IFT 6135 - W2019 - Assignment 1

Question 1 - Multilayer Perceptron for MNIST

Assignment Instructions: https://www.overleaf.com/read/msxwmbbvfxrd (https://www.overleaf.com/read/msxwmbbvfxrd)
Github Repository: https://github.com/stefanwapnick/IFT6135PracticalAssignments (https://github.com/stefanwapnick/IFT6135PracticalAssignments)
Developed in Python 3

Team Members:

- Mohamed Amine (id 20150893) (Q2)
- Oussema Keskes (id 20145195) (Q3)
- Stephan Tran (id 20145195) (Q3)
- Stefan Wapnick (id 20143021) (Q1)

In [1]:
```
from activations import Sigmoid, Tanh, Relu
from models import NN, NNFactory
from data import load_mnist, ResultsCache
from weight_initialization import Normal, Glorot, Zeros
from visualization import plot_gradient_difference, plot_training_stats
import numpy as np
```

Part 1 - Building the Model

Methodology

A standard feed-forward neural network (multilayer perceptron) was implemented using numpy and applied to the MNIST handwritten digit dataset. This dataset consists of 10 classes (digits) and 28x28 input images (or equivalently a 784 1d vector). The standard train/dev/test split of 50k/10k/10k recommended in the assignment 1 started code was used. The neural network implementation can be found in `models.py` .
Loss is implemented using multi-class cross entropy and optimized with stochasctic gradient descent.

$$L = -\frac{1}{M}\sum_{i}^{M}\sum_{c} 1_{y=c} log(f_c(x_i))$$

M = number of samples in the mini-batch, c = class index, f = softmax

The dimensionality of the hidden layers, weight initialization, activation function (with the exception of the last layer being softmax), learning rate, and mini-batch size are parameterized.

Training - Forward Pass: The forward pass is computed by calculating the pre and post-activation functions at each layer:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$
$$a^{(l)} = g(z^{(l)})$$

z = pre-activation, a = layer output (post-activation), g = activation function

The activation function of the final output layer is taken to be the softmax function.

Training - Back Propagation: Backward propegation is done by calculating gradient quantities iteratively for each layer:

$$\nabla_{z^{(l)}} L = \nabla_{a^{(l)}} L \odot g'(z^{(l)})$$
$$\nabla_{a^{(l-1)}} L = (W^{(l)})^T \nabla_{z^{(l)}} L$$
$$\nabla_{W^{(l)}} L = \nabla_{z^{(l)}} L (a^{(l-1)})^T$$
$$\nabla_{b^{(l)}} L = \nabla_{z^{(l)}} L$$

The weights and bias terms are then adjusted:

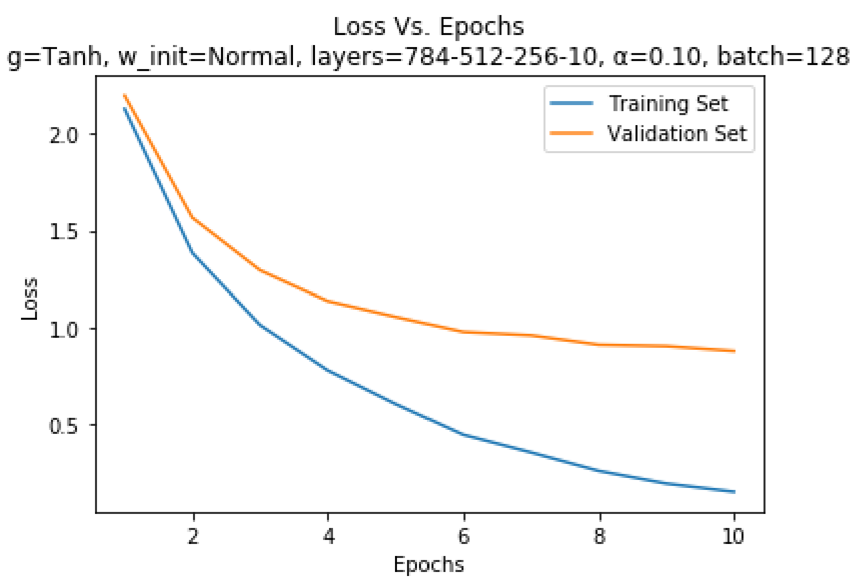
$$W^{(l)} \leftarrow W^{(l)} - \alpha \nabla_{W^{(l)}} L$$
$$b^{(l)} \leftarrow b^{(l)} - \alpha \nabla_{b^{(l)}} L$$

