward pass and update beta and gamma

It works with any variant of the gradient descent, including RMSProp and Adam

**Why does Batch Norm work?**

There is an issue called covariance shift, that means that the distribution of the data can change between training and testing data

A picture containing child art, drawing, diagram, line

Description automatically generated

For example you trained your network to classify black cats and at test time the cats have different colors

The batch normalization techniques helps to train deeper networks as it standardizes the layers, in this way changes in the first layers do not affect that much the deeper layers => decoupling effect

Additionally, batch norm acts as a regularization because each mini-batch is scaled by the mean and variance computed on that mini-batch

This adds some noise to the values z[l] within that mini-batch => similarly, it adds noise to each hidden layer’s activation

Smaller mini-batches + batch normalization imply stronger regularization

So the regularization is only a side effect

**Batch Norm at Test Time**

If we use batch norm with mini-batches during training, for each mini-batch a new mean and std are computed

At the test time the solution is to approximate the mean and std by using exponentially weighted average; you use this technique for each layer on which batch norm was applied

**Multi-class Classification**

**Softmax Regression**

It’s a generalization of the Logistic Regression; the output layer has as many outputs as the number of classes

A picture containing text, handwriting, font, line

Description automatically generated

This layer has as the activation function one that it’s called the softmax, which gets and outputs as many values as the number of classes

The property of this function is that it outputs a vector of values between 0 and 1, whose sum is one (outputs the probability of each class)

A picture containing font, white, symbol, text

Description automatically generated

If we train a softmax classifier, so a NN with one layer with the softmax function as the activation one, it can partition n the space in multiple areas, by generating linear separations

A picture containing text, graphics, graphic design, screenshot

Description automatically generated

**Training a Softmax Classifier**

The function is called softmax because it maps the input values to output values, where the biggest input value remains the largest output one

A hard max function transforms the inputs to a one-hot encoded vector

A softmax regression with 2 classes is essentially a logistic regression

For multi-class classification, the loss function is called categorical cross entropy

A picture containing handwriting, font, calligraphy, typography

Description automatically generated (j -> 1 to number of classes)

Then, the cost function is:

A picture containing handwriting, font, calligraphy, typography

Description automatically generated

The gradient descent is the same, the difference is when we compute dZ [L] = y hat – y, which has the shape (classes no, 1)

**Introduction to Programming Frameworks**

**Deep Learning Frameworks**

Pick a deep learning framework based on:

Ease of programming (development and deployment)

Running speed

Truly open

**TensorFlow**

Mock-up example to minimize a cost function J(w)

You can optimize it in 2 ways:

1. Call the following function multiple times (like 1000 times)

A picture containing text, font, screenshot

Description automatically generated

tf.GradientTape – records the sequence of operations for the forward propagation and then you can “go back” on the tape to compute the backpropagation

Withing the gradient tape context, the tf.Varaible(s) is(are) watched

2. Have variable coefficients and instead of the last 3 lines and the gradient tape, use .minimize of the optimizer

A screenshot of a computer code

Description automatically generated with low confidence

While tf computes the forward propagation, it computes an execution graph that is used for the backpropagation

A picture containing handwriting, font, line

Description automatically generated

References:

[Introduction to gradients and automatic differentiation  |  TensorFlow Core](https://www.tensorflow.org/guide/autodiff)

[tf.GradientTape  |  TensorFlow v2.12.0](https://www.tensorflow.org/api_docs/python/tf/GradientTape)