Introduction to Deep Learning

Eugene Charniak

Abhijit Kumar Baruah,
Advisor: Dr. Dominique Guillot.

Department of Mathematical Sciences,
University of Delaware.

20 March 2019

Outline

Review

- Perceptron classifies inputs by finding the dot product of feature vector and weight vector and passing that number into a step function.

Review

- Perceptron classifies inputs by finding the dot product of feature vector and weight vector and passing that number into a step function.
- The Perceptron Learning Rule:
  - Predicts an output based on the current weights and inputs.
  - Compares it to the expected output, or label.
  - Update its weights, if the prediction != the label.
  - Iterate until the epoch threshold has been reached.

Review

- Perceptron classifies inputs by finding the dot product of feature vector and weight vector and passing that number into a step function.
- The Perceptron Learning Rule:
  - Predicts an output based on the current weights and inputs.
  - Compares it to the expected output, or label.
  - Update its weights, if the prediction != the label.
  - Iterate until the epoch threshold has been reached.
- To update the weights during each iteration, it:
  - Finds the error by subtracting the prediction from the label.
  - Multiplies the error and the learning rate.
  - Multiplies the result to the inputs.
  - Adds the resulting vector to the weight vector.

Review

- Perceptron classifies inputs by finding the dot product of feature vector and weight vector and passing that number into a step function.
- The Perceptron Learning Rule:
  - Predicts an output based on the current weights and inputs.
  - Compares it to the expected output, or label.
  - Update its weights, if the prediction != the label.
  - Iterate until the epoch threshold has been reached.
- To update the weights during each iteration, it:
  - Finds the error by subtracting the prediction from the label.
  - Multiplies the error and the learning rate.
  - Multiplies the result to the inputs.
  - Adds the resulting vector to the weight vector.
- For a linearly separable data, the perceptron algorithm guarantees a separating hyperplane.

Review

Theorem 1 (Mistake Bound Theorem (Novikoff 1962, Block 1962))

Let $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\}$ *be a sequence of training examples such that for all* i*, the feature vector* $\mathbf{x}_i \in \mathbb{R}^n, ||\mathbf{x}_i|| \leq R$ *and the label* $y_i \in \{-1, 1\}$*. Suppose there exists a unique vector* $\mathbf{u} \in \mathbb{R}^n$ *such that for some* $\gamma \in \mathbb{R}$ *and* $\gamma > 0$ *we have* $y_i(\mathbf{u}^T \mathbf{x}_i) \geq \gamma$*. Then, the perceptron algorithm will make at most* $(R/\gamma)^2$ *mistakes on the training sequence.*

Remark: The number of mistakes refers to the number of updates the perceptron algorithm makes in a training sequence.

Review

Theorem 1 (Mistake Bound Theorem (Novikoff 1962, Block 1962))

Let $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\}$ *be a sequence of training examples such that for all* i*, the feature vector* $\mathbf{x}_i \in \mathbb{R}^n, ||\mathbf{x}_i|| \leq R$ *and the label* $y_i \in \{-1, 1\}$*. Suppose there exists a unique vector* $\mathbf{u} \in \mathbb{R}^n$ *such that for some* $\gamma \in \mathbb{R}$ *and* $\gamma > 0$ *we have* $y_i(\mathbf{u}^T \mathbf{x}_i) \geq \gamma$*. Then, the perceptron algorithm will make at most* $(R/\gamma)^2$ *mistakes on the training sequence.*

Remark: The number of mistakes refers to the number of updates the perceptron algorithm makes in a training sequence.

- The loss function associated with the Perceptron is the Zero-One Loss.

Review

Theorem 1 (Mistake Bound Theorem (Novikoff 1962, Block 1962))

Let $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\}$ be a sequence of training examples such that for all i, the feature vector $\mathbf{x}_i \in \mathbb{R}^n, ||\mathbf{x}_i|| \leq R$ and the label $y_i \in \{-1, 1\}$. Suppose there exists a unique vector $\mathbf{u} \in \mathbb{R}^n$ such that for some $\gamma \in \mathbb{R}$ and $\gamma > 0$ we have $y_i(\mathbf{u}^T\mathbf{x}_i) \geq \gamma$. Then, the perceptron algorithm will make at most $(R/\gamma)^2$ mistakes on the training sequence.

Remark: The number of mistakes refers to the number of updates the perceptron algorithm makes in a training sequence.

- The loss function associated with the Perceptron is the Zero-One Loss.
- Zero-One Loss does not work with the Gradient Descent learning algorithm.

Review

Theorem 1 (Mistake Bound Theorem (Novikoff 1962, Block 1962))

Let $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\}$ *be a sequence of training examples such that for all* i*, the feature vector* $\mathbf{x}_i \in \mathbb{R}^n, ||\mathbf{x}_i|| \leq R$ *and the label* $y_i \in \{-1, 1\}$*. Suppose there exists a unique vector* $\mathbf{u} \in \mathbb{R}^n$ *such that for some* $\gamma \in \mathbb{R}$ *and* $\gamma > 0$ *we have* $y_i(\mathbf{u}^T \mathbf{x}_i) \geq \gamma$*. Then, the perceptron algorithm will make at most* $(R/\gamma)^2$ *mistakes on the training sequence.*

Remark: The number of mistakes refers to the number of updates the perceptron algorithm makes in a training sequence.

- The loss function associated with the Perceptron is the Zero-One Loss.
- Zero-One Loss does not work with the Gradient Descent learning algorithm.
- Cross-entropy loss function is: $X(\Phi, x) = -\ln p_\Phi(a_x)$.

Derivatives

- This is what we have so far.

Derivatives

- This is what we have so far.

$$l_j = b_j + \mathbf{x}.\mathbf{w}_j \tag{1}$$

$$p(a) = \sigma_a(l) = \frac{e^{l_a}}{\sum_i e^{l_i}} \tag{2}$$

$$X(\Phi, x) = -\ln p(a) \tag{3}$$

Derivatives

- This is what we have so far.

$$l_j = b_j + \mathbf{x}.\mathbf{w}_j \tag{1}$$

$$p(a) = \sigma_a(l) = \frac{e^{l_a}}{\sum_i e^{l_i}} \tag{2}$$

$$X(\Phi, x) = -\ln p(a) \tag{3}$$

- This process of going from input to the loss is called the *forward pass* of the learning algorithm.

Derivatives

- This is what we have so far.

$$l_j = b_j + \mathbf{x}.\mathbf{w}_j \tag{1}$$

$$p(a) = \sigma_a(l) = \frac{e^{l_a}}{\sum_i e^{l_i}} \tag{2}$$

$$X(\Phi, x) = -\ln p(a) \tag{3}$$

- This process of going from input to the loss is called the *forward pass* of the learning algorithm.
- This computes the values to be used in the *backward pass* - the weight adjustment pass.

Derivatives

- This is what we have so far.

$$l_j = b_j + \mathbf{x}.\mathbf{w}_j \quad (1)$$

$$p(a) = \sigma_a(l) = \frac{e^{l_a}}{\sum_i e^{l_i}} \quad (2)$$

$$X(\Phi, x) = -\ln p(a) \quad (3)$$

- This process of going from input to the loss is called the *forward pass* of the learning algorithm.
- This computes the values to be used in the *backward pass* - the weight adjustment pass.
- One method is the *stochastic gradient descent*.

Gradient Descent

- The name comes from looking at the slope of the loss function (its gradient) and then having the system lower its loss (descend) by following the gradient.

Gradient Descent

- The name comes from looking at the slope of the loss function (its gradient) and then having the system lower its loss (descend) by following the gradient.
- The learning method as a whole is known as *back propagation*.

Gradient Descent

- The name comes from looking at the slope of the loss function (its gradient) and then having the system lower its loss (descend) by following the gradient.
- The learning method as a whole is known as *back propagation*.
- We start by looking at the simplest case of gradient estimation, that for one of the biases, b_j and error induced by a single training example.

