|  |
| --- |
| Setting up your Machine Learning Application |
| Regularizing your neural network |
| Setting up your optimization problem |
| Optimization algorithms |
| Hyperparameter tuning |
| Batch Normalization |
| Multi-class classification |
| Introduction to programming frameworks |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Setting up your Machine Learning Application**  **Train/Dev/Test Set:**   * There are n numbers of parameters for DL/ML * It is very difficult/ impossible to figure out the different parameters of the model beforehand for different problems. * Hence, applied ML/Deep Learning process is highly iterative in nature. * Faster we can iterate, we can make efficient model faster * Train/Dev/Test dataset segregation is a strategy to support this approach * Dev and test set should come from same distribution  |  |  |  |  | | --- | --- | --- | --- | |  | Train | Dev | Test | | Significance | Which is used to train the model | To find which model perform well | Unknown dataset / real case scenarios | | Division (Pre-Big Data) | 70%/60% | 15%/20% | 15%/20% | | Division (Big – Data era) | 98%/99% | 1%/0.5% | 1%/0.5% |   **Bias/Variance:**   * **Bias** signifies how the model fits the data in Test Set **(Underfit = High Bias, Good fit = Low bias, overfit = very low bias)** * **Variance** signifies how the model performs on Dev/Test set of data. If differs a lot w.r.t Train set then we have high Variance, if the difference is very little then we have low variance * In Pre-deep learning era, we used to hear trade-off between bias and variance. Now it is not heard much rather in any model we have some bias and variance        * Looking into the Train set error we can identify the bias of the model * Looking at the Dev set error we can identify the variance of the model   **Basic Recipe for Bias/Variance problem in Deep NN :**   * First test in train set if the model is having high bias   If Yes 🡪 One of the following technique could be helpful   * Using bigger network * Trying different NN architecture * Train for longer period of time * Then to test the variance problem check the performance of the model in the dev set   If there is high variance then following technique could be helpful   * Using more data to train * Use regularization technique * Trying different NN architecture   In pre-deep learning era we often heard bias/variance trade- off. The reason behind that was not much availability of good tools which can only reduce bias without impacting variance and vice versa. But, now in deep learning we can reduce bias by using bigger netork and would not impact the variance. Likewise, if we use more data to train we will only address variance without impacting bias. |
| **Regularizing your neural network**  **Regularization:**  Regularization is a technique through which we can reduce variance in the neural network.  In Logistic Regression –  We calculate cost function J(W, b) and try to minimize J by adjusting parameters W/b  Below is the cost function with regularization term.  **Regularization in Neural Network:**  Likewise, we can apply the same into NN as follows  L2 regularization is called weight decay – |
| **Why L2 regularization helps to reduce Variance/ Overfitting?**  **Intuition 1:**  The L2 regularization works as if we are making W parameter near equals zero thus the overall network becomes very simple NN or a linear network    **Intuition 2:**  Let say we are using activation function tanh. So from the figure below we can see that if we have value of z very high then it will fall in non-linear region of the tanh function. Using regularization, we are trying to control parameter w to not become very large. Hence, z will oprate mostly in the linear region of the tanh function. So, the model will not experience too much non-linearity.    **Dropout Regularization:**  Dropout is another popular regularization technique which is used to get rid out of overfitting.  With this technique we are randomly dropping some connections of neurons at each layer during training time. Hence, it would help removing dependency of any neuron to any specific feature and it would try to give weightage (may be small) to other features as well.    **Implementation of dropout (“Inverted Dropout”)**   * Set one keep-prob (0-1) * Apply dropout at each layer with probability = 1 – keep-prop * Calculate activation with a scaling factor of keep-prop to compensate the reduction of values due to drop out * The drop-out is applied only in train set, while test we remove dropout * Based on the architecture different keep-prop can be applied at different layers. For example, the layers with more hidden units can have less keep prob whereas with less hidden units having bigger keep-prop   **Why does dropout work?**   * it force the network to not depend on only one / some specific set of features. Thus, the network tries to distribute the weights across features * after applying dropout the network becomes relatively small. So it less non-linear in nature and helps removing overfitting   **Downside of dropout regularization:**  With dropout applied we won’t have well defined monotonically decreasing J w.r.t iteration. So, it is hard to debug    **Other Regularization Techniques:**  **Data Augmentation :**  It is a cost effective way to synthesize more data from the data in hand. Then use it to tran the model |
