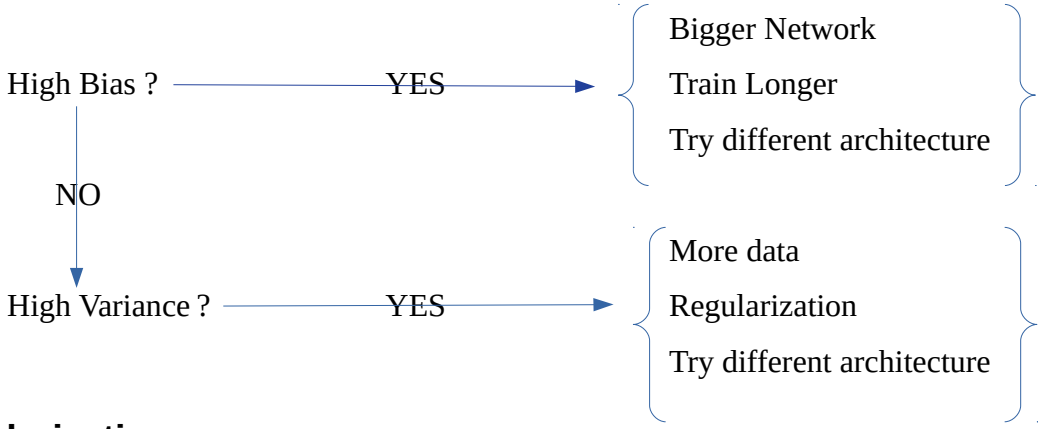
Hyper parameter Tuning

I have not put all the details neither all the topics, these are just for my reference. Although if you are seeing this,it can be a review for you if you have gone through these topics in details earlier. :)

Train / Validation / Test Data set

In traditional machine learning we used 70-30 or 60-20-20 train-test split. But as Deep Learning is growing we have more data. So don't have an exact ratio in which we can split the data. Example, if we have 1M examples then we can use 10k for validation and 10k for test and rest of data is used for training our model.
Make sure that test and validation set come from same distribution as your train test, and its okay to have no test set, validation set will work.

Basic recipe for Machine Learning



Regularization

#L2 regularization Adding L2 norm to our loss function

$$J(W,b)=1/m\sum_{i=1}^{m} L(y^{(i)},\hat{y^{(i)}})+\lambda/2m\|W\|_F^2$$

(m is the number of examples)This is also known as weight decay. Similarly we can take L1 norm also.
#Dropout with probability 'p' we keep neurons from a layer and drop neurons with probability '1-p' . Then we calculate the output of the layer by dividing by probability 'p'.
Intuition: Cannot rely on a particular neuron, so have to spread out the weights
One way of dropout is initially set p=1, if loss function is decreasing then we can do dropout, if this work then we keep working on this.
#Data Augmentation flip, crop, random rotation of images
#Early Stopping

Weight Initialization

Use random initialization for the weights of the network and divide by some appropriate number, we can choose this number to be a hyper-parameter. In Xavier Initialization we use *np.random.rand(shape)/sqrt(1/n^{l-1})*, when we are using ReLU we can choose factor to be *sqrt(2/n^{l-1})*, while some researchers also use the factor *sqrt(2/$n^{l-1}+n^{l}$)*, n^{l} refers to number of nodes in layer l.

Mini- Batch Gradient descent

Divide the whole data in mini batches, example if you have 5M examples then break it into batches if 10k, then we will have 500 such batches. Now we will pass these 500 mini batches of the entire data and compute 500 gradient descent steps and going through the entire data is called 1 epoch. The loss function in batch gradient decent goes down at every pass of data, while loss function in mini batch gradient descent is kind of noisy: it sometimes goes up and sometimes goes down, but overall it goes down.
Another Hyper-parameter : Size of mini batch, (usually 32,64,128,256)

Exponential Weighting Average

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

With different values of β,we get different moving averages, and on approximation we are averaging the data over 1/(1-β) past values.
#Bias Correction Initially we take $v_0 = 0$, So our $v_1 = (1-\beta)v_0$ and this goes on, so this in turn result in error in initial terms. To overcome this term we can do something known as bias correction i.e. $v_t = v_t/(1-\beta^t)$

Gradient descent with Momentum

On iteration 't'

Computer dW,db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)\, dW$$

$$v_{db} = \beta v_{db} + (1-\beta)\, db$$

$$W = W - \alpha v_{dW}, b = b - \alpha v_{db}$$

(Intuition: Ball rolling down a hill).

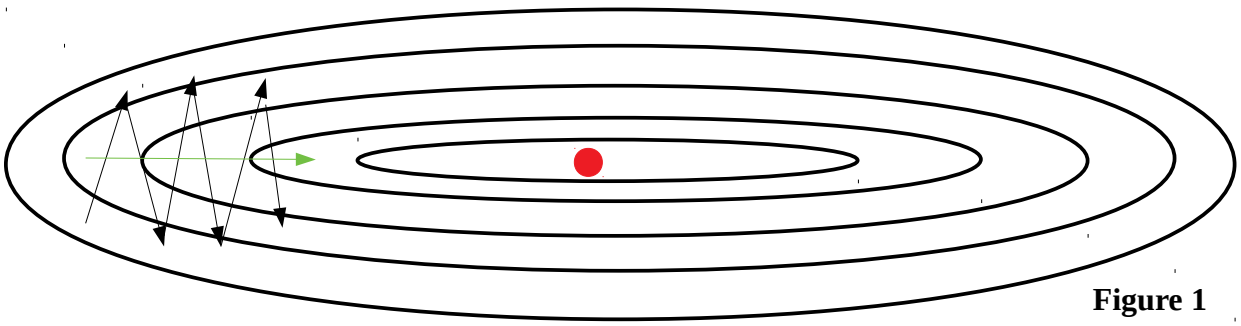


Figure 1

RMSprop (Root Mean Square Prop)