Gradient Descent

- The name comes from looking at the slope of the loss function (its gradient) and then having the system lower its loss (descend) by following the gradient.
- The learning method as a whole is known as *back propagation*.
- We start by looking at the simplest case of gradient estimation, that for one of the biases, b_j and error induced by a single training example.
- $\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j} \frac{\partial X(\Phi)}{\partial l_j}$

Gradient Descent

- The name comes from looking at the slope of the loss function (its gradient) and then having the system lower its loss (descend) by following the gradient.
- The learning method as a whole is known as *back propagation*.
- We start by looking at the simplest case of gradient estimation, that for one of the biases, b_j and error induced by a single training example.
- $\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j} \frac{\partial X(\Phi)}{\partial l_j}$
- $\frac{\partial X(\Phi)}{\partial l_j} = \frac{\partial p_a}{\partial l_j} \frac{X(\Phi)}{\partial p_a}$, where p_i is the probability assigned to class i by the network.

Gradient Descent

- The name comes from looking at the slope of the loss function (its gradient) and then having the system lower its loss (descend) by following the gradient.
- The learning method as a whole is known as *back propagation*.
- We start by looking at the simplest case of gradient estimation, that for one of the biases, b_j and error induced by a single training example.
- $\frac{\partial X(\Phi)}{\partial b_j} = \frac{\partial l_j}{\partial b_j} \frac{\partial X(\Phi)}{\partial l_j}$
- $\frac{\partial X(\Phi)}{\partial l_j} = \frac{\partial p_a}{\partial l_j} \frac{X(\Phi)}{\partial p_a}$, where p_i is the probability assigned to class i by the network.
- After some derivation, we have

$$\frac{\partial X(\Phi)}{\partial l_j} = -\frac{1}{p_a} \begin{cases} (1 - p_j)p_a & \text{if } a = j \\ -p_j p_a & \text{it } a \neq j \end{cases} = \begin{cases} -(1 - p_j) & \text{if } a = j \\ p_j & \text{it } a \neq j \end{cases} \tag{4}$$

- Rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) \text{ if } a = j \\ -p_j \text{ if } a \neq j \end{cases} \tag{5}$$

- Rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) \text{ if } a = j \\ -p_j \text{ if } a \neq j \end{cases} \tag{5}$$

- We have similarly the equation for changing weight parameters:

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}} \frac{\partial X(\Phi)}{\partial l_j} \tag{6}$$

- Rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) \text{ if } a = j \\ -p_j \text{ if } a \neq j \end{cases} \tag{5}$$

- We have similarly the equation for changing weight parameters:

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}} \frac{\partial X(\Phi)}{\partial l_j} \tag{6}$$

- After some calculation we have our equation for weight updates as:

$$\Delta w_{i,j} = -\mathcal{L} x_i \frac{\partial X(\Phi)}{\partial l_j} \tag{7}$$

- Rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) \text{ if } a = j \\ -p_j \text{ if } a \neq j \end{cases} \tag{5}$$

- We have similarly the equation for changing weight parameters:

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}} \frac{\partial X(\Phi)}{\partial l_j} \tag{6}$$

- After some calculation we have our equation for weight updates as:

$$\Delta w_{i,j} = -\mathcal{L} x_i \frac{\partial X(\Phi)}{\partial l_j} \tag{7}$$

- We have derived how the parameters of our model should change in light of a single training example.

- Rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) \text{ if } a = j \\ -p_j \text{ if } a \neq j \end{cases} \tag{5}$$

- We have similarly the equation for changing weight parameters:

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}} \frac{\partial X(\Phi)}{\partial l_j} \tag{6}$$

- After some calculation we have our equation for weight updates as:

$$\Delta w_{i,j} = -\mathcal{L} x_i \frac{\partial X(\Phi)}{\partial l_j} \tag{7}$$

- We have derived how the parameters of our model should change in light of a single training example.
- The gradient descent algorithm now needs to pass though all the training examples.

