Naomi Kaduwela

Assignment 1 - Due 5/8/2019

Simple neural network

Build a simple neural network from first principles and understand the underlying mechanisms of how the system works.

In this assignment, students will:

- Build a simple neural network from basic operations using numpy
- Implement the core neural network in 9 simple lines of code
- Make improvements to the simple neural network
- Visualize the network to understand weights, activations, gradients and class separation power

----

Final Best Code

- Below is the example, and header sections for each question with the related work, commented out, as requested.

Original Code

In [374]:

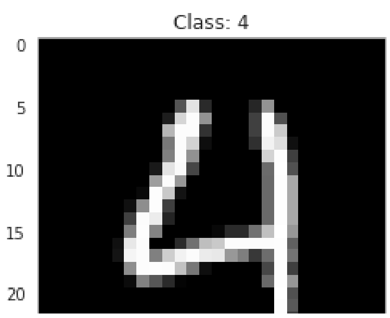
```
import os
import numpy as np
import matplotlib.pyplot as plt


# Load MNIST dataset
from tensorflow.keras.datasets import mnist
# X is an array of 60,000 28x28 black and white images with pixel values from 0-255
# Y is an array of 60,000 labels with values from 0-9 denoting the image class
(X, Y), (_, _) = mnist.load_data()
```

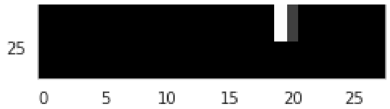
In [375]:

```
# Show some random example images from X
random_digit = np.random.randint(0, len(X) - 1)
plt.imshow(X[random_digit], cmap='gray')
# Show image class
plt.title('Class: %01d' % Y[random_digit])
```

Out[375]:

```
Text(0.5, 1.0, 'Class: 4')
```





In [376]:

```
# Do some basic transforms of data

# We want the pixels in X to have values between 0-1. We'll need to cast to float32 to support the
range 0-1.
X = X.astype('float32') / 255.0

# As mentioned in lecture 3, we'll need to flatten X to input it into our basic neural network
X = X.reshape((len(X), -1)).T

# Y should be a 1-hot vector with a 1 indicating the digit class and 0 elsewhere
T = np.zeros((len(Y), 10), dtype='float32').T
for i in range(len(Y)):
    T[Y[i], i] = 1
```

In [377]:

```
def calculate_loss():
    global losses
    loss = np.sum((L3 - T)**2)/len(T.T)
    losses.append(loss)
    #print("[%04d] MSE Loss: %0.6f" % (i, loss))
```

In [378]:

```
def accuracy():
    global L3, accpct, accuracies
    predictions = np.zeros(L3.shape, dtype='float32')
    for j, m in enumerate(np.argmax(L3.T, axis=1)): predictions[m,j] = 1
    acc = np.sum(predictions*T)
    accpct = 100*acc/X.shape[1]
    accuracies.append(accpct)
```

In [379]:

```
def hw_update():
    global L1, L2, L3, dW1, dW2, dW3, hw1, hw2, hw3
    uw1, uw2, uw3 = np.dot(dW1, X.T), np.dot(dW2, L1.T), np.dot(dW3, L2.T)
    hw1.append(lr*np.abs(uw1).mean()), hw2.append(lr*np.abs(uw2).mean()), hw3.append(lr*np.abs(uw3)
.mean())
```

In [380]:

```
# Helper function to draw confusion matrix
def confusion_matrix():
    global X, L1, L2, L3, W1, W2, W3, dW1, dW2, dW3, accuracies, losses, accpct
    import seaborn
    os.makedirs('train', exist_ok=True)
    predictions = np.zeros(L3.shape, dtype='float32')
    for j, m in enumerate(np.argmax(L3.T, axis=1)): predictions[m,j] = 1
    acc = np.sum(predictions*T)
    accpct = 100*acc/X.shape[1]
    # accuracies.append(accpct)

    data = np.zeros((10,10,))
    for z, c in enumerate(np.argmax(T.T, axis=1)): data[c][np.argmax(predictions.T[z])] += 1
    for z, s in enumerate(data.sum(axis=0)): data[:,z] /= (s + 1e-5)
    seaborn.set_style("whitegrid", {'axes.grid' : False})
    seaborn.heatmap(data, annot=data*100, fmt='0.0f', cmap='Wistia')
    plt.xlabel('Actual'), plt.ylabel('Predicted'), plt.title('Confusion matrix (ACC %0.2f%%)' % acc
pct)
    plt.savefig(os.path.join('train', 'confusion-%03d-%r.png' % (i,lr)))
# edited train function name to include learning rate
    #plt.savefig(os.path.join('train', 'tsne-%03d-%r-%r.png' % (i, k,lr)))
    plt.show(), plt.close()
```

