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:

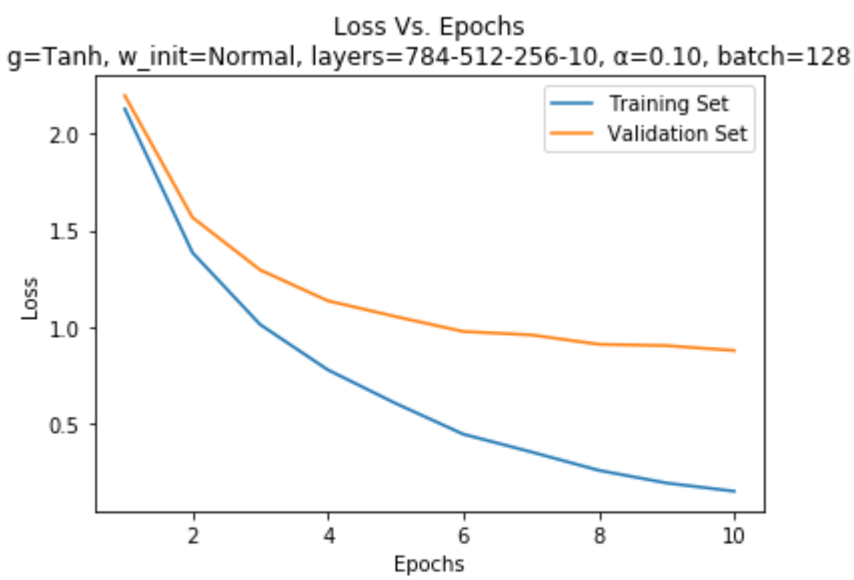
$$W^{(l)} \leftarrow W^{(l)} - \alpha \nabla_{W^{(l)}} L$$
$$b^{(l)} \leftarrow b^{(l)} - \alpha \nabla_{b^{(l)}} L$$

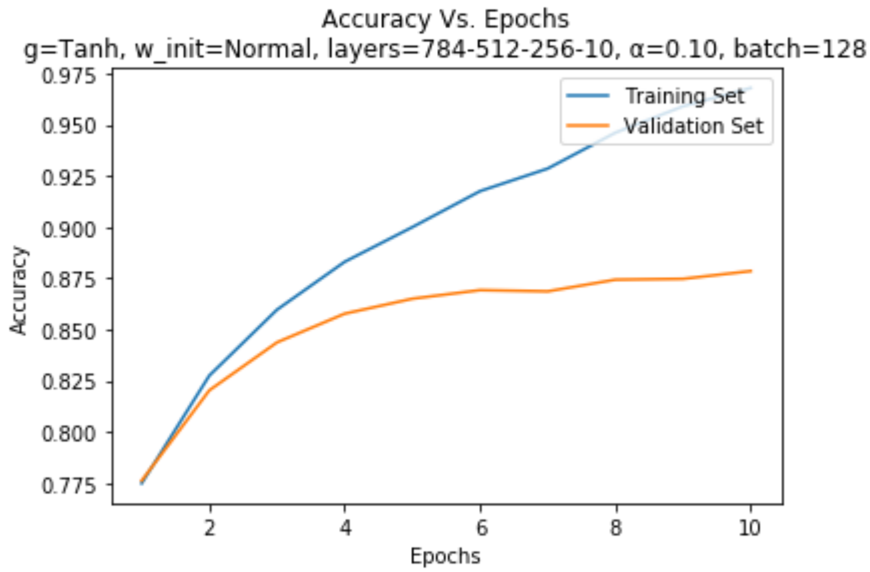
Sample Training Results

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)

