Neural Networks: Practical Issues

Michael

School of Mathematics and Statistics, UCD

School of Economics, University of Nottingham

1 Introduction

When we apply our machine learning (ML) or deep learning (DL) models in different problems, we will encounter so many different practical issues, such as choices of activation functions, value of learning rate, etc. Knowing how to do hyper-parameter tuning, regularization and optimization is essential for improving our ML/DL models. According to Andrew Ng, practicing ML/DL models is a process of iteration, in which you have to learn to find the best choices of hyper-parameters by keeping trying different methods.

In this set of notes, we will discuss the following topics:

- Model assessment
- Regulation
- Optimization
- Hyper-parameters tuning and programming framework

2 Model Assessment

The generalization performance of a learning method relates to its prediction capability on independent test data. Assessment of this performance is extremely important in practice, since it guides the choice of learning method or model, and gives us a measure of the quality of the ultimately chosen model.

To assess our model, we need split our data into three different datasets:

- Training set
- Validation set
- Test set

When your dataset id not very big, the ratio for those datasets could be: 60/20/20. However, when you have a large dataset (for instance, 1 million), then the ratio for those datasets should be around: 98/1/1.

Before we continue to discuss model assessment, we have to know the difference between model selection and model assessment:

- Model selection: estimating the performance of different models in order to choose the best one.
- Model assessment: having chosen a final model, estimating its prediction error (generalization error) on new data.

The training set is used to fit the models; the validation set is used to estimate prediction error for model selection; the test set is used for assessment of the generalization error of the final chosen model. Ideally, the test set should be kept in a "valut", and be brought out only at the end of the data analysis.

2.1 The Bias-Variance Decomposition

Andrew Ng states that almost every good machine learning practitioner has a very sophisticated view on understanding of bias-variance decomposition. Hence, it is very importance to know what kind of factors will affect the bias-variance trade-off.

Now, if we assume that $Y = f(X) + \epsilon$ where $E(\epsilon) = 0$ and $\mathrm{Var}(\epsilon) = \sigma_\epsilon^2$, we can derive an expression for the expected prediction error of a regression fit $\hat{f}(X)$ at an input point $X = x_0$, using squared-error loss:

$$\begin{aligned}\mathrm{Err}(x_0) &= E[(Y - \hat{f}(x_0))^2 | X = x_0] \\ &= \sigma_\epsilon^2 + [E\hat{f}(x_0) - f(x_0)]^2 + E[\hat{f}(x_0) - E\hat{f}(x_0)]^2 \\ &= \sigma_\epsilon^2 + \mathrm{Bias}^2(\hat{f}(x_0)) + \mathrm{Var}(\hat{f}(x_0)) \\ &= \text{Irreducible Error} + \text{Bias}^2 + \text{Variance}\end{aligned}$$

The first term is the variance of the target around its true mean $f(x_0)$ and cannot be avoided no matter how well we estimate $f(x_0)$, unless $\sigma_\epsilon^2 = 0$. The second term is the squared bias, the amount by which the average of our estimate differs from the true mean; the last term is the variance; the expected squared deviation of $\hat{f}(x_0)$ around its mean. *Typically the more complex we make the model $\hat{f}$, the lower the (squared) bias but the higher the variance.*

When we had higher variance, we say our model is over-fitting, and when we had higher bias, we say our model is under-fitting.

Problem	Recipe
High bias	big neural network model, or more complex model
High variance	more data, regulation

One should realize that sometimes it is difficult to reduce bias and variance at the same time. In the era of big data, the bias-variance trade-off can be tackled in some sense when you got both complex neural network model and big dataset.

2.2 Regularization

The purpose of doing regularization is mainly to reduce the variance. Generally, we have L1 and L2 regularization. Taking regression models as examples, with L2 regularization, we have the following cost function:

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

With L1 regularization, the Lasso regression add s the absolute value of magnitude of coefficient as penalty term to the loss function:

$$\sum_{i=1}^{n}(yi - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

In neural network model, the formulas of cost function have to adjusted based on the above regularization. One thing I have to mention is that one should realize that when you change the cost function, the formulas for doing gradient descent will also change.

There is another way for doing regularization, which avoids the optimization process through the cost function. It is called dropout regularization. You can read more on this method of regularization in this website. By the way, dropout regularization is very frequently used by ML practitioners as it is more efficient.

There are other techniques that can help us to get the same effects of regularization, which include:

- Data augmentation
- Early stopping of iteration (no over-fitting)

2.3 Normalizing Inputs

It has become a standard step for normalizing inputs whenever you are doing ML model. Normalizing inputs could help us speed up the gradient descent process. Also, normalizing inputs could help us to avoid the exploration or vanishing issue of gradient descent.

In addition, **weight initialization** also will help us speed up the gradient descent part. There are some techniques on weight initialization, which you could find in this website. To help us track the process of gradient descent, sometimes we also employ the numerical approximation of gradient to do gradient checking.

3 Optimization Algorithms

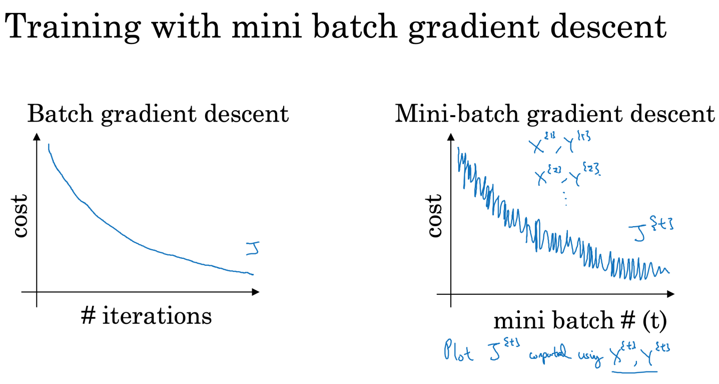
Once the dataset becomes large and model becomes more complex, then we need some techniques to speed up our optimization process. In this section, we will learn those techniques.

3.1 Mini-batch Gradient Descent

If out dataset is relative small, then we can do gradient descent by vectorizing our data. However, when the dataset becomes larger and larger, the gradient descent will become very slow even if we employed the vectorization. The key thing we concern during optimization process is the **convergency rate**. If we could find a way of updating our coefficients faster, then it will save us lots of time.

Of course, you could update your weights once after you going through all you data. But it would be smarter by updating our weights more frequently when we have a large dataset[1]. Mini-batch gradient descent is the way of updating gradient more frequent. After going through all mini-batch dataset, we say we finished one 'epoch' process.

Since we updating the gradient descent with each mini-batch, then our cost will not decrease consistently like we did it with the whole batch. The difference is presented in the following figure.