| **Early Stopping**  With this technique we stop early in training phase before the model overfits.  We plot the cost function value w.r.t iteration for train and dev. Right when the dev curve diverges, we stop training more.  One downside of this approach it works contrary to Orthogonalization (separate task for cost optimization and overfitting) |
| **Setting up your optimization problem**  **Normalization**   * It is a technique which helps to reduce the training time / help quickly optimization of the model * We need to subtract mean from the data and divide by variance     **Why normalizes input?**  Below is a very simple illustration of the cost function in 3D/ 2D surface. It turns out if we don’t normalize the input and there is a huge variation of ranges of input then the cost function is a elongated version. Where while training it might oscillate very much and take more time to converge. On the other hand, upon normalization, the cost function takes a well defined bow shaped where the convergence happens more smoothly.    **Vanishing and Exploding Gradients:**   * In deep neural network if we initialize w with slightly large or smaller than 1 then with each layer the action function value is exponentially increasing/decreasing. Thus, it causes gradient to become huge or very small. This problem is called exploding / vanishing gradient problem. |
| **Weight initialization for Deep Network:**   * This partially solves the vanishing or exploding gradient problem * There are different initialization techniques for different activation functions * **Generic (1/n) / (2/n) , where n = number of inputs** * **For relu, 2/(n^(l-1)** * **For tanh, sqroot(1/n^(l-1))** |
| **Numerical Approximation of Gradient Descent:**   * Instead of nudging function to one side, better to nudge in both sides * Below derivation proves that with this we can validate the derivative with numerical approximation to very small scale of error * Using this technique, we can easily verify if our gradient descent calculation is correct or wrong     **Gradient Checking:**   * with this technique we can validate if the backprop is working correctly or not * if the diff between numerical apx and the backprop cal is in the order of 10^-7 it signifies that the algorithm is perfect/ if in the order of 10^-5 then we need to check carefully/ if in order of 10^-3 then there is some issue in the model. |
|  |
| **Optimization algorithms**  **Mini-Batch Gradient Descent:**   * As applied machine learning is highly iterative in nature, the lower the training time will improve productivity * With this in mind we have applied vectorization for training to train the model with entiire training set with examples m and get rid out of explicit FOR loops. * While this approach helps hugely if the dataset size is small in order of thousands. But with the advent of big data era, we have dataset of size in order of milions or bilions. Then with the vectorization the performance is not as we expect * This makes necessity of following another approach of traning which is called mini-batch gradient decsent. * In this approach instead of training the model with entire dataset and update the parameters after 1 epoch of entire dataset we divide the dataset into small batch of data (64/128/256/512/1024). Then after each mini-batch iteration we update the parameters. * This process is faster for model convergance than gradient descent   **Illustration :**  Let say, batch size = 1000  **X =**   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **X(1)** | **X(2)** | **X(3)** | **…..** | **X(1000)** | **….** | **X(2000)** | **…** |  | **X(m)** |   **Y =**   |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **y(1)** | **y(2)** | **y(3)** | **…..** | **y(1000)** | **….** | **y(2000)** | **…** |  | **y(m)** |   **Notation :**   1. **= ith example**   **[l] = lth layer in the NN**  **{t} = t th batch of mini-batch gradient descent**   1. Divide the dataset into mini-batches of size = batchsize ( if data is 5,000,000 and batch size is 1,000. Then we will have 5,000 batches) 2. LOOP through the batches and update the parameters at each iteration. As if those each mini-batches are one gradient descent 3. At completion of all batches – repeat the process to the desired number of epochs |
| **Understanding Mini-batch gradient descent:**  **Different Types of Gradient Descents and applicability:**     |  |  |  | | --- | --- | --- | | **Gradient Descent** | **Mini-batch Gradient Descent** | **Stochastic Gradient Descent** | | batch size = m | 1<batch size<m  Conventional way is to take size in the order of 2  (64/128/256/512/1024) to bet fit into memory | batch size = 1 | | Takes huge time if data size is huge (order M/B) | Useful for huge dataset size (M/B) | Training time is worse for huge dataset as can’t get the benefit of vectorization | | Cost function is monotonically decreasing over iterations | Cost function is oscillating but trend is downward over iterations | Cost function is very much oscillating, but trend is downward over iterations | |  |  |  | |
