

Ahmad Hussameldin Hamed Hassan

Shared Git-hub link: <https://github.com/ahmadhassan1993/sharing-github>

Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Week 2 Summary

We need optimization algorithms to fast our NN in large data set.

- 1- One possible algorithm is Mini-Batch gradient descent, which divide the whole m examples of data set to mini batches:

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(1000)} & x^{(1001)} & \dots & x^{(2000)} & \dots & x^{(m)} \end{bmatrix}$$

(n_x, m) $X^{\{1\}}$ $(n_x, 1000)$ $X^{\{2\}}$ $(n_x, 1000)$ $X^{\{5,000\}}$ $(n_x, 1000)$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & y^{(1001)} & \dots & y^{(2000)} & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$ $Y^{\{1\}}$ $(1, 1000)$ $Y^{\{2\}}$ $(1, 1000)$ $Y^{\{5,000\}}$ $(1, 1000)$

What if $m = 5,000,000$?
5,000 mini-batches of 1,000 each
Mini-batch t : $X^{\{t\}}, Y^{\{t\}}$

Andrew Ng

We use $\{t\}$ to index the mini batch number. For example, if $m=5000$ then t will range from 1 to 5000.

Mini-batch gradient descent

repeat $\{$ for $t = 1, \dots, 5000 \}$

Forward prop on $X^{(t)}$.

$$Z^{(t)} = W^{(t)} X^{(t)} + b^{(t)}$$

$$A^{(t)} = g(Z^{(t)})$$

$$\vdots$$

$$A^{(t)} = g(Z^{(t)})$$

Vertical propagation (1000 examples)

Compute cost $J^{(t)} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{i=1}^{1000} \|W^{(t)}\|_F^2$

Backprop to compute gradients w.r.t $J^{(t)}$ (using $X^{(t)}, Y^{(t)}$)

$$W^{(t+1)} = W^{(t)} - \alpha \frac{\partial J}{\partial W}, \quad b^{(t+1)} = b^{(t)} - \alpha \frac{\partial J}{\partial b}$$

“1 epoch”
pass through training set.

1 step of gradient descent using $X^{(t+1)}, Y^{(t+1)}$ (as if $t=1000$)

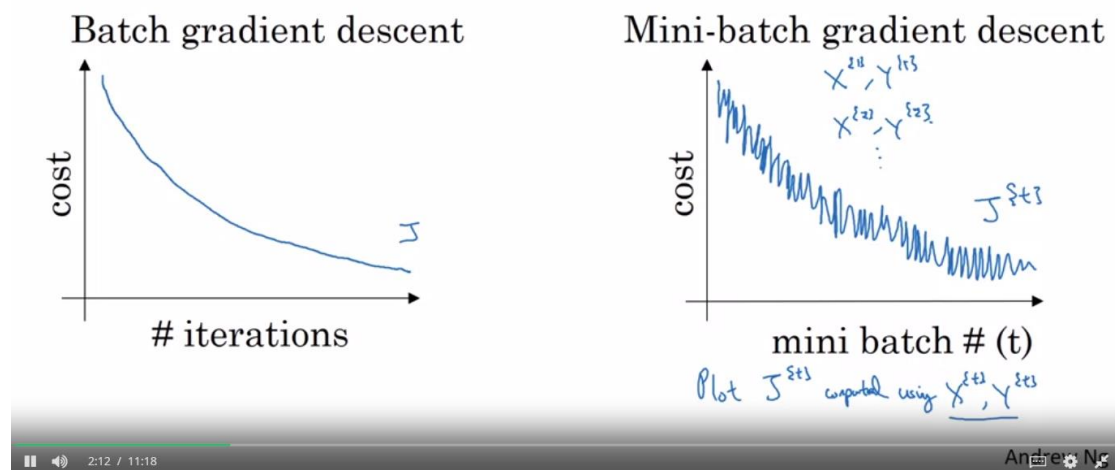
X, Y

Andrew Ng

1 epoch is the single pass through training set, i.e, the output of both forward and backward propagation for the whole batch.

The cost function plot for $\{t\}$ will be oscillating:

Training with mini batch gradient descent



Because different epochs will have different strength on the cost function.

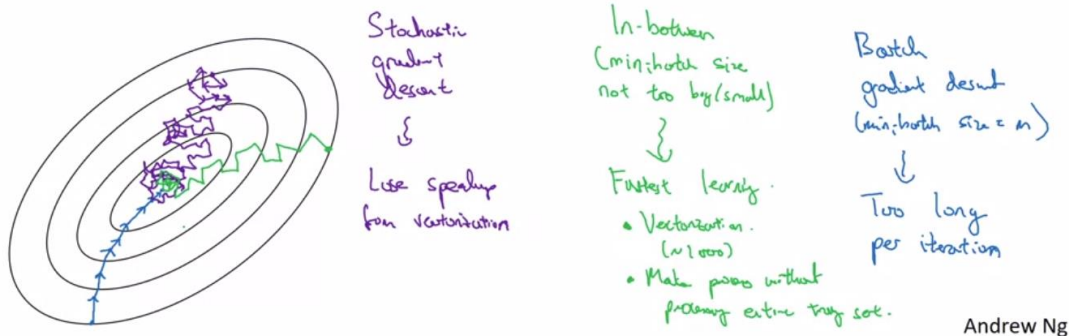
Choosing the size of the mini batch:

Choosing your mini-batch size

→ If mini-batch size = m : Batch gradient descent. $(X^{(1)}, Y^{(1)}) = (X, Y)$

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is its own mini-batch.
 $(X^{(1)}, Y^{(1)}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$ mini-batch.

In practice: Search in-between 1 and m



If mini batch size = m , so the full batch is processed at a time and longer time to solve. If size = 1, called stochastic, we will lose both speed up from vectorization and minimum is not reached. Optimum is to take size not too big and not too small. We will see progress without even completing the full data set.