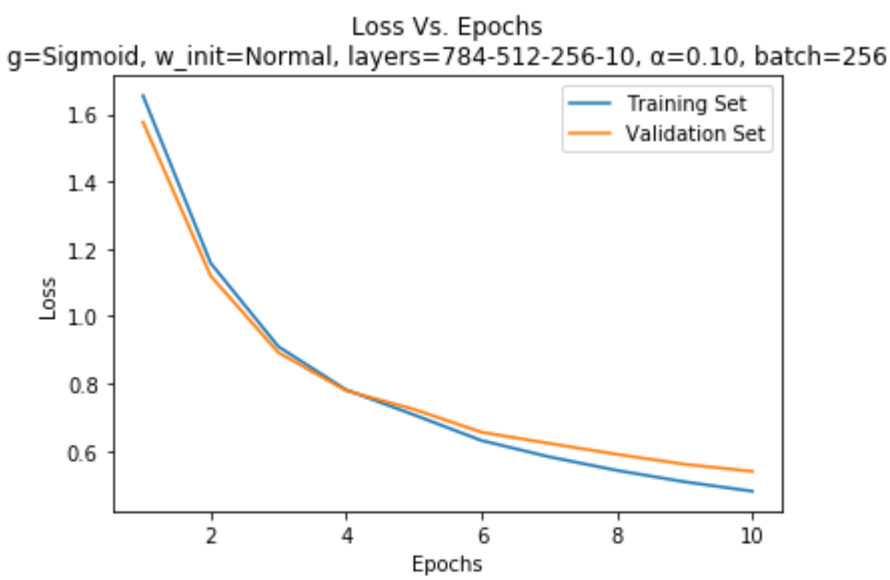
Results

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

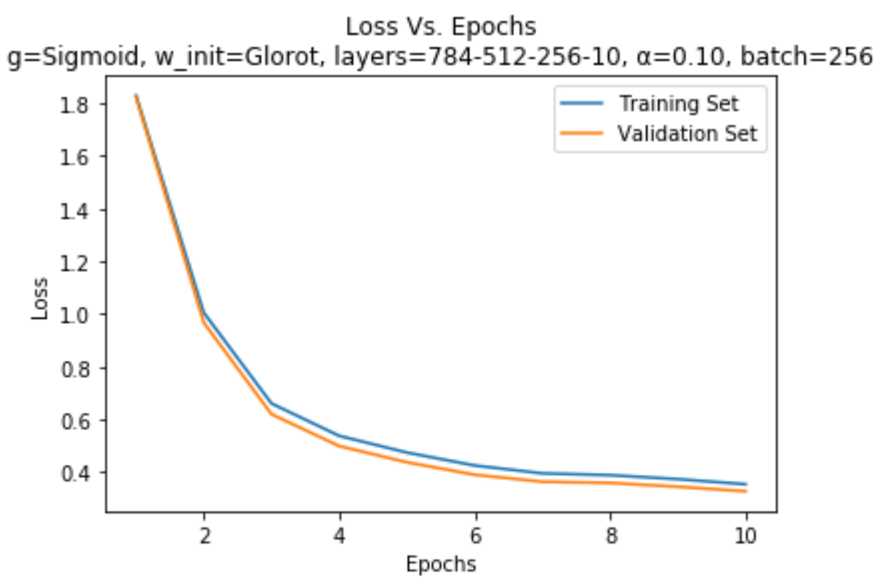
In [3]:
```
%matplotlib inline
train_set, valid_set, _ = load_mnist()
weight_inits = [Zeros, Normal, Glorot]

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:

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

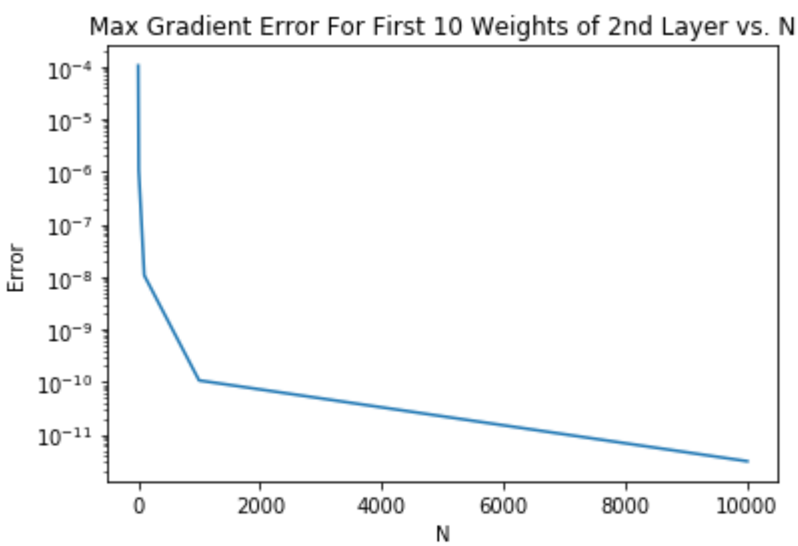
For each N value, the maximum difference of the 10 inspected weights is calculated and plotted:

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

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.

IFT 6135 - W2019 - Assignment 1

Question 2 - CNN 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)

Part 1 - CNN Model

Methodology

A CNN model using keras and tensorflow is implemented for application on the MNIST dataset consisting of 2 series of convolutional and max pooling layers. A final dense and softmax layer for classification terminate the CNN. The following sections further describe the methodology followed for data preprocessing and hyper-parameter tuning.

In [4]:
```
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.layers import Flatten,  MaxPooling2D, Conv2D
from sklearn.model_selection import train_test_split
from tensorflow.python.keras.optimizers import sgd
import matplotlib.pyplot as plt
import pandas as pd
```

Data Preprocessing

The MNIST dataset is loaded. This dataset consists of 10 classes (digits) and 28x28 input images (or equivalently a 784 1d vector). Labels are one-hot encoded. The standard train/dev/test split of 50k/10k/10k recommended in the assignment 1 started code was used.

In [5]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(60000, 28, 28, 1).astype('float32') / 255
x_test = x_test.reshape(10000, 28, 28, 1).astype('float32') / 255
n_classes = 10
y_train = keras.utils.to_categorical(y_train, n_classes)
y_test = keras.utils.to_categorical(y_test, n_classes)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=
1/6, random_state=1)

print("Size of:")
print("- Training-set:\t\t{}".format(x_train.shape[0]))
print("- Validation-set:\t{}".format(x_val.shape[0]))
print("- Test-set:\t\t{}".format(x_test.shape[0]))
print(" Shape of train target set:{}".format(y_train.shape))
```

```
Size of:
- Training-set:		50000
- Validation-set:	10000
- Test-set:		10000
 Shape of train target set:(50000, 10)
```

Hyperparameter Search

Hyper-parameters are briefly tuned on the validation dataset for model selection. The training and validation accuracies and losses are reported. The following parameters are tested:

Parameter	Value
learning rate	0.05, 0.01
batch size	128, 256
layer dimensions (conv1, conv2, dense)	(128, 256, 64), (64, 150, 128)

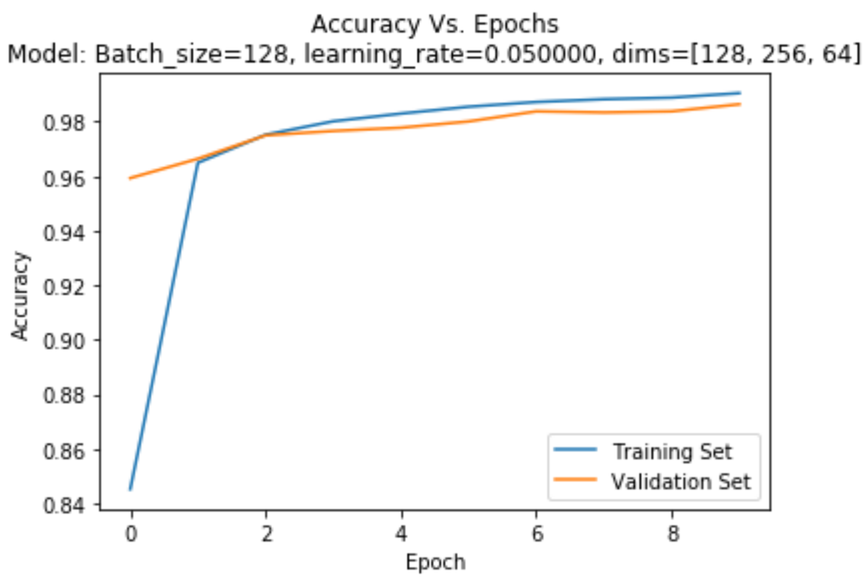
In [6]:
```
def create_model(learning_rate=0.001, layer_dims=[128, 256, 64]):
    model = Sequential()
    model.add(Conv2D(layer_dims[0], kernel_size=(5, 5), activation='relu', inp
ut_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(layer_dims[1], kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(layer_dims[2], activation='relu'))
    model.add(Dense(n_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer=sgd(lr=learning_r
ate), metrics=['accuracy'])
    return model
```

