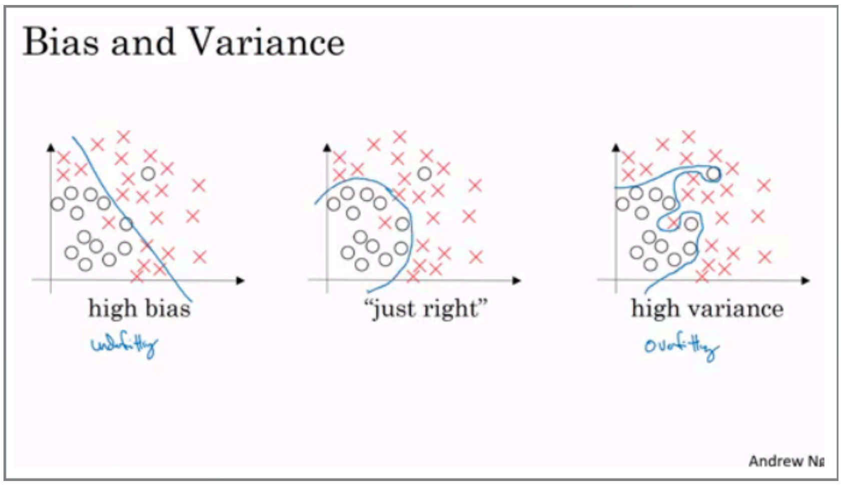
Part2. Improving Deep Neural Networks: Hyperparameters tuning, Regularization and Optmization

1. Practical aspects of Deep Learning

Train/Dev/Test sets

- It is impossible to get all your hyperparameters right on a new application from the first time. You should go through the loop: IDEA, CODE, EXPERIMENT many times to figure out your optimal hyperparameters.
- Your data would be split into three parts: train set, development set, test set.You will try to build a model upon 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 test set.
- The trend on the ratio of splitting the models:
  - if size of the dataset is small: 60/20/20
  - if size of the dataset is big enough: 98/1/1 or 99.5/0.25/0.25
- Make sure the dev and test set are coming from the same distribution.
- The dev set rule is to try them on some of the good models you've created. It's OK to only have a dev set without a test 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 it is used in development phase.

Bias and Variance



Bias and Variance

Such technique are easy to learn, but difficult to master:

- If your model is underfitting, it has a high "bias" and high "training_error".
- If your model is overfitting, it has a high "variance" and high "dev_error".
- These assumptions came from that human error is 0%. If the problem isn't like that, you will need to use human error as the baseline.

Basic Recipe for Machine Learning

- If your algorithm has a high bias:
  - Try to make your NN bigger (in size of hidden units or number of layers)
  - Try a different model that is suitable for your data
  - Try to run it longer
  - Use different (more advanced) optimization algorithms
- If your algorithm has a high variance:
  - More data
  - Try <u>regularization</u>
  - Try a different model that is suitable for your data
- You should try the previous two points until you have both low bias and low variance. (Training a bigger neural network never hurts!)
- In the older days before deep learning, there was a "Bias/Variance tradeoff". But because now you have more options and tools for solving the bias and variance problem, it is really helpful to use deep learning.

