Weight Initialization

In this lesson, you'll learn how to find good initial weights for a neural network. Having good initial weights can place the neural network close to the optimal solution. This allows the neural network to come to the best solution quicker.

Testing Weights

Dataset

To see how different weights perform, we'll test on the same dataset and neural network. Let's go over the dataset and neural network.

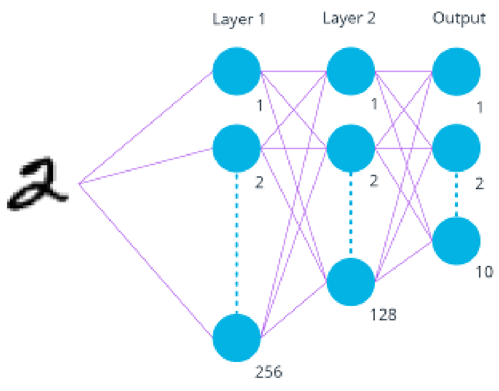
We'll be using the MNIST dataset to demonstrate the different initial weights. As a reminder, the MNIST dataset contains images of handwritten numbers, 0-9, with normalized input (0.0 - 1.0). Run the cell below to download and load the MNIST dataset.

In [2]:

```
%matplotlib inline

import tensorflow as tf
import helper

from tensorflow.examples.tutorials.mnist import input_data

print('Getting MNIST Dataset...')
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
print('Data Extracted.')
```

```
Getting MNIST Dataset...
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Data Extracted.
```

Neural Network



For the neural network, we'll test on a 3 layer neural network with ReLU activations and an Adam optimizer. The lessons you learn apply to other neural networks, including different activations and optimizers.

In [3]:

```
# Save the shapes of weights for each layer
layer_1_weight_shape = (mnist.train.images.shape[1], 256)
layer_2_weight_shape = (256, 128)
layer_3_weight_shape = (128, mnist.train.labels.shape[1])
```

Initialize Weights

Let's start looking at some initial weights.

All Zeros or Ones

If you follow the principle of Occam's razor, you might think setting all the weights to 0 or 1 would be the best solution. This is not the case.

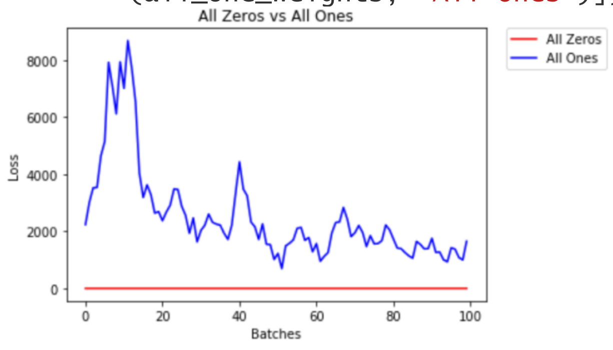
With every weight the same, all the neurons at each layer are producing the same output. This makes it hard to decide which weights to adjust.

Let's compare the loss with all ones and all zero weights using `helper.compare_init_weights`. This function will run two different initial weights on the neural network above for 2 epochs. It will plot the loss for the first 100 batches and print out stats after the 2 epochs (~860 batches). We plot the first 100 batches to better judge which weights performed better at the start.

Run the cell below to see the difference between weights of all zeros against all ones.

In [4]:

```
all_zero_weights = [
    tf.Variable(tf.zeros(layer_1_weight_shape)),
    tf.Variable(tf.zeros(layer_2_weight_shape)),
    tf.Variable(tf.zeros(layer_3_weight_shape))
]

all_one_weights = [
    tf.Variable(tf.ones(layer_1_weight_shape)),
    tf.Variable(tf.ones(layer_2_weight_shape)),
    tf.Variable(tf.ones(layer_3_weight_shape))
]

helper.compare_init_weights(
    mnist,
    'All Zeros vs All Ones',
    [
        (all_zero_weights, 'All Zeros'),
        (all_one_weights, 'All Ones')])
```



```
After 858 Batches (2 Epochs):
Validation Accuracy
   11.260% -- All Zeros
    9.900% -- All Ones
Loss
    2.300  -- All Zeros
  372.644  -- All Ones
```

As you can see the accuracy is close to guessing for both zeros and ones, around 10%.