The mini batch size is a Hybrid Parameter that we should choose well. If the training set is small (< 2000), then we should use the batch gradient descent. Typical size takes the power of 2:

Choosing your mini-batch size

If small training set : Use batch gradient descent.
 $(m \leq 2000)$

Typical mini-batch sizes:

→ 64, 128, 256, 512, 1024
 $2^6, 2^7, 2^8, 2^9, 2^{10}$

Make sure mini-batch fit in CPU/GPU memory.
 $X^{(k)}, Y^{(k)}$

Andrew Ng

We also must be sure that the size fits the memory of our computer.

2- Exponentially Weighted Average (moving average):

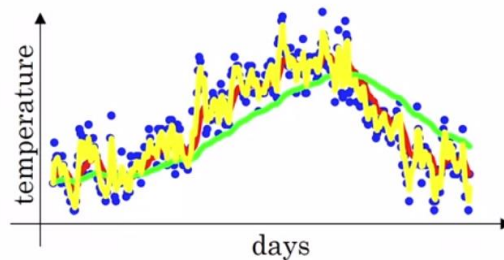
Exponentially weighted ^{moving} averages

$$V_t = \beta V_{t-1} + (1-\beta) \theta_t \leftarrow$$

$\beta = 0.9$: ≈ 10 days' temperature
 $\beta = 0.98$: ≈ 50 days
 $\beta = 0.5$: ≈ 2 days

V_t is approximately
 average over
 $\rightarrow \approx \frac{1}{1-\beta}$ days' temperature.

$$\frac{1}{1-0.98} = 50$$



Andrew Ng

In this example we see the trend (average) of varying temperature over the whole year. A hybrid parameter β is used to decide over which number of days we take the average ($1/(1-\beta)$). We should choose value that give us a suitable number of days, not too big not too small.

Exponentially weighted averages

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

$$\begin{aligned}
 v_{100} &= 0.9v_{99} + 0.1\theta_{100} \\
 v_{99} &= 0.9v_{98} + 0.1\theta_{99} \\
 v_{98} &= 0.9v_{97} + 0.1\theta_{98} \\
 &\dots
 \end{aligned}$$

$$\begin{aligned}
 \rightarrow v_{100} &= 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98}) \\
 &= 0.1\theta_{100} + 0.1 \times 0.9 \theta_{99} + 0.1(0.9)^2 \theta_{98} + 0.1(0.9)^3 \theta_{97} + 0.1(0.9)^4 \theta_{96} + \dots \\
 0.9^{\textcircled{10}} &\approx 0.35 \approx \frac{1}{e}
 \end{aligned}$$

$(1-\epsilon)^{1/\epsilon} = \frac{1}{e}$
 $\frac{1-\epsilon}{0.9} \leftarrow \epsilon = 0.02 \rightarrow 0.98 \approx \frac{1}{e}$

Andrew Ng

It is called exponential, because over n days the temperature degree increases exponentially from day 0 to day n . this is according to $\beta^{(1/\beta)}$.

Implementation by single line of code:

Implementing exponentially weighted averages

$$v_0 = 0$$

$$v_1 = \beta v_0 + (1 - \beta) \theta_1$$

$$v_2 = \beta v_1 + (1 - \beta) \theta_2$$

$$v_3 = \beta v_2 + (1 - \beta) \theta_3$$

...

$$V_\theta := 0$$

$$V_\theta := \beta V + (1 - \beta) \theta_1$$

$$V_\theta := \beta V + (1 - \beta) \theta_2$$

⋮

$$\rightarrow V_0 = 0$$

Repeat {

Get next θ_t

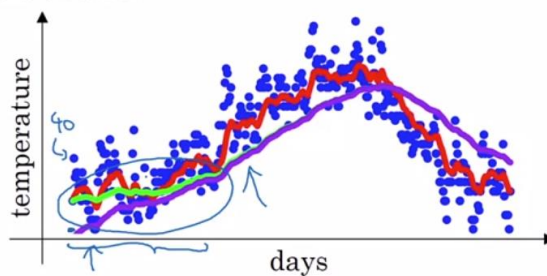
$$V_\theta := \beta V_\theta + (1 - \beta) \theta_t \leftarrow$$

}

Andrew Ng

We use Bias to correct the first values:

Bias correction



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

$$v_0 = 0$$

$$v_1 = 0.98 v_0 + 0.02 \theta_1$$

$$v_2 = 0.98 v_1 + 0.02 \theta_2$$

$$= 0.98 \times 0.02 \times \theta_1 + 0.02 \theta_2$$

$$= 0.0196 \theta_1 + 0.02 \theta_2$$

$$\frac{v_t}{1 - \beta^t}$$

$$t=2: 1 - \beta^t = 1 - (0.98)^2 = 0.0396$$

$$\frac{v_2}{0.0396} = \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396}$$

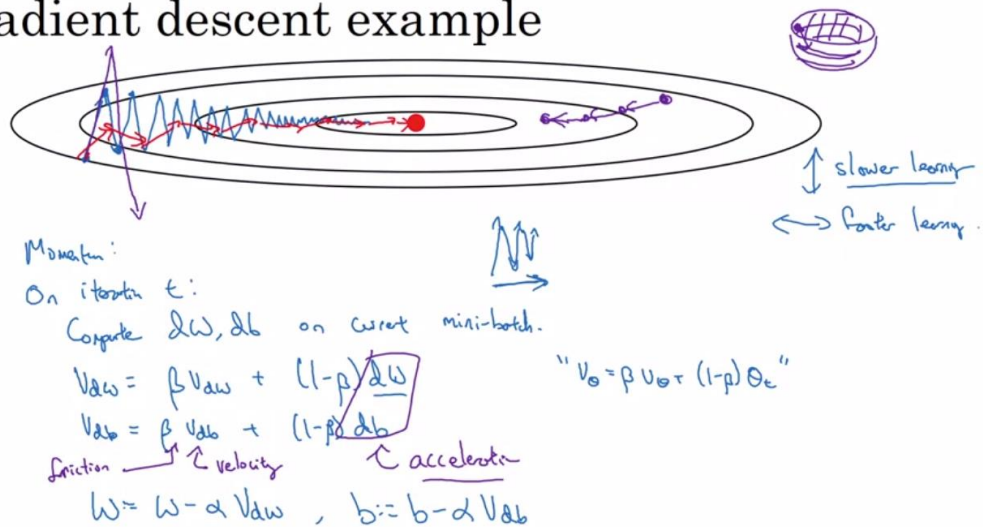
Andrew Ng

Bias = $1 - \beta^t$, where t is the number of the day that we want to calculate temperature in it. We will divide the calculated average by this bias for the whole days.