| **Exponentially weighted Averages:**  Below is the graph showing the temperature in London for each day in a year.    Let say to we would like to plot a line graph that fits the data and we are using below equation to derive the value of temperature of one day  v = 0 initialize with zero  v\_1 = 0.9 \* v\_0 + 0.1 theta\_1 ; theta1 is the temperature of day 1  v\_2 = 0.9 \* v\_1 + 0.1 theta\_2 ; theta2 is the temperature of day 2  …  …  v\_t = 0.9\*v\_t-1 + 0.1 theta\_t  If we plot the graph for the equation above for each day then the graph will look like below  We can generalize the equation as follows:  This is equation for moving average/ exponentially weighted average. If we use value of beta = 0.9 then it roughly calculates average of (1/1-beta) = 10 previous days average  See below the impact of beta on the moving average line for the given data    This is a computationally efficient way to calculate exponentially weighted average.  We can calculate through moving window as well but that would be computationally more expensive as we need to stone n-size window data into memory |
| **Bias correction in EWA:**   * In practice we see if we select beta value very large close to one then it becomes less sensitive to current data. * As you can see that for beta = 0.9 we have the purple line not the green line. Which is shifted to right. * As shown below if we select v\_0 = 0 then in initial stage v changes very slowly * To reduce this issue we have one technique called bias correction, where instead of taking actual value of v\_t we take **v\_t = v\_t/(1-beta^t)** where t = # day   **Gradient Descent with Momentum:**   * In Gradient descent as illustrated below it oscillates and converges to minima * **Momentum** is introduced into gradient descent to reduce the oscillation, so that the model can converge faster. * **Momentum helps to reduce divergence in other direction and maintain the acceleration towards minima. ???**  * In Gradient descent with momentum we use EWA to update the parameters   Below is the implementation details.    **RMSprop**   * RMSProp helps to speed up optimization like momentum but here the update of parameters are different   Let say for sake of simplicity we have b and w 2 parameters and having direction as follows in the cost function space.  So, we want our algorithm to move towards more in w direction and less in b direction to quickly converge. So we need to reduce the effect of db and increase the effect of dW in our learning process. This RMSProp works to achieve this.   1. We calculate S\_dw 2. We calculate S\_db 3. We update the model parameters as follows 4. **If db is large then S\_db will be large and it will reduce the learning rate in b and if dW is small then it would increase the learning rate in W direction???** |
| **Adam Optimization** (Adaptive momentum Optimization):   * It worked well for almost all scenarios to optimize gradient descent * It uses best of momentum and RMSProp to update the parameters |
| **Best Practices in Selecting Hyper Parameters for Adam Optimization:**   |  |  | | --- | --- | | Beta 1 | 0.9 (dw) | | Beta 2 | 0.999 (dW)2 | | epsilon | 10^-8 | | alpha | Needs to be tuned |   **Learning Rate Declay:**   * Learning rate decay is another hyper-parameter which could be used for optimization problem * For mini-batch gradient descent we have more oscillation if we use same learning rate at every epoch. Also, it fails to reach to minima and oscillates near to minima. * If the learning rate is fixed and relatively large, then it would oscillate around a bigger region of the minima. If we select very small learning rate, then it would take ages to converge * To solve this problem, learning rate decay technique which can be used to reduce this problem.  |  |  | | --- | --- | |  |  | |  | exponential | |  | t = batch number | |  | Discrete decay | |
|  |
| **Hyperparameter** **tuning**   * As there are lots of hyperparameters available in deep neural network which can affect the performance of the model, often it becomes very difficult to follow good approach and systematically tune hyperparameters. * Although there are several params available all those are not equally important, so below approach can be a best practice to systematically tune the params  |  |  | | --- | --- | | **α(learning rate)** | **Priority 1** | | **β(momentum) ~ 0.9** | **Priority 2** | | **Ɛ (epsilon)** | **Priority 2** | | **# hidden units** | **Priority 2** | | **mini-batch size** | **Priority 2** | | **# layers** | **Priority 3** | | **#learning rate decay** | **Priority 3** | | **β\_1 ~ 0.9, β\_1 ~0.99 (Adam optimization)** | **Very Rarely required tuning** |  |  |  | | --- | --- | | **Commonly adapted approach** | **Best Practice** | | Hyperparameter 1  Hyperparameter 2 | Hyperparameter 1  Hyperparameter 2 | | In this approach let say we have 2 hyperparameters to tune we create grid of values and test each value | In this approach we randomly select some values from the 2 hyperparameter space. It ca also be extended to other dimensional space | | Works well if #params are small | Works well even if the #no of params are large |   **Coarse to fine:**  in this approach we first randomly put the values in hyperparameter space. Then when we find 2 or more datapoint for which we have better performance we try to sample more in the near by region of the same space. |