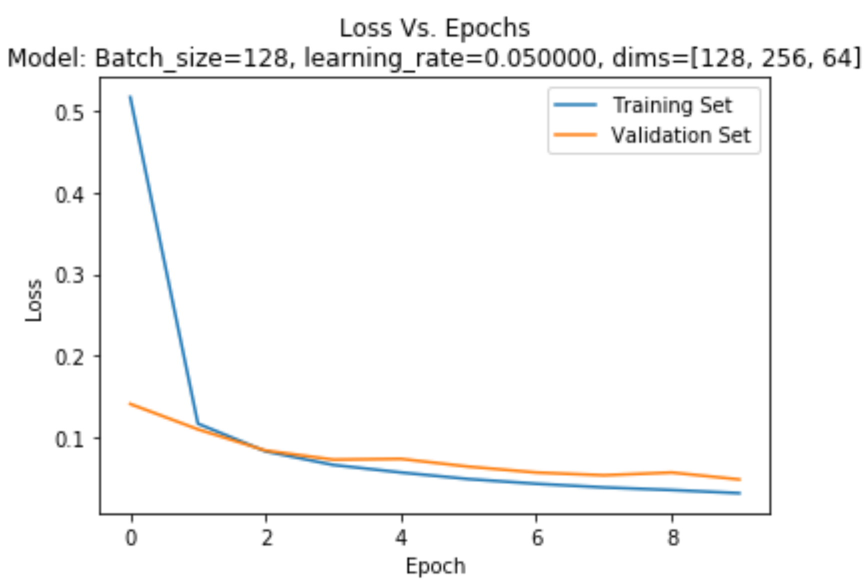
In [7]:
```
batch_sizes = [128, 256]
learning_rates = [0.05, 0.01]
layer_dims = [[128, 256, 64], [64, 150, 128]]
params = [(batch, alpha, dims) for batch in batch_sizes for alpha in learning_
rates for dims in layer_dims]
best_model = None

print("\nHyper-Parameter Search:")
for (batch_size, learning_rate, dims) in params:
    model = create_model(learning_rate, dims)
    history = model.fit(x_train, y_train, batch_size=batch_size, epochs=10, ve
rbose=0, validation_data=(x_val, y_val))
    print("Batch_size=%d, learning_rate=%f, dims=%s, val-acc=%f" % (batch_size
, learning_rate, dims, history.history['val_acc'][-1]))
    if best_model is None or history.history['val_acc'][-1] > best_model[0].hi
story['val_acc'][-1]:
        best_model = (history, model, (batch_size, learning_rate, dims))

history, model, stats = best_model
print("\nBEST MODEL: Batch_size=%d, learning_rate=%f, dims=%s, val-acc=%f" % (
*stats, history.history['val_acc'][-1]))
print(pd.DataFrame(history.history))
model.summary()
```

```
Hyper-Parameter Search:
Batch_size=128, learning_rate=0.050000, dims=[128, 256, 64], val-acc=0.986400
Batch_size=128, learning_rate=0.050000, dims=[64, 150, 128], val-acc=0.983700
Batch_size=128, learning_rate=0.010000, dims=[128, 256, 64], val-acc=0.972600
Batch_size=128, learning_rate=0.010000, dims=[64, 150, 128], val-acc=0.971500
Batch_size=256, learning_rate=0.050000, dims=[128, 256, 64], val-acc=0.980700
Batch_size=256, learning_rate=0.050000, dims=[64, 150, 128], val-acc=0.982700
Batch_size=256, learning_rate=0.010000, dims=[128, 256, 64], val-acc=0.962100
Batch_size=256, learning_rate=0.010000, dims=[64, 150, 128], val-acc=0.951500

BEST MODEL: Batch_size=128, learning_rate=0.050000, dims=[128, 256, 64], val-
acc=0.986400
   val_loss  val_acc      loss      acc
0  0.140735   0.9593  0.516678  0.84516
1  0.109671   0.9664  0.116625  0.96494
2  0.083642   0.9750  0.082670  0.97522
3  0.072500   0.9766  0.066031  0.98016
4  0.073429   0.9778  0.056796  0.98298
5  0.063924   0.9801  0.048702  0.98550
6  0.056722   0.9838  0.042981  0.98724
7  0.053344   0.9834  0.038480  0.98826
8  0.056751   0.9838  0.035274  0.98884
9  0.048227   0.9864  0.031436  0.99048
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 24, 24, 128)       3328      
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 128)       0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 256)       295168    
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 256)         0         
_________________________________________________________________
flatten (Flatten)            (None, 6400)              0         
_________________________________________________________________
dense (Dense)                (None, 64)                409664    
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650       
=================================================================
Total params: 708,810
Trainable params: 708,810
Non-trainable params: 0
_________________________________________________________________
```

Plots

In [8]:
```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Accuracy Vs. Epochs\nModel: Batch_size=%d, learning_rate=%f, dims=%
s' % (*stats,))
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training Set', 'Validation Set'])
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss Vs. Epochs\nModel: Batch_size=%d, learning_rate=%f, dims=%s' %
(*stats,))
plt.ylabel('Loss')
plt.legend(['Training Set', 'Validation Set'])
plt.xlabel('Epoch')
plt.show()
```





Test Set Results

Now the sequential model is evaluated using the test set. The accuracy and the loss are shown below.