In [381]:

```
# Helper function to visualize training
def triple_plot():
    global X, L1, L2, L3, W1, W2, W3, dW1, dW2, dW3, c, losses, accpct, mean_activ, hw1, hw2, hw3
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12,3))
    testi = np.random.choice(range(60000))
    ax1.imshow(X.T[testi].reshape(28,28), cmap='gray')
    ax1.set_xticks([]), ax1.set_yticks([])
    cls = np.argmax(L3.T[testi])
    ax1.set_title("Prediction: %d confidence=%0.2f" % (cls, L3.T[testi][cls]/np.sum(L3.T[testi])))

    ax2.plot(losses, color='blue')
    ax2.set_title("Loss"), ax2.set_yscale('log')
    ax3.plot(accuracies, color='blue')
    ax3.set_ylim([0, 100])
    ax3.axhline(90, color='red', linestyle=':')      # Aim for 90% accuracy in 200 epochs
    ax3.set_title("Accuracy: %0.2f%%-%03d-%r" % (accpct,i, lr))
    #ax3.set_title("Accuracy: %0.2f%%-%03d-%r" % (accpct,i, k))
    plt.show(), plt.close()

    fig, ((ax1, ax2, ax3, ax4), (ax5, ax6, ax7, ax8), (ax9, ax10, ax11, ax12)) = plt.subplots(3, 4,
figsize=(10,10))
    ax1.imshow(np.reshape(L1.mean(axis=1), (16, 16,)), cmap='gray', interpolation='none'), ax1.set_
title('L1 $\mu$=%0.2f $\sigma$=%0.2f' % (L1.mean(), L1.std()))
    ax2.imshow(np.reshape(L2.mean(axis=1), (16, 8,)),  cmap='gray', interpolation='none'), ax2.set_
title('L2 $\mu$=%0.2f $\sigma$=%0.2f' % (L2.mean(), L2.std()))
    ax3.imshow(np.reshape(L3.mean(axis=1), (10, 1,)),  cmap='gray', interpolation='none'), ax3.set_
title('L3 $\mu$=%0.2f $\sigma$=%0.2f' % (L3.mean(), L3.std())), ax3.set_xticks([])
    activations = np.concatenate((L1.flatten(), L2.flatten(), L3.flatten()))
    try:
        ax4.hist(activations)
    except ValueError:
        pass
    ax4.set_title('Activation histogram')

    ax5.imshow(np.reshape(W1.mean(axis=0), (28, 28,)), cmap='gray', interpolation='none'), ax5.set_
title('W1 $\mu$=%0.2f $\sigma$=%0.2f' % (W1.mean(), W1.std()))
    ax6.imshow(np.reshape(W2.mean(axis=0), (16, 16,)), cmap='gray', interpolation='none'), ax6.set_
title('W2 $\mu$=%0.2f $\sigma$=%0.2f' % (W2.mean(), W2.std()))
    ax7.imshow(np.reshape(W3.mean(axis=0), (16, 8, )), cmap='gray', interpolation='none'), ax7.set_
title('W3 $\mu$=%0.2f $\sigma$=%0.2f' % (W3.mean(), W3.std())), ax7.set_xticks([])
    ax8.plot(accuracies, color='blue'), ax8.set_title("Accuracy: %0.2f%%-%03d-%r" % (accpct,i,lr)),
ax8.set_ylim(0, 100)

    uw1, uw2, uw3 = np.dot(dW1, X.T), np.dot(dW2, L1.T), np.dot(dW3, L2.T)
    hw1.append(lr*np.abs(uw1).mean()), hw2.append(lr*np.abs(uw2).mean()), hw3.append(lr*np.abs(uw3)
.mean())
    ax9.imshow(np.reshape(uw1.sum(axis=0),  (28, 28,)), cmap='gray', interpolation='none'), ax9.set
_title ('$\Delta$W1: %0.2f E-5' % (1e5 * lr * np.abs(uw1).mean()), color='r')
    ax10.imshow(np.reshape(uw2.sum(axis=0), (16, 16,)), cmap='gray', interpolation='none'), ax10.se
t_title('$\Delta$W2: %0.2f E-5' % (1e5 * lr * np.abs(uw2).mean()), color='g')
    ax11.imshow(np.reshape(uw3.sum(axis=0), (16, 8, )), cmap='gray', interpolation='none'), ax11.se
t_title('$\Delta$W3: %0.2f E-5' % (1e5 * lr * np.abs(uw3).mean()), color='b'), ax11.set_xticks([])

    ax12.plot(hw1, color='r', label='dW1'), ax12.plot(hw2, color='g', label='dW2'), ax12.plot(hw3,
color='b', label='dW3'), ax12.set_title('Weight update magnitude')
    ax12.legend(loc='upper right'), ax12.set_yscale('log')

    plt.suptitle("Weight and update visualization ACC: %0.2f%% LR=%0.8f" % (accpct, lr))
    plt.savefig(os.path.join('train', 'train-%03d-%r.png' % (i,lr)))                   # edited t
ain function name to include learning rate
    #plt.savefig(os.path.join('train', 'tsne-%03d-%r-%r.png' % (i, k,lr)))
    plt.show(), plt.close()
```