- Rule for updating the NN bias parameters:

$$\Delta b_j = \mathcal{L} \begin{cases} (1 - p_j) \text{ if } a = j \\ -p_j \text{ if } a \neq j \end{cases} \tag{5}$$

- We have similarly the equation for changing weight parameters:

$$\frac{\partial X(\Phi)}{\partial w_{i,j}} = \frac{\partial l_j}{\partial w_{i,j}} \frac{\partial X(\Phi)}{\partial l_j} \tag{6}$$

- After some calculation we have our equation for weight updates as:

$$\Delta w_{i,j} = -\mathcal{L} x_i \frac{\partial X(\Phi)}{\partial l_j} \tag{7}$$

- We have derived how the parameters of our model should change in light of a single training example.
- The gradient descent algorithm now needs to pass though all the training examples.
- Modify each parameter by the sum of the changes from the individual examples.

Stochastic Gradient Descent

- Because gradient descent algorithm is slow, in practice we update parameters every m (batch size) examples, for m much less than the size of the training set. This process is called *stochastic gradient descent*.

Stochastic Gradient Descent

- Because gradient descent algorithm is slow, in practice we update parameters every m (batch size) examples, for m much less than the size of the training set. This process is called *stochastic gradient descent*.
- The smaller the batch size, the smaller the learning rate $\mathcal{L}$ should be.

Stochastic Gradient Descent

- Because gradient descent algorithm is slow, in practice we update parameters every m (batch size) examples, for m much less than the size of the training set. This process is called *stochastic gradient descent*.
- The smaller the batch size, the smaller the learning rate $\mathcal{L}$ should be.
- Idea: Any one example is going to push the weights toward classifying that example correctly at the expense of the others.

Pseudocode for simple feed-forward stochastic gradient descent algorithm

1. Set b_j randomly but close to zero.
2. Set weights $w_{i,j}$ similarly.
3. Until development accuracy stops increasing
   (a) for each training example k in batches of m examples
      i. do the forward pass using equations (4),(5) and (6).
      ii. do the backward pass using equations (7),(8) and (10).
      iii. every m examples, modify all Φ with the summand updates
   (b) compute the accuracy of the model by running the forward pass on all examples in the development set.
4. output the Φ from the iteration before the decrease in development accuracy.

TensorFlow

Preliminaries

- Tensorflow is an open-source programming language developed by Google.

Preliminaries

- Tensorflow is an open-source programming language developed by Google.
- It is easily trainable on CPU as well as GPU for distributed computing.

Preliminaries

- Tensorflow is an open-source programming language developed by Google.
- It is easily trainable on CPU as well as GPU for distributed computing.
- Very efficient on large scale systems.

Preliminaries

- Tensorflow is an open-source programming language developed by Google.
- It is easily trainable on CPU as well as GPU for distributed computing.
- Very efficient on large scale systems.
- The library provides distribution functions.

Preliminaries

- Tensorflow is an open-source programming language developed by Google.
- It is easily trainable on CPU as well as GPU for distributed computing.
- Very efficient on large scale systems.
- The library provides distribution functions.
- On Anaconda Python 3.5 (which you can find here: `https://www.continuum.io/`) run *conda install tensorflow* from a command line interface to install TensorFlow using Conda packages.

Preliminaries

- Tensorflow is an open-source programming language developed by Google.
- It is easily trainable on CPU as well as GPU for distributed computing.
- Very efficient on large scale systems.
- The library provides distribution functions.
- On Anaconda Python 3.5 (which you can find here: `https://www.continuum.io/`) run *conda install tensorflow* from a command line interface to install TensorFlow using Conda packages.
- Traditional first program:

```
import tensorflow as tf
x=tf.constant("Hello World")
sess=tf.Session()
print(sess.run(x))
```

Preliminaries

- Tensorflow is an open-source programming language developed by Google.
- It is easily trainable on CPU as well as GPU for distributed computing.
- Very efficient on large scale systems.
- The library provides distribution functions.
- On Anaconda Python 3.5 (which you can find here: `https://www.continuum.io/`) run *conda install tensorflow* from a command line interface to install TensorFlow using Conda packages.
- Traditional first program:

```python
import tensorflow as tf
x=tf.constant("Hello World")
sess=tf.Session()
print(sess.run(x))
```

- This prints out "Hello World"

Preliminaries

- TF functions define a computation that is executed only when we call the **run** command.