Regularization

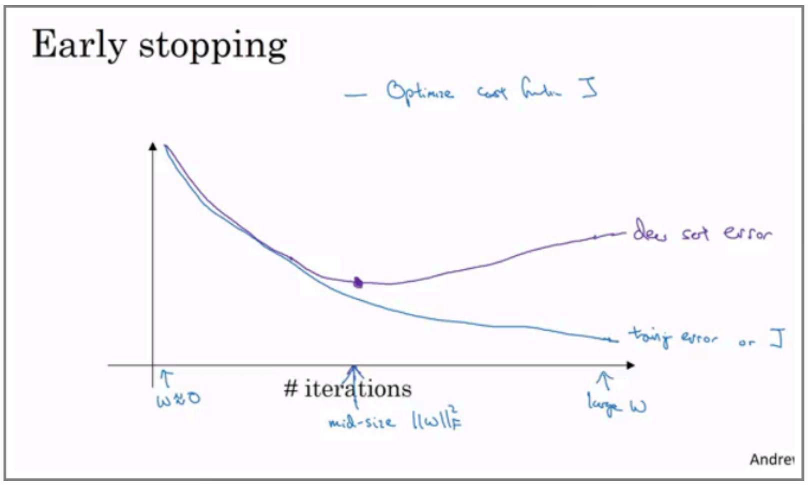
- L1 matrix norm: $||W|| = \sum |w_{ij}|$
- L2 matrix norm: $||W||^2 = \sum (w_{ij})^2$
- L1 regularization: $J(w,b) = \frac{1}{m}\sum L(y_i, y_i') + \frac{\lambda}{2m}\sum |w_i|$ (here, λ is hyperparemter)
- L2 regulariztion: $J(w,b) = \frac{1}{m}\sum L(y_i, y_i') + \frac{\lambda}{2m}\sum (w_i)^2$
- New weight update step: $w[l] = (1 - \frac{\alpha\lambda}{m})w[l] - \alpha\frac{dJ}{dw[l]}$
  - The new term $(1 - \frac{\alpha\lambda}{m})w[l]$ causes the <u>weights to decay</u> in propoertion to its size.
- The L1 regularization version makes a lot of w values become zeros, which makes the model size smaller. Thus, L2 regularization is being used much more often.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias. In practice, this penalizes large weights and effectively limits the freedom in your model.
- 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 <u>monotonically</u> after every eleation of gradient descent with regularization. If you plot the old definition of J without 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. A most common technique to implement dropout is called "inverted dropout".
- Dropout can have different "keep_prob" per layer.
  - The input layer dropout has to be near 1 (or without dropout) since you don't want to eliminate a lot of input features.
  - Having different "keep_prob" 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.
- A lot of researchers are using dropout with computer vision because they have a very big input size and almost never have enough data, thus having overfitting problem almost for every incident.
- A downside of dropout is that the cost function J is not well-defined and it will be hard to debug.
  - To solve it, you'll need to turn off dropout (set all "keep_prob" to 1), and then run the code, checking that J monotonically decreases and then turn on the dropouts again.
- At the test time, don't use dropout. If you implement dropout at test time, it would add noise to predicitions.

Other regularization methods

1. Data augmentation:
- Flip pictures horizontally, Apply a random rotation or distortion etc.
- New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
2. Early Stopping



Early Stopping

- 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. Pick the point at which the training set error and dev set error are optimal and take as the best parameters.
- Not recommended since this technique simultaneously tries to minimize the cost function and not to overfit which contradicts the orthogonalization approach.
- Its advantage is that you don't need to search for hyperparameters like in other regularization approaches.

3. Model Ensembles

- Train multiple independent models and average their results at test time.

Normalizing Inputs

- If you normalize your inputs, this will speed up the training process a lot.
- These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set.)
- Normalization are going on these steps:
  - Get the mean of the training set: $\mu = \frac{1}{m} \sum x_i$
  - Subtract the mean for each input, making your inputs centered around 0: $x_i = x_i - \mu$
  - Get the variance of the training set: $\sigma^2 = \frac{1}{m} \sum x_i^2$
  - Normalize the variance: $x_i = \frac{x_i}{\sigma^2}$
- 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 (looking more symmetric in a circular shape) and we can use a larger learning rate, making the optimization faster.

Vanishing/Exploding Gradients

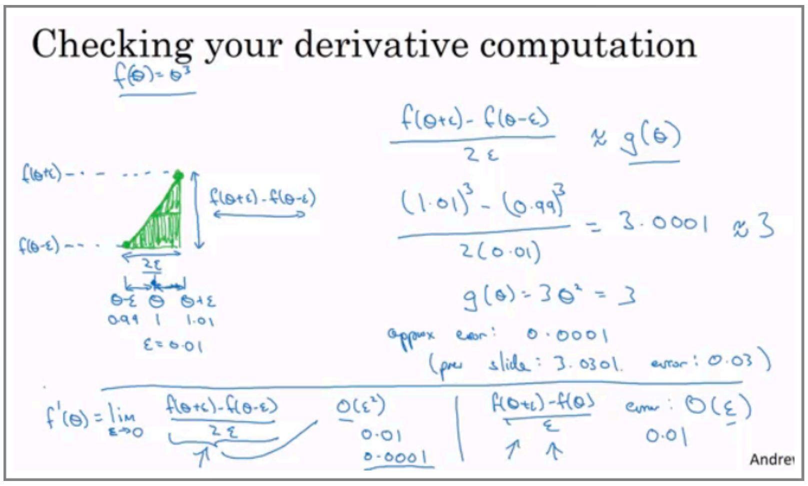
- The Vanishing / Exploding gradients occurs when your derivatives become very small or very big.
  - If $W > I$, the activation and gradients will explode.
  - If $W < I$, the activation and gradients will vanish.
- Recently, Microsoft trained 152 layers (ResNet) which is a really big number of layers. With such a deep neural network, your activations or gradients might increase or

decrease exponentially as a function of L. This makes training much difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps, taking a long time for gradient descent to learn anything.

Weight Initialization for Deep Networks

- A partial solution to the Vanishing/Exploding gadients in NN is better choice of the random initialization of weights.
- <u>He Initialization</u>: $W^{[l]} \sim N(0, \frac{2}{(1+a^2)\times n^{[l-1]}})$ (better to use with ReLU activation)
- <u>Xavier Initialization</u>: $W^{[l]} \sim N(0, \frac{1}{n^{[l-1]}})$ (better to use with tanh activation)

Numerical approximation of Gradients



Gradient Checking

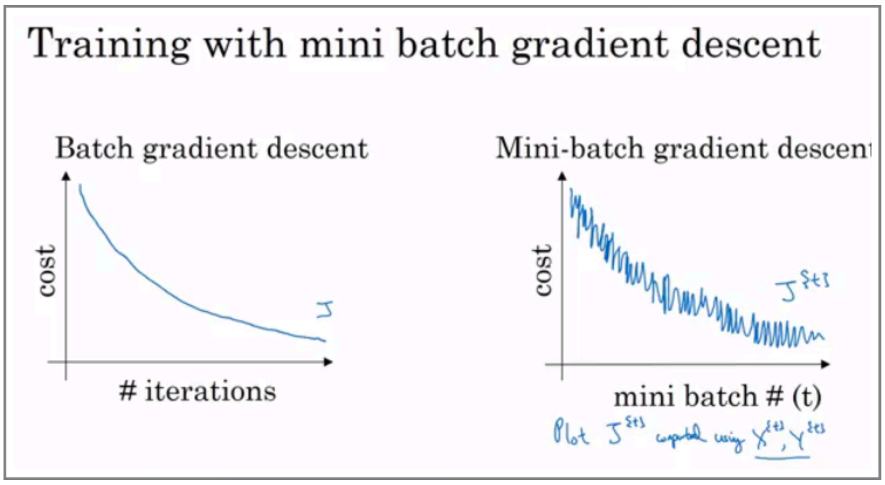
- There is an technique called gradient checking which tells you if your implementation of back-propagation is correct. Gradient checking approximates the gradients and is very helpful for finding errors in your back-propagation implementation, but it's much slower than the gradient descent. So don't use it at trianing time, just use for debugging.
- Gradient checking doesn't work with dropout because J is not consistent. You can first turn off dropout, run gradient checking and then turn on dropout again.
- Gradient checking is going on these steps:
  - Reshape $W[1], b[1], \ldots, W[L], b[L]$ into one big vector θ with cost function $L(\theta)$
  - Reshape $dW[1], db[1], \ldots, dW[L], db[L]$ into one big vector d_θ
  - Compute $\text{approx}(d_\theta) = \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$ and $x = \frac{||\text{approx}(d_\theta) - d_\theta||}{||\text{approx}(d_\theta)|| + ||d_\theta||}$ with $\epsilon = 10^{-7}$

- If $x < 10^{-7}$: Great! Very likely the back-propagation implementation is correct.
- If x is around 10^{-5}: OK, but need to inspect if there are no particularly big values in vector x.
- If $x \geq 10^{-3}$: Bad.. Probably there is a bug in back-propagation implementation.