3- Gradient Descent with Momentum:

We need to go faster horizontally not vertically to the solution:

Gradient descent example



Andrew Ng

The most common value for β is 0.9. in that case, we neglect the following in equations as well as the Bias:

Implementation details

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

On iteration t :

Compute dW, db on the current mini-batch

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

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

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

$$\frac{v_{dw}}{1-\beta^t}$$

Hyperparameters: α, β

$$\beta = 0.9$$

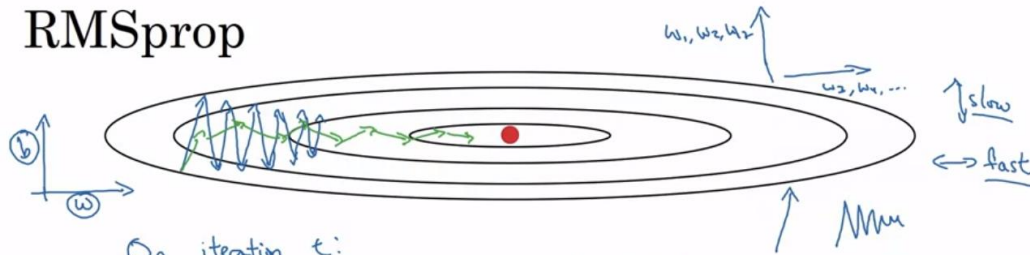
average over last ≈ 10 gradients

Andrew Ng

4- RMSprop in Gradient Descent:

We use the following equations:

RMSprop



On iteration t :

Compute dw, db on current mini-batch

$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) \underbrace{dw^2}_{\text{element-wise}} \leftarrow \text{small}$$

$$\rightarrow S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \leftarrow \text{large}$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw} + \epsilon}} \quad b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$$

$$\epsilon = 10^{-8}$$

Andrew Ng

Where β_2 is different than β that used in momentum. We add ϵ (very small) to ensure that the root is zero it will not affect the result.

5- Adam (Adaptive Moment estimation) Optimization Algorithm:

It is a combination of two algorithms: momentum and RMSprop:

Adam optimization algorithm

$$V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0$$

On iteration t :

Compute dw, db using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \leftarrow \text{"momentum"} \beta_1$$

$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \leftarrow \text{"RMSprop"} \beta_2$$

$$V_{dw}^{\text{corrected}} = V_{dw} / (1 - \beta_1^t), \quad V_{db}^{\text{corrected}} = V_{db} / (1 - \beta_1^t)$$

$$S_{dw}^{\text{corrected}} = S_{dw} / (1 - \beta_2^t), \quad S_{db}^{\text{corrected}} = S_{db} / (1 - \beta_2^t)$$

$$w := w - \alpha \frac{V_{dw}^{\text{corrected}}}{\sqrt{S_{dw}^{\text{corrected}} + \epsilon}} \quad b := b - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}} + \epsilon}}$$

Andrew Ng

Hyperparameters choice:

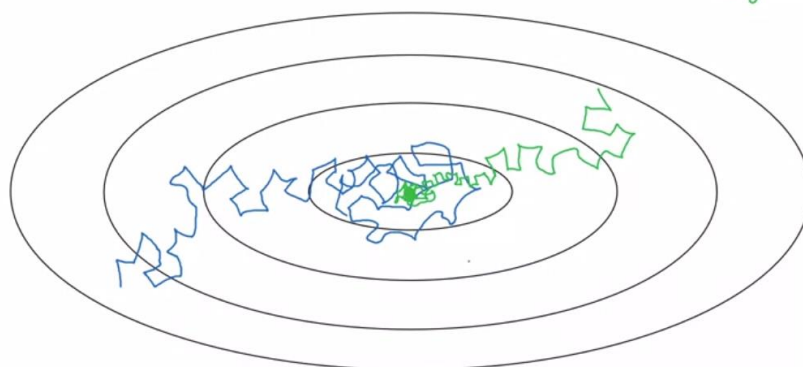
→ α : needs to be tune
→ β_1 : 0.9 → (\underline{dw})
→ β_2 : 0.999 → ($\underline{dw^2}$)
→ ϵ : 10^{-8}

Adam: Adaptive moment estimation

Andrew Ng

- Learning rate decay:

Learning rate decay



Andrew Ng

We should make the learning rate begin large then decay when approaching the solution, but slowly.

Method 1:

Learning rate decay

1 epoch = 1 pass through data.

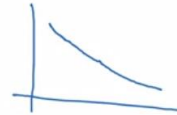
$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
\vdots	\vdots



$$\alpha_0 = 0.2$$

$$\text{decay-rate} = 1$$



Andrew Ng

Other Methods:

Other learning rate decay methods

Formula

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0 \quad \text{— exponentially decay.}$$

$$\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0$$

discrete staircase

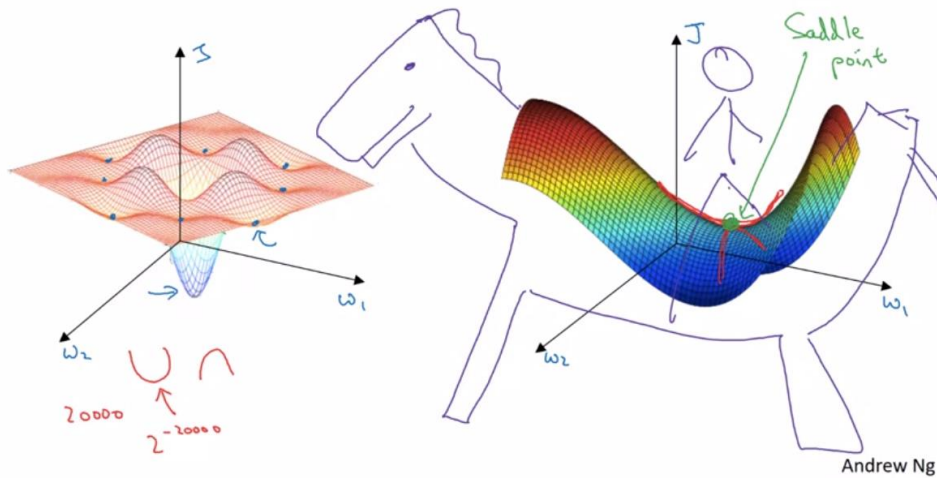
Manual decay.