Preliminaries

- TF functions define a computation that is executed only when we call the **run** command.
- TF function **Session** creates a session, associated with which is a graph defining a computation.

Preliminaries

- TF functions define a computation that is executed only when we call the **run** command.
- TF function **Session** creates a session, associated with which is a graph defining a computation.
- Commands like **constant** add elements to this computation.

Preliminaries

- TF functions define a computation that is executed only when we call the **run** command.
- TF function **Session** creates a session, associated with which is a graph defining a computation.
- Commands like **constant** add elements to this computation.
- The third line tells TF to evaluate the TF variable pointed to by **x** inside the graph associated with variable **sess**.

Preliminaries

- TF functions define a computation that is executed only when we call the **run** command.
- TF function **Session** creates a session, associated with which is a graph defining a computation.
- Commands like **constant** add elements to this computation.
- The third line tells TF to evaluate the TF variable pointed to by **x** inside the graph associated with variable **sess**.
- Contrast this behavior if the last line is replaced with **print(x)**.

TF Placeholder

- In **CODE 2**, **z** is a Python variable whose value is a TF placeholder.

TF Placeholder

- In **CODE 2**, **z** is a Python variable whose value is a TF placeholder.
- Like a formal variable in a programming language function

```
x = 2.0
def sillyAdd(z):
    return z+x
print(sillyAdd(3)) # Prints out 5.0
```

TF Placeholder

- In **CODE 2**, **z** is a Python variable whose value is a TF placeholder.
- Like a formal variable in a programming language function

```python
x = 2.0
def sillyAdd(z):
    return z+x
print(sillyAdd(3)) # Prints out 5.0
```

- 'z' is the name of the sillyAdd function's argument.

TF Placeholder

- In **CODE 2**, **z** is a Python variable whose value is a TF placeholder.
- Like a formal variable in a programming language function

```
x = 2.0
def sillyAdd(z):
    return z+x
print(sillyAdd(3)) # Prints out 5.0
```

- 'z' is the name of the sillyAdd function's argument.
- TF placeholders are given value in a similar manner.

```
print(sess.run(comp, feed_dict={z:3.0}))
```

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.
- First 'sess.run' prints out the sum of 2.0 and 3.0.

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.
- First 'sess.run' prints out the sum of 2.0 and 3.0.
- Second 'sess.run' prints out the sum of 2.0 and 16.0.

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.
- First 'sess.run' prints out the sum of 2.0 and 3.0.
- Second 'sess.run' prints out the sum of 2.0 and 16.0.
- Third 'sess.run' does not require a placeholder value for computation.

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.
- First 'sess.run' prints out the sum of 2.0 and 3.0.
- Second 'sess.run' prints out the sum of 2.0 and 16.0.
- Third 'sess.run' does not require a placeholder value for computation.
- Fourth 'sess.run' requires a value for computation which is not supplied, hence an error.

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.
- First 'sess.run' prints out the sum of 2.0 and 3.0.
- Second 'sess.run' prints out the sum of 2.0 and 16.0.
- Third 'sess.run' does not require a placeholder value for computation.
- Fourth 'sess.run' requires a value for computation which is not supplied, hence an error.
- Placeholders may be simple scalars or multidimensional tensors.