One should also realize that the *stochastic gradient descent* method updates the weights faster than others, but it does not guarantee that the weights will converge steadily. Also, once you have a large dataset, it almost loses the advantage of speeding up optimization.

For the size of mini-batch, normally one could follow this rule: $2^n, n \in \mathbb{Z}^+$.

3.2 Exponentially Weighted Averages

In time series, there is a very famous model called *moving average model.* We will review this moving average model first, then introduce the exponentially weighted average, and also explain how it can help us to speed up the optimization process.

The class of stochastic processes we will consider is known as the *Autoregressive Moving Average (ARMA)* class. This is a class of linear time series models with the general form,

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \epsilon_t - \theta_1 \epsilon_{t-1} - \theta_2 \epsilon_{t-2} - \cdots - \theta_q \theta_{t-q}$$

[1]It's all about how much information we can use, and how much we want to use.

We take ϵ_t to be a stochastic process such that

$$
\begin{aligned}
E(\epsilon_t) &= 0 \quad \forall\ t \\
V(\epsilon_t) &= \sigma^2 \quad \forall\ t \\
C(\epsilon_t, \epsilon_{t-s}) &= 0 \ \ \forall\ t
\end{aligned}
$$

The moving average process of order 1, MA(1) is given by

$$y_t = \epsilon_t - \theta \epsilon_{t-1}$$

The autoregressive process of order 1, AR(1) is given by

$$y_t = \phi y_{t-1} + \epsilon_t$$

It is easy to show that AR(1) is equivalent to MA(∞) process,

$$y_t = \epsilon_t + \phi \epsilon_{t-1} + \phi^2 \epsilon_{t-2} + \cdots$$

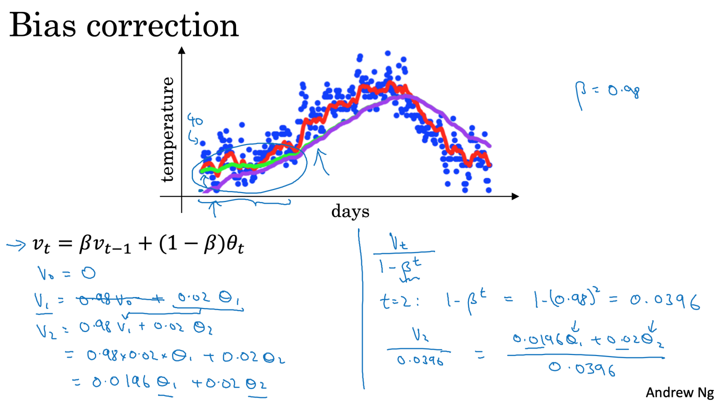
The exponentially weighted average is given by

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t \tag{3.1}$$

Again, rewrite the above equation recursively, then you would get the equivalent moving average model

$$V_t = (1-\beta)\theta_t + (1-\beta)\beta\theta_{t-1} + (1-\beta)\beta^2\theta_{t-2} + \cdots$$

Since $\beta < 1$, then the weight of θ decays very fast with decreasing time, and the decay rate is close to $1/e$, which is the reason of calling this model *exponentially weighted averages* model. Sometimes, we also do bias correction in exponentially weighted averages by dividing the right side of equation (3.1) by $1-\beta^t$, which will make the average more close to the real data at the early stage.



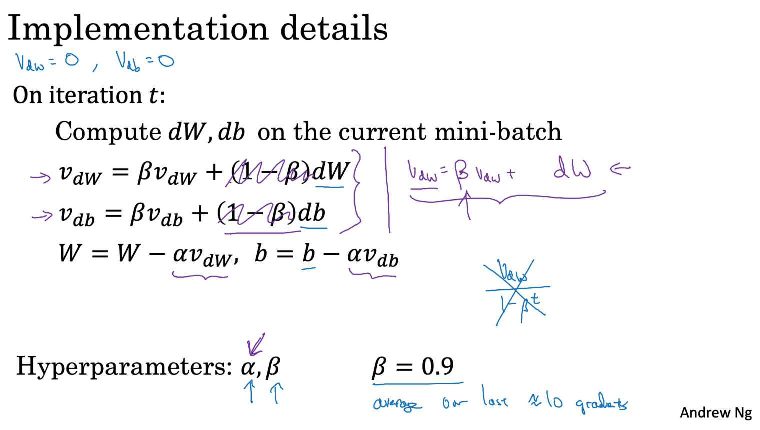
Why do we use exponentially weighted average model?

Once you have the very large dataset, then it is better to extract the key information from you data and then use those information to train you model. For instance, in the above picture, we just use moving average line rather than all data plotted in dots. You can understand it as a techniques of dimension reduction.

3.3 Gradient Descent with Momentum

When we are training our model, we need calculate the derivatives based on cost function before updating the gradient. Once we have more than one dimensions, convergence behavior of weights is full of noisy even thought it approaches to the convergent value. *To get rid of those noisy behaviors*, we apply the exponentially weighted average techniques to derivatives of weighs, and call this method *gradient descent with momentum*.

The following figure gives the details of implementing gradient descent with momentum in your model.



Implementation details

$V_{dW}=0$, $V_{db}=0$

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

$V_{dW} = \beta V_{dW} + dW \leftarrow$

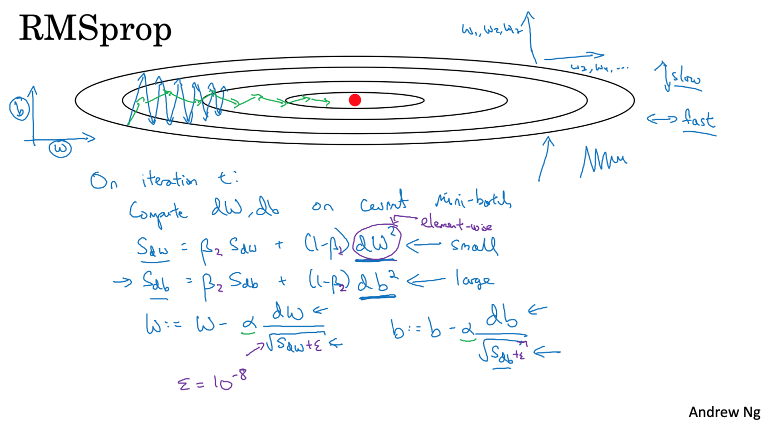
$\cancel{\frac{V_{dW}}{1-\beta^t}}$

Hyperparameters: α, β $\quad \beta = 0.9$

average over last ≈10 gradients

Andrew Ng

There is another method for speeding up the gradient descent updating process, which is called RMSprop and presented in the following figure.