Andrew Ng

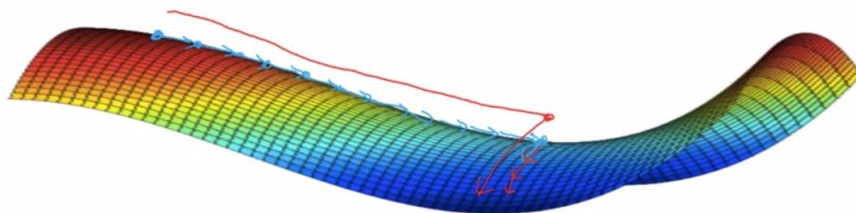
- The problem of local minimum:

We are unlikely to have a problem with local minimum in large data set. However, we have a problem in the slow of the gradient. So, the aforementioned optimization algorithms will make gradient descent faster.

Local optima in neural networks



Problem of plateaus



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

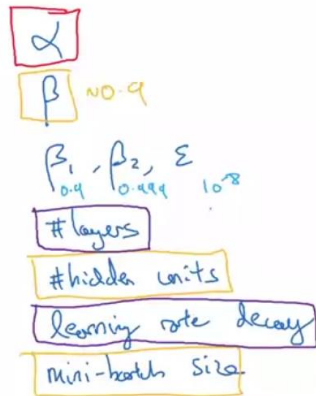
Andrew Ng

Week 3 summary

1) Hybrid parameters tuning:

The updated hybrid parameters so far are:

Hyperparameters

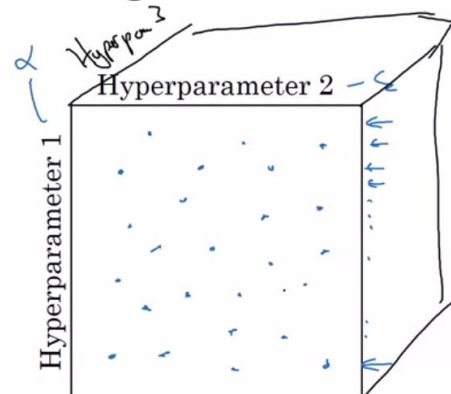
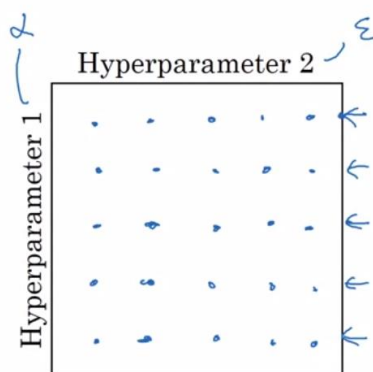


Andrew Ng

Red colored is the most important and must be tuned well. Yellow ones are less importance. Finally, burble colored are the least parameters.

Try random values:

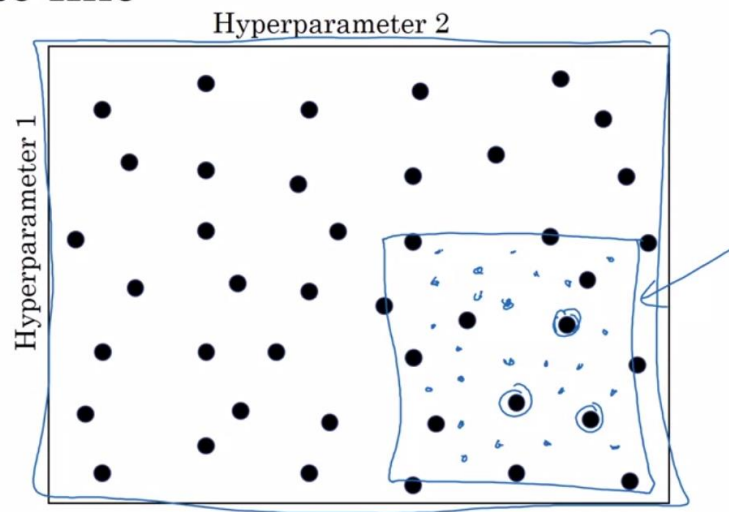
Try random values: Don't use a grid



Andrew Ng

Try to fine these random results by focusing on a small box containing the most expected suitable values and choose of them:

Coarse to fine



Andrew Ng

Number of layers and number of hidden units are mostly picked at uniform random:

Picking hyperparameters at random

$$\rightarrow n^{\text{test}} = 50, \dots, 100$$



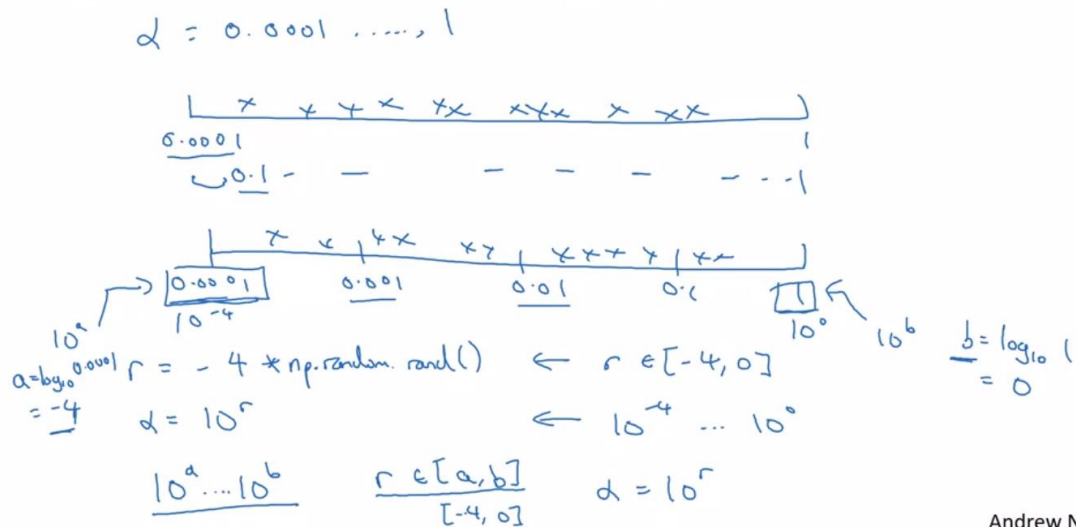
$$\rightarrow \# \text{layers } L: 2 - 4$$

$$2, 3, 4$$

Andrew Ng

Try to use the appropriate scale so that to consider the full range. For example, log scale for learning rate α :