TF Placeholder

- We have a named argument of **run** which takes as possible values Python dictionaries.
- First 'sess.run' prints out the sum of 2.0 and 3.0.
- Second 'sess.run' prints out the sum of 2.0 and 16.0.
- Third 'sess.run' does not require a placeholder value for computation.
- Fourth 'sess.run' requires a value for computation which is not supplied, hence an error.
- Placeholders may be simple scalars or multidimensional tensors.
- Example: For Mnist digit recognition program, the image defined as placeholder will be of type 'float32' and shape $[28, 28]$ or $[784]$, depending on whether the program was handed a two- or one-dimensional Python list.

```
img=tf.placeholder(tf.float32,shape=[28,28])
```

Another TF data structure

- NN models are defined by their parameters and the program's architecture.

Another TF data structure

- NN models are defined by their parameters and the program's architecture.
- The parameters are initialized randomly and the NN modifies them to minimize the loss on training data.

Another TF data structure

- NN models are defined by their parameters and the program's architecture.
- The parameters are initialized randomly and the NN modifies them to minimize the loss on training data.
- Three stages to create TF parameter:
    1. Create a tensor with initial values.
    2. Turn the tensor into a variable (TF parameters).
    3. Initializing the variables/TF-parameter.

Another TF data structure

- NN models are defined by their parameters and the program's architecture.
- The parameters are initialized randomly and the NN modifies them to minimize the loss on training data.
- Three stages to create TF parameter:
  1. Create a tensor with initial values.
  2. Turn the tensor into a variable (TF parameters).
  3. Initializing the variables/TF-parameter.
- Lets create the parameters we need for the feed-forward Mnist pseudo code (as seen in the first talk).

Creating TF parameters for Mnist pseudocode

- First the bias terms 'b', then the weights 'W'

```
bt = tf.random_normal([10], stddev=.1)
b = tf.Variable(bt)
W = tf.Variable(tf.random_normal([784,10],stddev=.
sess=tf.Session()
sess.run(tf.initialize_all_variables())
print(sess.run(b))
```

- First line creates a tensor of shape [10] with ten values randomly generated from a normal distribution with standard deviation 0.1.

Creating TF parameters for Mnist pseudocode

- First the bias terms 'b', then the weights 'W'

```
bt = tf.random_normal([10], stddev=.1)
b = tf.Variable(bt)
W = tf.Variable(tf.random_normal([784,10],stddev=.
sess=tf.Session()
sess.run(tf.initialize_all_variables())
print(sess.run(b))
```

- First line creates a tensor of shape [10] with ten values randomly generated from a normal distribution with standard deviation 0.1.
- Second line takes 'bt' and creates a piece of the TF graph that will create a variable with the same shape and values.

Creating TF parameters for Mnist pseudocode

- First the bias terms 'b', then the weights 'W'

```
bt = tf.random_normal([10], stddev=.1)
b = tf.Variable(bt)
W = tf.Variable(tf.random_normal([784,10],stddev=.
sess=tf.Session()
sess.run(tf.initialize_all_variables())
print(sess.run(b))
```

- First line creates a tensor of shape [10] with ten values randomly generated from a normal distribution with standard deviation 0.1.
- Second line takes 'bt' and creates a piece of the TF graph that will create a variable with the same shape and values.
- Third line combines the above two events and creates variable 'W'.

Creating TF parameters for Mnist pseudocode

- First the bias terms '*b*', then the weights '*W*'

```
bt = tf.random_normal([10], stddev=.1)
b = tf.Variable(bt)
W = tf.Variable(tf.random_normal([784,10],stddev=.
sess=tf.Session()
sess.run(tf.initialize_all_variables())
print(sess.run(b))
```

- First line creates a tensor of shape [10] with ten values randomly generated from a normal distribution with standard deviation 0.1.
- Second line takes 'bt' and creates a piece of the TF graph that will create a variable with the same shape and values.
- Third line combines the above two events and creates variable 'W'.
- Fifth line initializes 'b' and 'W' in the session before we can use them.

Reference

Eugene Charniak
Introduction to Deep Learning

Anaconda 3.5
https://www.continuum.io/

Tensorflow installation guide for Anaconda
https://www.anaconda.com/tensorflow-in-anaconda/

THANK YOU!