2. Optimization algorithms

Mini-batch Gradient Descent

- Training NN with a large data is slow. It turns out that you can make a faster algorithm by making gradient descent process some of your items even before you finish the whole data. Such approach works much faster in the large datasets.
- In Batch Gradient Descent, we run the gradient descent on the whole dataset.
    - Too long per iteration
    - `mini_batch_size = m`
- In Stochastic Gradient Descent, we run the gradient descent on a single dataset.
    - Too noisy regarding cost minimization (can be reduced by using smaller α); won't ever converge to reach the minimum cost
    - Lose speedup from vectorization
    - `mini_batch_size = 1`
- In Mini-Batch Gradient Descent, we run the gradient descent on the mini datasets.
    - Faster learning: have the vectorization advantage, make progress without waiting to process the entire training set
    - Doesn't always exactly converge; oscillates in a very small region (can be reduced by using smaller α)
- In mini-batch algorithm, the cost won't go 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 decreases on each iteration.)



Batch Gradient and Mini-batch Gradient

- Guidelines for choosing hyperparameter mini-batch size:
    - if small training set (<2000), use batch gradient descent.

  - 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
  - Make sure that mini-batch fits in CPU/GPU memory

Exponentially Weighted Averages

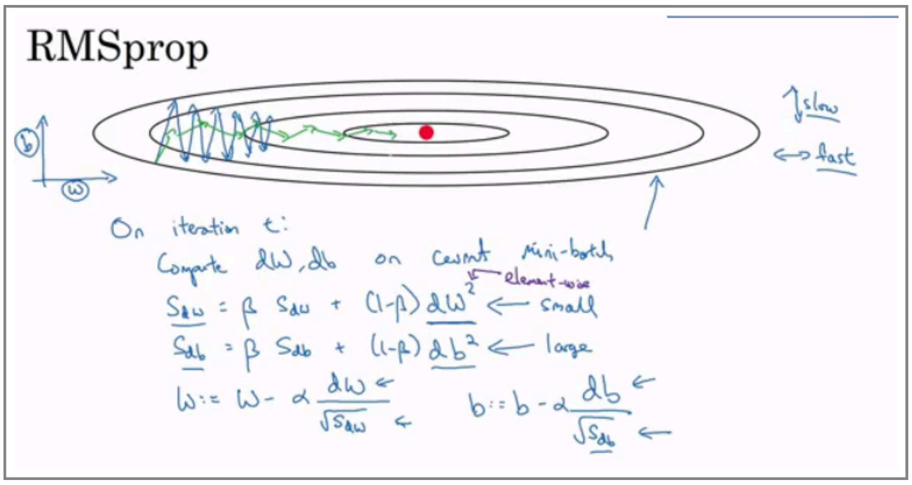
- $V(t) = \beta V(t-1) + (1-\beta)\theta(t), V(0) = 0$: represent averages over $\dfrac{1}{1-\beta}$ entries
- The bias correction helps make the exponentially weighted averages more accurate.
  - Because $V(0) = 0$, the bias of the weighted averages is shifted and the accuracy suffers at the start.
  - $V(t) = \dfrac{\beta V(t-1) + (1-\beta)\theta(t)}{1-\beta^t}$ solves the bias issue as t becomes larger, $1-\beta^t$ becomes close to 1.
- β is another hyperparameter. $\beta = 0.9$ is very common and works well in most cases.
- In practice, people don't bother implementing bias correction.