When you want to increase speed in one direction rather than other, in the above figure we want to want to move in right direction more rather than moving up and down
On iteration 't'

Computer dW,db on the current mini-batch

$$s_{dW} = \beta s_{dW} + (1-\beta)\, dW^2$$

$$s_{db} = \beta s_{db} + (1-\beta)\, db^2$$

$$W = W - \alpha(v_{dW}/\sqrt{s_{dW}}), b = b - \alpha(v_{db}/\sqrt{s_{db}})$$

Sometimes, this may happen that the denominator term might be zero or very close to zero. So we end up, adding a term ϵ in the denominator i.e. $W = W - \alpha(v_{dW}/\sqrt{s_{dW} + \varepsilon})$
Both Momentum and RMSprop are used for damping out the oscillations

Adam (Adaptive moment estimation)

Set $v_{dw}=0, s_{dw}=0, v_{db}=0, s_{dw}=0,$

On iteration 't'

Computer dW,db on the current mini-batch

$$v_{dW}=\beta_1 v_{dW}+(1-\beta_1)dW, v_{db}=\beta_1 v_{db}+(1-\beta_1)db$$

$$s_{dW}=\beta_2 s_{dW}+(1-\beta_2)dW^2, s_{db}=\beta_2 s_{db}+(1-\beta_2)db^2$$

$$v_{dW}^{corrected}=v_{dW}/(1-\beta_1^t), v_{db}^{corrected}=v_{db}/(1-\beta_1^t),$$

$$s_{dW}^{corrected}=s_{dW}/(1-\beta_2^t), s_{db}^{corrected}=s_{db}/(1-\beta_2^t),$$

$$W=W-\alpha(v_{dW}/\sqrt{s_{dW}+\varepsilon}), b=b-\alpha(v_{db}/\sqrt{s_{db}+\varepsilon})$$

Author of Adam paper recommend $\beta_1 = 0.9$,$\beta_2 = 0.999$ and $\epsilon = 10^{-8}$

Learning Rate Decay

Since we are using mini batch gradient descent for our weights update, with constant learning rate the search goes on and on and there is no stopping, but if we reduce the learning rate with epochs the minimum get stuck in a region rather than wondering around.

So one way to decay learning rate is to use new learning rate $\alpha=\left(\frac{1}{1+decay\ rate*epoch\ num}\right)\alpha_0$

or we can use $\alpha=0.95^{epoch\ num}\ \alpha_0$ or sometime people also use step decay i.e. half the learning rate after certain number of epochs.

Tuning Process

- Some hyper parameters are important than others, try to tune them first, if some standard values are provided by literature use that for low priority hyper-parameters
- **Don't use grid**. Computationally expensive!!
- Try random values
- **Coarse to fine**. If you find some hyper-parameters are good in a region, then zoom in and randomly sample more values of hyper-parameters to get better results

Sampling hyper-parameters

Use log scale to sample hyper-parameters. Example if you are sampling from 0.0001 to 1, and if ypu randomly sample in this range then more sampling would be done from 0.1 to 1 than sampling from 0.0001 to 0.1, use r = -4*np.random.rand() and samples = 10^r .

Similarly if you have samples from range 10^a to 10^b , then r = a + (b-a)np.random.rand() ans take samples = 10^r.

Batch Normalization

Just like we normalize the input of logistic regression model in machine Learning, to get a better model. We try to normalize the output of all the all the layer so that model can learn better and faster. Usually the output of the neuron (i.e. before applying the activation function of the layer) is normalized, rather than the post activated outputs.

Given intermediate outputs $z^{(1)},z^{(2)},...,z^{(m)}$. we calculate mean $\mu=\frac{1}{m}\sum_{i=1}^{m} z^{(i)}$ and variance

$\sigma=\frac{1}{m}\sum_{i=1}^{m}(z^{(i)}-\mu)^2$ then the transformed outputs are $z_{norm}^{(i)}=\frac{z^{(i)}-\mu}{\sqrt{\sigma+\varepsilon}}$

Another way of normalizing outputs is to use $\widetilde{z^{(i)}}=\gamma z^{(i)}+\beta$ where γ and β are learnable parameters.

#Working with mini-batches

$$X^{\{1\}} \xrightarrow{W^{[1]},b^{[1]}} z^{\{1\}} \xrightarrow{BN,\ \beta^{[1]},\gamma^{[1]}} z^{`\{1\}} \xrightarrow{Activation} a^{\{1\}}$$

We calculate the output of the first layer, then we apply batch normalization in which we normalize the whole batch by subtracting mean and dividing by variance, after obtaining the normalized inputs we multiply and add the learn-able parameters β and ɣ. In the final step of first layer we then apply the activation function and this process repeats every layer.

Why batch-norm works?

Batch normalization reduces the amount by what the hidden unit values shift around (co-variance shift).

Batch norm at test time

At test time we might have different mini batch size or we might be processing 1 example at a time. So we need to mean and variance to normalize the input, but with one example we cannot do that. To deal with problem we take exponential average or running average of the mean and variance during the training time and use it while testing the data.

References:

https://www.youtube.com/playlist?list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc