Practical aspects of Deep Learning

Train / Dev / Test sets

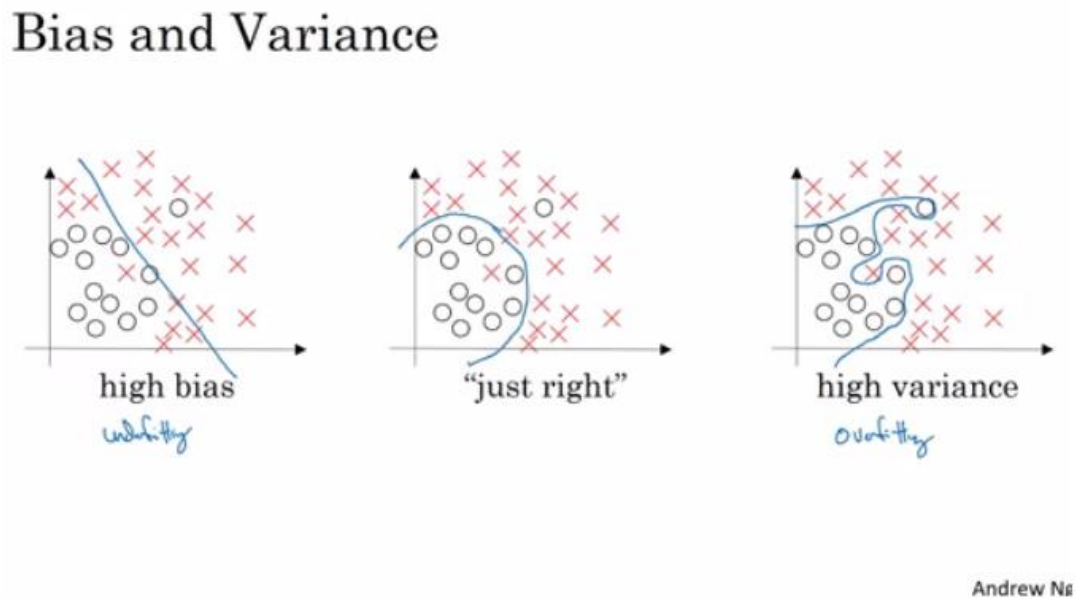
Its impossible to get all your hyperparameters right on a new application from the first time.

- go through the loop: `Idea ==> Code ==> Experiment`.
- Your data will be split into three parts:
  - Training set. (Has to be the largest set)
  - Development or "dev" set.
  - Testing set.
- ratio of splitting the models:
  - If size of the dataset is 100 to 1000000 ==> 60/20/20
  - If size of the dataset is 1000000 to INF ==> 98/1/1

- build a model using training set then try to optimize hyperparameters on dev set as much as possible. Then after your model is ready you try and evaluate the testing set.
- **Make sure the dev and test set are coming from the same distribution**.
- Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

Bias / Variance

- explanation of Bias / Variance:
  - high bias -model is under fitting (logistic regression of non linear data).
  - high variance - model is overfitting



- High variance (overfitting) for example:
  - Training error: 1%
  - Dev error: 11%
- high Bias (underfitting-big training error) for example:
  - Training error: 15%
  - Dev error: 14%
- high Bias (underfitting) && High variance (overfitting) for example:
  - Training error: 15%
  - Test error: 30%
- Best:
  - Training error: 0.5%
  - Test error: 1%

Basic Recipe for Machine Learning

high bias:

- Try to make your NN bigger (size of hidden units, number of layers)
- Try a different model that is suitable for your data.
- Try to run it longer.
- Different (advanced) optimization algorithms.

high variance:

    - More data.
    - Try **regularization**.
    - Try a different model(NN architecture) that is suitable for your data.

Regularization against height variance

- Regularization(Lagrange multiplier):
    - The L2(Norm2) regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(||w[i]||^2)`
    - The L1(Norm1) regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|)`
    - The L1 regularization version makes a lot of w values become zeros, which makes the model size smaller.
    - L2 regularization is being used much more often.
    - `lambda` here is the regularization parameter (hyperparameter)
- Regularization back prop:

dw=$\frac{dL}{dz}\frac{dz}{dw} + \frac{dL}{dw} = dw_{old} + \frac{\lambda}{m}\omega^{[l]}$

Why regularization reduces overfitting?

Here are some intuitions:

- Intuition 1:
    - If `lambda` is too large - a lot of w's will be close to zeros which will make the NN simpler (you can think of it as it would behave closer to logistic regression).
    - If `lambda` is good enough it will just reduce some weights that makes the neural network overfit.
- Intuition 2 (with *tanh* activation function):
    - If `lambda` is too large, w's will be small (close to zero) - will use the linear part of the *tanh* activation function, so we will go from non linear activation to *roughly* linear which would make the NN a *roughly* linear classifier.
    - If `lambda` good enough it will just make some of *tanh* activations *roughly* linear which will prevent overfitting.
- Intuition 3: Using regularization we create a new condition minimizing the size of all ω. The problem in overfitting is unbalanced weights which mean one ω is very strong compare to other. Smaller values of ω is a key to

more balanced weights. Regularization work like Lagrange multiplier the equality constraints is the size of ω.

Implementation tip: if you implement gradient descent, one of the steps to debug gradient descent is to plot the cost function J as a function of the number of iterations of gradient descent and you want to see that the cost function J decreases **monotonically** after every elevation of gradient descent with regularization. If you plot the old definition of J (no regularization) then you might not see it decrease monotonically.

Dropout Regularization

The dropout regularization eliminates some neurons/weights on each iteration based on a probability.

Code for Inverted dropout:

```
keep_prob = 0.8   # 0 <= keep_prob <= 1
l = 3  # this code is only for layer 3
# the generated number that are less than 0.8 will be dropped. 80% stay,
20% dropped

d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob
#create matrix 0 if <keep prob 1 otherwise.size same as a3.

a3 = np.multiply(a3,d3)   # keep only the values in d3

# increase a3 so the J will not change.
a3 = a3 / keep_prob
```

- It test time we don't use dropout. If you implement dropout at test time - it would add noise to predictions.

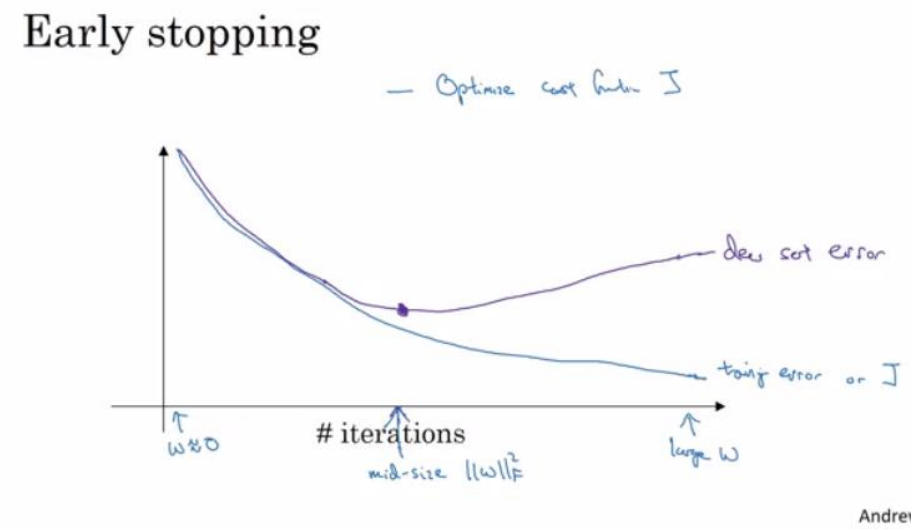
Understanding Dropout

- In the previous video, the intuition was that dropout randomly knocks out units in your network. So it's as if on every iteration you're working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.
- Another intuition: can't rely on any one feature, so have to spread out weights.
- It's possible to show that dropout has a similar effect to L2 regularization.

- Dropout can have different `keep_prob` per layer.
- If you're more worried about some layers overfitting than others, you can set a lower `keep_prob` for some layers than others. The downside is, this gives you even more hyperparameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and then just have one hyperparameter, which is a `keep_prob` for the layers for which you do apply dropouts.
- A lot of researchers are using dropout with Computer Vision (CV) because they have a very big input size and almost never have enough data, so overfitting is the usual problem. And dropout is a regularization technique to prevent overfitting.
- A downside of dropout is that the cost function J is not well defined and it will be hard to debug (plot J by iteration).
    - To solve that you'll need to turn off dropout, set all the `keep_prob`s to 1, and then run the code and check that it monotonically decreases J and then turn on the dropouts again.

Other regularization methods

- **Data augmentation**:
    - For example in a computer vision data:
        - You can flip all your pictures horizontally this will give you m more data instances.
        - You could also apply a random position and rotation to an image to get more data.
    - For example in OCR, you can impose random rotations and distortions to digits/letters.
    - New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
- **Early stopping**:
    - In this technique we plot the training set and the dev set cost together for each iteration. At some iteration the dev set cost will stop decreasing and will start increasing.
    - We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost).
    - We will take these parameters as the best parameters.



    - Andrew prefers to use L2 regularization instead of early stopping because this technique simultaneously tries to minimize the cost function and not to overfit which contradicts the orthogonalization approach (will be discussed further).
    - But its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like `lambda` in L2 regularization).
- **Model Ensembles**:
    - Algorithm:
        - Train multiple independent models.
        - At test time average their results.
    - It can get you extra 2% performance.
    - It reduces the generalization error.
    - You can use some snapshots of your NN at the training ensembles them and take the results.

Normalizing inputs

If you normalize your inputs this will speed up the training process a lot.

Normalization are going on these steps:

i. Get the mean of the training set: `mean = (1/m) * sum(x(i))`
ii. Subtract the mean from each input: `X = X - mean`
    - This makes your inputs centered around 0.
iii. Get the variance of the training set: `variance = (1/m) * sum(x(i)^2)`
iv. Normalize the variance. `X /= variance`

These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set).

Why normalize?

- If we don't normalize the inputs our cost function will be deep and its shape will be inconsistent (elongated) then optimizing it will take a long time.
- But if we normalize it the opposite will occur. The shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.
- In multi variance Normal Distribution the points of constant density tend to look like hyper ellipsoids. The derivatives tend to be lower in ellipsoids then in cycle for randomly chosen point on the space. The extreme example $\sigma_1^2 \gg \sigma_2^2$ in the covariance matrix creating ellipsoid where a>>b.

Vanishing / Exploding gradients

- The Vanishing / Exploding gradients occurs when your derivatives become very small or very big.
- To understand the problem, suppose that we have a deep neural network with number of layers L, and all the activation functions are **linear** and each `b = 0`

Examples:

```
(1)Y' = W[L]W[L-1].....W[2]W[1]X
```

we have 2 hidden units per layer and x1 = x2 = 1, we result in:

Exploding gradient:

```
if W[l] = [1.5   0]
          [0   1.5]

Y' = W[L] [1.5  0]^(L-1) X = 1.5^L       # which will be very large
          [0  1.5]
```

Vanishing gradient:

```
if W[l] = [0.5  0]
          [0  0.5]

Y' = W[L] [0.5  0]^(L-1) X = 0.5^L       # which will be very small
          [0  0.5]
```

Summary:

- The activations (and similarly derivatives) will be decreased/increased exponentially as a function of number of layers.
- If W > I (Identity matrix) the activation and gradients will explode.
- If W < I (Identity matrix) the activation and gradients will vanish.
- Recently Microsoft trained 152 layers (ResNet)! which is a really big number. With such a deep neural network, if your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small. And this makes training difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.

partial solution- careful choice of how you initialize the weights

Partial solution-Weight Initialization for Deep Networks

In a single neuron (Perceptron model): `Z = w1x1 + w2x2 + ... + wnxn` So if `n_x` is large we want `W`'s to be smaller to not explode the cost.

Options:

Paper name: [Xavier, glorot &yoshua Bengio's- Understanding the difficulty of training deep feedforward]

- Turn the variance equals 1/n_x to be the range of W's.

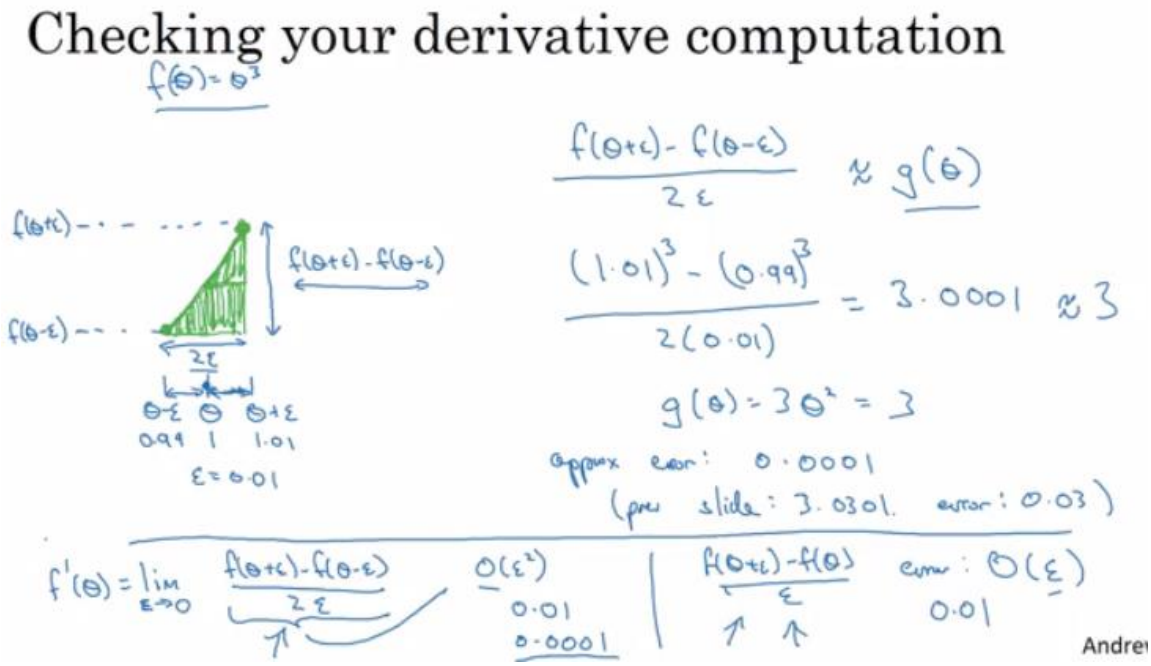
Tanh, sigmoid:

```
np.random.rand(shape) * np.sqrt(1/n[l-1])
```

Relu: sqrt to `2/n[l-1]` for is better:

- `np.random.rand(shape) * np.sqrt(2/n[l-1])`

Numerical approximation of gradients:



Gradient checking Algorithm:

- First take `W[1],b[1],...,W[L],b[L]` and reshape into one big vector (`theta`)
- The cost function will be `L(theta)`
- Then take `dW[1],db[1],...,dW[L],db[L]` into one big vector (`d_theta`)
- ```
  eps = 10^-7   # small number
  for i in len(theta):
    d_theta_approx[i] = (J(theta1,...,theta[i] + eps) - J(theta1,...,theta[i] - eps)) / 2*eps
  ```
- Finally we evaluate this formula `(||d_theta_approx - d_theta||) / (||d_theta_approx||+||d_theta||)` (`||` - Euclidean vector norm) and check (with eps = 10^-7):
  - if it is < 10^-7 - great, very likely the backpropagation implementation is correct
  ```

- if around 10^-5 - can be OK, but need to inspect if there are no particularly big values in `d_theta_approx - d_theta` vector
- if it is >= 10^-3 - bad, probably there is a bug in backpropagation implementation

Optimization algorithms

Mini-batch gradient descent

Mini-batch reasons:

- huge data won't fit in the memory at once we need other processing to make such a thing.
- Wants gradient descent move in smaller steps.

How?

split m to **mini batches** of size 1000.

mini batches ==> `t: X{t}, Y{t}`

- `X{1} = 0 ... 1000`
- `X{2} = 1001 ... 2000`
- `...`
- `X{bs} = ...`

Mini-batch size:

- (`mini batch size = m`) ==> Batch gradient descent
- (`mini batch size = 1`) ==> Stochastic gradient descent (SGD)
- (`mini batch size = between 1 and m`) ==> Mini-batch gradient descent

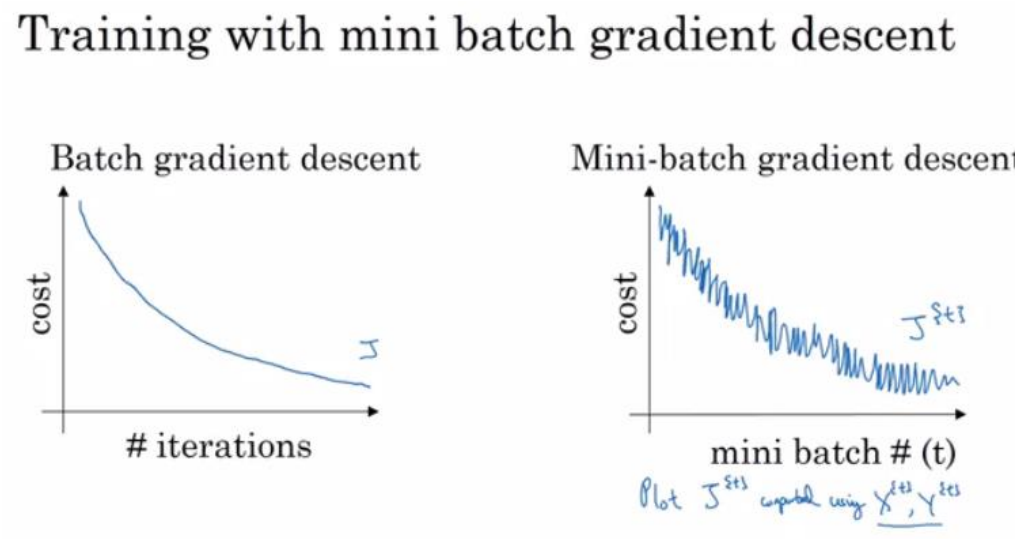
Mini-Batch algorithm pseudo code:

```
for t = 1:No_of_batches                                # this is called an epoch
      AL, caches = forward_prop(X{t}, Y{t})
      cost = compute_cost(AL, Y{t})
      grads = backward_prop(AL, caches)
      update_parameters(grads)
```

J mini batch vs Batch gradient descent

- In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm. It could contain some ups and downs but generally it

has to go down (unlike the batch gradient descent where cost function descreases on each
iteration).



Guidelines for choosing mini-batch size:

i. If small training set (< 2000 examples) - use batch gradient descent.
ii. It has to be a power of 2 (because of the way computer memory is layed out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2): `64, 128, 256, 512, 1024, ...`
iii. Make sure that mini-batch fits in CPU/GPU memory.

- Mini-batch size is a `hyperparameter`.

Exponentially weighted averages(used in stocks)

If we have data like the temperature of day through the year it could be like this:

```
T(1) = 40
T(2) = 49
T(3) = 45
...
T(180) = 60
...
```

If we plot this data we will find it some noisy. We can use exponentially weighted averages to smooth the data.

Exponentially weighted averages algorithm:

```
V(t) = β * v(t-1) + (1- β) * θ(t)
```

(1) $V(t) = \sum_{n=0}^{\infty} \beta^{N} (1-\beta) * \theta(t-N)$

- `1-B-The influence of today temperature on V(t) result.`

- represent averages over ~ `(1 / (1 - beta))` entries:
  - `beta = 0.9` will average last 10 entries
  - `beta = 0.98` will average last 50 entries
  - `beta = 0.5` will average last 2 entries
- Best beta average for our case is between 0.9 and 0.98

Example(Blue-no weighted averages, Red-with weighted averages):

exponentially weighted averages Algorithm

```
v = 0
Repeat
{
  Get θ (t)
  v = β * v + (1-β) * θ (t)
}
```

Bias correction in exponentially weighted averages

> Problem: Because `v(0) = 0`, the bias of the weighted averages is shifted and the accuracy suffers at the start. $\theta(1)$ has an influence of `(1- β)`.
>
> Solution:
>
> $$V(t) = \frac{V(t)}{1-\beta^t}$$

This way $\theta(1)$ has a influence of $\frac{\theta(1)(1-\beta)}{1-\beta} = \theta(1)$. The influence is strong only at the beginning t<10.

Gradient descent with momentum

The momentum algorithm almost always works faster than standard gradient descent. The idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values. Momentum helps the cost function to go to the minimum point in a faster and consistent way.

Why it works? if we can imagine a ping pong ball inside a bowl, gradient descent is looking for the maximum acceleration(slope) at each step ignoring previews momentum. using the past information is kind of adding the velocity to determine the next step instead of just use the acceleration in each point. Using the velocity, we can get over small obstacle instead of going around it clearly achieving faster converges.

Pseudo code:

```
vdW = 0, vdb = 0
on iteration t:
     # can be mini-batch or batch gradient descent
     compute dw, db on current mini-batch

     vdW = beta * vdW + (1 - beta) * dW
     vdb = beta * vdb + (1 - beta) * db
     W = W - learning_rate * vdW
     b = b - learning_rate * vdb
```

RMSprop(Root mean square prop)

$$s_{d\omega} = \beta s_{d\omega} + (1-\beta) d\omega^2$$

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

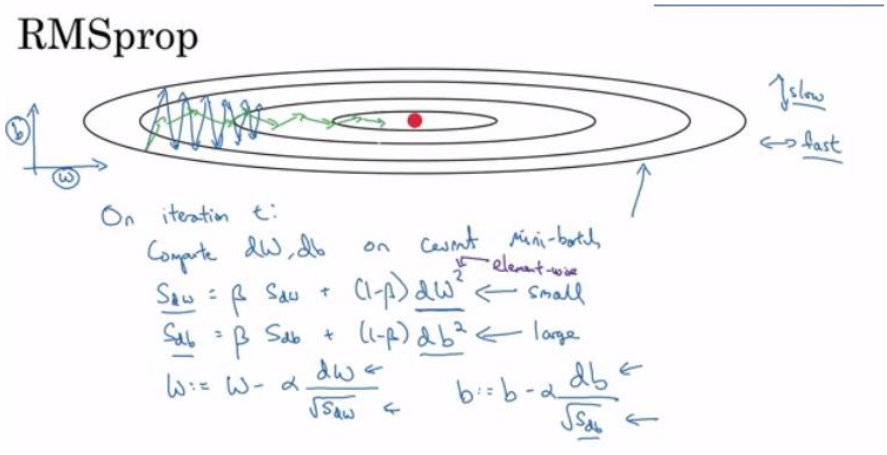
$$\omega = \omega - \frac{\alpha d\omega}{s_{d\omega}}, b = b - \frac{\alpha db}{s_{db}}$$

Why?RMS prop Divide the gradient by a running average of its recent magnitude. Which is like using the sign of the gradient. The magnitude of the gradient can be very different for different weights and can change during learning. This makes it hard to choose a single global learning rate. using only the sign solve this problem making the steps at each direction similar.

Pseudo code:

```
sdW = 0, sdb = 0
on iteration t:
     # can be mini-batch or batch gradient descent
     compute dw, db on current mini-batch

     sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise
     sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise
     W = W - learning_rate * dW / sqrt(sdW)
     b = B - learning_rate * db / sqrt(sdb)
```



[Geoffrey Hinton RMSprop]

Adam optimization algorithm(RMSprop + momentum)

Pseudo code:

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
     # can be mini-batch or batch gradient descent
     compute dw, db on current mini-batch

     vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
     vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

     sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
     sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

     vdW = vdW / (1 - beta1^t)      # fixing bias
     vdb = vdb / (1 - beta1^t)      # fixing bias

     sdW = sdW / (1 - beta2^t)      # fixing bias
     sdb = sdb / (1 - beta2^t)      # fixing bias

     W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
     b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

- Hyperparameters recommended for Adam:
  - Learning rate: needed to be tuned.
  - $\beta 1$: parameter of the momentum - `0.9` is recommended by default.
  - $\beta 2$: parameter of the RMSprop - `0.999` is recommended by default.
  - default $\epsilon = 10^{-8}$ is recommended by.

Learning rate decay

Slowly reduce learning rate when we are closer to the real solution.

Methods:

(1) `learning rate=`$\frac{1}{(1+\gamma * epoch_{num})} * \alpha_0, \quad \gamma = decay\ rate$

(2) `learning rate=`$0.95^{epoch_{num}} * \alpha_0$

(3) `learning rate=`$\frac{K}{\sqrt{epoch_{num}}} * \alpha_0$

Andrew Ng, learning rate decay has less priority.

The problem of local optima

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.

Hyperparameter tuning, Batch Normalization and Programming Frameworks

Tuning process

- We need to tune our hyperparameters to get the best out of them.
- Hyperparameters importance are (as for Andrew Ng):
  i. Learning rate-α.
  ii. Momentum beta .
  iii. Mini-batch size.
  iv. No. of hidden units.
  v. No. of layers.
  vi. Learning rate decay.
  vii. Regularization lambda.
  viii. Activation functions.
  ix. Adam β_1 & β_2
- Its hard to decide which hyperparameter is the most important in a problem. It depends a lot on your problem.

  Generic method:

  - Random values of Hyperparameters.
  - When you find some hyperparameters values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space.

Using an appropriate scale to pick hyperparameters

Scaling problem:

Example $\beta(momentum)$: $\frac{1}{1-\beta}$=number of days we average.

change from 0.9-0.9005$\left(\frac{1}{1-0.9005}-\frac{1}{1-0.9}=10\right)$ change 0.999-0.9995(1000). We want to sample more in between 0.999-0.9995.

Solution:

```
Using logarithmic scale.
```

Pseudo code:

```
a_log = log(a) # e.g. a = 0.0001 then a_log = -4

b_log = log(b) # e.g. b = 1 then b_log = 0

r = (a_log - b_log) * np.random.rand() + b_log

result = 10^r

# In the example the range would be from [-4, 0] because rand range
[0,1]
```

Hyperparameters tuning in practice: Pandas vs. Caviar

- "Pandas"- don't have much computational resources:
  - Day 0 you might initialize your parameter as random and then start training.
  - Then you watch your learning curve gradually decrease over the day.
  - And each day you nudge your parameters a little during training.
- “Caviar”-have enough computational resources,
  - run some models in parallel and at the end of the day(s) you check the results.

Batch Normalization activations in a network

What?

- Forcing the inputs to a distribution with zero mean and variance of 1.

Why?

- normalized input by subtracting the mean and dividing by variance shapes the cost function(circles instead of ellipses) and helps reaching the minimum point faster.

Algorithm for normalizing:

- Given `Z[1] = [z(1), ..., z(m)]`, i = 1 to m (for each input)
- Compute `mean = 1/m * sum(z[i])`
- Compute `variance = 1/m * sum((z[i] - mean)^2)`
- Then `Z_norm[i] = (z(i) - mean) / np.sqrt(variance + epsilon)` (add `epsilon` for numerical stability if variance = 0)

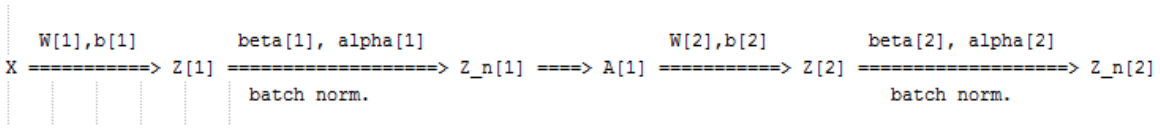
Normalizing as hyperparameters:

- `Z_tilde[i] = gamma * Z_norm[i] + beta`
- gamma and beta are learnable parameters of the model.
- Making the NN learn the distribution of the outputs.

Note: There are some debates in the deep learning literature about whether you should normalize values before the activation function `Z[1]` or after applying the activation function `A[1]`. In practice, normalizing `Z[1]` is done much more often than normalizing `A[1]`.

Fitting Batch Normalization into a neural network

- Using batch norm in 3 hidden layers
  NN:



- Our NN parameters will be:
  - `W[1]`, `b[1]`, ..., `W[L]`, `b[L]`, `beta[1]`, `gamma[1]`, ..., `beta[L]`, `gamma[L]`

Note: If we are using batch normalization parameters `b[1]`, ..., `b[L]` doesn't count because they will be eliminated after mean subtraction step, so:

Batch normalization at test time

- In testing we might need to process examples one at a time. The mean and the variance of one example won't make sense.
- We have to compute an estimated value of mean and variance to use it in testing time.
- We can use the weighted average across the mini-batches.

Softmax Regression

used if we don't have binary classification problem (sigmoid function in Y).

Example: classifying by classes `dog`, `cat`, `baby chick` and `none`

- None `class = 0->vector= [1 0 0 0]`
- Dog `class = 1->vector= [0 1 0 0]`
- Cat `class = 2->vector= [0 0 1 0]`
- Baby chick `class =3->vector=[0 0 0 1]`

Notations:

- `C = no. of classes`
- Range of classes is `(0, ..., C-1)`

Output Y:

- Each of C values in the output layer will contain a probability of the example to belong to each of the classes.
- In the last layer we will have to activate the Softmax activation function instead of the sigmoid activation.

Softmax activation Pseudo code:

```
t = e^(Z[L])                        # shape(C, m)
A[L] = e^(Z[L]) / sum(t)            # shape(C, m), sum(t) - sum of t's for
each example (shape (1, m))
```

hard max

- gets 1 for the maximum value and zeros for the others.

Code:

```
a.max(axis=1,keepdims=1) ==a
```

Softmax Loss:

The loss function used with softmax:

```
L(y, ŷ) = - sum(y[j] * log(ŷ[j])) # j = 0 to C-1
```

The cost function used with softmax:

```
J(w[1], b[1], ...) = - 1 / m * (sum(L(y[i], ŷ [i]))) # i = 0 to m
```

Back propagation with softmax:

The derivative of softmax is: $\hat{y}$* (1 - $\hat{y}$)

```
dZ[L] = ŷ- Y
```

Deep learning frameworks

- It's not practical to implement everything from scratch. Our numpy implementations were to know how NN works.
- Here are some of the leading deep learning frameworks:
  - Caffe/ Caffe2
  - CNTK
  - DL4j
  - Keras
  - Lasagne
  - mxnet
  - PaddlePaddle
  - TensorFlow
  - Theano
  - Torch/Pytorch
- Comparison between them can be found here.