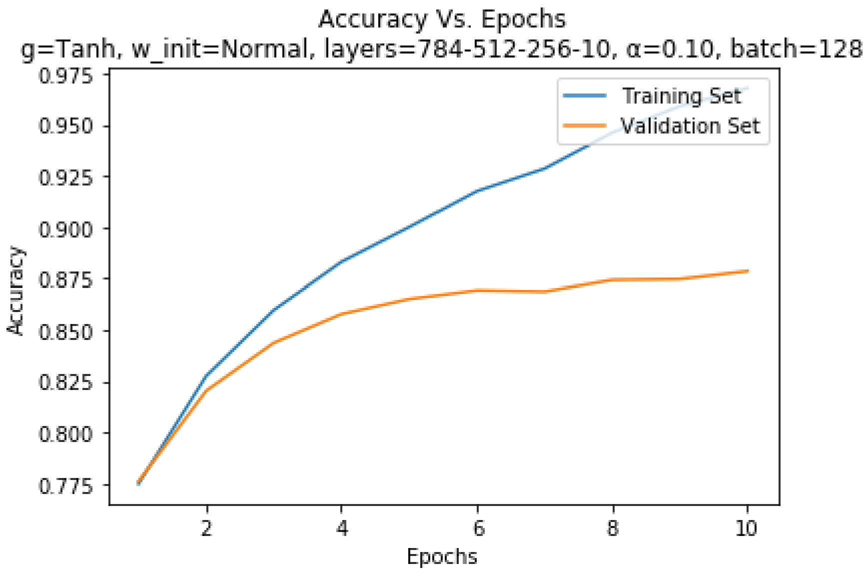
Sample Training Results

The code block below shows sample results which plot the loss and accuracy over number of epochs during training. **Tanh** activation function and **Normal** distribution weight initialization are used.

In [2]:
```
%matplotlib inline
train_set, valid_set, _ = load_mnist()
nn = NNFactory.create(hidden_dims=[512, 256], activation=Tanh, weight_init=Nor
mal)
stats = nn.train(train_set, valid_set, alpha=0.1, batch_size=128)
plot_training_stats(stats, plot_title=nn.training_info_label, plot_acc=True)
```

```
TRAINING: g=Tanh, w_init=Normal, layers=784-512-256-10, α=0.10, batch=128
Epoch 1: TrainLoss=2.127284, TrainAcc=0.774920, ValidLoss=2.196374, ValidAcc=
0.776200
Epoch 2: TrainLoss=1.384241, TrainAcc=0.827500, ValidLoss=1.565976, ValidAcc=
0.820300
Epoch 3: TrainLoss=1.012352, TrainAcc=0.859640, ValidLoss=1.294797, ValidAcc=
0.843700
Epoch 4: TrainLoss=0.778200, TrainAcc=0.883020, ValidLoss=1.135239, ValidAcc=
0.857700
Epoch 5: TrainLoss=0.605040, TrainAcc=0.899940, ValidLoss=1.053738, ValidAcc=
0.865000
Epoch 6: TrainLoss=0.445588, TrainAcc=0.917500, ValidLoss=0.976493, ValidAcc=
0.869200
Epoch 7: TrainLoss=0.354145, TrainAcc=0.928520, ValidLoss=0.959031, ValidAcc=
0.868600
Epoch 8: TrainLoss=0.258702, TrainAcc=0.946020, ValidLoss=0.910475, ValidAcc=
0.874300
Epoch 9: TrainLoss=0.193244, TrainAcc=0.958900, ValidLoss=0.903415, ValidAcc=
0.874700
Epoch 10: TrainLoss=0.150637, TrainAcc=0.967840, ValidLoss=0.878393, ValidAcc
=0.878500
DONE (100s): g=Tanh, w_init=Normal, layers=784-512-256-10, α=0.10, batch=128
- ValidAcc=0.878500
```





