Applied ML is highly iterative process
Idea → code → experiment
Use experiment results to refine idea and repeat

To be more efficient on the iterative process

1. Split data into train/ dev/ test sets
    - Dev set = hold-out cross validation set
        - Used to see which model perform best on the dev set, pick the best model to evaluate on test set (to get unbiased estimate)
    - 60:20:20 split ratio rule on small dataset
    - For large dataset e.g. 1M rows, 10K (1%) examples for each evaluation set (1-5% (small portion) of examples are enough)
    - Dev and test set should come from same distribution
    - Test set is optional but should exist if we don't want to make model overfit to dev set (not overfit = perform well on other unseen data) i.e.want to get unbiased performance estimate of selected model
2. Bias and Variance
    - It is hard for models to give val performance more than train performance **
    - High bias = underfitting -- not perform well on seen data (train set) -- train & val performance low (nearly the same)
    - High variance = overfitting -- not perform well on unseen data (val set) -- train performance good but val bad performance
    - High bias, high variance = underfitting + overfitting -- not perform well on train set but perform better on train set than val set (worst case)
    - Low bias, low variance = good fit -- perform well on train & val set (nearly same performance)
    - bias variance trade off -- when bias decrease variance increase and vice versa
    - Train error high → high bias
    - Val error high → high variance
3. High bias solution -- solve before high variance problem
    - Try bigger & more complex network
    - Train longer
4. High variance solution
    - Add more data
        - Data augmentation e.g. in computer vision, flip or rotate image etc.
    - Early stopping
        - Stop training when val error increase
        - Help prevent overfitting but cost function is not minimized as much as possible
    - Use regularization
        - L1 regularization -- ends up making weight matrix sparse with a lot of zeroes
        - L2 regularization -- similar to weight decay -- weights are updated (subtraction & multiplication with fraction (< 1)) -- weight is more close to

zero → simpler network, less over fitting (effects of hidden units are still used but the effects are reduced as weight is not zero)
    - Regularization param is tunable should not be too high (underfitting as many of neurons are not considered) or too low (overfitting as model has a lot of neurons or has high complexity)
- Use Dropout
    - Not using some proportion of neurons in specified layers.
    - Not using = removing all incoming and outgoing edges from the discarded neurons.
    - Inverted drop out
        - Create random number matrix with same shape as layer weights (layer weight = matrix containing all of layer's neuron weight)
        - Each random number belongs to each weight (edge incoming to neuron in the layer). If the rand number is less than keep probability → keep the weight else reset the edge weight to zero.
        - Compute layer output using weights processed in previous step
        - Correct (scale) layer output by dividing original output with keep probability
        - Drop out is used in both forward and backward propagation
    - Don't use drop out at test time as it will add noise to prediction
    - Drop out make the model not rely on a particular feature for prediction, drop out help spread neuron weights → reduce weight for each feature similar to L2 regularization
    - Layers that have many parameters can overfit. To dropout → set low neuron keep probability for the layer (drop more)
    - Using dropout make loss harder to calculate

** The number of neurons in the previous layer gives us the number of columns of the weight matrix, and the number of neurons in the current layer gives us the number of rows in the weight matrix.

5. Normalize inputs
    - Normalize = zero out the mean, make the variance = 1
    - Help speed up training
    - Make cost function look more symmetric, faster to converge to minimum cost function value
6. Exploding & vanishing gradient problem
   For deep neural networks
    - If large weight value is initialized → exploding gradient -- activation values become larger & larger & very large at last layer -- gradient is large
    - If small weight value is initialized → vanishing gradient -- activation values become smaller & smaller & very smaller at last layer -- gradient is small -- gradient descent take tiny little steps (slower to converge)
    - Exploding & vanishing gradient problem can be mitigated by careful weight initialization (suitable layer weight initialization for layer activation)

Deep learning tends to work well on big data but large amount of training data make the training slow but optimization algorithms can speed up the training process
Solution: consider only subsets of dataset to speed up training -- the subsets is called mini-batches

- Batch gradient descent -- process entire train set when training
- Mini-batch gradient descent -- process subset of training data -- loss might fluctuate when training on different mini-batches because some mini-batch data is harder to predict.
- Training with -- mini-batch size = amount of training data -- is batch gradient descent
    - Batch gradient descent finally converges
    - Take too long to train per epoch due to large amount of training data
- If mini-batch size = 1 -- is stochastic gradient descent
    - Stochastic gradient descent will not ever converge but loss can be less oscillating by using smaller learning rate
    - Inefficiently process each training example -- not helping to speed up
- Moderate mini-batch size
    - help learn fastest among batch & stochastic gradient descent
    - Oscillate in smaller religion then stochastic gradient descent
    - May or may not converge to minimum loss

How to choose

- Train set size <= 2000 examples → use batch gradient descent
- Otherwise, use mini-batch gradient descent
    - Typically mini-batch sizes are power of 2 e.g. 64, 128, 256, 512 etc.
    - Use mini-batch size that fit in CPU/ GPU memory (RAM) -- look at how big is each training example

Epoch = single pass to training set

If we split training set into mini batches, we can run through training set many times with each round of running through train set becomes much faster

Optimization algorithm

Exponentially weighted averages

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

t = current (time)
t - 1 = previous (time)
beta is decimal
theta = variable to compute average of

exponentially weighted moving averages

- Faster than gradient descent
- Computationally efficient to calculate that is why used in ML

- v_t is exponentially decaying function
- v_t is like averaging over 1 / (1 - beta) time units
- It will take around 1 / (1 - beta) time units to get decay weight value around 1 / e
    - beta ** (1 / (1 - beta)) = 1 / e
- The v_t formula introduces bias because the computed average by the formula and the correct average is not nearly the same at small t -- need to make some changes to the formula i.e. do the bias correction if we not want bias estimate at small t

    - Now v_t = (original formula's v_t) / (1 - (beta ** t))

      The v_t value will be close to correct average

- If beta is large we rely more on previous average to compute current average & current theta has less contribution to the current average -- current average will not change much from previous average i.e. change slower from previous average -- graph is smoother

Gradient descent with momentum

- Use average (weighted average) of gradient to adjust weight instead of using the gradient
- Reduce oscillations while trying to minimize loss function → loss value converge to minimum loss function value faster

On iteration t:

Compute dW, db on the current mini-batch

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

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

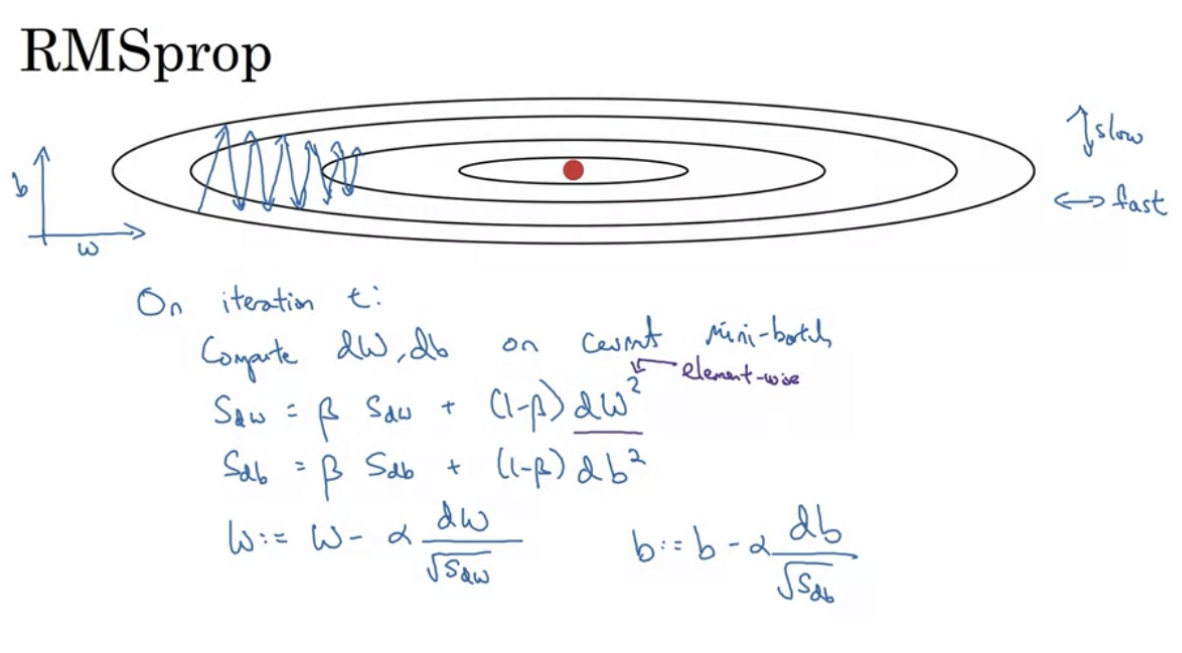
$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Initial v_{db} and v_{dw} are initialized to matrix of zeros

- If training for many iterations e.g. > 10 epoch we can automatically get unbiased estimate of gradient average without correcting the bias
- Common beta value is 0.9 -- which make the model use average of gradient of over last 10 epochs to update weights

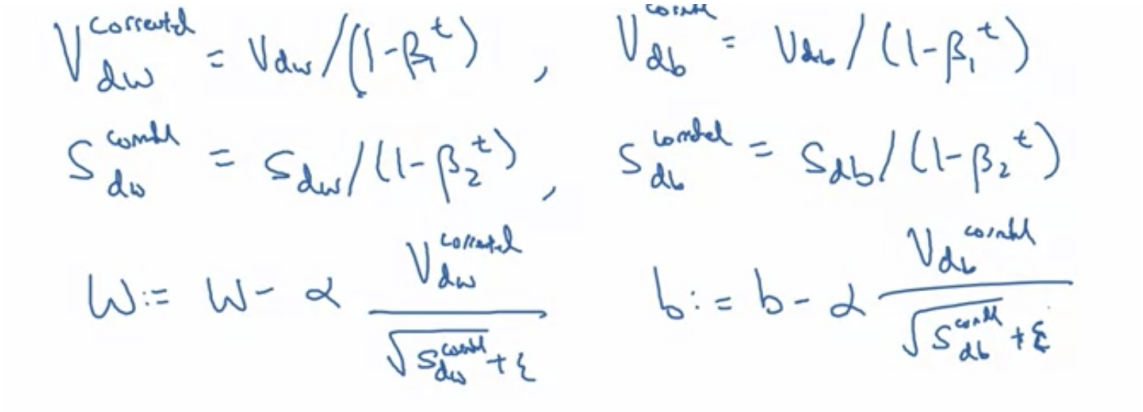
RMS prop

- Helps reduce oscillations and helps loss to converge to minimum faster
- Take derivative (compute gradient) first, then square
- S is moving average in the RMS prop figure below



Adam

- Helps reduce oscillations and helps loss to converge to minimum faster
- Momentum & RMS prop
    - V is average used in momentum gradient descent
    - S is average used in RMS prop
    - beta1 and beta2 can have same or different value
    - Epsilon is added to make the denominator non zero



Learning rate (LR) decay

- LR decay = process of slowly decreasing learning rate over time
- Help to learn faster comparing to fixing LR
    - LR_{new} = CONSTANT * LR_{old}

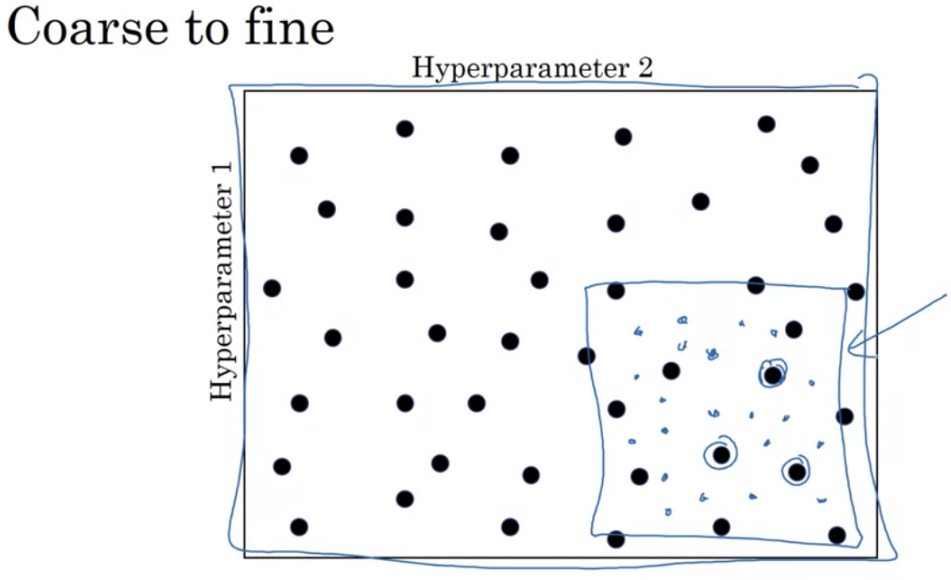
CONSTANT may vary from epoch to epoch e.g. by using other variable(s) (which change) to compute this constant

Tips (by Dr. Andrew) for tuning hyper parameters
Parameters grouped by priority for tuning

- Highest priority
  - Learning Rate
- Medium priority
  - Momentum
  - Mini-batch size
  - No. of hidden units
- Lower priority
  - Number of layers
  - Learning rate decay

Recommendations

- try random hyperparameter values i.e. do random search because you will have a chance to try more values than grid search (more exploration)
- Coarse to fine grain tuning (optional)
  - Come up with boundary values where optimal parameters may reside in then zoom in to the region



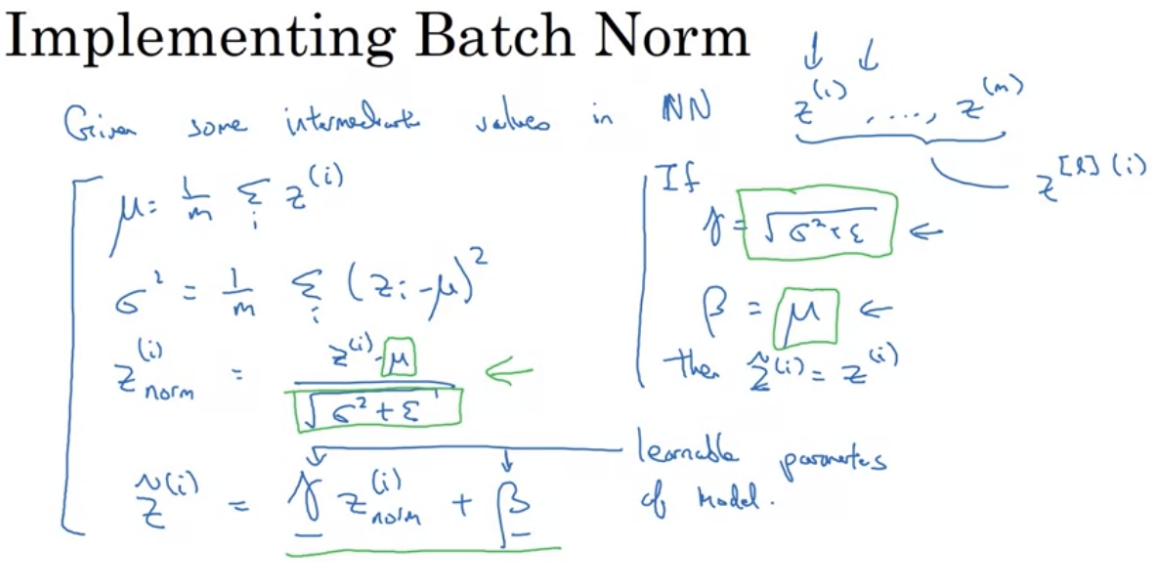
For parameter tuning, pick parameters uniformly and use appropriate scale e.g. linear scale, log scale for values much less than 1
For log scale determine lowest and highest possible value of parameter → take log base 10 of both → pick random value between lower log and higher log uniformly → 10 ** log_base_10(random_value) will be the parameter value to try

Hyper param tuning practice

- Pandas approach -- build small no. of models try tune the built models -- good if few computational resources are available or models take long to train (may be due to training data has high volume etc.)
- Caviar approach -- build many models tune and pick the best model -- good if a lot of computational resource are available

Batch normalization

- Normalizes input to neurons e.g. hidden layer neuron inputs' weighted sum (sum before activation i.e. z) or neuron's activation value (a) for that mini-batch
- Can speed up learning for some networks
- Has slight regularization effect
- Make downstream neurons rely not much on any specific upstream neurons



**** teacher normalized weighted sum (z)

z = value before activation (weighted sum)
a = activation function result of neuron
v = value of neuron to normalize which can be either z or a
m = no. of samples in that minibatch
μ = mean of neuron's v in mini-batch
σ ** 2 = variance of neuron's v in mini-batch
β, δ = learnable params of batch norm (ensures to make mean and variance of normalized features as you want i.e. make mean and variance near or equal to 0 and 1 respectively by default)
ε = small constant that prevent division by zero

Beta control mean & gamma control variance of normalized features
Beta & gamma are updated the same way as weight e.g. gradient descent or other algorithms

Batch normalization reduces the effects of previous layers on current layer i.e. each layer can learn independently

- In test time you might not have mini-batches; thus we use exponentially weighted average (roughly estimated average) of μ and σ ** 2 values got from training on mini-batches to scale neuron inputs at test time