RMSprop

On iteration t:

Compute dW, db on current mini-batch

$S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2 \leftarrow$ small (element-wise)

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

$W := W - \alpha \frac{dW}{\sqrt{S_{dW}+\varepsilon}}$ $\quad b := b - \alpha \frac{db}{\sqrt{S_{db}+\varepsilon}}$

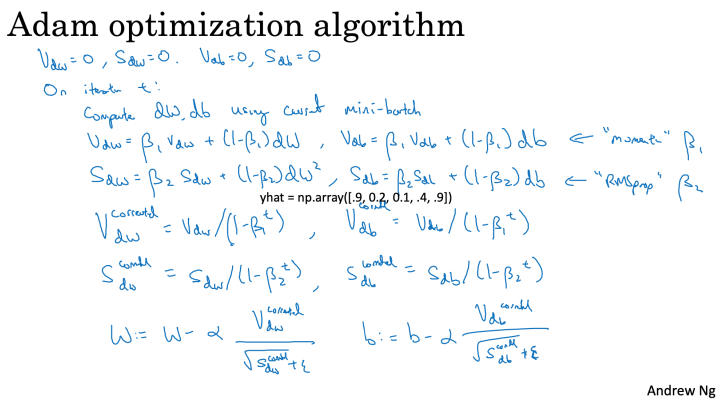
$\varepsilon = 10^{-8}$

Andrew Ng

It is all about oscillation

Similar to gradient descent with momentum, RMSprop reduces oscillation of derivatives more by using the square of derivative. I have to say that this method is quite smart as it basically just restricts the gradient descent to one direction (either up or down), which means the convergence rate should be around twice faster than normal optimization.

Now, we can combine gradient descent with momentum with RMSprop, and get a new algorithm, called Adam (adaptive momentum estimation) optimization algorithm. The implementation details are shown in the next figure.



Adam optimization algorithm

$V_{dw}=0,\ S_{dw}=0.\quad V_{db}=0,\ S_{db}=0$

On iteration t:

Compute dW, db using current mini-batch

$V_{dw}=\beta_1 V_{dw}+(1-\beta_1)dW,\quad V_{db}=\beta_1 V_{db}+(1-\beta_1)db \leftarrow$ "momentum" β_1

$S_{dw}=\beta_2 S_{dw}+(1-\beta_2)dW^2,\quad S_{db}=\beta_2 S_{db}+(1-\beta_2)db \leftarrow$ "RMSprop" β_2

yhat = np.array([.9, 0.2, 0.1, .4, .9])

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

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

$W:=W-\alpha\frac{V_{dw}^{\text{corrected}}}{\sqrt{S_{dw}^{\text{corrected}}}+\varepsilon}\qquad b:=b-\alpha\frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}}}+\varepsilon}$

Andrew Ng

Institution for this optimization is very straightforward: first you extract momentum of derivatives, then you restrict the momentum of derivatives into one direction. *So, if RMSprop moves forward by two steps, then Adam is more like moving forward by two steps along the exponentially moving average line rather following the very sparse data.*

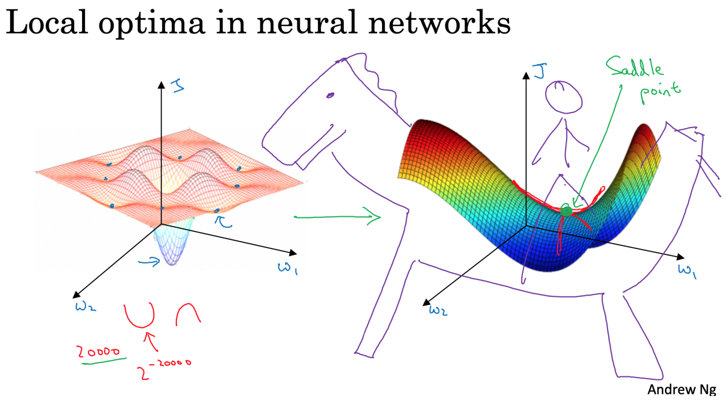
3.4 Learning Rate Decay

The intuition for learning rate decay is to improve the accuracy of gradient descent process by decaying the learning rate when weights are approaching to the convergent value[2].

Since learning rate decay improves the accuracy of estimation, sometimes it will slow down the optimization process.

3.5 Be Careful on Plateaus

What a lovely picture to finish this subsection.



4 Implementations in Python

The code examples below illustrate the difference between stochastic gradient descent and (batch) gradient descent.

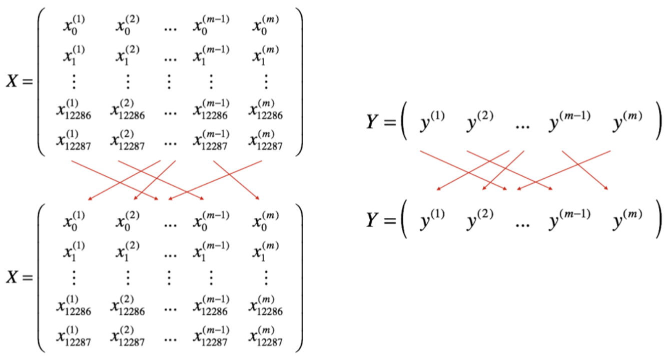
```
# (Batch) Gradient Descent
X = data_input
```

[2]Remember, computer scientists are living in a discreet world!

```
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    # Forward propagation
    a, caches = forward_propagation(X, parameters)
    # Compute cost.
    cost += compute_cost(a, Y)
    # Backward propagation.
    grads = backward_propagation(a, caches, parameters)
    # Update parameters.
    parameters = update_parameters(parameters, grads)

# Stochastic Gradient Descent
X = data_input
Y = labels
parameters = initialize_parameters(layers_dims)
for i in range(0, num_iterations):
    for j in range(0, m):
        # Forward propagation
        a, caches = forward_propagation(X[:,j], parameters)
        # Compute cost
        cost += compute_cost(a, Y[:,j])
        # Backward propagation
        grads = backward_propagation(a, caches, parameters)
        # Update parameters.
        parameters = update_parameters(parameters, grads)
```

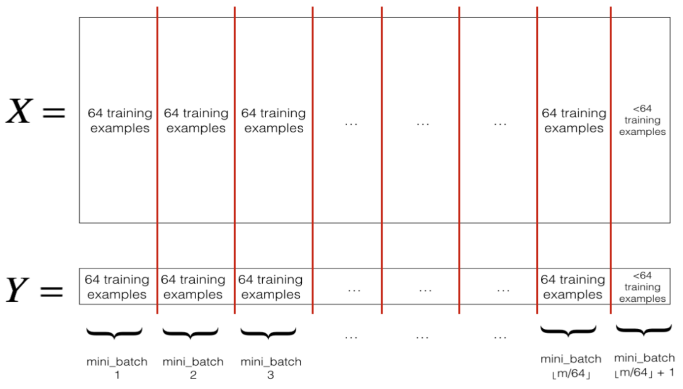
Now, let's learn how to build mini-batches from the training set (X, Y). There are two steps:

- Shuffle: create a shuffled version of the training set (X, Y) as shown below. Each column of X and Y represents a training example. Note that the random shuffling is done *synchronously* between X and Y.