In [382]:

```
# Helper function to visualize via T-SNE
def tsne_viz():
    global X, L1, L2, L3, W1, W2, W3, dW1, dW2, dW3, accuracies, losses
    from matplotlib import cm
    import sklearn.manifold
    colors = iter(cm.rainbow(np.linspace(0, 1, 10)))
    X_embedded = sklearn.manifold.TSNE(n_components=2).fit_transform(L2.T[:500])
    for digit in range(10):
```

```
        xx = X_embedded[Y[:500] == digit, 0]
        yy = X_embedded[Y[:500] == digit, 1]
        plt.scatter(xx, yy, c=[next(colors)], label=digit)
        t = plt.text(np.median(xx), np.median(yy), digit, fontsize=24)
        t.set_bbox({'facecolor': 'white', 'alpha': 0.75})
    plt.title('T-SNE viz - Accuracy: %0.2f%%' % accpct), plt.legend()
    plt.savefig(os.path.join('train', 'tsne-%03d-%r.png' % (i, lr)))            # edited t
ain function name to include learning rate
    #plt.savefig(os.path.join('train', 'tsne-%03d-%r-%r.png' % (i, k,lr)))
    plt.show(), plt.close()
```

In [383]:

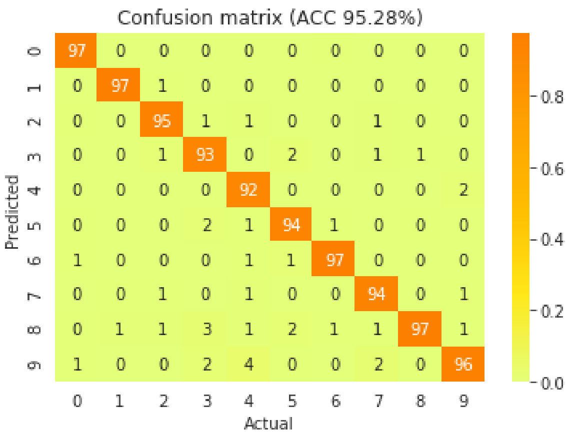
```
def checknan(x):
    # Checks if matrix contains NaN
    return np.any(np.isnan(x))
```

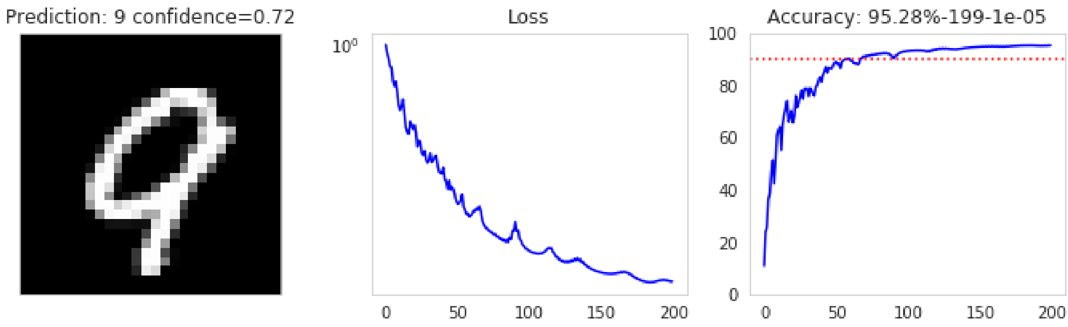
In [ ]:

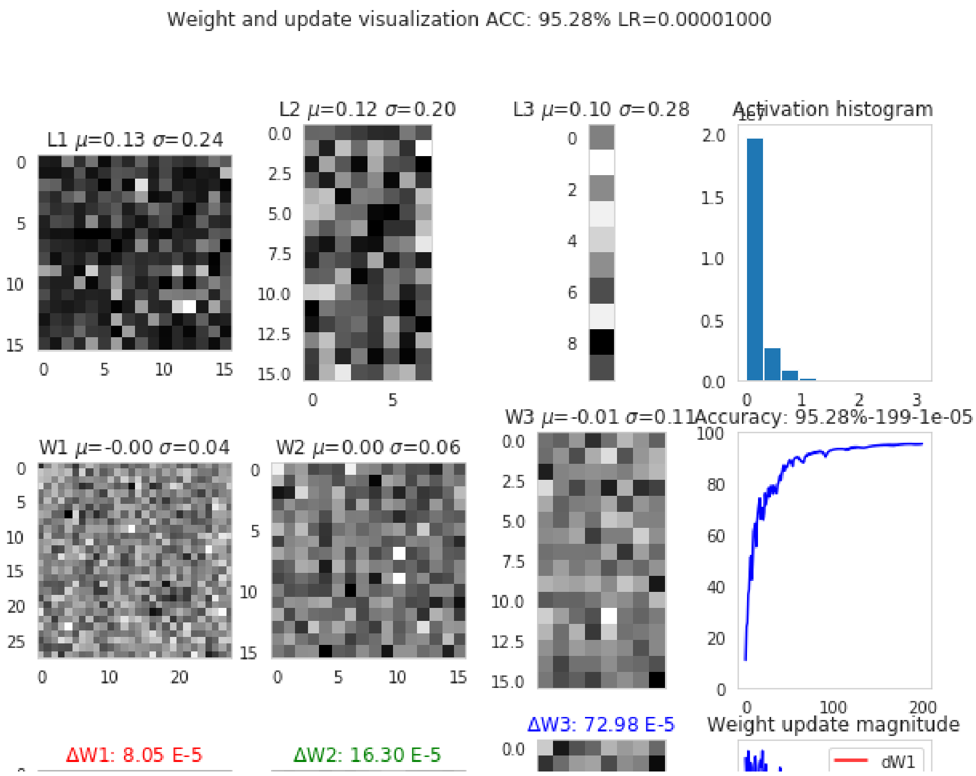
In [384]:

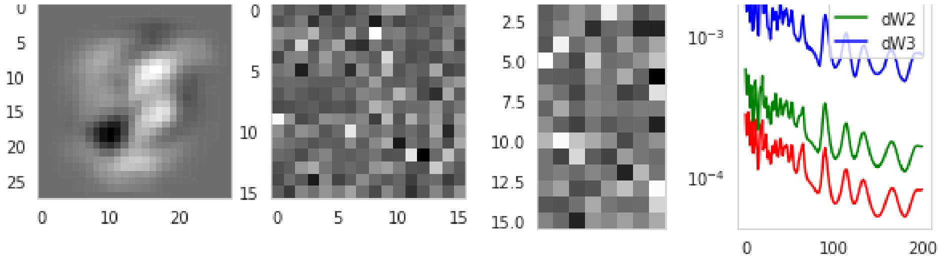
```
def relu(x): return np.maximum(x, 0)
def drelu(x): return 1. * (x>0)

# Define forward pass
def forward_pass_relu(X, W1, W2, W3):
    L1 = relu(W1.dot(X))
    L2 = relu(W2.dot(L1))
    L3 = relu(W3.dot(L2))
    return L1, L2, L3

# Define backward pass
def backward_pass_relu(L1, L2, L3, W1, W2, W3):
    dW3 = (L3 - T) * drelu(L3)
    dW2 = W3.T.dot(dW3) * drelu(L2)
    dW1 = W2.T.dot(dW2) * drelu(L1)
    return dW1, dW2, dW3

def update_weights_relu(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2):
    W3 -= lr*np.dot(dW3, L2.T)
    W2 -= lr*np.dot(dW2, L1.T)
    W1 -= lr*np.dot(dW1, X.T)
    return W1, W2, W3

# Monitoring variables
accuracies = []
losses     = []
mean_activ = []
hw1, hw2, hw3 = [], [], []

#%% Setup: 784 -> 256 -> 128 -> 10
W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)

# Main loop
from IPython.display import clear_output
from tensorflow.keras.utils import Progbar
progbar = Progbar(200)

# Learning rate, decrease if optimization isn't working
lr = 1e-5

for i in range(200):
    L1, L2, L3     = forward_pass_relu(X, W1, W2, W3)
    dW1, dW2, dW3 = backward_pass_relu(L1, L2, L3, W1, W2, W3)
    W1, W2, W3     = update_weights_relu(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)

    progbar.update(i % 200)
    calculate_loss()
    accuracy()
    hw_update()
```