The neural network is having a hard time determining which weights need to be changed, since the neurons have the same output for each layer. To avoid neurons with the same output, let's use unique weights. We can also randomly select these weights to avoid being stuck in a local minimum for each run.

A good solution for getting these random weights is to sample from a uniform distribution.

Uniform Distribution

A uniform distribution has the equal probability of picking any number from a set of numbers. We'll be picking from a continous distribution, so the chance of picking the same number is low. We'll use TensorFlow's `tf.random_uniform` function to pick random numbers from a uniform distribution.

tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)

Outputs random values from a uniform distribution.

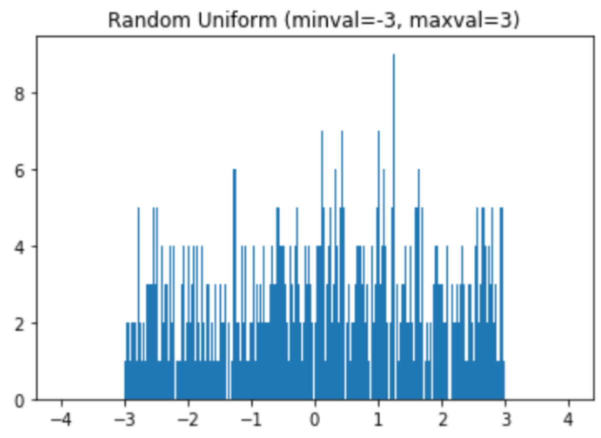
The generated values follow a uniform distribution in the range [minval, maxval). The lower bound minval is included in the range, while the upper bound maxval is excluded.

- **shape:** A 1-D integer Tensor or Python array. The shape of the output tensor.
- **minval:** A 0-D Tensor or Python value of type dtype. The lower bound on the range of random values to generate. Defaults to 0.
- **maxval:** A 0-D Tensor or Python value of type dtype. The upper bound on the range of random values to generate. Defaults to 1 if dtype is floating point.
- **dtype:** The type of the output: float32, float64, int32, or int64.
- **seed:** A Python integer. Used to create a random seed for the distribution. See tf.set_random_seed for behavior.
- **name:** A name for the operation (optional).

We can visualize the uniform distribution by using a histogram. Let's map the values from `tf.random_uniform([1000], -3, 3)` to a histogram using the `helper.hist_dist` function. This will be `1000` random float values from `-3` to `3`, excluding the value `3`.

In [5]:

```
helper.hist_dist('Random Uniform (minval=-3, maxval=3)',
tf.random_uniform([1000], -3, 3))
```



The histogram used 500 buckets for the 1000 values. Since the chance for any single bucket is the same, there should be around 2 values for each bucket. That's exactly what we see with the histogram. Some buckets have more and some have less, but they trend around 2.

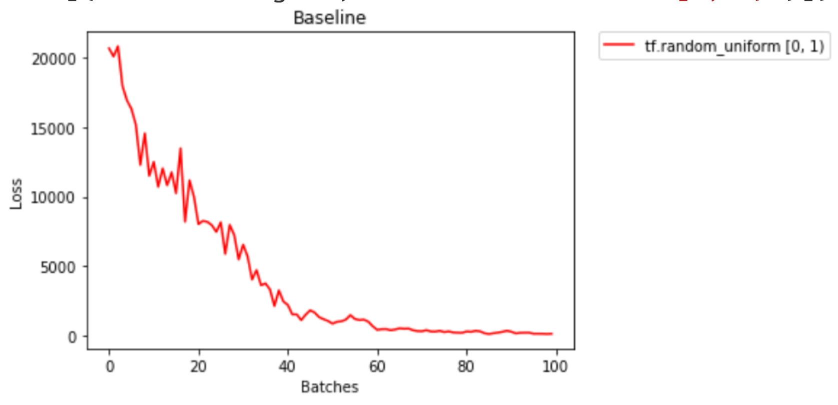
Now that you understand the `tf.random_uniform` function, let's apply it to some initial weights.

Baseline

Let's see how well the neural network trains using the default values for `tf.random_uniform`, where `minval=0.0` and `maxval=1.0`.