| **Using an appropriate scale to pick hyperparameters:**   * In last example we saw how randomly selecting hyperparameter value gives better result than grid search * However, we cannot apply uniformly distributed random parameters for all parameters. Below is the justification   Let say, we want to tune the # layers between 2-6 layers or we want to identify the correct value of the hidden layer (50-100). In this case we can follow uniformly distributed sample random values.  But if we want to find the value for alpha / beta then instead of following linear scale we should follow logarithmic scale.  Like alpha – tanging from 0.001 to 1, if use linear scale then 90% values will lie between 0.1 to 1  So here we can follow below  r = -4 \* np.random.randn()  alpha = 10^r  Same approach can be taken for beta values.  For beta if we want to tune between 0.9 to 0.999 it is better to use the scale for (1-beta) and follow the same approach as we did for alpha |
|  |
| **Batch Normalization:**   * Like we did the input normalization to improve the earning of the logistic regression, we can also normalize the layers of DNN to achieve better result of learning of hidden layers   Like we did the normalization for X to have faster learning of W,b for logistic regression,  We can apply this to the inner layer of DNN to normalize the value of activation a^[l] to have better learning W^[l+1], b^[l+1]. In practice instead of normalizing activation (a) we normalize z^[l] |
| **Implementing Batch normalization:**  Given some intermediate value in NN z^(1),z^(2),……,z^(m) for layer l   1. Calculate the mean 2. Calculate the variance 3. Calculate normalization 4. Calculate where   With which we can control the range of values to be used for Z norm |
| Let see how we can implement batch norm with 2 hidden layers.  So, we end up with following learning parameters **[**  We can apply gradient descent / RMS Prop or Adam optimization to learn beta and gamma just we did in W and b |
| **Why does batch normalization work?**  Let say we have a logistic network which is designed for Cat/Non-Cat classification.  It is trained with only images with black cat. Now if it is tested using other coloured cat images then it will not perform well. Because the distribution of test and train data is different. Even though the model function does not change it would not be able to predict correctly. This is called co-variant shift.  What normalization does is even though the data distribution changes it keeps the mean and variance intact so that the ranges of values does not change much. That helps in better learning efficiency for the model.  Now let’s analyse the case one DNN. Here if we consider the activation at layer 2 which is acting as input to layer 3. So, the data distribution of a^[2] is responsible or learning of W^[3] and b^[3].  Let’s assume that layer 2 is the input for the DNN wih 2 layers and we hide the previous layers.  So, if we apply batch normalization at a^[2] then it will help to improve learning for W^[3] and b^[3]. But question may be asked why we will have co-variant shift problem here.  The reason for that is we have hidden the earlier layers which is not reality. So the value of a^[2] depends n z^[2]. That depends on W^[2], b^[2]  As those parameters are continuously changing so a^[2] will face the problem of co-variant shift. Thus, applying batch normalization will help to improve the learning for W^[3],b^[3].  We can apply this to other layers as well and they will experience similar improvement  **Regularization effect of Batch Normalization:**   * Each mini-batch is scaled by the mean/variance computed on just that mini-batch. * This adds some noise to the values within that minibatch. So like dropout, it adds some noise to each hidden layer’s activations. * This has a slight regularization effect.   **Batch normalization at Test time:**   * During training we use mean and variance of the mini-batch [X^{t},Y{t}] * During testing, we would have one example at a time so it does not make sense of taking mean and variance of one example to calculate the batch norm. * Therefore, the standard practice is to use EWA of mean and variance form training mini-batches to use in testing |
| **Softmax regression:** |
|  |
|  |
| **Deep Learning programming framework:**  • Caffe/Caffe2  • CNTK  • DL4J  • Keras  • Lasagne  • mxnet  • PaddlePaddle  • TensorFlow  • Theano  • Torch  Choosing deep learning frameworks   * Ease of programming (development and deployment) * Running speed * Truly open (open source with good governance) |
|  |
| 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)) |