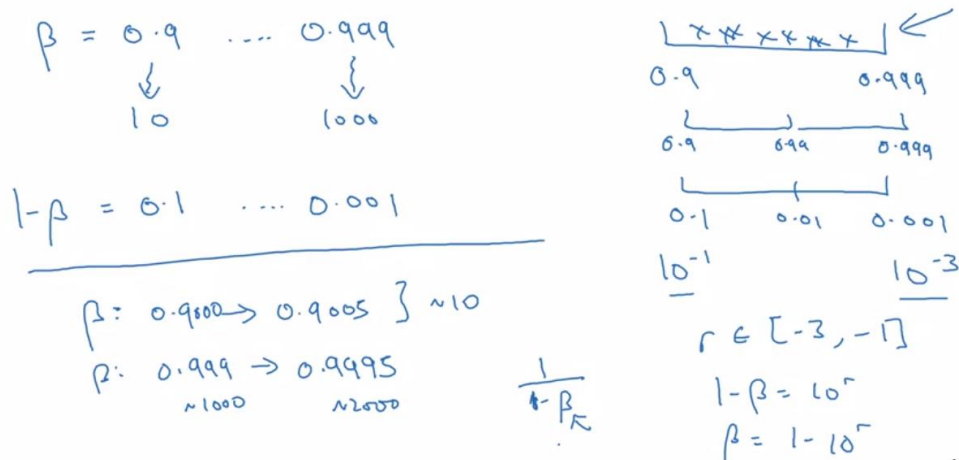
Appropriate scale for hyperparameters



Andrew Ng

For exponentially weighted average, we also use log scale as in α . As β near 1 and $1-\beta$ is near 0:

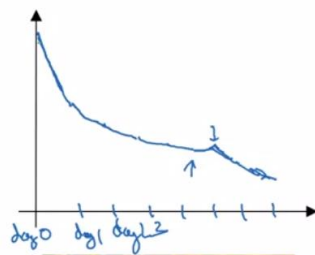
Hyperparameters for exponentially weighted averages



Andrew Ng

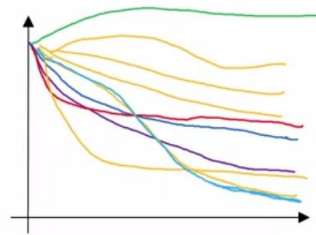
Try to use different models if you have enough resources, like computers:

Babysitting one model



Panda ←

Training many models in parallel



Caviar ←

Andrew Ng

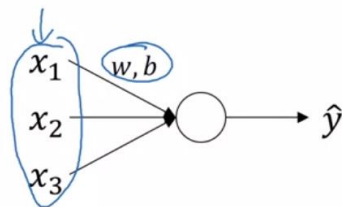
2) Batch Normalization (BN):

- Normalize activation functions:

This is to faster the W and b computations.

It is better to normalize Z before going to the activation function.

Normalizing inputs to speed up learning

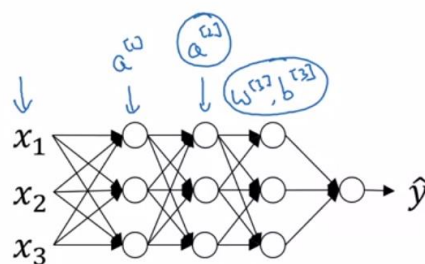


$$\mu = \frac{1}{m} \sum_i x^{(i)}$$

$$X = X - \mu$$

$$\sigma^2 = \frac{1}{m} \sum_i x^{(i)2} \quad \leftarrow \text{element-wise}$$

$$X = X / \sigma^2$$

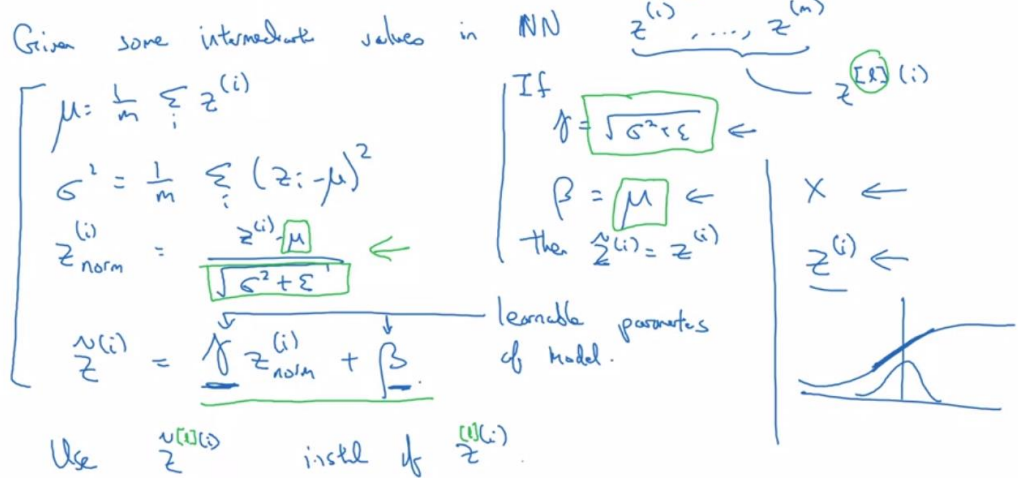


Can we normalize $\frac{a^{[2]}}{w^{[2]}, b^{[2]}}$ so as to train faster

Normalize $\frac{z^{[2]}}{\uparrow}$

Andrew Ng

Implementing Batch Norm



Andrew Ng

So, the mean and variance could be any values rather than 0 and 1, respectively.