In [9]:
```
result = model.evaluate(x_test, y_test)
print('Test Set Results:')
for name, value in zip(model.metrics_names, result):
    print(name, value)
```

```
10000/10000 [==============================] - 1s 78us/step
Test Set Results:
loss 0.03629222674376797
acc 0.9869
```

Part 2 - Comparison to MLP Discussion

The CNN model achives an accuracy of approximately 1% higher than the MLP designed in question 1 (approx. 97.5% vs. 98.5%) when tested on the validation set. Although a small quantity, in the context of the MNIST dataset where overall accuracy values are high, it is significant.
CNNs are particular adept at processing images given that the convolution operator, with various learned kernels, can be tuned to detect various patterns in an image. These patterns encoded in trained kernel weights begin as simple edges and curves but build in into more complex recognition patterns in later layers. In this way, a CNN can better analyze an image. Conversely, a MLP simply examines pixel by pixel and so is less apt at determining overall patterns.

In [ ]:
```
```

IFT 6135 - W2019 - Assignment 1

Question 3 - Cats Vs. Dogs

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)

Kaggle Team name: **Doge** Submission to Kaggle done by Stephan Anh Vu Tran

Inspired by the IFT6135-H19 PyTorch Tutorial

Importing libraries

In [24]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms, datasets
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt
import csv
import os
import pandas as pd
from IPython.display import display, Markdown
```

Defining paremeters and variables

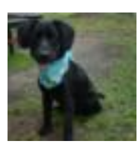
In [ ]:
```
RUN_LOCAL = True
VALID_RATIO = 0.10
TRAIN_RATIO = 1.0 - VALID_RATIO
LEARNING_RATE = 0.00001
KERNEL_SIZE = 3
PAD = 1
BATCH_SIZE = 128
MAX_EPOCH = 186

SAVE_MODEL = False  # Save model after each epoch
LOAD_MODEL = False # Skip training phase
SAVE_PATH = 'BestModel.pwf'

if RUN_LOCAL:
    TRAIN_IMG_DIR ='trainset/trainset'
    TEST_IMG_DIR = 'testset/'
else:
    # Kaggle directory
    TRAIN_IMG_DIR ='../input/trainset/trainset'
    TEST_IMG_DIR = '../input/testset'

cuda_available = torch.cuda.is_available()
# print(cuda_available)
```

Importing datasets and preprocessing

1. Load training and testing images
2. Data augmentation:
   In order to learn features from various images of position of dog and cats, we have augmented the dataset examples with
   a) **Horizontal flipping**: the input image is randomly mirrored so the model could learn features of a dog or cat even if it is flipped. The dog shown in the left side below and the right side below should both be recognised as a dog by the model.

   b) **Rotation**: Similarily to the horizontal flipping process, we have added random rotation (up 10 degrees roration) to the training set to learn features of a dog or cat no matter how it is oriented. This is meant to make the model able to recognize a dog or cat even if it is slighty rotated.

   c) **Random resize and scale**: the training images have been cropped and then rescale back to the original size. The purpose of the operation is to build some invariance to scaling of dogs and cats features in our model.



3. Transform images into tensor so they could be processed
4. **Split dataset into training and validation set**: To determine the appropriate the hyperparameters (number of convoution layers, kernel size, learning rate, choice of activation function, etc.) for the current task, we have split the initial training set into a final training set and a validation set with a ratio of 90 % and 10 % respectively. This way, we should be able to have enough training examples to learn the features and enough validation set examples so the result of the validation can be considered a good representation of unknown inputs (test set). In this case, we have:

- Training set total examples : 19,998 items
- Training set after split (90%) : 17,999 items
- Validation set after split (10%) : 1,999 items

In [11]:
```
# Class to return image folder with paths inspired from andrewjong
class ImageFolderWithPaths(datasets.ImageFolder):
    def __getitem__(self, index):
        original_tuple = super(ImageFolderWithPaths, self).__getitem__(index)
        path = self.imgs[index][0]
        tuple_with_path = (original_tuple + (path,))
        return tuple_with_path

# Apply a combination of transforms on all images
image_transform = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomHorizontalFlip(),
    transforms.RandomResizedCrop(64, scale=(0.9, 1.0)),
    transforms.ToTensor(),
    transforms.Normalize((0.48,0.45,0.40), (0.20,0.20,0.20))
])

# No transforms on the image
image_transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.48,0.45,0.40), (0.20,0.20,0.20))
])

train_dataset = datasets.ImageFolder(root=TRAIN_IMG_DIR, transform=image_trans
form)
valid_dataset = datasets.ImageFolder(root=TRAIN_IMG_DIR, transform=image_trans
form_test)

# Dataset splitting (train-validation)
dataset_size = len(train_dataset)
indices = list(range(dataset_size))
split = int(np.floor(VALID_RATIO * dataset_size))

np.random.seed(42)
np.random.shuffle(indices)

train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)


# Load dataset
trainset_loader = torch.utils.data.DataLoader(train_dataset, batch_size = BATC
H_SIZE,
                                              num_workers=4, sampler=train_samp
ler)

validset_loader = torch.utils.data.DataLoader(valid_dataset, batch_size = 1,
                                              num_workers=4, sampler=valid_samp
ler)


test_dataset = ImageFolderWithPaths(root=TEST_IMG_DIR, transform=image_transfo
rm_test)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1, shuffle=
```

```
False,
                                               num_workers=4)
```

Showing some samples of the training set

In [12]:
```
def imshow(img):
    x, y = img
    x = x.numpy().transpose((1, 2, 0))
    print(x.shape)
    plt.imshow(x)
    if y == 0:
        print('Meow!')
    else:
        print('Barf!')

# plt.figure()
# plt.subplot(1,2,1)
# imshow(train_dataset[800])
# plt.subplot(1,2,2)
# imshow(valid_dataset[100])

def get_mean_std(dataset):
    r_mean_list = []
    g_mean_list = []
    b_mean_list = []
    r_std_list = []
    g_std_list = []
    b_std_list = []

    for i in range(100):
        x = dataset[10*i][0]
        x = np.asarray(x)
        r_mean, g_mean, b_mean = np.mean(x[0]), np.mean(x[1]), np.mean(x[2])
        r_std,g_std,b_std = np.std(x[0]), np.std(x[1]), np.std(x[2])

        r_mean_list.append(r_mean)
        g_mean_list.append(g_mean)
        b_mean_list.append(b_mean)
        r_std_list.append(r_std)
        g_std_list.append(g_std)
        b_std_list.append(b_std)

    print(dataset[100][0])

    mean = [np.mean(r_mean_list), np.mean(g_mean_list), np.mean(b_mean_list)]
    std = [np.mean(r_std_list), np.mean(g_std_list), np.mean(b_std_list)]

    return mean, std

# print(get_mean_std(train_dataset))
```

Defining CNN architecture

We use a CNN which is inspired from VGGNet model with 4 convolution layers, a ReLU activation function after each convolution stage followed by max pooling layers from beginning to end and 3 fully connected linear layers as shown in the output below. Here are additional description of the model:

- The size of the convolutional filters is (3x3) with padding p= 1
- Max pooling windows is (2, 2) with stride s=2.
- The output activation function is a sigmoid so we can have an output ranging from [0,1]
- The total number of parameters is 4846401.
- Optimizer: Stochastic Gradient Descent with a learning rate of 0.00001
- Loss Function: Binary Cross Entropy
- The batch size is 128
- The number of epochs is 186

In [18]:
```
class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            # Layer 1: Convolution - ReLU - Max pooling
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(KERNEL_SIZE
, KERNEL_SIZE), padding=PAD),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 2: Convolution - ReLU - Max pooling
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(KERNEL_SIZ
E, KERNEL_SIZE), padding=PAD),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 3: Convolution - ReLU - Max pooling
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(KERNEL_SI
ZE, KERNEL_SIZE), padding=PAD),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 4: Convolution - ReLU - Max pooling
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=(KERNEL_S
IZE, KERNEL_SIZE), padding=PAD),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2)

        )
        # Layer 5-6-7: Fully connected linear layers
        self.fc1 = nn.Linear(256*4*4, 1024)
        self.fc2 = nn.Linear(1024, 256)
        self.fc3 = nn.Linear(256, 1)


    def forward(self, x):
        temp = self.conv(x)
        temp = temp.view(-1,256*4*4)
        temp = self.fc1(temp)
        temp = self.fc2(temp)
        temp = self.fc3(temp)
        temp = torch.sigmoid(temp)
        return temp

cnn = Classifier()
if cuda_available:
    cnn = cnn.cuda()

optimizer = torch.optim.SGD(cnn.parameters(), lr=LEARNING_RATE)
criterion = nn.BCELoss()

pytorch_total_params = sum(p.numel() for p in cnn.parameters())
print('Number of parameters in the model: %d' % pytorch_total_params)
print(cnn)
```

```
Number of parameters in the model: 4846401
Classifier(
  (conv): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_
mode=False)
    (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_
mode=False)
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_
mode=False)
    (9): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil
_mode=False)
  )
  (fc1): Linear(in_features=4096, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=1, bias=True)
)
```

Classifier training and validation

In [ ]:
```
best_epoch = 0
best_acc = 0
log_train_loss = []
log_train_acc = []
log_valid_loss = []
log_valid_acc = []

if LOAD_MODEL == False:
    for epoch in range(MAX_EPOCH):

        train_losses = []
        valid_losses = []

        total = 0
        correct = 0

        ##### TRAINING PHASE ####
        # Set the model in training mode
        for batch_idx, (inputs, labels) in enumerate(trainset_loader):

            # Data conversion to float and cuda
            labels_flt = torch.tensor(labels, dtype=torch.float)
            if cuda_available:
                inputs, labels_flt, labels = inputs.cuda(), labels_flt.cuda(),
labels.cuda()

            # Compute forward phase
            outputs = cnn(inputs).squeeze()

            # Compute training loss (Binary Cross Entropy)
            loss = criterion(outputs, labels_flt)
            train_losses.append(loss.data.item())

            # Prediction: since the output is between 0 and 1 du to sigmoid ac
tivation function, the
            # prediction value will be 0 when the output is [0,0.5] and 1 when
it is ]0.5,1]
            predicted = outputs > 0.5
            if cuda_available:
                predicted = torch.tensor(predicted, dtype=torch.long).cuda()
            else:
                predicted = torch.tensor(predicted, dtype=torch.long)

            # Compute training accuracy
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            train_acc = 100*correct/total

            # Backward propagation
            loss.backward()
            optimizer.step()

        # Log the training loss and accuracy with respect to the epoch number
        log_train_loss.append([epoch,np.mean(train_losses)])
        log_train_acc.append([epoch,train_acc])
```

```
        # Save model into a file for later use
        if SAVE_MODEL:
            if epoch%5 == 0:
                torch.save(cnn.state_dict(), "CNN{0:03d}.pwf".format(epoch))


        ##### VALIDATION PHASE ####
        total = 0
        correct = 0

        # Set the model in evaluation mode
        with torch.no_grad():
            for batch_idx, (inputs, labels) in enumerate(validset_loader):
                # Data conversion to float and cuda
                labels_flt = torch.tensor(labels, dtype=torch.float)
                if cuda_available:
                    inputs, labels_flt, labels = inputs.cuda(), labels_flt.cud
a(), labels.cuda()

                # Compute forward phase
                outputs = cnn(inputs).squeeze()

                # Compute validation loss (Binary Cross Entropy)
                loss = criterion(outputs, labels_flt)
                valid_losses.append(loss.data.item())

                # Prediction
                predicted = outputs > 0.5
                predicted = torch.tensor(predicted, dtype=torch.long).cuda()

                # Compute training accuracy
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                valid_acc = 100*correct/total