Analysis

Typical loss and accuracy versus epochs curves are obtained. Validation accuracy and loss begin to plateau after 10 epochs while training results continue to improve, indicating the start of some overfitting.
The validation accuracy is not very high in this instance, however with the correct set of hyper-parameters >97% accuracy can be obtained (see the hyper-parameter search section for more details).

Part 2 - Weight Initialization

Methodology

This section examines the effects of different weight initialization schemes:

- **Zeros**: All weights are initialized to 0
- **Normal**: Initialized from a standard normal distribution $\mathscr{N}(w_{ij}^l; 0, 1)$ (mean=0, variance=1)
- **Glorot**: Initialized from a uniform distribution $\mathscr{U}(w_{ij}^l; -d^l, d^l)$ where $d^l = \sqrt{\frac{6}{h^{l-1}+h^l}}$ (h^l denotes the number dimension of layer l)

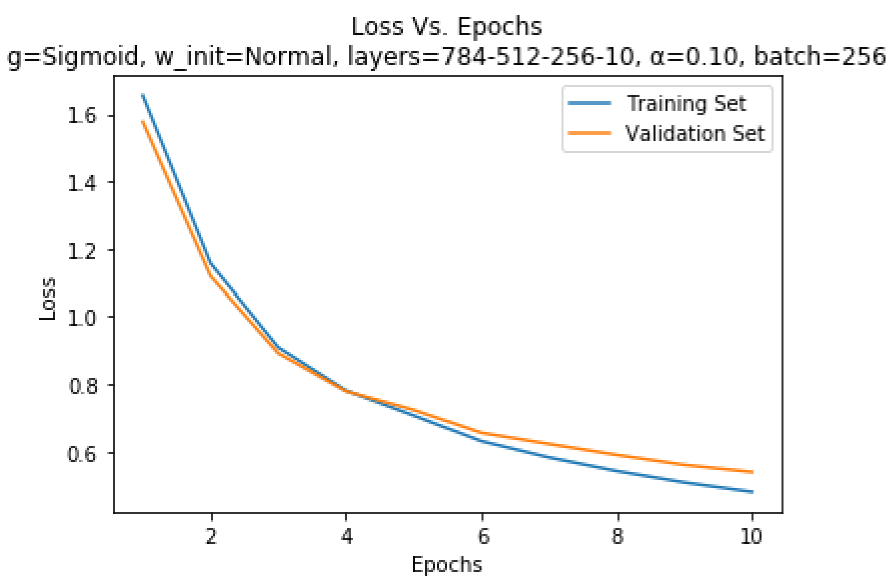
Results

The following code block plots the loss versus training epochs for the different weight schemes: Zeros, Normal, Glorot

