# Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization (Coursera course notes)

Created by Aldo Zaimi

October 25, 2019

- **Week 1: Practical aspects of deep learning**
  - **Train/dev/test sets:**
    - Standard split: training set, development set (also called hold-out, cross-validation or validation set) and test set.
    - Standard procedure: use the training set to train your model, use the dev set to optimize your model (hyperparameters) and select the best model, use the test set to evaluate the performance of the selected model.
    - When working with smaller datasets (100-10,000 samples): use traditional split 70/30%.
    - When working with big data (more than 1,000,000 samples): use a higher portion for the training (e.g. 98/2% split).
    - Make sure the training and dev/test sets come from the same distribution.
  - **Bias/variance:**
    - High bias leads to underfitting, while high variance leads to overfitting.
    - Overfitting (high variance): when the dev error is much higher than the training error.
    - Underfitting (high bias): when both the training and dev errors are similar and high.
    - High bias and high variance: when the training error is high, and the dev error is much higher than the training error.
    - Low bias and low variance: when the training error is low, and the dev error is similar to the training error (typically a little higher).
    - To detect high variance: look at the difference between the training error and the dev error (higher difference usually indicates inability to generalize well).
    - To detect high bias: look at the training set error (higher training error usually indicates high bias).
  - **Basic recipe for machine learning:**
    - Step 1: train a baseline network.
    - Step 2: Detect if the network has a high bias (monitor training error). If it is the case, reduce the bias by using a bigger network, training longer and/or using a different architecture.
    - Step 3: Detect if the network has a high variance (monitor difference between training and dev errors). If it is the case, reduce the variance by using regularization, using more data and/or using a different network architecture.

    * Note that in the modern deep learning era, the bias/variance trade-off is much less frequent (it is easier to have both low variance and bias).

- **Regularization:**
    * L2 regularization (the most commonly used in deep learning): minimizing the squared norm of the parameters (weights) of the network.
    * L1 regularization: minimizing the sum of the parameters (weights) of the network.
    * Regularization can be controlled via the regularization parameter *lambda*.
    * L2 regularization is also called weight decay.
    * If the L2 regularization parameter is too restrictive (high), the weights of the network will tend to zero and lead to higher bias (simpler model not capable of fitting the data).
- **Dropout regularization:**
    * Dropout: adding a probability of dropping neurons in the network.
    * Dropout has a regularization effect that spreads out the weights (and shrinks them) across the neurons of the network.
    * Inverted dropout technique: to take into account the lower values obtained during training (due to using less nodes because of the dropout rate), the outputs of the layers are scaled up by a factor corresponding to the dropout parameter (i.e. the keep probability). This is equivalent to scaling down by the same factor during test time (instead of scaling up during training).
    * Different dropout parameters (between 0 and 1) can be used across the different layers, depending on the risk of overfitting (e.g. layers more prone to overfitting can have a higher dropout).
    * Mostly used in computer vision because of the lack of data (more risk of overfitting).
    * Implementation tip: first make sure the standard cost function (without dropout) is decreasing during training before adding dropout (to make sure that your dropout implementation does not add any bugs to the pipeline).
- **Other regularization methods:**
    * Data augmentation: adding random transformations and distortions to your dataset (flipping, rotation, rescaling).
    * Make sure that your data augmentation procedure doesn't affect the integrity and quality of your data (only use realistic distortions for the given application).
    * Early stopping: when plotting the training and dev errors vs the number of iterations, stop the training process when the dev error starts to increase significantly (to avoid overfitting).
    * Downside of early stopping: it focuses on reducing overfitting instead of optimizing the loss function.
- **Input normalization:**
    * Standard normalization procedure of input features: subtract average and divide by the variance.
    * The same normalization procedure should also be applied to the test data.
    * Advantage of input normalization: makes the cost function easier to converge.
- **Vanishing/exploding gradients:**
    * Cause: gradients can become very small (vanishing) or very large (exploding) when training very deep networks, affecting learning.

    - One solution: better weight initialization.
  - **Weight initialization for deep networks:**
    - Variance that can be used for weight initialization: 1/(number of weights in layer).
    - Multiply every weight by the square root of the chosen variance.
    - When using ReLU activations: variance of 2/(number of weights) is more optimal.
    - When using tanh activation: variance of 1/(number of weights) is more optimal (also called Xavier initialization).
    - Note that the variance used to initialize the weights can be tuned like an hyperparameter of the network.

