**Week 1 – Notes**

**Setting up your Machine Learning Application**

**Train / Dev / Test sets**

A mistake made by professionals who transition from a domain to another (like from NLP to Computer Vision) is that they use the same intuition and the same space of hyperparameters, but this is wrong because for each domain you have to try something different

For small data sets (for example 10,000 examples) is ok to split the data into 60/20/20, but if the data set has like 1,000,000 examples, is enough to use 10,000 examples for development / test sets

For example, a split like 98 / 1 / 1 or 99.5 / .4 / .1 is enough

It’s extremely important to have at least for dev and test data sets the same distributions

Ex: train set – images from web, dev and test sets – images taken by users

Ideally, train / dev / test should have examples drawn from the same distribution

It’s alright to have only train and dev sets, as long as you acknowledge that the estimates made on the dev set are biased (probably you don’t need an unbiased estimate)

**Bias / Variance**

Bias (underfitting) and variance (overfitting)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Train set error | 1% | 15% | 15% | 0.5% |
| Dev set error | 11% | 16% | 30% | 1% |
| Bias / variance | High variance | High  bias | High bias &  High variance | Low bias &  Low variance |

These cases should be considered in relation with the optimal error and the quality of the data set; for example if the data is bad and even humans have a performance of 15%, then the case in which the train error is 15% and the dev error is 16% cannot be considered as a high bias case

When comparing the train and dev metrics, we don’t have to consider that 0.5% error on train and 1% error on dev is a case of overfitting only because the error is double; we have to consider the absolute values

A case of both high bias and variance is when we have non-linearly separable data and the classifier draw a barrier that is mostly linear, but with some irregularities

A drawing of a butterfly

Description automatically generated with medium confidence

**Basic Recipe for Machine Learning**

Initially you have to check it your network is biased (training data performance); solutions: bigger network, train longer, and eventually perform NN architecture search

Then you check for high variance (dev data performance); solutions: more data, regularization, and perhaps NN architecture search

Classical ML: there was a bias – variance tradeoff, because if you minimize one, the other one increases, but for DL there isn’t this tradeoff, because you can minimize one issue, w/o increasing the other one

**Regularizing your Neural Network**

**Regularization**

For logistic regression

A close-up of a handwritten note

Description automatically generated with low confidence

You add to the cost function a regularization term

In this case, the regularization term is (alpha / ( 2 \* m)) \* L2 or L1 regularization

A picture containing handwriting, text, font, calligraphy

Description automatically generated

A picture containing text, font, handwriting, white

Description automatically generated

L1 regularization gives sparse results, thus some weights will be zero, so the model will be simpler after regularization

However, L2 regularization is used more

For neural networks

A close-up of writing on a white board

Description automatically generated with low confidence

Here, instead of the vector norm, we use the summation of each the Frobenius norm of each W matrix

A picture containing text, font, diagram, white

Description automatically generatedA picture containing handwriting, text, font, calligraphy

Description automatically generated

This regularization is also called weight decay because is exactly as you would subtract in the gradient descent step also the (lambda / m) \* Wl (Wl being the weights of that layer, while updating Wl)

A close-up of a sign

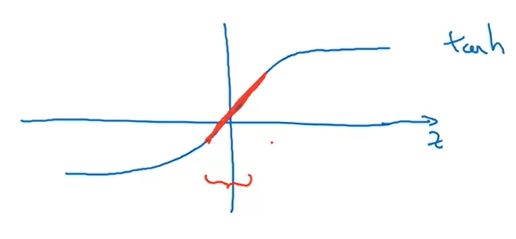
Description automatically generated with low confidence

Weight decay is a regularization technique (such as L2 regularization) that results in gradient descent shrinking the weights on every iteration

**Why Regularization Reduces Overfitting?**

When you add it, if you use a large lambda, a lot of weights will be zero, so then the network will become simpler, so it learns simpler functions

Additionally, because the weights will become smaller, then only the “linear” part of the activation function will be used, thus the overall network will be simpler



**Dropout Regularization**

While training, you shut off randomly a % of neurons, but you do not reset them to 0 (that will remove the learned info), you just don’t use them for the particular training example

A picture containing diagram, line, circle, origami

Description automatically generated

The most used implementation of dropout is the inverted dropout when you multiply the a value of each layer with zero values based on a keep\_prob threshold; it’s extremely important to divide the a with keep\_prob threshold, so that the next layers are not affected because of the shutting down of some neurons; additionally, this helps while testing when you do not apply dropout

However, you can apply dropout at test time, but you have to predict for each input example multiple times and in the end to compute the mean of the predictions, because otherwise the predictions would be random noise

**Understanding Dropout**

Because a lot of weights are randomly turned off, the NN is forced to spread out the weights, so there isn’t too much emphasis on one feature

For different layers there are used different values of the keep\_prob, for example the large layers where the overfitting is most likely, there lower keep\_prob values are utilized

Use dropout only if your NN overfits, not just as a default setting as many computer vision practitioners do

The downside of using this technique it that you cannot check the well-defined cost function (now being more irregular) – you can check it by removing the dropout layers