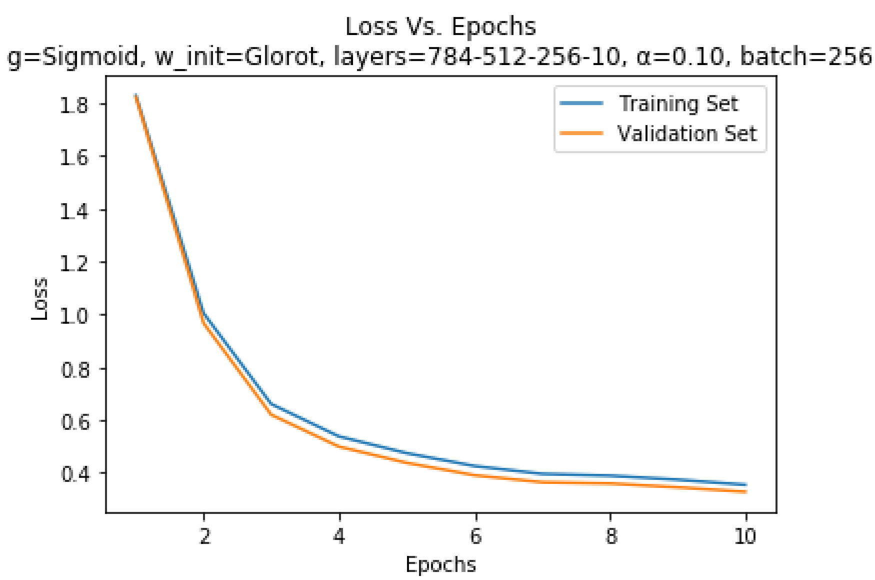
In [3]:
```
%matplotlib inline
train_set, valid_set, _ = load_mnist()
weight_inits = [Zeros, Normal, Glorot]

for weight_init in weight_inits:
    nn = NNFactory.create(hidden_dims=[512, 256], activation=Sigmoid, weight_i
nit=weight_init)
    stats = nn.train(train_set, valid_set, alpha=0.1, batch_size=256)
    plot_training_stats(stats, plot_title=nn.training_info_label)
```

```
TRAINING: g=Sigmoid, w_init=Zeros, layers=784-512-256-10, α=0.10, batch=256
Epoch 1: TrainLoss=2.337794, TrainAcc=0.098640, ValidLoss=2.335217, ValidAcc=
0.099100
Epoch 2: TrainLoss=2.306142, TrainAcc=0.103500, ValidLoss=2.305279, ValidAcc=
0.109000
Epoch 3: TrainLoss=2.313480, TrainAcc=0.099020, ValidLoss=2.313204, ValidAcc=
0.096700
Epoch 4: TrainLoss=2.316895, TrainAcc=0.113560, ValidLoss=2.318640, ValidAcc=
0.106400
Epoch 5: TrainLoss=2.311022, TrainAcc=0.102020, ValidLoss=2.311380, ValidAcc=
0.103000
Epoch 6: TrainLoss=2.313465, TrainAcc=0.099760, ValidLoss=2.315412, ValidAcc=
0.096100
Epoch 7: TrainLoss=2.312609, TrainAcc=0.099760, ValidLoss=2.315774, ValidAcc=
0.096100
Epoch 8: TrainLoss=2.311059, TrainAcc=0.099760, ValidLoss=2.310917, ValidAcc=
0.096100
Epoch 9: TrainLoss=2.313262, TrainAcc=0.113560, ValidLoss=2.315789, ValidAcc=
0.106400
Epoch 10: TrainLoss=2.309388, TrainAcc=0.113560, ValidLoss=2.311756, ValidAcc
=0.106400
DONE (82s): g=Sigmoid, w_init=Zeros, layers=784-512-256-10, α=0.10, batch=256
- ValidAcc=0.106400
```

```
TRAINING: g=Sigmoid, w_init=Normal, layers=784-512-256-10, α=0.10, batch=256
Epoch 1: TrainLoss=1.655491, TrainAcc=0.623220, ValidLoss=1.576994, ValidAcc=
0.639200
Epoch 2: TrainLoss=1.157830, TrainAcc=0.713100, ValidLoss=1.119534, ValidAcc=
0.724600
Epoch 3: TrainLoss=0.909710, TrainAcc=0.765840, ValidLoss=0.891969, ValidAcc=
0.769500
Epoch 4: TrainLoss=0.781428, TrainAcc=0.792120, ValidLoss=0.778922, ValidAcc=
0.794500
Epoch 5: TrainLoss=0.707129, TrainAcc=0.810680, ValidLoss=0.723576, ValidAcc=
0.808000
Epoch 6: TrainLoss=0.630673, TrainAcc=0.827460, ValidLoss=0.655415, ValidAcc=
0.822600
Epoch 7: TrainLoss=0.582187, TrainAcc=0.838380, ValidLoss=0.622224, ValidAcc=
0.831900
Epoch 8: TrainLoss=0.541821, TrainAcc=0.848360, ValidLoss=0.589698, ValidAcc=
0.839700
Epoch 9: TrainLoss=0.507923, TrainAcc=0.857280, ValidLoss=0.560104, ValidAcc=
0.846100
Epoch 10: TrainLoss=0.479925, TrainAcc=0.864280, ValidLoss=0.538935, ValidAcc
=0.851200
DONE (83s): g=Sigmoid, w_init=Normal, layers=784-512-256-10, α=0.10, batch=25
6 - ValidAcc=0.851200
```



```
TRAINING: g=Sigmoid, w_init=Glorot, layers=784-512-256-10, α=0.10, batch=256
Epoch 1: TrainLoss=1.829079, TrainAcc=0.484820, ValidLoss=1.822579, ValidAcc=
0.499300
Epoch 2: TrainLoss=1.005384, TrainAcc=0.743600, ValidLoss=0.967466, ValidAcc=
0.765000
Epoch 3: TrainLoss=0.660813, TrainAcc=0.834420, ValidLoss=0.620112, ValidAcc=
0.851500
Epoch 4: TrainLoss=0.537883, TrainAcc=0.859000, ValidLoss=0.499165, ValidAcc=
0.871000
Epoch 5: TrainLoss=0.474474, TrainAcc=0.869500, ValidLoss=0.437632, ValidAcc=
0.882400
Epoch 6: TrainLoss=0.425391, TrainAcc=0.883860, ValidLoss=0.390326, ValidAcc=
0.895400
Epoch 7: TrainLoss=0.395646, TrainAcc=0.888840, ValidLoss=0.363747, ValidAcc=
0.899100
Epoch 8: TrainLoss=0.388785, TrainAcc=0.889940, ValidLoss=0.359251, ValidAcc=
0.898500
Epoch 9: TrainLoss=0.373719, TrainAcc=0.890480, ValidLoss=0.344760, ValidAcc=
0.900900
Epoch 10: TrainLoss=0.354258, TrainAcc=0.897560, ValidLoss=0.327695, ValidAcc
=0.906700
DONE (83s): g=Sigmoid, w_init=Glorot, layers=784-512-256-10, α=0.10, batch=25
6 - ValidAcc=0.906700
```