$X = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m-1)} & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m-1)} & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{12286}^{(1)} & x_{12286}^{(2)} & \dots & x_{12286}^{(m-1)} & x_{12286}^{(m)} \\ x_{12287}^{(1)} & x_{12287}^{(2)} & \dots & x_{12287}^{(m-1)} & x_{12287}^{(m)} \end{pmatrix}$
$Y = \begin{pmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m-1)} & y^{(m)} \end{pmatrix}$

- Partition: partition the shuffled (X, Y) into mini-batches of size `mini_batch_size`(e.g., 64). Note that the number of training examples is not always divisible by `mini_batch_size`. The last mini batch might be smaller, but you don't need to worry about this.



Here is the code of doing mini-batch in Python.

```
# GRADED FUNCTION: random_mini_batches

def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):
    """
    Creates a list of random minibatches from (X, Y)

    Arguments:
    X -- input data, of shape (input size, number of examples)
    Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (1,
                                           number of examples)
    mini_batch_size -- size of the mini-batches, integer

    Returns:
    mini_batches -- list of synchronous (mini_batch_X, mini_batch_Y)
    """

    np.random.seed(seed)            # To make your "random" minibatches the
                                           same as ours
    m = X.shape[1]                  # number of training examples
    mini_batches = []

    # Step 1: Shuffle (X, Y)
    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((1,m))

    # Step 2: Partition (shuffled_X, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m/mini_batch_size) # number of
                                          mini batches of size
                                          mini_batch_size in your
                                          partitionning
    for k in range(0, num_complete_minibatches):
        ### START CODE HERE ### (approx. 2 lines)
        mini_batch_X = shuffled_X[:, k*mini_batch_size : (k+1)*
                                              mini_batch_size]
        mini_batch_Y = shuffled_Y[:, k*mini_batch_size : (k+1)*
                                              mini_batch_size]
        ### END CODE HERE ###
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        ### START CODE HERE ### (approx. 2 lines)
        mini_batch_X = shuffled_X[:, -(m-(mini_batch_size*
                                              num_complete_minibatches)):]
        mini_batch_Y = shuffled_Y[:, -(m-(mini_batch_size*
                                              num_complete_minibatches)):]
        ### END CODE HERE ###
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batches
```

Now, let's implement gradient descent with momentum based on the following update rule, for $l = 1, \cdots, L$:

$$\begin{aligned}
v_{dW^{[l]}} &= \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\
W^{[l]} &= W^{[l]} - \alpha v_{dW^{[l]}} \\
v_{db^{[l]}} &= \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\
b^{[l]} &= b^{[l]} - \alpha v_{db^{[l]}}
\end{aligned}$$

```
# GRADED FUNCTION: initialize_velocity

def initialize_velocity(parameters):
    """
    Initializes the velocity as a python dictionary with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the
                                                      corresponding
                                                      gradients/parameters.
    Arguments:
    parameters -- python dictionary containing your parameters.
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl

    Returns:
    v -- python dictionary containing the current velocity.
                    v['dW' + str(l)] = velocity of dWl
                    v['db' + str(l)] = velocity of dbl
    """

    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}

    # Initialize velocity
    for l in range(L):
        ### START CODE HERE ### (approx. 2 lines)
        v["dW" + str(l+1)] = np.zeros((parameters['W'+str(l+1)].shape))
        v["db" + str(l+1)] = np.zeros((parameters['b'+str(l+1)].shape))
        ### END CODE HERE ###

    return v


# GRADED FUNCTION: update_parameters_with_momentum

def update_parameters_with_momentum(parameters, grads, v, beta,
                                           learning_rate):
```

```
    """
    Update parameters using Momentum

    Arguments:
    parameters -- python dictionary containing your parameters:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients for each
                                               parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    v -- python dictionary containing the current velocity:
                    v['dW' + str(l)] = ...
                    v['db' + str(l)] = ...
    beta -- the momentum hyperparameter, scalar
    learning_rate -- the learning rate, scalar

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- python dictionary containing your updated velocities
    """

    L = len(parameters) // 2 # number of layers in the neural networks

    # Momentum update for each parameter
    for l in range(L):

        ### START CODE HERE ### (approx. 4 lines)
        # compute velocities
        v["dW" + str(l+1)] = beta * v['dW'+str(l+1)] + (1 - beta) * grads['
                                          dW' + str(l+1)]
        v["db" + str(l+1)] = beta * v['db'+str(l+1)] + (1 - beta) * grads['
                                          db' + str(l+1)]
        # update parameters
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                                          learning_rate * v["dW" + str(
                                          l+1)]
        parameters["b" + str(l+1)] = parameters['b' + str(l+1)] -
                                          learning_rate * v['db' + str(
                                          l+1)]
        ### END CODE HERE ###

    return parameters, v
```

How do we choose β?

- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for $beta$ range from 0.8 to 0.999. If you don't feel inclined to tune this, $beta = 0.9$ is often a reasonable default.
- Turning the optimal β for your model might need trying several values to see what works best in term of the value of the cost function J.

Adam is one of the most effective optimization algorithm for training neural networks. It combines ideas from RMSProp and Momentum. Here is how it works:

1. It calculates an exponentially weighted average of past gradient, and stores it in variable v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction)
3. It updates parameters in a direction based on combining information from “1” and “2”.

The update rule is, for $l = 1, \cdots, L$:

$$\begin{aligned}
v_{dW^{[l]}} &= \beta_1 v_{dW^{[l]}} + (1-\beta_1)\frac{\partial J}{\partial W^{[l]}} \\
v^{corrected}_{dW^{[l]}} &= \frac{v_{dW^{[l]}}}{(1-\beta_1)^t} \\
s_{dW^{[l]}} &= \beta_2 v_{dW^{[l]}} + (1-\beta_2)(\frac{\partial J}{\partial W^{[l]}})^2 \\
s^{corrected}_{dW^{[l]}} &= \frac{s_{dW^{[l]}}}{(1-\beta_2)^t} \\
W^{[l]} &= W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}} + \varepsilon}
\end{aligned}$$

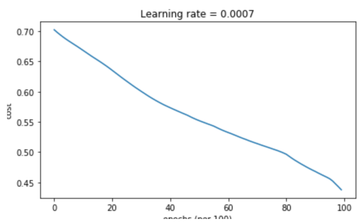
Here is the code in Python

```
# GRADED FUNCTION: initialize_adam

def initialize_adam(parameters) :
    """
    Initializes v and s as two python dictionaries with:
                - keys: "dW1", "db1", ..., "dWL", "dbL"
                - values: numpy arrays of zeros of the same shape as the
                                                    corresponding
                                                    gradients/parameters.

    Arguments:
    parameters -- python dictionary containing your parameters.
                    parameters["W" + str(l)] = Wl
                    parameters["b" + str(l)] = bl

    Returns:
    v -- python dictionary that will contain the exponentially weighted
                                            average of the gradient.
                    v["dW" + str(l)] = ...
                    v["db" + str(l)] = ...
    s -- python dictionary that will contain the exponentially weighted
                                            average of the squared gradient.
                    s["dW" + str(l)] = ...
                    s["db" + str(l)] = ...

    """
    L = len(parameters) // 2 # number of layers in the neural networks
    v = {}
    s = {}

    # Initialize v, s. Input: "parameters". Outputs: "v, s".
    for l in range(L):
    ### START CODE HERE ### (approx. 4 lines)
        v["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape))
        v["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape))
        s["dW" + str(l+1)] = np.zeros((parameters["W" + str(l+1)].shape))
        s["db" + str(l+1)] = np.zeros((parameters["b" + str(l+1)].shape))
    ### END CODE HERE ###

    return v, s


# GRADED FUNCTION: update_parameters_with_adam

def update_parameters_with_adam(parameters, grads, v, s, t, learning_rate =
                                    0.01,
                                beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8
                                                                        ):
    """
    Update parameters using Adam

    Arguments:
    parameters -- python dictionary containing your parameters:
                    parameters['W' + str(l)] = Wl
                    parameters['b' + str(l)] = bl
    grads -- python dictionary containing your gradients for each
                                      parameters:
                    grads['dW' + str(l)] = dWl
                    grads['db' + str(l)] = dbl
    v -- Adam variable, moving average of the first gradient, python
                                      dictionary
    s -- Adam variable, moving average of the squared gradient, python
                                      dictionary
    t --- counts the number of steps taken of Adam
    learning_rate -- the learning rate, scalar.
    beta1 -- Exponential decay hyperparameter for the first moment
                                      estimates
    beta2 -- Exponential decay hyperparameter for the second moment
                                      estimates
    epsilon -- hyperparameter preventing division by zero in Adam updates

    Returns:
    parameters -- python dictionary containing your updated parameters
    v -- Adam variable, moving average of the first gradient, python
                                      dictionary
    s -- Adam variable, moving average of the squared gradient, python
                                      dictionary
    """

    L = len(parameters) // 2                 # number of layers in the
                                      neural networks
    v_corrected = {}                         # Initializing first moment
                                      estimate, python dictionary
    s_corrected = {}                          # Initializing second moment
                                            estimate, python dictionary

    # Perform Adam update on all parameters
    for l in range(L):
        # Moving average of the gradients. Inputs: "v, grads, beta1".
                                                Output: "v".
        ### START CODE HERE ### (approx. 2 lines)
        v["dW" + str(l+1)] = beta1 * v["dW" + str(l+1)] + (1 - beta1) *
                                                grads['dW' + str(l+1)]
        v["db" + str(l+1)] = beta1 * v["db" + str(l+1)] + (1 - beta1) *
                                                grads['db' + str(l+1)]
        ### END CODE HERE ###

        # Compute bias-corrected first moment estimate. Inputs: "v, beta1,
                                                t". Output: "v_corrected".
        ### START CODE HERE ### (approx. 2 lines)
        v_corrected["dW" + str(l+1)] = v["dW" + str(l+1)] / (1 - np.power(
                                                beta1, t))
        v_corrected["db" + str(l+1)] = v["db" + str(l+1)] / (1 - np.power(
                                                beta1, t))
        ### END CODE HERE ###

        # Moving average of the squared gradients. Inputs: "s, grads, beta2
                                                ". Output: "s".
        ### START CODE HERE ### (approx. 2 lines)
        s["dW" + str(l+1)] = beta2 * s["dW" + str(l+1)] + (1 - beta2) * np.
                                                power(grads['dW' + str(l+1)],
                                                 2)
        s["db" + str(l+1)] = beta2 * s["db" + str(l+1)] + (1 - beta2) * np.
                                                power(grads['db' + str(l+1)],
                                                 2)
        ### END CODE HERE ###

        # Compute bias-corrected second raw moment estimate. Inputs: "s,
                                                beta2, t". Output: "
                                                s_corrected".
        ### START CODE HERE ### (approx. 2 lines)
        s_corrected["dW" + str(l+1)] = s["dW" + str(l+1)] / (1 - np.power(
                                                beta2, t))
        s_corrected["db" + str(l+1)] = s["db" + str(l+1)] / (1 - np.power(
                                                beta2, t))
        ### END CODE HERE ###

        # Update parameters. Inputs: "parameters, learning_rate,
                                                v_corrected, s_corrected,
                                                epsilon". Output: "parameters
                                                ".
        ### START CODE HERE ### (approx. 2 lines)
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
                                                learning_rate * v_corrected["
                                                dW" + str(l+1)] /(np.sqrt(
                                                s_corrected["dW" + str(l+1)]+
                                                epsilon))
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
                                                learning_rate * v_corrected["
                                                db" + str(l+1)] /(np.sqrt(
                                                s_corrected["db" + str(l+1)]+
                                                              epsilon))
        ### END CODE HERE ###

    return parameters, v, s
```

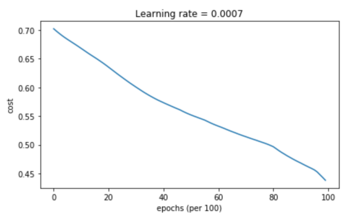
Here is the code of full model

```
def model(X, Y, layers_dims, optimizer, learning_rate = 0.0007,
                                         mini_batch_size = 64, beta = 0.9,
          beta1 = 0.9, beta2 = 0.999,  epsilon = 1e-8, num_epochs = 10000,
                                                  print_cost = True):
    """
    3-layer neural network model which can be run in different optimizer
                                         modes.

    Arguments:
    X -- input data, of shape (2, number of examples)
    Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (1,
                                          number of examples)
    layers_dims -- python list, containing the size of each layer
    learning_rate -- the learning rate, scalar.
    mini_batch_size -- the size of a mini batch
    beta -- Momentum hyperparameter
    beta1 -- Exponential decay hyperparameter for the past gradients
                                         estimates
    beta2 -- Exponential decay hyperparameter for the past squared
                                         gradients estimates
    epsilon -- hyperparameter preventing division by zero in Adam updates
    num_epochs -- number of epochs (number of iterations)
    print_cost -- True to print the cost every 1000 epochs

    Returns:
    parameters -- python dictionary containing your updated parameters
    """

    L = len(layers_dims)                 # number of layers in the neural
                                             networks
    costs = []                           # to keep track of the cost
    t = 0                                # initializing the counter required
                                             for Adam update
    seed = 10                            # For grading purposes, so that your "
                                             random" minibatches are the same
                                             as ours
    m = X.shape[1]                       # number of training examples

    # Initialize parameters
    parameters = initialize_parameters(layers_dims)

    # Initialize the optimizer
    if optimizer == "gd":
        pass # no initialization required for gradient descent
    elif optimizer == "momentum":
        v = initialize_velocity(parameters)
    elif optimizer == "adam":
        v, s = initialize_adam(parameters)

    # Optimization loop
    for i in range(num_epochs):
        # Define the random minibatches. We increment the seed to reshuffle
                                                     differently the dataset
                                                    after each epoch
        seed = seed + 1
        minibatches = random_mini_batches(X, Y, mini_batch_size, seed)
        cost_total = 0

        for minibatch in minibatches:

            # Select a minibatch
            (minibatch_X, minibatch_Y) = minibatch

            # Forward propagation
            a3, caches = forward_propagation(minibatch_X, parameters)

            # Compute cost and add to the cost total
            cost_total += compute_cost(a3, minibatch_Y)

            # Backward propagation
            grads = backward_propagation(minibatch_X, minibatch_Y, caches)

            # Update parameters
            if optimizer == "gd":
                parameters = update_parameters_with_gd(parameters, grads,
                                                       learning_rate)
            elif optimizer == "momentum":
                parameters, v = update_parameters_with_momentum(parameters,
                                                        grads, v, beta,
                                                       learning_rate)
            elif optimizer == "adam":
                t = t + 1 # Adam counter
                parameters, v, s = update_parameters_with_adam(parameters,
                                                       grads, v, s, t,
                                                       learning_rate, beta1,
                                                        beta2,  epsilon)

        cost_avg = cost_total / m

        # Print the cost every 1000 epoch
        if print_cost and i % 1000 == 0:
            print ("Cost after epoch %i: %f" %(i, cost_avg))
        if print_cost and i % 100 == 0:
            costs.append(cost_avg)

    # plot the cost
    plt.plot(costs)
    plt.ylabel('cost')
    plt.xlabel('epochs (per 100)')
    plt.title("Learning rate = " + str(learning_rate))
    plt.show()

    return parameters
```

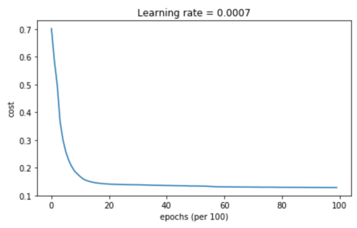
The following figure shows how the optimization progresses in terms of cost reduction.



(a) Gradient Descent



(b) Momentum



(c) Adam

I have to say the performance of Adam is pretty amazing.

5 Hyperparameters Tuning

There are many parameters needed to be tuned in our model. Based on the importance of those parameters, we make a ranking as follows:

- learning rate α
- gradient momentum β, number of hidden units, and mini-batch size
- number of layers, learning rate decaying, and parameters for Adam optimization

Since the combination of those parameters will determine the performance of your model, it is better to randomize our parameters before we training our model. In addition, we also need pay attention on the scale of parameters. To get the random parameters uniformly, we need generate our data on the log scale. Here is the implementation code in python

```python
r = -4 * np.random.rand()   # r in [-4, 0]
alpha = 10**r   # 10^-4...10^0
```

To organize our hyperparameters tuning efficiently, we need a framework. There are two ways:

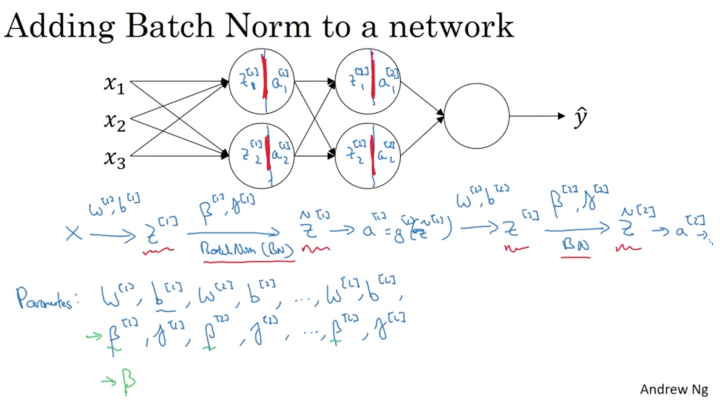
- babysitting your model
- training many models in parallel

You need consider your computational cost when you chose different hyperparameters searching process.

In the optimization section, we have shown that normization of input would speed up our training process. But inside the model, we did not do further normization for different layers. Batch normalization will do this task. It will not only normalize the the input, but also the activation values for each layer. However, we don't want to our neural network has zero mean in each layer. Therefore, we adopt our activation value based on the following formula:

$$z_{norm} = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
$$\hat{z} = \gamma z_{norm} + \beta$$

where γ and β are learnable parameters. The following figures presents the how batch normalization works in neural network model.



Why does batch normalization work?

- it can avoid the covariate shift, and our model could cope with different datasets generally
- it makes our channel of neural network become more neat (or regulated), so our model could learn parameters well or efficiently.

One way of understanding batch normalization is to regard it as the regularization.

6 TensorFlow

Writing and running programs in TensorFlow has the following steps:

- Create Tensors (variables) that are not yet executed/evaluated.
- Write operations between those Tensors.
- Initialize your Tensors.
- Create a Session.
- Run the Session. This will run the operations you'd written above.