In [6]:

```
# Default for tf.random_uniform is minval=0 and maxval=1
basline_weights = [
    tf.Variable(tf.random_uniform(layer_1_weight_shape)),
    tf.Variable(tf.random_uniform(layer_2_weight_shape)),
    tf.Variable(tf.random_uniform(layer_3_weight_shape))
]

helper.compare_init_weights(
    mnist,
    'Baseline',
    [(basline_weights, 'tf.random_uniform [0, 1)')])
```



```
After 858 Batches (2 Epochs):
Validation Accuracy
   65.340% -- tf.random_uniform [0, 1)
Loss
   64.356  -- tf.random_uniform [0, 1)
```

The loss graph is showing the neural network is learning, which it didn't with all zeros or all ones. We're headed in the right direction.

General rule for setting weights

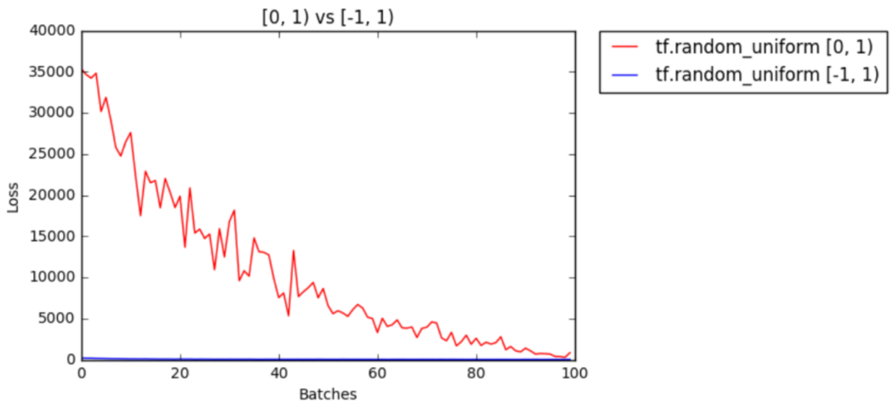
The general rule for setting the weights in a neural network is to be close to zero without being too small. A good pracitce is to start your weights in the range of [−y,y][−y,y] where y=1/n−−√y=1/n (nn is the number of inputs to a given neuron).

Let's see if this holds true, let's first center our range over zero. This will give us the range [-1, 1).

In [6]:

```
uniform_neg1to1_weights = [
    tf.Variable(tf.random_uniform(layer_1_weight_shape, -1, 1)),
    tf.Variable(tf.random_uniform(layer_2_weight_shape, -1, 1)),
    tf.Variable(tf.random_uniform(layer_3_weight_shape, -1, 1))
]
```

```
helper.compare_init_weights(
    mnist,
    '[0, 1) vs [-1, 1)',
    [
        (basline_weights, 'tf.random_uniform [0, 1)'),
        (uniform_neg1to1_weights, 'tf.random_uniform [-1, 1)')])
```



```
After 858 Batches (2 Epochs):
Validation Accuracy
   73.840% -- tf.random_uniform [0, 1)
   89.360% -- tf.random_uniform [-1, 1)
Loss
   13.700  -- tf.random_uniform [0, 1)
    5.470  -- tf.random_uniform [-1, 1)
```

We're going in the right direction, the accuracy and loss is better with [-1, 1). We still want smaller weights. How far can we go before it's too small?

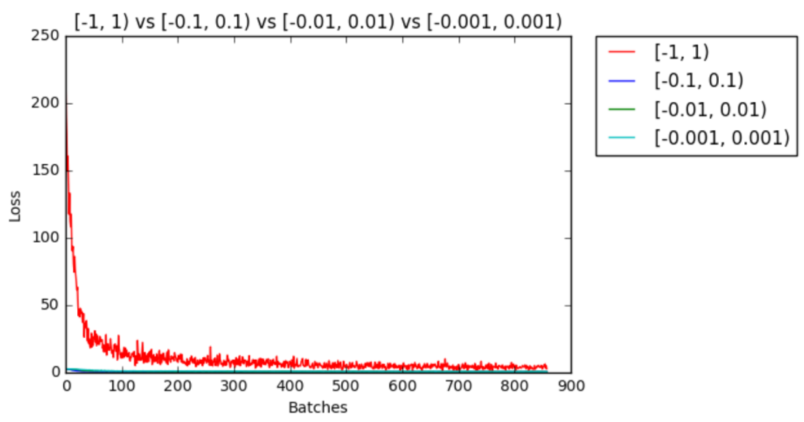
Too small

Let's compare [-0.1, 0.1), [-0.01, 0.01), and [-0.001, 0.001) to see how small is too small. We'll also set `plot_n_batches=None`to show all the batches in the plot.