        # Log the training loss and accuracy with respect to the epoch number
        log_valid_loss.append([epoch,np.mean(valid_losses)])
        log_valid_acc.append([epoch,valid_acc])

        if valid_acc > best_acc:
            best_epoch = epoch
            best_acc = valid_acc

    log_train_loss = np.swapaxes(np.asarray(log_train_loss),0,1)
    log_train_acc = np.swapaxes(np.asarray(log_train_acc),0,1)
    log_valid_loss = np.swapaxes(np.asarray(log_valid_loss),0,1)
    log_valid_acc = np.swapaxes(np.asarray(log_valid_acc),0,1)

    print('Test Acc : %.3f Best epoch: %d Best acc: %.3f' % (epoch, valid_acc
, best_epoch, best_acc))
    print('-----------------------------------------------------------------------------
-----------------')
else:
    cnn.load_state_dict(torch.load(SAVE_PATH))
    cnn.eval()
```

Plot training and validation loss/accuracy

Refer to curves below

The first trend we can observe is the more training epochs we execute the more the accurate the training and validation are until a certain point. However, as we could see from the validation loss curve, the model starts to overfit on the training set after about 225 epochs. This is because the neural network is now modeled after the training set which degrades generalization perfgormance of the network.

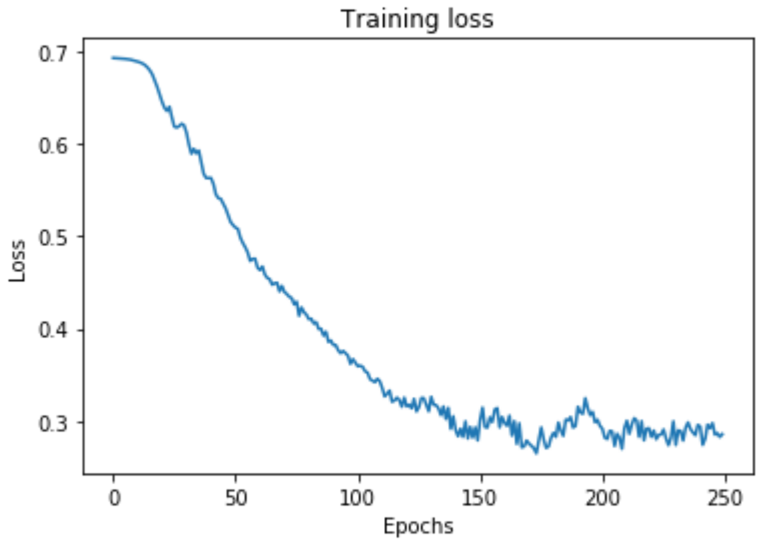
Also, we can see a spike in the training loss and validation loss at about epoch 180. This may be due to an explosion of the gradient somehow. The model then converges back to a suboptimal point in epoch ~ 210.

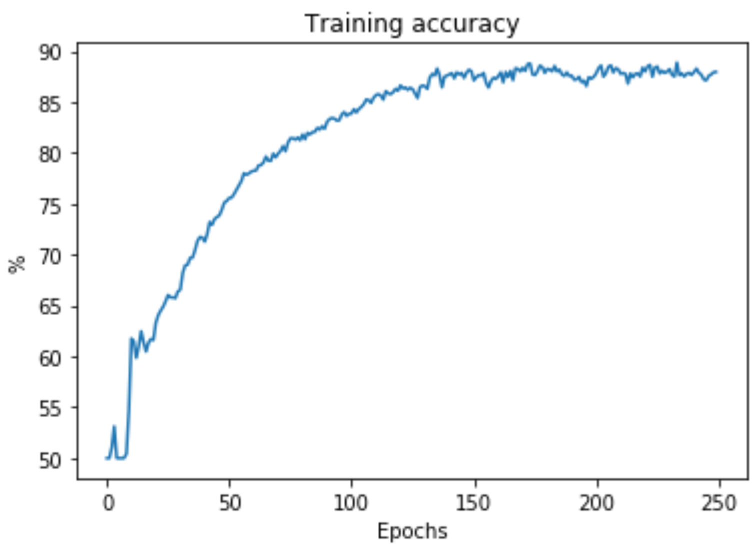
In order to improve our validation performance and hopefully the test accuracy, we could have added some regularization techniques in the process so our model could be better at generalization. For example, we could have implementend

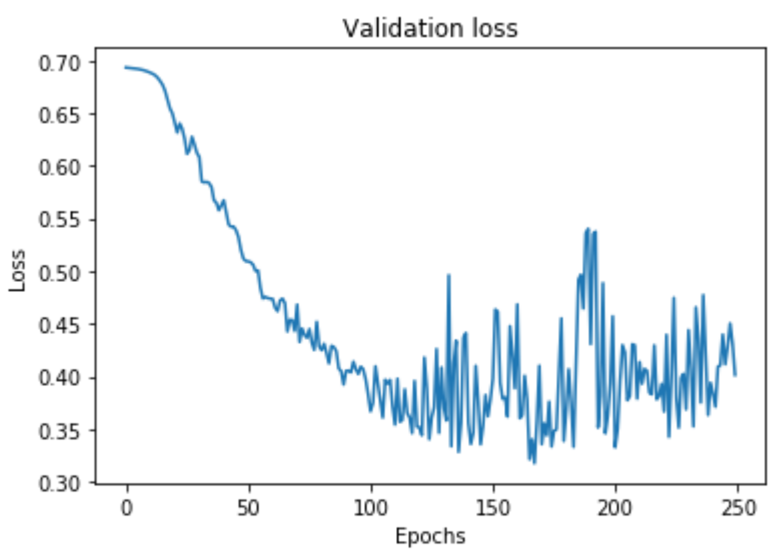
- Dropout: randomly removing some nonoutput units during training phase. This way, the learned features could be spread across more neurons instead of being “concentred” in certain neurons in the network.
- L1 or L2 regularization: adding a regularizer term in the objective function to “penalize” the weight.

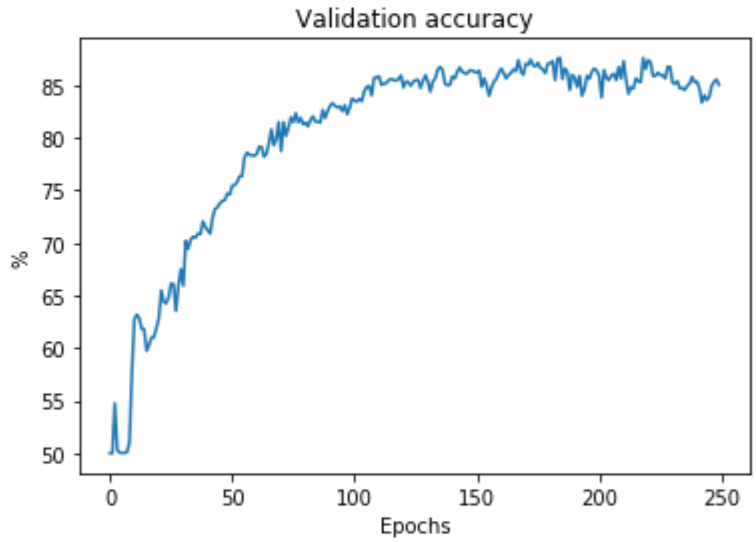
We obtained the best accuracy in validation of about 84.75% with the hyperparameters described in 3.1. Then, we submitted our model to the test set which results in a score of 85.74%. From these numbers, we can suggest that the validation process was good enough so we could have a similar performance with the test set. Our model was able to generalize to the test examples and didn’t overfit either to the training set or the validation set.

In [22]:
```
if LOAD_MODEL == False:
    plt.figure(1)
    plt.plot(log_train_loss[0], log_train_loss[1])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training loss')
    plt.savefig('train_loss.png')