For instance,

```
a = tf.constant(2)
b = tf.constant(10)
c = tf.multiply(a, b)
print(c)
```

As expected, you will not see 20! You got a tensor saying that the result is a tensor that does not have the shape attribute, and is of type "int32". All you did was put in the 'computation graph', but you have not run this computation yet. In order to actually multiply the two numbers, you will have to create a session and run it.

```
sess = tf.Session()
print(sess.run(c))
```

Next, you will also have to know about placeholders. A placeholder is an object whose value you can specify only later. To specify values for a placeholder, you can pass in values by using a "feed dictionary"(`feed_dict variable`). Below, we created a placeholder for x. This allows us to pass in a number later when we run the session.

```
# Change the value of x in the feed_dict

x = tf.placeholder(tf.int64, name = 'x')
print(sess.run(2 * x, feed_dix = {x : 3}))
sess.close()
```

Now, we use TensorFlow to calculate a linear function.

```
def linear_function():
  """
  Implements a linear function:
    Initialize X to be a random tensor of shape(3, 1)
    Initialize W to be a random tensor of shape(4, 3)
    Initialize b to be a random tensor of shape(4, 1)
  Returns:
    results -- runs the session for Y = WX + b
  """

  np.random.seed(1)

  X = tf.constant(np.random.randn(3, 1), name = 'X')
  W = tf.constant(np.random.randn(4, 3), name = 'W')
  b = tf.constant(np.random.randn(4, 1), name = 'b')
  Y = tf.add(tf.matmul(W, X), b)

  sess = tf.Session()
  result  = sess.run(Y)

  sess.close()

  return result
```

Now, we use TensorFlow to create a sigmoid function.

```
def sigmoid(z):
  """
  Compute the sigmoid of z

  Arguments: z -- input values, scalar or vector
  Returns: results -- the sigmoid of z
  """

  x = tf.placeholder(tf.float32, name = 'x')

  sigmoid = tf.sigmoid(x)

  with tf.Session() as sess:
    result = sess.run(sigmoid, feed_dict = {x : z})
  return result
```

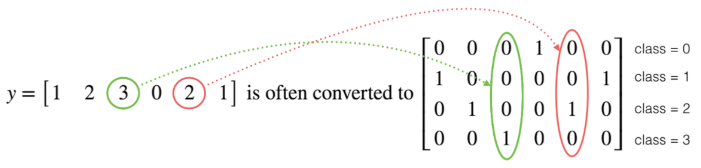
To summarize, you now know how to:

1. create a constant or a placeholder
2. specify the computation graph corresponding to operations you want to compute
3. create the session
4. run the session, using a feed dictionary if necessary to specify placeholder variables' values.

Now, we use TensorFlow to create a cost function

```
def cost(logits, labels):
  """
  Computes the cost using the sigmoid cross entropy
  Arguments:
  logits -- vector containing z, output of the last linear unit
  labels -- vector of labels y (1 or 0)

  Returns:
  cost -- runs the session of the cost
  """

  z = tf.placeholder(tf.float32, name = 'z')
  y = tf.placeholder(tf.float32, name = 'y')

  cost = tf.nn.sigmoid_cross_entropy_with_logits(logits=z, labels = y)   #
                                                  be careful, you need put the name of
                                                  arguments
  sess = tf.Session()
  cost = sess.run(cost, feed_dict = {z: logits, y: labels})
  sess.close()

  return cost
```

Many times in deep learning you will have a vector with numbers from 0 to C-1, where C is the number of classes. If C is for example 4, then you might have the following vector which you will need to convert as follows:



```
def one_hot_matrix(labels, C):
  """
  Create a matrix where could do one-hot encoding
  Arguments:
    labels -- vector containing the labels
    C -- number of classes, the depth of the one hot dimension
  Returns:
```

```
  one_hot  -- one hot matrix
  """

  C = tf.constant(C, name = 'C')
  one_hot_matrix = tf.one.hot(labels, C, axis=0)

  sess = tf.Session()
  one_hot = sess.run(one_hot_matrix)
  sess.close()

  return one_hot
```

```
def ones(shape):
  ones = tf.ones(shape)

  sess = tf.Session()
  ones = sess.run(ones)
  sess.close()

  return ones
```