In [7]:

```
uniform_neg01to01_weights = [
    tf.Variable(tf.random_uniform(layer_1_weight_shape, -0.1, 0.1)),
    tf.Variable(tf.random_uniform(layer_2_weight_shape, -0.1, 0.1)),
    tf.Variable(tf.random_uniform(layer_3_weight_shape, -0.1, 0.1))
]

uniform_neg001to001_weights = [
    tf.Variable(tf.random_uniform(layer_1_weight_shape, -0.01, 0.01)),
    tf.Variable(tf.random_uniform(layer_2_weight_shape, -0.01, 0.01)),
    tf.Variable(tf.random_uniform(layer_3_weight_shape, -0.01, 0.01))
]

uniform_neg0001to0001_weights = [
    tf.Variable(tf.random_uniform(layer_1_weight_shape, -0.001, 0.001)),
    tf.Variable(tf.random_uniform(layer_2_weight_shape, -0.001, 0.001)),
    tf.Variable(tf.random_uniform(layer_3_weight_shape, -0.001, 0.001))
]
```

```
helper.compare_init_weights(
    mnist,
    '[-1, 1) vs [-0.1, 0.1) vs [-0.01, 0.01) vs [-0.001, 0.001)',
    [
        (uniform_neg1to1_weights, '[-1, 1)'),
        (uniform_neg01to01_weights, '[-0.1, 0.1)'),
        (uniform_neg001to001_weights, '[-0.01, 0.01)'),
        (uniform_neg0001to0001_weights, '[-0.001, 0.001)')],
    plot_n_batches=None)
```

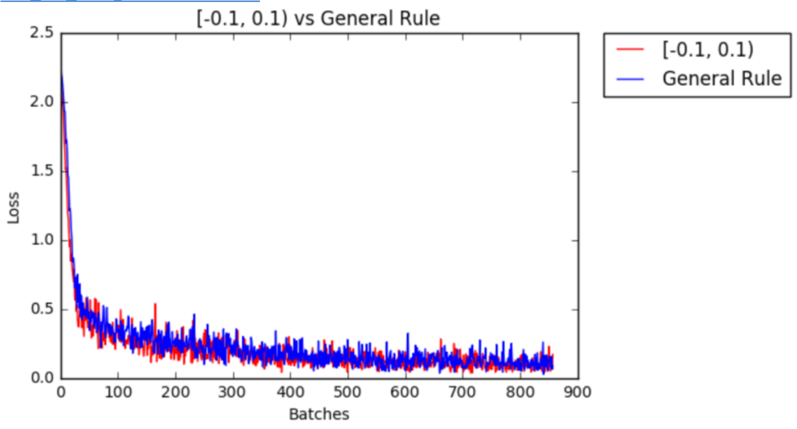


```
After 858 Batches (2 Epochs):
Validation Accuracy
   91.000% -- [-1, 1)
   97.220% -- [-0.1, 0.1)
   95.680% -- [-0.01, 0.01)
   94.400% -- [-0.001, 0.001)
Loss
    2.425  -- [-1, 1)
    0.098  -- [-0.1, 0.1)
    0.133  -- [-0.01, 0.01)
    0.190  -- [-0.001, 0.001)
```

Looks like anything [-0.01, 0.01) or smaller is too small. Let's compare this to our typical rule of using the range y=1/n−−√y=1/n.

In [8]:

```
import numpy as np

general_rule_weights = [
    tf.Variable(tf.random_uniform(layer_1_weight_shape, -
1/np.sqrt(layer_1_weight_shape[0]), 1/np.sqrt(layer_1_weight_shape[0]))),
    tf.Variable(tf.random_uniform(layer_2_weight_shape, -
1/np.sqrt(layer_2_weight_shape[0]), 1/np.sqrt(layer_2_weight_shape[0]))),
    tf.Variable(tf.random_uniform(layer_3_weight_shape, -
1/np.sqrt(layer_3_weight_shape[0]), 1/np.sqrt(layer_3_weight_shape[0])))]

helper.compare_init_weights(
    mnist,
    '[-0.1, 0.1) vs General Rule',
    [(uniform_neg01to01_weights, '[-0.1, 0.1)'),
        (general_rule_weights, 'General Rule')],
    plot_n_batches=None)
```



```
After 858 Batches (2 Epochs):
Validation Accuracy
   96.880% -- [-0.1, 0.1)
   96.900% -- General Rule
Loss
    0.169  -- [-0.1, 0.1)
    0.088  -- General Rule
```

The range we found and y=1/n−−√y=1/n are really close.