Analysis

Glorot weight initialization appears to produce the best results. Ideally weight initialization should set weights with small non-zero values, such that activations functions are not saturated and produce strong gradient signals, with evenly spread values to encourage diversity of weight exploration (break symmetry between units) during training.

- **Zeros**: Results in little change because it prevents gradients can be propagated backwards ($\nabla_{a^{(l-1)}} L = (W^{(l)})^T \nabla_{z^{(l)}} L = 0$). Some flunctuations still occur because weights of the last layer (those computed from the softmax) are still adjusted somewhat however the effects are negligible.
- **Normal**: Normal distribution weight initialization produces moderate results however this scheme is outperformed by Glorot initialization that encourages a more even spread of weights. Likewise, the implemented scheme also lacks a scaling term (such as that used in Glorot initialization) and so the values sampled by the standard Normal distribution (with mean of 0, variance of 1) may be overly large in certain cases.
- **Glorot**: Glorot initialization appears to yield the best results. It possesses a faster convergence and lower overall loss after 10 epochs. Glorot initialization produces an even spread and scales terms as a function of the layer dimensionality to produce small non-zero values (ensuring a strong gradient signal, non-saturated activation function).

Part 3 - Hyperparameter Search

Methodology

In his section, the effects of different hyper-parameters on the performance of the model are explored. Hyper-parameters are tuned on the validation set to select the model that appears to generalize best. The following parameters are tested:

Parameter	Value
learning rate	0.1, 0.01
batch size	128, 256
hidden layer dimensions	(512, 256), (512, 512), (784, 256)
activation functions	sigmoid, tanh, relu

Results

The following results summarize how validation accuracy changes for different hyper-parameters. Results are ordered by descending value of validation accuracy.