Gradient Descent with Momentum

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

RMS Prop (Root Mean Square Prop)

- RMS Prop wil make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example:
  - Ensure that s_{dW} is not zero by adding a small value $\epsilon = 10^{-8}$



RMSprop

On iteration t:

Compute dW, db on current mini-batch

$S_{dW} = \beta S_{dW} + (1-\beta) dW^2 \leftarrow$ small

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

$W := W - \alpha \dfrac{dW}{\sqrt{S_{dW}}}$ $\qquad b := b - \alpha \dfrac{db}{\sqrt{S_{db}}}$

RMS Prop

Adam Optimization Algorithm (Adaptive Moment Estimation)

- Adam optimization and RMS Prop are among the optimization algorithms that worked very well with a lot of NN architectures.
- Adam optimization simply puts RMP prop and momentum together!

Learning Rate Decay

- decay_rate = d, epoch_num = e
- $\alpha = \frac{1}{1+de}\alpha$: slowly reduce learning rate.
  - $\alpha = 0.95^e\alpha$
  - $\alpha = \frac{k}{\sqrt{e}}\alpha$
- Some people are making changes to the learning rate manually.
- As mentioned before, mini-batch gradient descent won't reach the optimum point to converge. But by making the learning rate decay with iterations, it will be much closer to the opt because the steps (and possibly oscillations) near the optimum are smaller.
- 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. Thus, it's unlikely to get stuck in a bad local optima in high dimensions.
- It is much more likely to get to the saddle point rather than the lcoal minima. Pleateaus can make learning slow:
  - Plateau is a region where the derivative is close to zero for a long time.
  - This is where algorithms like momentum, RMS Prop and Adam can help.

3. Hyperparameter tuning, Batch Normalization and Programming Frameworks

Tuning Process

- Need to tune the hyperparameters to get the best out of them
- Importance of hyperparameters
  - Learning rate, Momentum beta, Mini-batch size, number of hidden units and layers, learning rate decay, regularization lambda, activation functions, and Adam β_1 and β_2
- It is hard to decide which hyperparameter is the most important in a problem. It depends a lot on your problem.
- He ways to tune is to sample a grid with N hyperparameter settings and then try all setting combinations on your problem, or you can just use random values.
- Or you can use Coarse to fine sampling scheme.
  - When you find some hyperparameters that give you a better performance- zoom into a smaller region around these values and sample more densely within this space.
  - Can be automated.

Using an Appropriate Scale to Pick Hyperparameters

Let's say you have a specific range for a hyperparameters from $a \leq x \leq b$. It is better to search for the right ones using the logarithmic scale rather then in linear scale.

Hyperparameters Tuning in Practice: Pandas vs. Caviar

Intuitions about hyperparameters settings from one application area may or may not transfer to a different one.

- You can use the "babysitting model" if you don't have much computational resources.
  - You might initialize your parameters as random and then start training.
  - Then you watch your learning curve gradually decrease over the day. You also nudge your parameters a little during training.
- You can run some models in parallel and at the end, you can check the results.

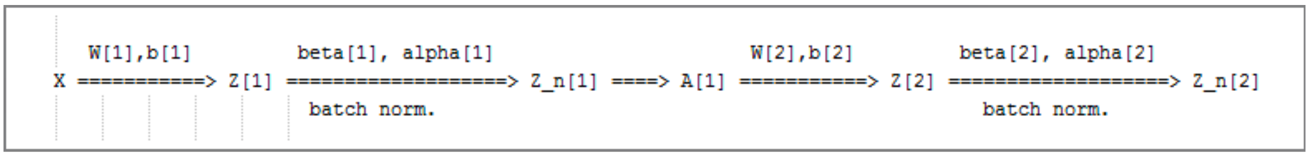
Normalization Activations in a Network

In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization created by two researchers, Sergey Ioffe and Christian Szegedy. It speeds up learning a lot.

Before, we normalized input by subtracting the mean and dividing by variance. This helped a lot for the shape of the cost functio and for reaching the minimum point faster. The question is: for any hidden leayer, can we normalize $A[l]$ to train $W[l]$, $b[l]$ faster?