Since the uniform distribution has the same chance to pick anything in the range, what if we used a distribution that had a higher chance of picking numbers closer to 0. Let's look at the normal distribution.

Normal Distribution

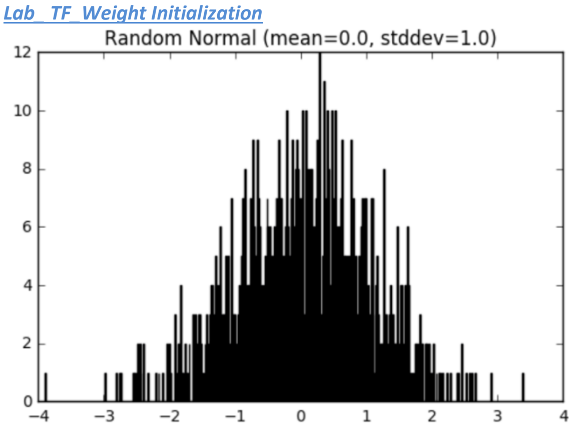
Unlike the uniform distribution, the normal distribution has a higher likelihood of picking number close to it's mean. To visualize it, let's plot values from TensorFlow's `tf.random_normal` function to a histogram.
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

Outputs random values from a normal distribution.

- **shape:** A 1-D integer Tensor or Python array. The shape of the output tensor.
- **mean:** A 0-D Tensor or Python value of type dtype. The mean of the normal distribution.
- **stddev:** A 0-D Tensor or Python value of type dtype. The standard deviation of the normal distribution.
- **dtype:** The type of the output.
- **seed:** A Python integer. Used to create a random seed for the distribution. See tf.set_random_seed for behavior.
- **name:** A name for the operation (optional).

In [9]:

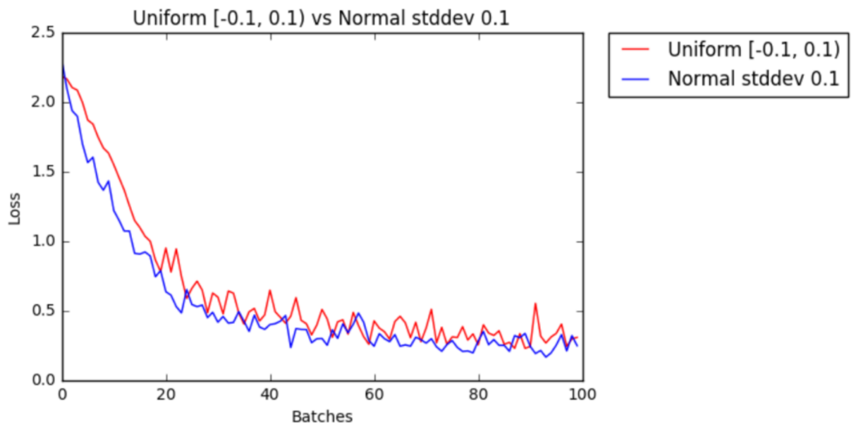
```
helper.hist_dist('Random Normal (mean=0.0, stddev=1.0)',
tf.random_normal([1000]))
```



Let's compare the normal distribution against the previous uniform distribution.

In [10]:

```
normal_01_weights = [
    tf.Variable(tf.random_normal(layer_1_weight_shape, stddev=0.1)),
    tf.Variable(tf.random_normal(layer_2_weight_shape, stddev=0.1)),
    tf.Variable(tf.random_normal(layer_3_weight_shape, stddev=0.1))
]

helper.compare_init_weights(
    mnist,
    'Uniform [-0.1, 0.1) vs Normal stddev 0.1',
    [
        (uniform_neg01to01_weights, 'Uniform [-0.1, 0.1)'),
        (normal_01_weights, 'Normal stddev 0.1')])
```



```
After 858 Batches (2 Epochs):
Validation Accuracy
   96.920% -- Uniform [-0.1, 0.1)
   97.200% -- Normal stddev 0.1
Loss
    0.103  -- Uniform [-0.1, 0.1)
    0.099  -- Normal stddev 0.1
```

The normal distribution gave a slight increasse in accuracy and loss. Let's move closer to 0 and drop picked numbers that are `x`number of standard deviations away. This distribution is called Truncated Normal Distribution.