**Other Regularization Methods**

Data augmentation: for example flip images, rotate them, zoom in, add distortions

Early stopping: stop the training process when the dev set error increases

During the training process, the weights W become larger, so you may want to stop the process earlier so that the model doesn’t overfit

In deep learning, the training and optimization process is best to be done in 2 steps:

1. Focus on the optimization of the cost function (minimize it)
2. Only after that you may want to deal with the overfitting problem

This 2 step process is called orthogonalization

The downside of early stopping is that you have to do these 2 steps at once

**Setting Up your Optimization Problem**

**Normalizing Inputs**

Subtract the mean and divide by sigma; use the same mean and sigma extracted from the train data to normalize the test set

A picture containing diagram, drawing, circle

Description automatically generated

We want to normalize the inputs because if the inputs are not normalized, then the cost function has an odd shape and a smaller learning rate + many steps are needed to optimize the function, whereas if the features are normalized, it’s enough to use fewer steps

**Vanishing / Exploding Gradients**

When dealing with very deep neural networks, it’s important to initialize the weights correctly, because even if they are 0.5 or 1.5 after a forward pass through the network, they may explode (1.5^number of layers) or vanish (0.5^number of layers)

**Weight Initialization for Deep Networks**

When initializing weights, it’s important to consider that when there are multiple inputs in a layer, the values of the weights should be smaller because otherwise the output of the neurons will be larger => exploding gradients

A picture containing handwriting, line, text, font

Description automatically generated

One rule for initialization would be to initialize them such that the variance is 1 / number of neurons in the previous layer / input

Multiply the random weights (0 – 1)

For ReLU: by np.sqrt(2/number of neurons in the previous layer)

For Tanh: by np.sqrt(1 / number of neurons in the previous layer) – Xavier initialization

Another example: np.sqrt(2 / (number of neurons in the previous layer + number of neurons in the current layer)

**Numerical Approximation of Gradients**

We want to check if we have implemented correctly the gradients, so now we compute them with a numerical approach

The original derivative formula is not as precise as the two-sided differenceA picture containing handwriting, text, font, calligraphy

Description automatically generated

A picture containing handwriting, line, text, diagram

Description automatically generated

A picture containing handwriting, font, text, calligraphy

Description automatically generated - this is the definition for extremely small values of theta

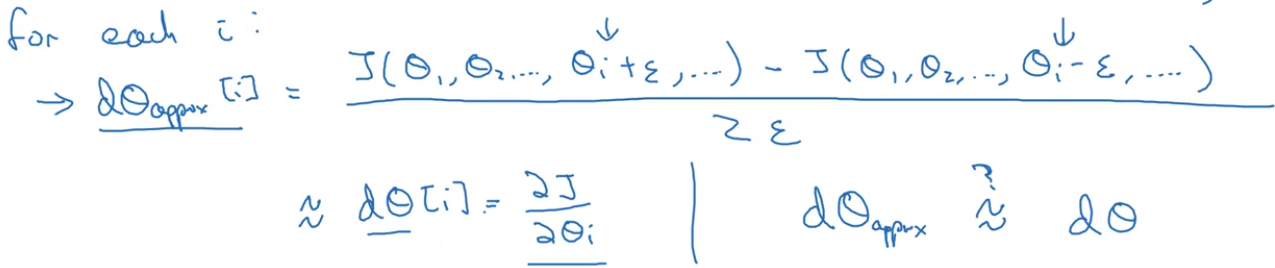
**Gradient Checking**

****

****

So that J(W1, b1,…. WL, bL) = J(theta1, theta 2,… theta L)

The idea is that for each theta you compute the numerical approx. of the gradient and compare it with the original gradient



The comparison should be based on

A picture containing handwriting, font, calligraphy, line

Description automatically generated (the normalization is included in case of very small d theta approx. and d theta)

For example, if you set the epsilon around 10^-7, then you expect that the above formula would yield a value around 10^-7. However, if the value is lager, you have to check your gradients (10^-5 – check, 10^-3 – worry)

**Gradient Checking Implementation Notes**

The numerical approximations of the gradients are slow to compute, so do not use them while training instead of the gradients computed with the original approach

If algorithm fails grad check, then consider verifying individual components such as db, dW

Do not forget to include the regularization while computing the gradients

This check doesn’t work with dropout because you don’t know for each example which nodes where turned off, and in the end you compute the derivative of J with respect to Wl for example

Perform the check with multiple random initialization because maybe for small values of W and b the check is ok, but for greater values there might be problems

**Initialization Assignment**

A well-chosen initialization can speed-up the convergence of gradient descent

Initializing W and b with zeros: the model doesn’t learn anything and it doesn’t break the symmetry, so each layer is equivalent to only one neuron because all learn the same function

Initializing W randomly and b with zeros: breaks the symmetry, but however there might appear the problem of vanishing / exploding gradients and the training process is slower; for initialization use randn (normal distribution) not rand (uniform distribution)

He initialization: sqrt(2 / layers\_dims[l-1]) – only for W, but b is initialized with zeros; this initialization works well with ReLU activations