IST 690 Independent Study Deep Learning
Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

by

Luigi Penaloza

Professor Stephen Wallace

The second part of Andrew Ng's specialization focuses on getting the most out of a neural network, and what is learned in this course can be directly applied unto the first course where Andrew taught how to build them. By teaching hyperparameter tuning, regularization and optimization, I learned how to improve on existing neural networks. We learn how to fine tune them by addressing frequent problems encountered such as overfitting which means that the model learns details and noise in the training data to an extent that it negatively impacts the performance of the model on new data. Another frequent problem we learn to deal with is vanishing/exploding gradients, which has the effect of making it hard for neural networks to learn or tune parameters early. Especially when the number of layers in the neural network architecture increases.

This course teaches how to find the right weight initialization, how to use dropouts, regularization and normalization. It also does a good job at showing how the different variants of optimization algorithms work and which ones are the right ones to choose from, depending on the needs or the problem to be solved.

Interestingly a takeaway point from this course is that randomness can at times be a better option. When hyperparameter tuning, random values seem to work better than following a standard approach because by doing so in a defined space on a right scale works better than using a grid search per se.

The course ends with an introduction to TensorFlow, an open source library supported in Python for machine learning applications such as neural networks. TensorFlow allows for the creation of neural networks without having to do it from scratch.

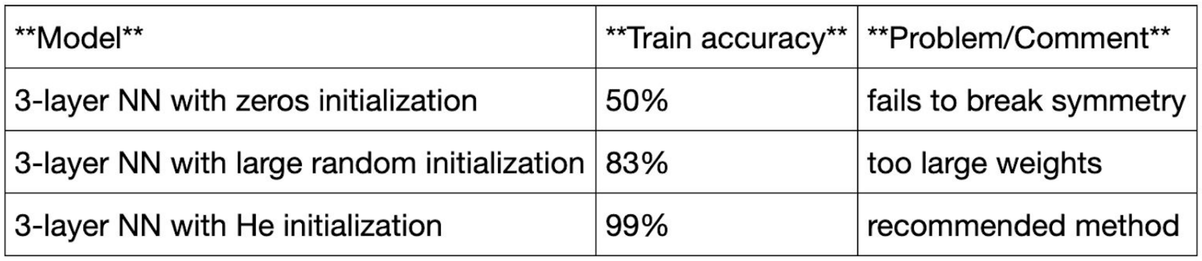
With the culmination of course 2, I have now learned to build a basic neural network and how to tune it with initialization, gradient descent, regularization and batch normalization. I've had practice in Python and worked with NumPy and TensorFlow, all useful skills for future career.

Below is a brief summary of the different steps learned in this course regarding hyperparameter tuning:

Initialization refers to the initial value of the weights given to train the neural network. If initialization is well chosen, then the neural network can learn better, as it can speed the convergence of gradient descent and increase the odds of gradient descent converging to a lower training and generalization error.

The table below compares three different types of initializations. All with the same number of iterations and hyperparameters.

- Zeroes initialization sets zeroes as the input argument.
- Random initializations sets the weights to large random values.
- He initialization sets the weights to random values according to a paper by He et al., 2015



Model	**Train accuracy**	**Problem/Comment**
3-layer NN with zeros initialization	50%	fails to break symmetry
3-layer NN with large random initialization	83%	too large weights
3-layer NN with He initialization	99%	recommended method

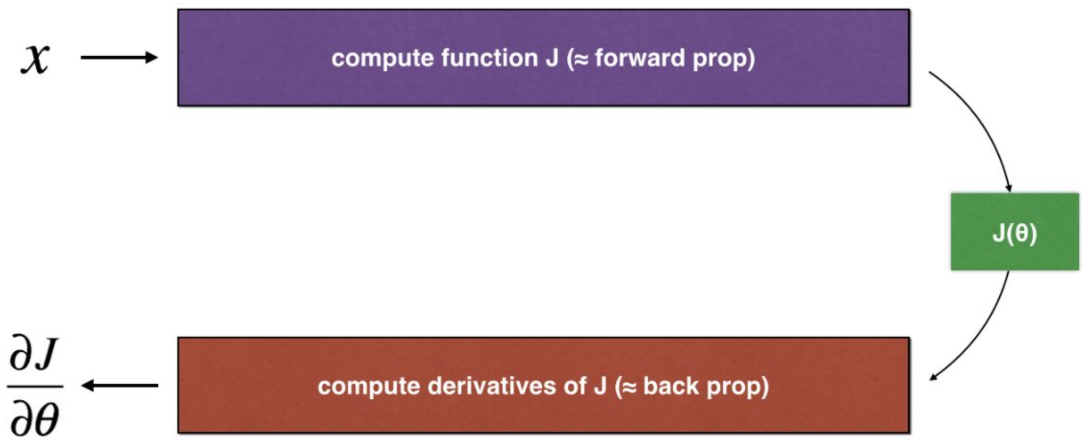
Regularization is the technique used to avoid overfitting. Because deep learning models have a lot of flexibility and capacity, overfitting can be a big problem if training dataset is not big enough. It might do well on the training set, but the learned network will not generalize to new examples that it hasn't seen.

The standard way to avoid overfitting is called L2 regularization, and it consists of appropriately modifying your cost function. Given that we modify the cost function we also need to change the backward propagation. This means that all the gradients need to be computed with respect to the new cost as well. As a result any overfitting model, should not be overfitting on the training data any more.

L2 regularization assumes that a model with small weights is simpler than one with large weights. Therefore by penalizing the square values of the weights in the cost function you're driving all the weights to smaller values. It becomes too costly for the cost to have large weights which leads to a smoother model where the output changes more slowly as input changes.

Gradient Checking is a method of numerically checking the derivatives computed by code to make sure the implementation is actually correct. Because forward propagation is easier to implement, we consider it to be correct, and gradient checking is a way to debug the back propagation algorithm used.

Diagram below shows the key computation steps for gradient checking:


x
compute function J (≈ forward prop)
J(θ)
compute derivatives of J (≈ back prop)
$\frac{\partial J}{\partial \theta}$

Optimization Methods that can speed up learning include:

1. Gradient Descent- Taking gradient steps with respect to all m examples on each step.

2. Mini Batch Gradient Descent- Like gradient descent, but each batch has just one example. Composed of shuffling, which creates a shuffled version of the training set and partition which divides the set into mini batches.

3. Momentum- A good analogy for this method would be a ball going downhill building speed adding momentum according to gradient/slope of the hill. Takes into consideration past gradients by using weighted average.

4. Adam- One of the most effective algorithms. Calculates exponentially weighted average and weighted average of the squares of past gradients to update parameters in a direction based on those averages.