Truncated Normal Distribution

tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

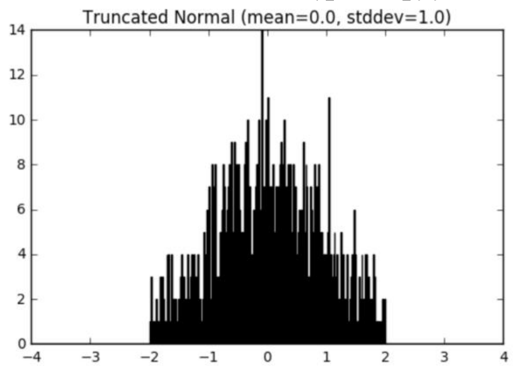
Outputs random values from a truncated normal distribution.

The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.

- **shape:** A 1-D integer Tensor or Python array. The shape of the output tensor.
- **mean:** A 0-D Tensor or Python value of type dtype. The mean of the truncated normal distribution.
- **stddev:** A 0-D Tensor or Python value of type dtype. The standard deviation of the truncated normal distribution.
- **dtype:** The type of the output.
- **seed:** A Python integer. Used to create a random seed for the distribution. See tf.set_random_seed for behavior.
- **name:** A name for the operation (optional).

In [11]:

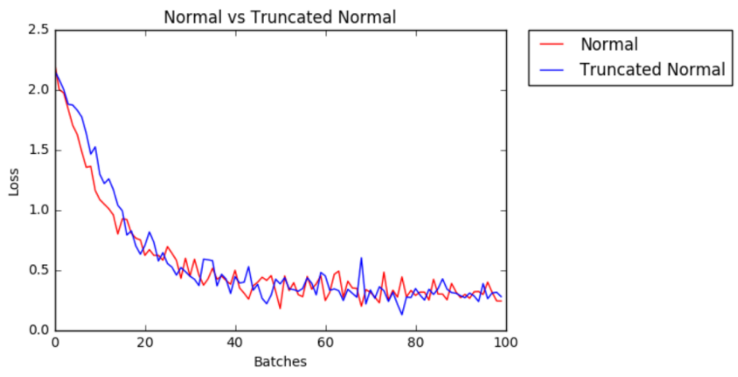
```
helper.hist_dist('Truncated Normal (mean=0.0, stddev=1.0)',
tf.truncated_normal([1000]))
```



Again, let's compare the previous results with the previous distribution.

In [12]:

```
trunc_normal_01_weights = [
    tf.Variable(tf.truncated_normal(layer_1_weight_shape, stddev=0.1)),
    tf.Variable(tf.truncated_normal(layer_2_weight_shape, stddev=0.1)),
    tf.Variable(tf.truncated_normal(layer_3_weight_shape, stddev=0.1))]

helper.compare_init_weights(
    mnist,
    'Normal vs Truncated Normal',
    [
        (normal_01_weights, 'Normal'),
        (trunc_normal_01_weights, 'Truncated Normal')])
```



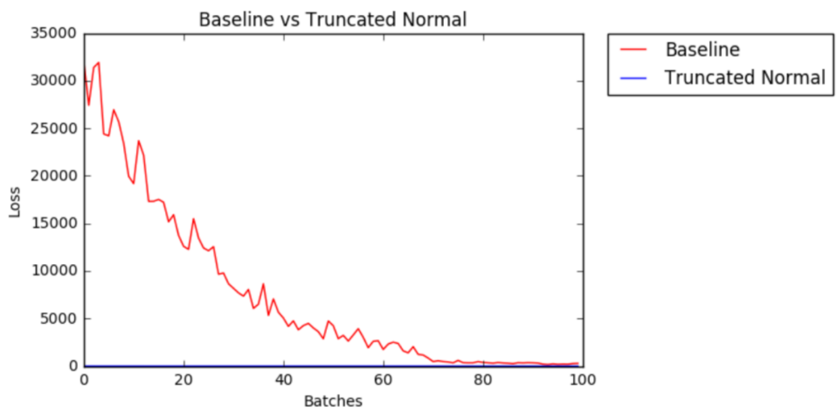
```
After 858 Batches (2 Epochs):
Validation Accuracy
   97.020% -- Normal
   97.480% -- Truncated Normal
Loss
    0.088  -- Normal
    0.034  -- Truncated Normal
```

There's no difference between the two, but that's because the neural network we're using is too small. A larger neural network will pick more points on the normal distribution, increasing the likelihood it's choices are larger than 2 standard deviations.