    plt.figure(2)
    plt.plot(log_train_acc[0], log_train_acc[1])
    plt.xlabel('Epochs')
    plt.ylabel('%')
    plt.title('Training accuracy')
    plt.savefig('train_acc.png')

    plt.figure(3)
    plt.plot(log_valid_loss[0], log_valid_loss[1])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Validation loss')
    plt.savefig('valid_loss.png')

    plt.figure(4)
    plt.plot(log_valid_acc[0], log_valid_acc[1])
    plt.xlabel('Epochs')
    plt.ylabel('%')
    plt.title('Validation accuracy')
    plt.savefig('valid_acc.png')

    plt.show()
```


Training loss
0.7
0.6
0.5
0.4
0.3
Loss
0
50
100
150
200
250
Epochs


Training accuracy
90
85
80
75
70
65
60
55
50
%
0
50
100
150
200
250
Epochs


Validation loss
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
Loss
0
50
100
150
200
250
Epochs

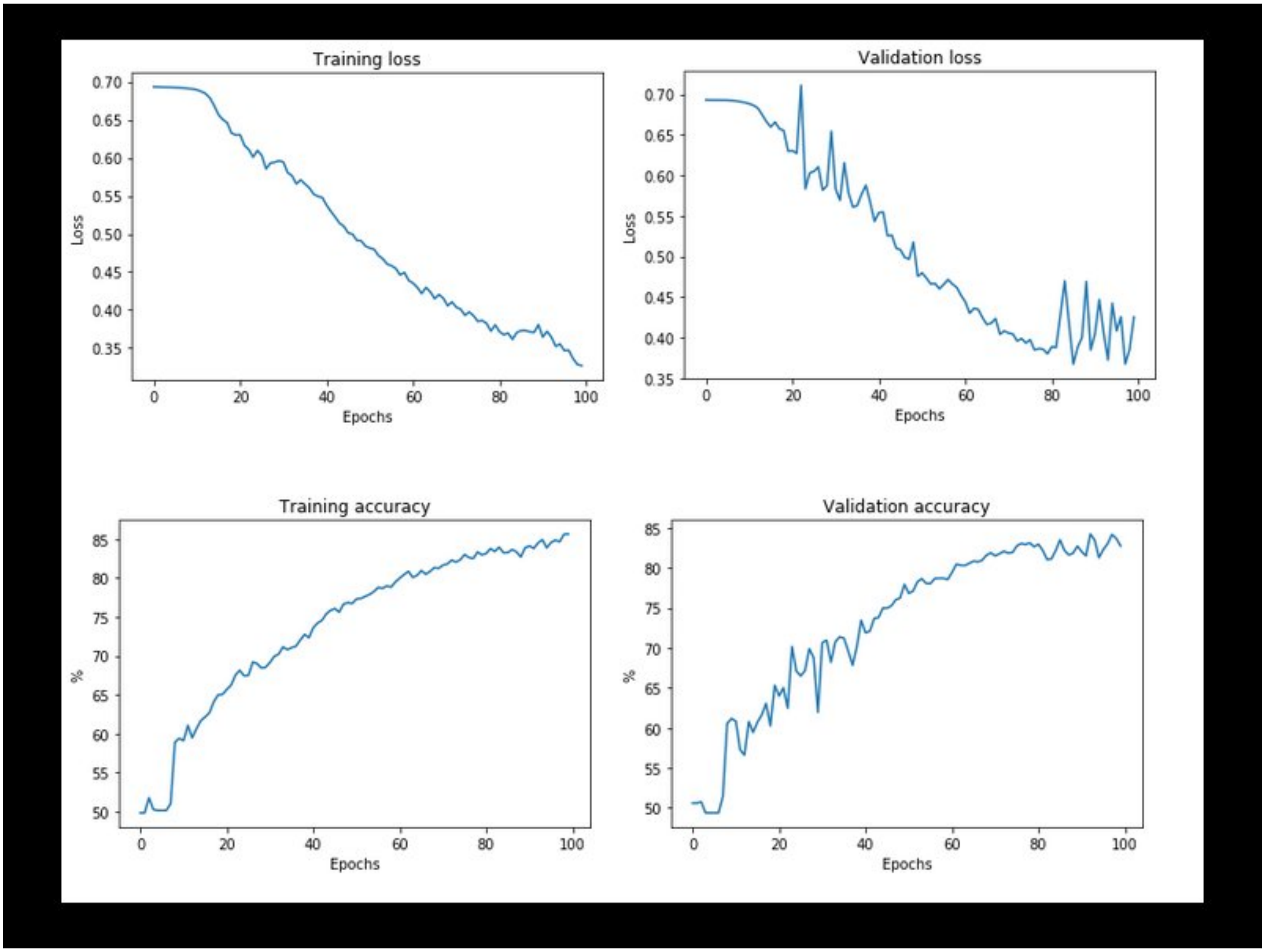

Validation accuracy
%
Epochs
50
55
60
65
70
75
80
85
0
50
100
150
200
250

Hyperparameters search

We added another linear layer and keep the other hyperparameters as is, we get:

```
print(cnn)
```

```
Classifier(
  (conv): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=4096, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=1, bias=True)
)
```



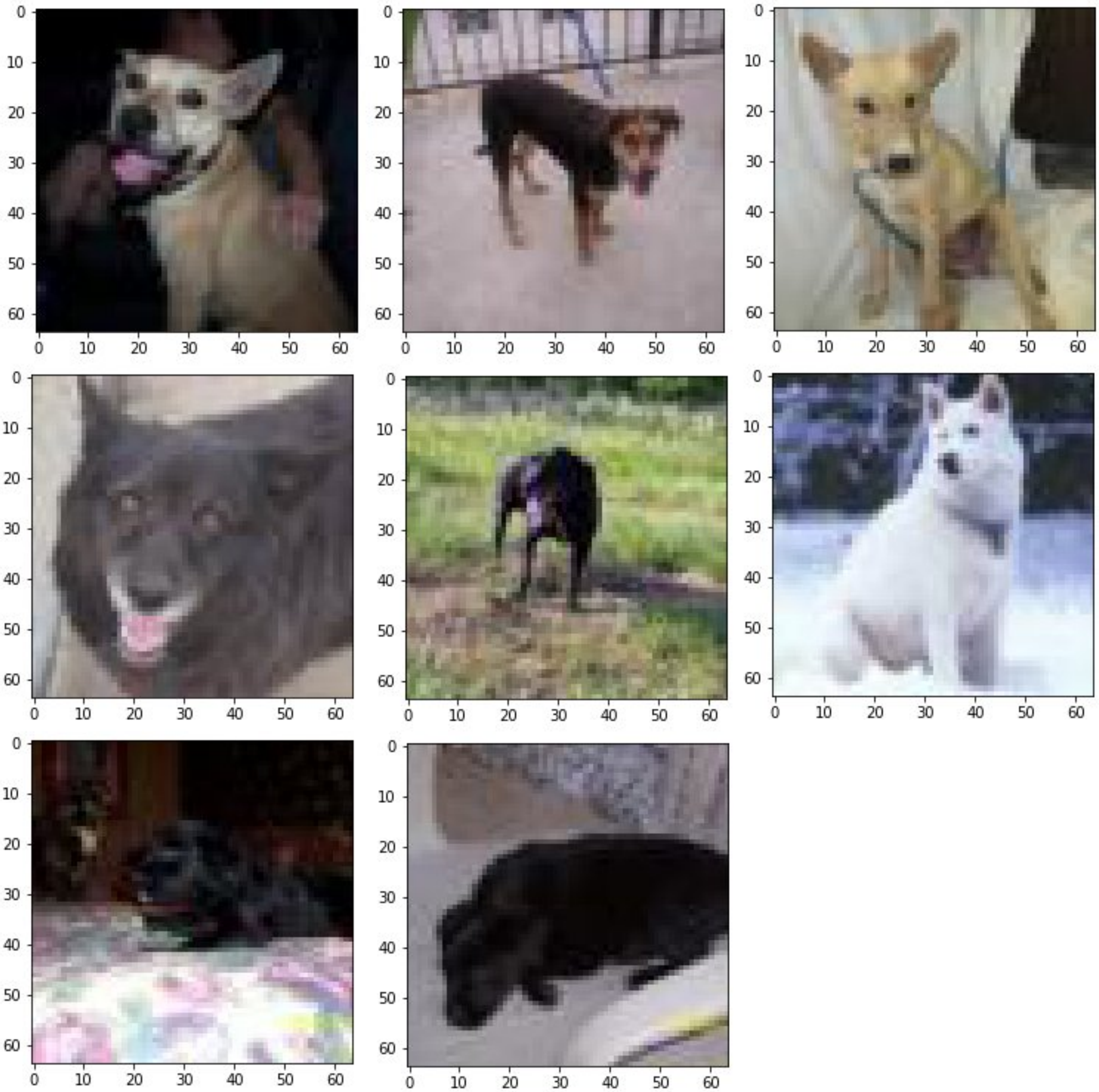
We also tried to increase the value of learning rate (Learning rate = 0.001) by keeping the other hyperparameters.

Predicting test dataset

In [ ]:
```
header = ['id', 'label']
prediction = []

with torch.no_grad():
    for batch_idx, (inputs, labels, paths) in enumerate(test_loader):
        if cuda_available:
            inputs, labels = inputs.cuda(), labels.cuda()
        outputs = cnn(inputs)
        _, predicted = torch.max(outputs.data, 1)
        predicted = outputs > 0.5
        predicted = torch.tensor(predicted, dtype=torch.long).cuda()

        filename = os.path.basename(paths[0])
        filename = os.path.splitext(filename)[0]

        if predicted == 0:
            data_out = [filename, 'Cat']
        else:
            data_out = [filename, 'Dog']
        prediction.append(data_out)

image_name = np.asarray(prediction)[:,0]
label = np.asarray(prediction)[:,1]
submission = pd.DataFrame({ 'id': image_name, 'label': label })
submission.to_csv("my_submission.csv", index=False)
```

Visualize misclassification and uncertain classification

Some of the misclassification examples that the network got are shown below



We notice from these samples that dogs with pointy ears are more prone to be classified as cats. Thus, it is highly probable that our model learned features on the ear of the animal and every photo where an animal has pointy ears, it is considered to be a cat. What we also observe is image where the object is not really distinguishable is quite hard for the network to classify as shown on image e).

Furthermore, for a couple of test images, the model ouputs a probability of about 50 % on both classes (we’ve displayed the input images when the output of the neural network is 0.45 < y < 0.55).

What we notice here is dark object of interested in the image make it harder to the network to be able to classify with high probability the class of the input as shown in the last row of the images above. There is a difficulty to detect facial features on the animal (hard to locate the nose, the eyes, etc.). Also, form the first image, we could clearly see why the convolutional neural network had a gard time predicting the class since the face of the cat is not visible. Another case where there is some difficulties for the model is the presence of multiple object as seen on image #2 on the first row. It is much easier to

To improve the whole model, we could perform additional pre-processing on the image in order to enhance details, edges and features in the image. This way, the network may be able to learn and detect features that were indistinguishable before. This could be done with a high-pass filter. As for dark object, we could add histogram equalization on the image to spread the contrast of the image and thus make it more clearer and brighter. Also, image segmentation could be perform first so we can extract the object of interest and suppress any background information.