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

- Given $Z[l] = [z(1), \ldots, z(m)]$, compute $\mu = \frac{1}{m} \sum z(i)$ and $\sigma^2 = \frac{1}{m} \sum (z(i) - \mu)^2$.
- $Z_n[i] = \frac{(z(i) - \mu)}{\sqrt{\sigma^2 + \epsilon}}$ (add ϵ for numerical stability if $\sigma^2 = 0$)
- $Z_{\tilde{i}} = \gamma Z_n[i] + \beta$ γ and β are learnable parameters of the model)
  - Allows inputs belong to other distribution (with other mean and variance)
  - Makes the NN learn the distribution of the outputs
- You can remove $b[l]$ or make it always zero if you are using batch normalization, because it always eliminate $b[l]$.



```
   W[1],b[1]        beta[1], alpha[1]                  W[2],b[2]        beta[2], alpha[2]
X ===========> Z[1] ===================> Z_n[1] ====> A[1] ===========> Z[2] ===================> Z_n[2]
                     batch norm.                                              batch norm.
```

Using Batch Normalization into a NN

- If you are using a deep learning framework, you won't have to implement batch norm yourself.
- Batch normalization is usually applied with mini-batches.

Why does Batch Normalization Work?

- The first reason is the same reason as why we normalize X.
- The second reason is that batch normalization reduces the problem of input values changing and shifting.
- Batch normalization does some regularization:
  - Each mini-batch is scaled by the mean/variance computed of that mini-batch. This adds some noise to the values $Z[l]$ within that mini-batch. So, similar to dropout, it adds some noise to each hidden layer's activations. Thus, it has a slight regularization effect.
  - Using bigger size of the mini-batch, you are reducing noise and therefore regularization effect.
  - Don't rely on batch normalization as a regularization. It's intended for normalization of hidden units, activations and therefore speeding up learning. For regularization, use other regularization techniques (L2 or dropout).

Batch Normalization at Test Time

- When we train a NN with batch normalization, we compute the mean and the variance of the mini-batch.
- In testing, we might need to process examples one at a time. The mean and the variance of one example won't make sense.
- Thus, 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. Such method is also sometimes called "Running average".
- In practice, you will most often use a deep learning framework and it will contain some default implmentation of doing such a thing.

Softmax Regression

- In every example, we have used so far we were talking about binary classification. There is a generalization of logistic regression called Softmax Regression that is used for multiclass classification/regression.
- There is an activation called hardmax, whcih gets 1 for the maximum value and zeros for the others. The Softmax came from softening the values in contrast to hardmax.
- $\sigma(z_l) = \frac{e^{z_l}}{\sum e^{z_i}}$

Deep Learning Frameworks

- It's not practical to implement everything from scratch. Deep learning is now in the phase of doing something with the frameworks and not from scratch to keep on going. There are many good deep learning frameworks you can use: Caffe, CNTK, Keras, Lasagne, TensorFlow, Torch/PyTorch etc. These frameworks are getting better month by month.
- How to choose deep learning framework:
  - Ease of programming in development and deployment
  - Running speed
  - Truly open source with good governance

Tensorflow Toy Code

```python
import numpy as np
import tensorflow as tf

w = tf.Variable(0, dtype=tf.float32) # Crate variable W
cost = tf.add(tf.add(w**2, tf.multiply(-10.0, w)), 25.0) # cost = w^2 - 10w + 25
train = tf.train.GradientDescentOptimizer.minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
session.run(w)
session.run(train)
# session.run(train, feed_dict = {x: coefficients}

print("W after one iteration: ", session.run(w))

for i in range(1000):
  session.run(train)
  # session.run(train, feed_dict = {x: coefficients}

print("W after 1000 iterations: ", session.run(w))
```

Python   Copy   Caption   •••

```python
# better for cleaning up in case of error/exception
with tf.Session() as session:
  session.run(init)
  session.run(w)

# cost function
tf.nn.sigmoid_cross_entropy_with_logits(logits=..., lables=...)

# reset the graph
tf.reset_default_graph()
```