We've come a long way from the first set of weights we tested. Let's see the difference between the weights we used then and now.

In [13]:

```
helper.compare_init_weights(
    mnist,
    'Baseline vs Truncated Normal',
    [
        (basline_weights, 'Baseline'),
        (trunc_normal_01_weights, 'Truncated Normal')])
```



```
After 858 Batches (2 Epochs):
Validation Accuracy
   66.100% -- Baseline
   97.040% -- Truncated Normal
Loss
   24.090  -- Baseline
    0.075  -- Truncated Normal
```

That's a huge difference. You can barely see the truncated normal line. However, this is not the end your learning path. We've provided more resources for initializing weights in the classroom!

Helper.py

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

```
def hist_dist(title, distribution_tensor, hist_range=(-4, 4)):
    """
    Display histogram of a TF distribution
    """
    with tf.Session() as sess:
        values = sess.run(distribution_tensor)

    plt.title(title)
    plt.hist(values, np.linspace(*hist_range, num=len(values)/2))
    plt.show()

def _get_loss_acc(dataset, weights):
    """
    Get losses and validation accuracy of example neural network
    """
    batch_size = 128
    epochs = 2
    learning_rate = 0.001

    features = tf.placeholder(tf.float32)
    labels = tf.placeholder(tf.float32)
    learn_rate = tf.placeholder(tf.float32)

    biases = [
        tf.Variable(tf.zeros([256])),
        tf.Variable(tf.zeros([128])),
        tf.Variable(tf.zeros([dataset.train.labels.shape[1]]))
    ]

    # Layers
    layer_1 = tf.nn.relu(tf.matmul(features, weights[0]) + biases[0])
    layer_2 = tf.nn.relu(tf.matmul(layer_1, weights[1]) + biases[1])
    logits = tf.matmul(layer_2, weights[2]) + biases[2]

    # Training loss
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels))

    # Optimizer
    optimizer = tf.train.AdamOptimizer(learn_rate).minimize(loss)

    # Accuracy
    correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # Measurements use for graphing loss
    loss_batch = []

    with tf.Session() as session:
        session.run(tf.global_variables_initializer())
        batch_count = int((dataset.train.num_examples / batch_size))
        # The training cycle
        for epoch_i in range(epochs):
            for batch_i in range(batch_count):
                batch_features, batch_labels = dataset.train.next_batch(batch_size)

                # Run optimizer and get loss
                session.run(
                    optimizer,
                    feed_dict={features: batch_features, labels: batch_labels,
learn_rate: learning_rate})
                l = session.run(
                    loss,
                    feed_dict={features: batch_features, labels: batch_labels,
learn_rate: learning_rate})
                loss_batch.append(l)

        valid_acc = session.run(
            accuracy,
            feed_dict={features: dataset.validation.images, labels:
dataset.validation.labels, learn_rate: 1.0})

    # Hack to Reset batches
    dataset.train._index_in_epoch = 0
    dataset.train._epochs_completed = 0

    return loss_batch, valid_acc

def compare_init_weights(
        dataset,
        title,
        weight_init_list,
        plot_n_batches=100):
    """Plot loss and print stats of weights using an example neural network"""
    colors = ['r', 'b', 'g', 'c', 'y', 'k']
    label_accs = []
    label_loss = []

    assert len(weight_init_list) <= len(colors), 'Too many inital weights to plot'

    for i, (weights, label) in enumerate(weight_init_list):
        loss, val_acc = _get_loss_acc(dataset, weights)

        plt.plot(loss[:plot_n_batches], colors[i], label=label)
        label_accs.append((label, val_acc))
        label_loss.append((label, loss[-1]))

    plt.title(title)
    plt.xlabel('Batches')
    plt.ylabel('Loss')
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()
```

```
print('After 858 Batches (2 Epochs):')
print('Validation Accuracy')
for label, val_acc in label_accs:
    print('  {:7.3f}% -- {}'.format(val_acc*100, label))
print('Loss')
for label, loss in label_loss:
    print('  {:7.3f}  -- {}'.format(loss, label))
```