- **Week 2: Optimization algorithms**
  - **Mini-batch gradient descent:**
    - Batch gradient descent: running through all the samples of the training dataset before updating the gradients and weights.
    - Mini-batch gradient descent: only using a subset of the training dataset before updating the gradients and weights.
    - Mini-batch gradient descent algorithm: (i) compute the forward pass of the mini-batch, (ii) compute the cost function using the samples of the mini-batch, (iii) apply back-propagation to compute the gradients and update the weights, (iv) repeat for every mini-batch.
    - Epoch: one pass through the entire training dataset (batch).
    - Visualization: the cost function curve vs the number of iterations is typically noisier when using mini-batch gradient descent instead of batch gradient descent.
    - The mini-batch size should be between 1 (stochastic gradient descent) and the number of training data (batch gradient descent).
    - Stochastic gradient descent: very noisy, high chances of getting stuck in local minima, very slow and difficult to converge, does not make use of vectorized implementation.
    - Batch gradient descent: takes too much time per iteration.
    - Mini-batch gradient descent: tradeoff between the two.
    - If small training dataset (less than 2,000 samples): use batch gradient descent instead.
    - Choosing the mini-batch size: typically between 64 and 512 (using powers of 2), depending on the application (size of each sample) and available CPU/GPU memory.
  - **Gradient descent with momentum:**
    - Exponentially weighted averages: algorithm that uses a moving average based on previous time points (controlled by the *beta* parameter).
    - When the parameter *beta* is large (close to 1): (i) gives more weight to previous data points, (ii) results in a smoother curve.
    - When the parameter *beta* is small (close to 0): (i) gives more weight to current data point, (ii) results in a noisier curve.
    - The exponentially weighted averages algorithm can be seen as an element-wise product between the data points and a corresponding exponentially decayed function.
    - Momentum in machine learning: (i) computes an exponentially weighted average of the gradient during gradient descent, (ii) helps reducing oscillations during gradient descent

steps (smoother learning), (iii) makes gradient descent run much faster and converge faster, (iv) avoids the need to use larger learning rates, (v) weights of the network are updated using the exponentially weighted gradients instead of the current gradients.
    * Analogy for momentum: (i) current gradients can be seen as an acceleration term, (ii) past gradients can be seen as a velocity term (averaged gradients from previous steps), (iii) the *beta* parameter can be seen as a friction coefficient (a larger *beta* will lead to smoother learning and give more weight to previous gradients).
    * Hyperparameter *beta*: (i) typically 0.9 (can be seen as approximately averaging over the last 10 gradient steps), (ii) can be tuned differently and in accordance with the learning rate hyperparameter.
  - **RMSprop (root means squared prop) algorithm:**
    * We want to (i) slow down the learning in the direction perpendicular to the minima and (ii) speed up the learning in the direction of the minima.
    * RMSprop: (i) computes an exponentially weighted average of the squares of the gradients, (ii) updates the network weights by dividing the current gradients by the square root of the exponentially weighted average.
    * Intuition: oscillating gradients will be slowed down.
  - **Adam optimization algorithm:**
    * Combination of momentum (using hyperparameter $beta1$) and RMSprop (using hyperparameter $beta2$).
    * Very commonly used learning algorithm that works for many architectures.
    * Three hyperparameters to tune: (i) learning rate, (ii) momentum hyperparameter $beta1$ (typically 0.9), (iii) RMSprop hyperparameter $beta2$ (typically 0.999).
    * In most cases, hyperparameter tuning is only performed on the learning rate (the default values are used for the momentum and RMSprop hyperparameters).
  - **Learning rate decay:**
    * Consists of slowly reducing the learning rate over time so that (i) you can take larger learning steps at the beginning of the learning and (ii) take smaller learning steps when the algorithm approaches convergence.
    * Implementation: (i) set an initial learning rate, (ii) add a decay rate hyperparameter that will gradually lower the initial learning rate depending on the number of epochs done.
    * Other forms: (i) exponential decay, (ii) discrete staircase decay, (iii) manual decay.
  - **The problem of local optima:**
    * In higher dimensional problems, most points with a zero gradient are saddle points (and not local optima).
    * Plateaus (regions with very small gradient) can slow down the learning.
    * Algorithms such as momentum, RMSprop and Adam can really help solving these issues during learning.