In [4]:
```
%matplotlib inline

activations = [Sigmoid, Tanh, Relu]
alphas = [0.1, 0.01]
batch_sizes = [128, 256]
hidden_layers = [[512, 256], [512, 512], [784, 256]]
weight_inits = [Glorot]

train_set, valid_set, _ = load_mnist()
results_cache = ResultsCache.load()
params = [(g, h, a, b, w)
          for g in activations for a in alphas for b in batch_sizes
          for h in hidden_layers for w in weight_inits]

for (g, h, a, b, w) in params:
    nn = NNFactory.create(h, activation=g, weight_init=w)
    _, _, _, valid_acc = nn.train(train_set, valid_set, alpha=a, batch_size=b,
verbose=False)
    results_cache.insert(nn, a, b, valid_acc[-1])
results_cache.display()
```

```
Parameter Search Results Summary:
   activation weight_init          layers  alpha  batch     acc
0        Relu      Glorot  784-512-512-10   0.10    128  0.9768
1        Relu      Glorot  784-784-256-10   0.10    128  0.9764
2        Relu      Glorot  784-512-256-10   0.10    128  0.9759
3        Tanh      Glorot  784-784-256-10   0.10    128  0.9714
4        Relu      Glorot  784-784-256-10   0.10    256  0.9707
5        Tanh      Glorot  784-512-256-10   0.10    128  0.9699
6        Relu      Glorot  784-512-256-10   0.10    256  0.9690
7        Tanh      Glorot  784-512-512-10   0.10    128  0.9689
8        Relu      Glorot  784-512-512-10   0.10    256  0.9681
9        Tanh      Glorot  784-784-256-10   0.10    256  0.9607
10       Tanh      Glorot  784-512-256-10   0.10    256  0.9585
11       Tanh      Glorot  784-512-512-10   0.10    256  0.9552
12       Relu      Glorot  784-784-256-10   0.01    128  0.9392
13       Relu      Glorot  784-512-256-10   0.01    128  0.9375
14       Relu      Glorot  784-512-512-10   0.01    128  0.9369
15       Tanh      Glorot  784-512-256-10   0.01    128  0.9271
16       Tanh      Glorot  784-784-256-10   0.01    128  0.9255
17       Tanh      Glorot  784-512-512-10   0.01    128  0.9251
18       Relu      Glorot  784-784-256-10   0.01    256  0.9228
19       Relu      Glorot  784-512-256-10   0.01    256  0.9227
20       Relu      Glorot  784-512-512-10   0.01    256  0.9218
21    Sigmoid      Glorot  784-512-256-10   0.10    128  0.9197
22    Sigmoid      Glorot  784-784-256-10   0.10    128  0.9187
23    Sigmoid      Glorot  784-512-512-10   0.10    128  0.9182
24       Tanh      Glorot  784-784-256-10   0.01    256  0.9165
25       Tanh      Glorot  784-512-512-10   0.01    256  0.9165
26       Tanh      Glorot  784-512-256-10   0.01    256  0.9160
27    Sigmoid      Glorot  784-512-256-10   0.10    256  0.9067
28    Sigmoid      Glorot  784-784-256-10   0.10    256  0.9059
29    Sigmoid      Glorot  784-512-512-10   0.10    256  0.9026
30    Sigmoid      Glorot  784-784-256-10   0.01    128  0.8202
31    Sigmoid      Glorot  784-512-256-10   0.01    128  0.8089
32    Sigmoid      Glorot  784-512-512-10   0.01    128  0.7894
33    Sigmoid      Glorot  784-784-256-10   0.01    256  0.6842
34    Sigmoid      Glorot  784-512-256-10   0.01    256  0.6839
35    Sigmoid      Glorot  784-512-512-10   0.01    256  0.6374
```

Analysis

The model achieving the highest validation accuracy was **0.9768** with parameters: learning rate = 0.1, batch_size = 128, hidden_layers = (512, 512), activation function = relu. However, several other models appear in close second with only fractionally worse results.