6.1 Case Study with TensorFlow

```
import math
import numpy as np
import h5py
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.python.framework import ops
from tf_utils import load_dataset, random_mini_batches, convert_to_one_hot,
                                        predict

%matplotlib inline
np.random.seed(1)

# Loading the dataset
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes =
                                       load_dataset()

# Example of a picture
index = 9
plt.imshow(X_train_orig[index])
print ("y = " + str(np.squeeze(Y_train_orig[:, index])))

# Flatten the training and test images
X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T
X_test_flatten = X_test_orig.reshape(X_test_orig.shape[0], -1).T
# Normalize image vectors
X_train = X_train_flatten/255.
X_test = X_test_flatten/255.
# Convert training and test labels to one hot matrices
Y_train = convert_to_one_hot(Y_train_orig, 6)
Y_test = convert_to_one_hot(Y_test_orig, 6)

print ("number of training examples = " + str(X_train.shape[1]))
print ("number of test examples = " + str(X_test.shape[1]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))


def create_placeholders(n_x, n_y):
    """
    Creates the placeholders for the tensorflow session.

    Arguments:
    n_x -- scalar, size of an image vector (num_px * num_px = 64 * 64 * 3 =
                                        12288)
    n_y -- scalar, number of classes (from 0 to 5, so -> 6)

    Returns:
    X -- placeholder for the data input, of shape [n_x, None] and dtype "tf
                                      .float32"
    Y -- placeholder for the input labels, of shape [n_y, None] and dtype "
                                      tf.float32"

    Tips:
    - You will use None because it let's us be flexible on the number of
                                      examples you will for the
                                      placeholders.
      In fact, the number of examples during test/train is different.
    """

    ### START CODE HERE ### (approx. 2 lines)
    X = tf.placeholder(tf.float32, shape=(n_x, None), name = 'X')
    Y = tf.placeholder(tf.float32, shape=(n_y, None), name = 'Y')
    ### END CODE HERE ###

    return X, Y


# GRADED FUNCTION: initialize_parameters

def initialize_parameters():
    """
    Initializes parameters to build a neural network with tensorflow. The
                                      shapes are:
                        W1 : [25, 12288]
                        b1 : [25, 1]
                        W2 : [12, 25]
                        b2 : [12, 1]
                        W3 : [6, 12]
                        b3 : [6, 1]

    Returns:
    parameters -- a dictionary of tensors containing W1, b1, W2, b2, W3, b3
    """

    tf.set_random_seed(1)                      # so that your "random" numbers
                                        match ours

    ### START CODE HERE ### (approx. 6 lines of code)
    W1 = tf.get_variable('W1', [25, 12288], initializer=tf.contrib.layers.
                                      xavier_initializer(seed=1))
    b1 = tf.get_variable('b1', [25, 1], initializer=tf.zeros_initializer())
    W2 = tf.get_variable('W2', [12, 25], initializer=tf.contrib.layers.
                                      xavier_initializer(seed=1))
    b2 = tf.get_variable('b2', [12, 1], initializer=tf.zeros_initializer())
    W3 = tf.get_variable('W3', [6, 12], initializer=tf.contrib.layers.
                                      xavier_initializer(seed=1))
    b3 = tf.get_variable('b3', [6, 1], initializer=tf.zeros_initializer())
    ### END CODE HERE ###

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2,
                  "W3": W3,
                  "b3": b3}

    return parameters


# GRADED FUNCTION: forward_propagation

def forward_propagation(X, parameters):
    """
    Implements the forward propagation for the model: LINEAR -> RELU ->
                                      LINEAR -> RELU -> LINEAR ->
                                      SOFTMAX

    Arguments:
    X -- input dataset placeholder, of shape (input size, number of
                                      examples)
    parameters -- python dictionary containing your parameters "W1", "b1",
                                      "W2", "b2", "W3", "b3"
                  the shapes are given in initialize_parameters

    Returns:
    Z3 -- the output of the last LINEAR unit
    """

    # Retrieve the parameters from the dictionary "parameters"
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    W3 = parameters['W3']
    b3 = parameters['b3']

    ### START CODE HERE ### (approx. 5 lines)              # Numpy
                                      Equivalents:
    Z1 = tf.add(tf.matmul(W1, X), b1)  # Z1 = np.dot(W1, X) + b1
    A1 = tf.nn.relu(Z1)  # A1 = relu(Z1)
    Z2 = tf.add(tf.matmul(W2, A1), b2)  # Z2 = np.dot(W2, A1) + b2
    A2 = tf.nn.relu(Z2)  # A2 = relu(Z2)
    Z3 = tf.add(tf.matmul(W3, A2), b3)  # Z3 = np.dot(W3, A2) + b3
    ### END CODE HERE ###

    return Z3
```

```
# GRADED FUNCTION: compute_cost

def compute_cost(Z3, Y):
    """
    Computes the cost

    Arguments:
    Z3 -- output of forward propagation (output of the last LINEAR unit),
                                         of shape (6, number of examples)
    Y -- "true" labels vector placeholder, same shape as Z3

    Returns:
    cost - Tensor of the cost function
    """

    # to fit the tensorflow requirement for tf.nn.
                                         softmax_cross_entropy_with_logits
                                         (...,...)
    logits = tf.transpose(Z3)
    labels = tf.transpose(Y)

    ### START CODE HERE ### (1 line of code)
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=
                                         logits, labels=labels))
    ### END CODE HERE ###

    return cost


def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.0001,
          num_epochs = 1500, minibatch_size = 32, print_cost = True):
    """
    Implements a three-layer tensorflow neural network: LINEAR->RELU->
                                         LINEAR->RELU->LINEAR->SOFTMAX.

    Arguments:
    X_train -- training set, of shape (input size = 12288, number of
                                         training examples = 1080)
    Y_train -- test set, of shape (output size = 6, number of training
                                         examples = 1080)
    X_test -- training set, of shape (input size = 12288, number of
                                         training examples = 120)
    Y_test -- test set, of shape (output size = 6, number of test examples
                                         = 120)
    learning_rate -- learning rate of the optimization
    num_epochs -- number of epochs of the optimization loop
    minibatch_size -- size of a minibatch
    print_cost -- True to print the cost every 100 epochs

    Returns:
    parameters -- parameters learnt by the model. They can then be used to
                                         predict.
    """

    ops.reset_default_graph()                             # to be able to rerun
                                          the model without overwriting tf
```

```
                                       variables
tf.set_random_seed(1)                             # to keep consistent
                                      results
seed = 3                                          # to keep consistent
                                      results
(n_x, m) = X_train.shape                          # (n_x: input size, m
                                       : number of examples in the
                                      train set)
n_y = Y_train.shape[0]                            # n_y : output size
costs = []                                        # To keep track of
                                      the cost

# Create Placeholders of shape (n_x, n_y)
### START CODE HERE ### (1 line)
X, Y = create_placeholders(n_x, n_y)
### END CODE HERE ###

# Initialize parameters
### START CODE HERE ### (1 line)
parameters = initialize_parameters()
### END CODE HERE ###

# Forward propagation: Build the forward propagation in the tensorflow
                                      graph
### START CODE HERE ### (1 line)
Z3 = forward_propagation(X_train, parameters)
### END CODE HERE ###

# Cost function: Add cost function to tensorflow graph
### START CODE HERE ### (1 line)
cost = compute_cost(Z3, Y_train)
### END CODE HERE ###

# Backpropagation: Define the tensorflow optimizer. Use an
                                      AdamOptimizer.
### START CODE HERE ### (1 line)
optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate).
                                      minimize(cost)
### END CODE HERE ###

# Initialize all the variables
init = tf.global_variables_initializer()

# Start the session to compute the tensorflow graph
with tf.Session() as sess:

    # Run the initialization
    sess.run(init)

    # Do the training loop
    for epoch in range(num_epochs):

        epoch_cost = 0.                       # Defines a cost related
                                              to an epoch
        num_minibatches = int(m / minibatch_size) # number of
                                              minibatches of size
                                              minibatch_size in the
                                              train set
```

```
            seed = seed + 1
            minibatches = random_mini_batches(X_train, Y_train,
                                               minibatch_size, seed)

            for minibatch in minibatches:

                # Select a minibatch
                (minibatch_X, minibatch_Y) = minibatch

                # IMPORTANT: The line that runs the graph on a minibatch.
                # Run the session to execute the "optimizer" and the "cost
                                                  ", the feedict should
                                                   contain a minibatch
                                                  for (X,Y).
                ### START CODE HERE ### (1 line)
                _ , minibatch_cost = sess.run([optimizer, cost], feed_dict=
                                                  {X: minibatch_X, Y:
                                                  minibatch_Y})
                ### END CODE HERE ###

                epoch_cost += minibatch_cost / minibatch_size

            # Print the cost every epoch
            if print_cost == True and epoch % 100 == 0:
                print ("Cost after epoch %i: %f" % (epoch, epoch_cost))
            if print_cost == True and epoch % 5 == 0:
                costs.append(epoch_cost)

        # plot the cost
        plt.plot(np.squeeze(costs))
        plt.ylabel('cost')
        plt.xlabel('iterations (per fives)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()

        # lets save the parameters in a variable
        parameters = sess.run(parameters)
        print ("Parameters have been trained!")

        # Calculate the correct predictions
        correct_prediction = tf.equal(tf.argmax(Z3), tf.argmax(Y))

        # Calculate accuracy on the test set
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

        print ("Train Accuracy:", accuracy.eval({X: X_train, Y: Y_train}))
        print ("Test Accuracy:", accuracy.eval({X: X_test, Y: Y_test}))

        return parameters
```