- **Week 3: Hyperparameter tuning, batch normalization and programming frameworks.**
  - **Hyperparameter tuning:**

  - The learning rate is usually the most important hyperparameter to tune.
  - Other secondary hyperparameters that may need tuning: momentum, Adam optimization parameters, number of layers, number of hidden units, learning rate decay, mini-batch size.
  - Use random sampling instead of grid search when you optimize hyperparameters.
  - Use the appropriate scale when sampling hyperparameters values: (i) standard linear scale for hyperparameters such as number of layers or number of hidden units, (ii) log scale for hyperparameters such as learning rate and momentum (e.g. sampling between 0.001 and 1).
  - Two main approaches depending on your computational power: (i) babysitting one model (do all the hyperparameters tunings on the same model), (ii) training many models in parallel (with different hyperparameter settings) and pick the one that performs best.

- **Batch normalization:**
  - General idea: normalize the outputs of the previous layers in order to help the learning of the next layer of the network.
  - Standard implementation: for a given layer, (i) compute the mean of the output values (before activation), (ii) compute standard deviation of the output values (before activation), (iii) normalize all the output values by subtracting the mean and dividing by the standard deviation to get a distribution with zero mean and variance of one.
  - If you want your hidden units to have a different distribution: add two learnable parameters *gamma* (factor to control variance) and *beta* (bias to control mean).

- **Batch normalization for neural networks:**
  - Batch normalization (BN) can be seen as a new layer added between the computation of outputs $z$ of a layer and the activation function $a$.
  - Each BN layer has two learnable parameters (can be learned by gradient descent as well) to control the mean and variance of the normalized distribution.
  - Example of workflow for a network with 2 hidden layers: (i) insert input tensor $X$, (ii) compute output tensor $Z1$ using $X$ and weights $W1$, (iii) compute $Z1bn$ by applying BN to $Z1$ using BN parameters $m1$ and $v1$, (iv) apply activation function to $Z1bn$ to get $A1$, (v) compute output tensor $Z2$ using $A1$ and weights $W2$, (vi) compute $Z2bn$ by applying BN to $Z2$ using BN parameters $m2$ and $v2$, (vii) apply activation function to $Z2bn$ to get $A2$, (viii) compute output layer $Y$ using $A2$ and weights $W3$.
  - BN is typically applied with mini-batch gradient descent.
  - Implementing BN with gradient descent: (i) compute forward propagation on a mini-batch, (ii) in each hidden layer, use BN to normalize the values just before the activation function, (iii) use backpropagation to compute gradients for the weights and BN parameters, (iv) update the weights and BN parameters of the network with gradient descent.
  - BN is already implemented as a layer in most deep learning frameworks.

- **Why batch normalization works:**
  - Covariance shift: (i) makes later layers' weights of the network more robust to changes in early layers' weights, (ii) weakens the coupling between the weights of the early layers of those of the later layers so each hidden layer can learn more independently from the others, (iii) speeds up the learning process overall.

  * Slight regularization effect: (i) scaling using mean and variance computed on a mini-batch adds noise to the activations of the hidden layers (similar to dropout), (ii) forces the network to generalize better and not rely too much on specific hidden layers/units, (iii) this added noise can be reduced by using a larger mini-batch size.
- **Batch normalization at test time:**
  * During training, the mean and variance of BN are computed on a mini-batch.
  * During test time, another approach is needed to estimate the mean and variance, as we are usually working with single inputs instead of mini-batches.
  * To get an estimate of the mean and variance for test time: (i) keep track of the means and variances from the mini-batches during training, (ii) use exponentially weighted average to get an estimate of the mean and variance, (iii) use these estimated values of the mean and variance at test time to scale the hidden layers values.
- **Softmax regression for multiclass classification:**
  * Softmax activation function: (i) takes a vector of the same size as the number of classes as input, (ii) outputs a vector of the same size that represents the probabilities of the sample being one of the classes.
  * The softmax layer is the output layer of a multiclass classification problem.
  * Loss function for a softmax layer: cross-entropy loss.
- **Deep learning frameworks:**
  * How to choose a deep learning framework: (i) ease of programming (development and deployment), (ii) running speed, (iii) open source.