Activation Function: Relu was found to produce the best results, followed by tanh and finally the sigmoid activation function. Tanh can be viewed as a re-scaling of the logistic sigmoid function $tanh(x) = 2\sigma(2x) - 1$ and posseses a range of $[-1, 1]$ instead of $[0, 1]$. Tanh possesses a stronger gradient signal near its active region which may help learning speed. Likewise, tanh produces ouputs values around 0 mean which may speed up convergence. To see this, consider the gradient update equation:

$$\nabla_{W^{(l)}} L = \nabla_{z^{(l)}} L (a^{(l-1)})^T$$

Updates to the i'th neuron weights are represented by the i'th row in $\nabla_{W^{(l)}} L$:

$$\nabla_{W^{(l)}} L_i = \nabla_{z^{(l)}} L_i (a^{(l-1)})^T$$

Thus the gradient for the i'th neuron is determined by the scalar multiplication of $\nabla_{z^{(l)}} L_i$ and the input vector $a^{(l-1)}$. If $a^{(l-1)}$ elements are all positive, then the direction of weight changes for the i'th neuron are determined by the sign of $\nabla_{z^{(l)}} L_i$ and all weight will either increase or decrease. This may cause a zig-zag effect as weights attempt to converge to the optimal value where some need to be higher and others lower than their current value, slowing down convergence. Thus it can be advantageous to have inputs centered at at mean 0 (output by the previous layer) instead of solely non-negative inputs (such as those produced by the logistic sigmoid) to avoid this problem.

Relu was found to exhibit the best performance. Some advantages of Relu over other activation functions are that is is less likely to exhibit vanishing gradient behavior given there is no saturation region for positive inputs and has a relatively large gradient signal for positive values.

Layer Dimensions: The dimensionality of layers was not found to substancially improved results in many models. It can be hypothesized that the MNIST dataset does not require high capacity to represent an adequate descision boundary to correctly classify most samples.

Learning Rate: In general, a higher learning rate can speed up learning by virtue or larger gradient updates however too large of a learning rate can cause oscillations around a optimum. A higher learning rate may also help escape poor a local minima during gradient descent. In this case, a larger learning rate most likely produced better results simply because training was done for a maximum of 10 epochs and so the lower learning rate was not given a adequate time to converge. More epochs could of been run however for practical purposes the training time started to become prohibitively long given the number of hyper-parameter combinations tested.

Batch Size: A smaller batch size of 128 was found to produce better results. This could be for several reasons: smaller batch sizes generally converge faster (since there are more distinct gradient updates) and can help escape local minimum due to noise in updates in order to converge to a better final solution.

Part 4 - Validate Gradients using Finite Difference

Methodology

Gradient computations are validated using the central finite difference approximation of the derivative:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} \approx \frac{L(w_{ij}^{(l)} + \epsilon) - L(w_{ij}^{(l)} - \epsilon)}{2\epsilon}$$

(For simplicity of notation, the loss function is only shown with one $w_{ij}^{(l)}$ argument)

In summary, the value of a weight $w_{ij}^{(l)}$ is manually offset by $+/-\epsilon$ and a new loss is recorded. The central finite different equation is then used to estimate the gradient of the loss with respect to this weight. This estimate will be compared with the value returned from the neural network during back-propagation. If the implementation back-propagation is working as intended these two quantities should be close.

The first 10 weights of the second layer of the network are inspected for different values of N:

$$N = [1, 10, 100, 1000, 10000] \qquad \epsilon = 1/N$$

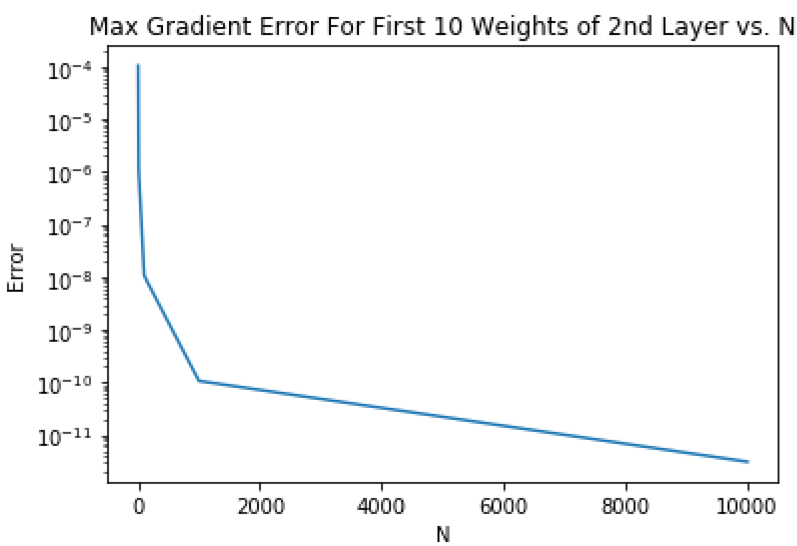
For each N value, the maximum difference of the 10 inspected weights is calculated and plotted:

$$max_{1<i<p} |\nabla_i^N - \partial L / \partial \theta_i|$$

Results

In [5]:
```
%matplotlib inline
layer = 2
M = 10
N = 10. ** (np.arange(5))
epsilons = np.reciprocal(N)
error = np.zeros(len(epsilons))

(x_train, y_train), valid_set, _ = load_mnist()
nn = NNFactory.create(hidden_dims=[512, 256], activation=Sigmoid, weight_init=
Glorot)
nn.train((x_train, y_train), valid_set)

# Take 1 training sample to use when comparing gradient calculations
x_sample = x_train[:, 0].reshape((-1, 1))
y_sample = y_train[:, 0].reshape((-1, 1))

# For the first 10 weights of the 2nd layer, calculate the max error
for i_eps, eps in enumerate(epsilons):
    for idx in range(M):
        # weight idx = layer #, neuron #, weight # for neuron
        # Inspect 10 first weights of 2nd layer, 1st neuron
        weight_idx = (layer, 0, idx)
        gradient_error = nn.estimate_finite_diff_gradient(x_sample, y_sample,
eps, weight_idx)
        error[i_eps] = max(error[i_eps], gradient_error)
plot_gradient_difference(N, error)
```

```
TRAINING: g=Sigmoid, w_init=Glorot, layers=784-512-256-10, α=0.10, batch=128
Epoch 1: TrainLoss=1.004191, TrainAcc=0.712860, ValidLoss=0.972711, ValidAcc=
0.738800
Epoch 2: TrainLoss=0.541155, TrainAcc=0.854620, ValidLoss=0.500941, ValidAcc=
0.870300
Epoch 3: TrainLoss=0.423884, TrainAcc=0.883640, ValidLoss=0.390488, ValidAcc=
0.893500
Epoch 4: TrainLoss=0.381097, TrainAcc=0.891480, ValidLoss=0.352300, ValidAcc=
0.899600
Epoch 5: TrainLoss=0.358063, TrainAcc=0.897500, ValidLoss=0.330352, ValidAcc=
0.905000
Epoch 6: TrainLoss=0.338225, TrainAcc=0.902380, ValidLoss=0.311613, ValidAcc=
0.911200
Epoch 7: TrainLoss=0.319275, TrainAcc=0.907560, ValidLoss=0.295628, ValidAcc=
0.915500
Epoch 8: TrainLoss=0.316658, TrainAcc=0.909120, ValidLoss=0.294879, ValidAcc=
0.914000
Epoch 9: TrainLoss=0.313108, TrainAcc=0.907100, ValidLoss=0.291814, ValidAcc=
0.912900
Epoch 10: TrainLoss=0.297042, TrainAcc=0.912740, ValidLoss=0.278098, ValidAcc
=0.919700
DONE (90s): g=Sigmoid, w_init=Glorot, layers=784-512-256-10, α=0.10, batch=12
8 - ValidAcc=0.919700
```



Analysis

Errors are plotted on a semi-log scale. The gradients computed during back-propegation and those estimated by the central finite difference approximation are a close match. Note that at lower values of N there is more error. However this is to be expected since the the finite difference approximation is less accurate for larger values of ϵ (smaller N). As ϵ decreases (larger N) the finite difference approximation becomes more accurate and the error was found to decrease.