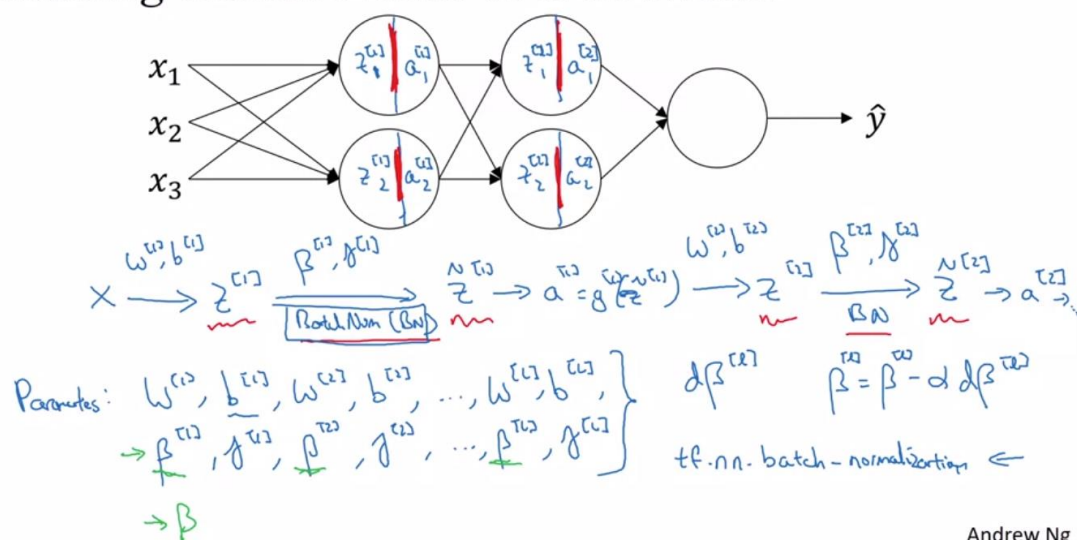
Batch Norm at test time

$$\begin{aligned} \rightarrow \mu &= \frac{1}{m} \sum_i z^{(i)} \\ \rightarrow \sigma^2 &= \frac{1}{m} \sum_i (z^{(i)} - \mu)^2 \\ \rightarrow z_{\text{norm}}^{(i)} &= \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \rightarrow \tilde{z}^{(i)} &= \gamma z_{\text{norm}}^{(i)} + \beta \end{aligned}$$

Andrew Ng

Updated parameters: W , b , β and γ where last two are for batch normalization:

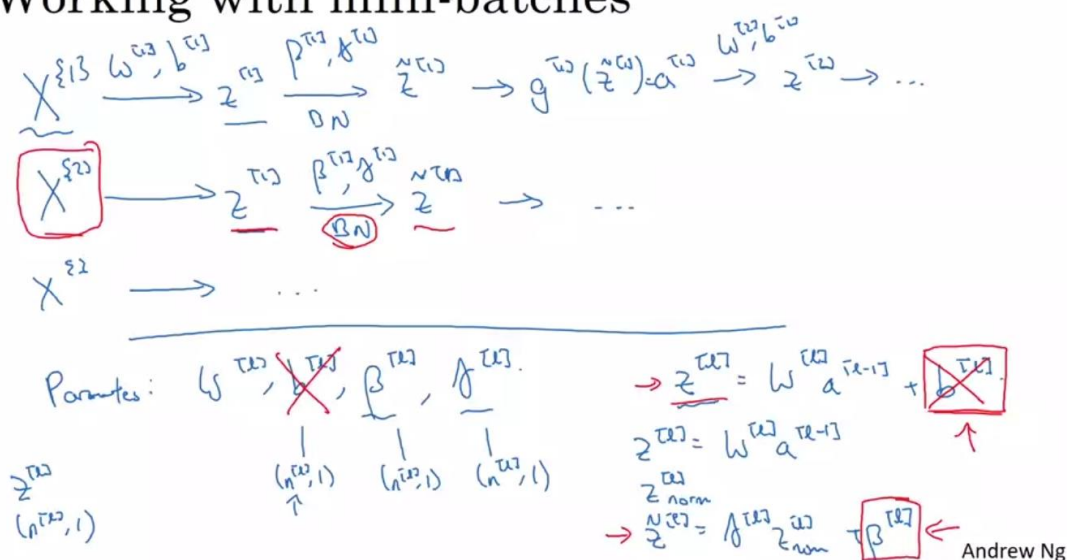
Adding Batch Norm to a network



Andrew Ng

We don't need to add parameter b , as it will be subtracted in the mean subtraction step in normalization. We instead use β and β & γ should be the same dimension as eliminated b to reserve Z dimension:

Working with mini-batches



Implementation of BN on Gradient Descent summary:

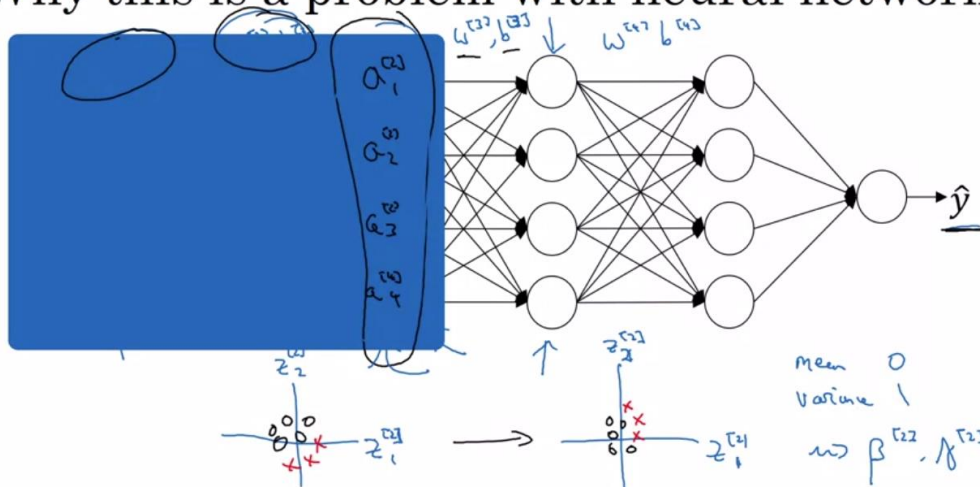
Implementing gradient descent

for $t = 1 \dots \text{num Mini Batches}$
 Compute forward pass on X^{test} .
 In each hidden layer, use BN to replace z with \hat{z} .
 Use backprop to compute $\frac{dw}{dz}$, $\frac{d\beta}{dz}$, $\frac{d\gamma}{dz}$.
 Update parameters $\left. \begin{aligned} W^{(i)} &:= W^{(i)} - \alpha \frac{dw^{(i)}}{dz} \\ \beta^{(i)} &:= \beta^{(i)} - \alpha \frac{d\beta^{(i)}}{dz} \\ \gamma^{(i)} &:= \gamma^{(i)} - \alpha \frac{d\gamma^{(i)}}{dz} \end{aligned} \right\} \leftarrow$
 Works w/ momentum, RMSprop, Adam.

Andrew Ng

The BN reduces the effect of changing the hidden layer (i) values on layer (i+1) values by making the mean and variance almost the same for all hidden layers:

Why this is a problem with neural networks?



Andrew Ng

BN adds noise as the dropout regularization does. To reduce this effect, use large sized mini batches:

Batch Norm as regularization

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch. X^{i+1}
 $z^{(a)}$
- This adds some noise to the values $z^{[l]}$ within that minibatch. So similar to dropout, it adds some noise to each hidden layer's activations. μ, σ^2
- This has a slight regularization effect.

mini-batch: 64

512

Andrew Ng

At test time, we shall use exponentially average to calculate the mean and variance:

Batch Norm at test time

$$\begin{aligned} \rightarrow \mu &= \frac{1}{m} \sum_i z^{(i)} \\ \rightarrow \sigma^2 &= \frac{1}{m} \sum_i (z^{(i)} - \mu)^2 \\ \rightarrow z_{\text{norm}}^{(i)} &= \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \rightarrow \tilde{z}^{(i)} &= \gamma z_{\text{norm}}^{(i)} + \beta \end{aligned}$$

μ, σ^2 : estimate using exponentially weighted average (across mini-batches).

$X^{(1)}, X^{(2)}, X^{(3)}, \dots$

\downarrow

$\mu^{(1)}, \mu^{(2)}, \mu^{(3)}, \dots$

\downarrow

$\sigma_1^2, \sigma_2^2, \sigma_3^2, \dots$

\downarrow

μ, σ^2

\leftarrow

$z_{\text{norm}} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}}$

$\tilde{z} = \gamma z_{\text{norm}} + \beta \leftarrow$

Andrew Ng

3) Multi-class Classification:

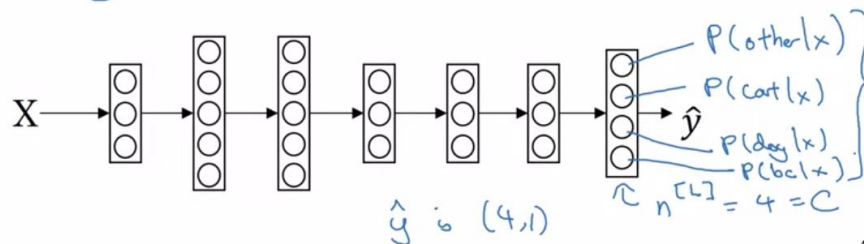
Rather than classifying 0-1 task (like is it a cat or not), we will classify multi things (like cat, dog or horse). C is the number of classes starting from class 0 to class c-1.

Recognizing cats, dogs, and baby chicks, *other*



3 1 2 0 3 2 0 1

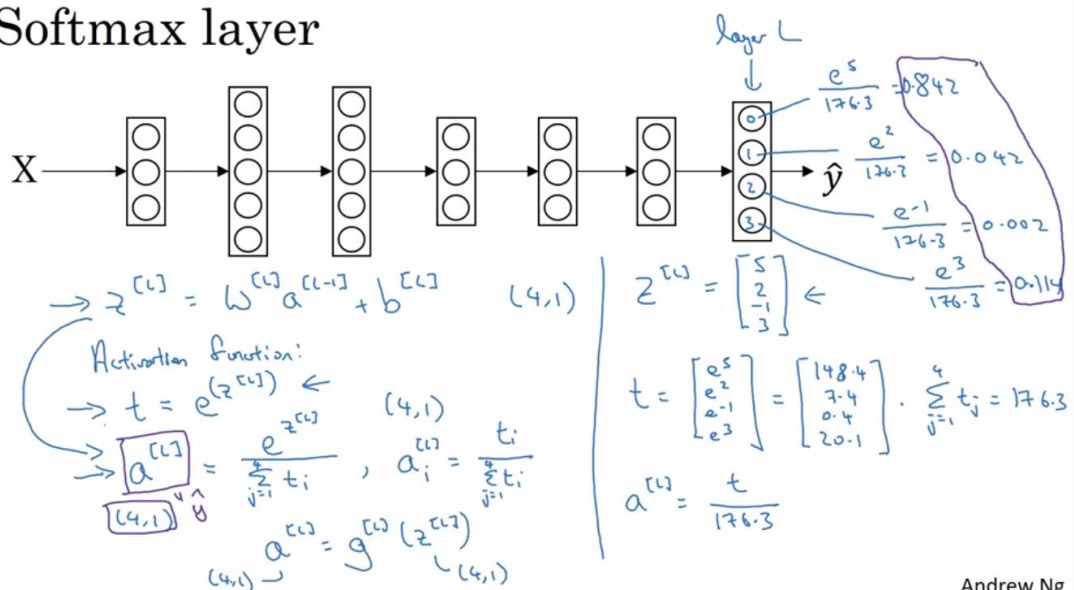
$C = \#classes = 4$ $(0, \dots, 3)$



Andrew Ng

For this task, the output layer is called Softmax Layer. This layer has exponential activation function:

Softmax layer



Andrew Ng

To train this NN, we need to make the specific output y^{\wedge} very large in order to make the loss in it very small. Also, we need to train over m examples:

Loss function

$(4,1)$
 $y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ ← cat $y_2 = 1$
 $y_1 = y_3 = y_4 = 0$
 $(4,1)$
 $\hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$
 $C = 4$

$$\mathcal{L}(\hat{y}, y) = - \sum_{j=1}^C y_j \log \hat{y}_j$$

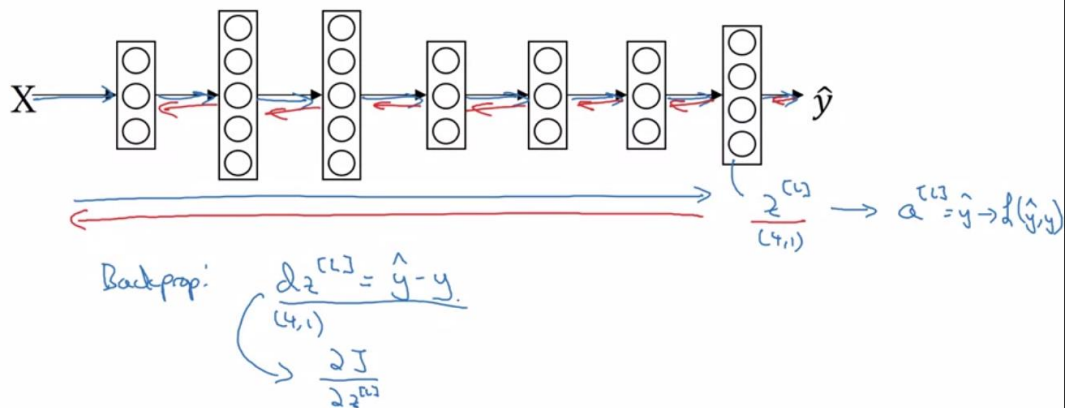
$$\mathcal{J}(w^{(1)}, b^{(1)}, \dots) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$-y_2 \log \hat{y}_2 = -\log \hat{y}_2$ make \hat{y}_2 big.
 $Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]$ $\hat{Y} = [\hat{y}^{(1)} \ \dots \ \hat{y}^{(m)}]$
 $= \begin{bmatrix} 0 & 0 & 1 & 0 & \dots \\ 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \end{bmatrix}$ $= \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 & \dots \end{bmatrix}$
 $(4, m)$ $(4, m)$

Andrew Ng

Also do the back propagation:

Gradient descent with softmax



4) Programming frameworks:

Deep learning frameworks

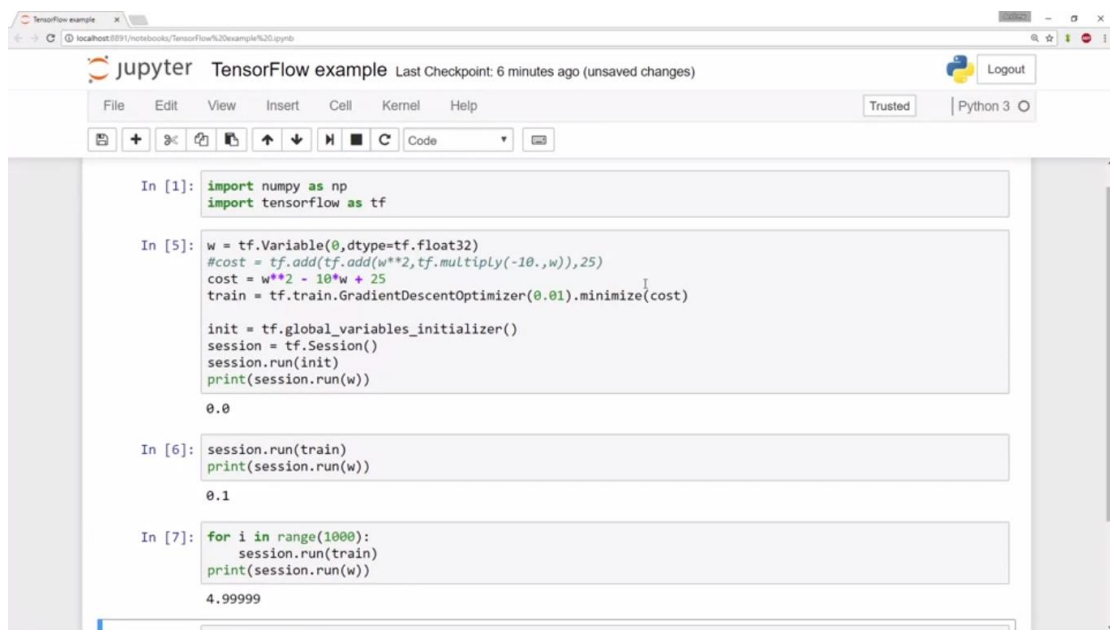
- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing deep learning frameworks

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)

Andrew Ng

TensorFlow:



```
In [1]: import numpy as np
import tensorflow as tf

In [5]: w = tf.Variable(0, dtype=tf.float32)
#cost = tf.add(tf.add(w**2, tf.multiply(-10., w)), 25)
cost = w**2 - 10*w + 25
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

0.0

In [6]: session.run(train)
print(session.run(w))

0.1

In [7]: for i in range(1000):
    session.run(train)
    print(session.run(w))

4.99999
```

- Backpropagation is built-in in tensorflow (no need to write it)

Code example

```
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))
```

```
with tf.Session() as session:
```

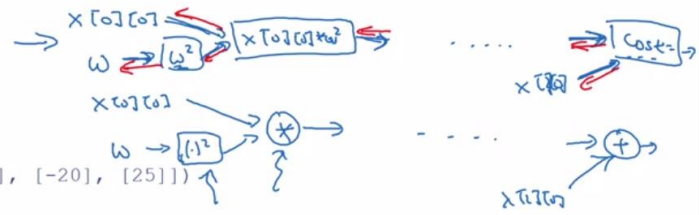
```
    session.run(init)
```

```
    print(session.run(w))
```

```
for i in range(1000):
```

```
    session.run(train, feed_dict={x:coefficients})
```

```
print(session.run(w))
```



Andrew Ng