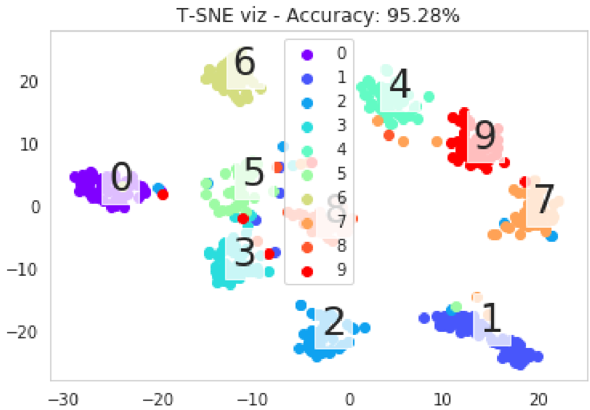
```
    if i == 199:
        if checknan(L1) or checknan(L2) or checknan(L3):
            print('\nNaN encountered in activations. Try lowering the learning rate and/or
correcting bugs in your code.')
            print('Optimization halted')
            break
        clear_output(wait=True)
        confusion_matrix()
        triple_plot()
        tsne_viz()


        #losses_dict[i] =losses #store losses for this round with the learning rate
        #accuracy_dict[i] =accuracies #store losses for this round with the learning rate
```











In [ ]:

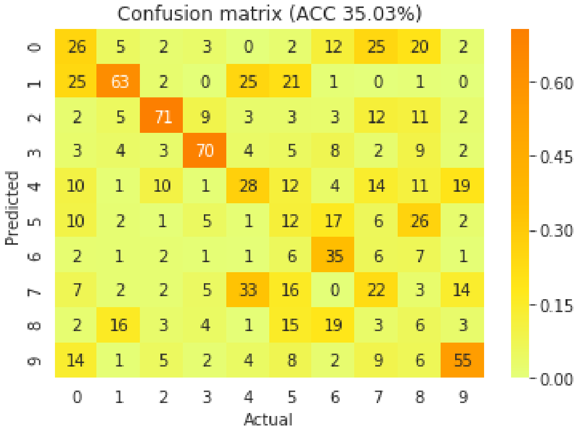
In [ ]:

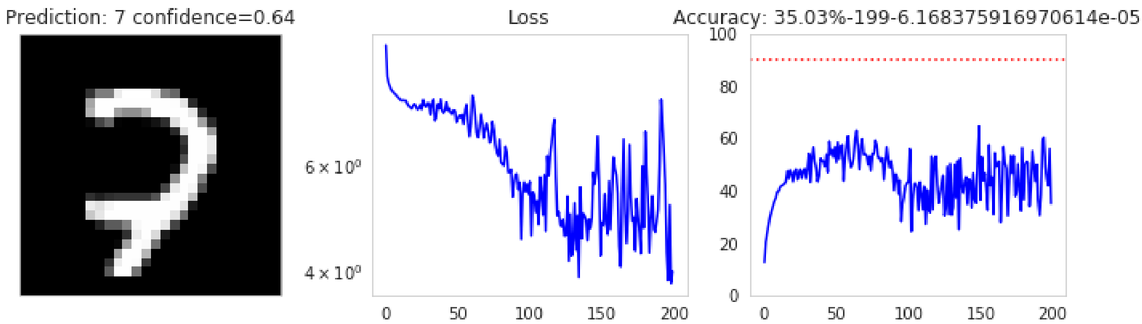
Extra Credit - Cosine learning rate

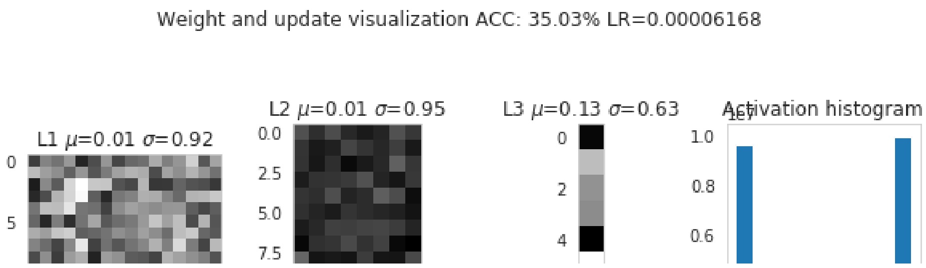
In [368]:

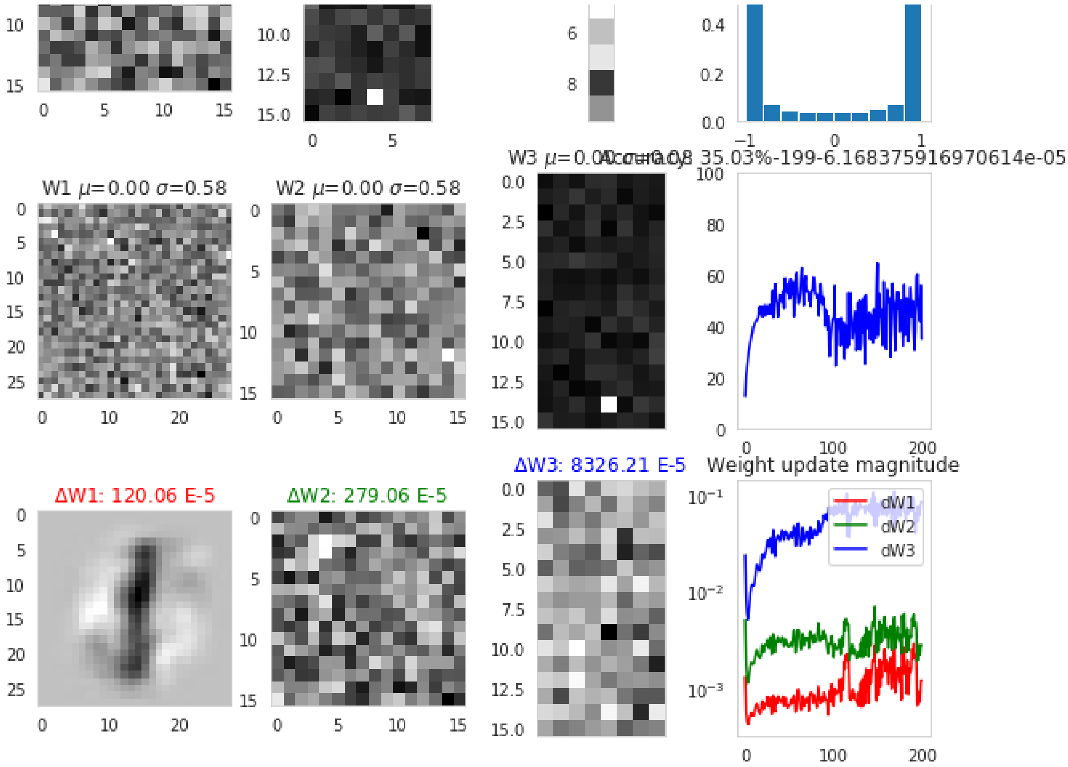
```
# Main loop
from IPython.display import clear_output
from tensorflow.keras.utils import Progbar
progbar = Progbar(200)

# Learning rate, decrease if optimization isn't working

#learning_rate_array = [3e-5, 2e-5, 1e-5, 1e-3, 1e-1] # created list of learning rates to try
losses_dict = {} # create dictionary to store resulting loss for learning rate trials to plot with
later
lr_accuracy_rate = {}


for j in range(1):

    # set updated learning rate each loop
    lr_min = 0
    lr_max = 1
    Epochs = 200
    cycles = 1


    lr = lr_min + 0.5 * (lr_max - lr_min) * (1 + np.cos((i / (Epochs/cycles)) * np.pi))

    # reinitialize weights to random %% Setup: 784 -> 256 -> 128 -> 10
    W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
```

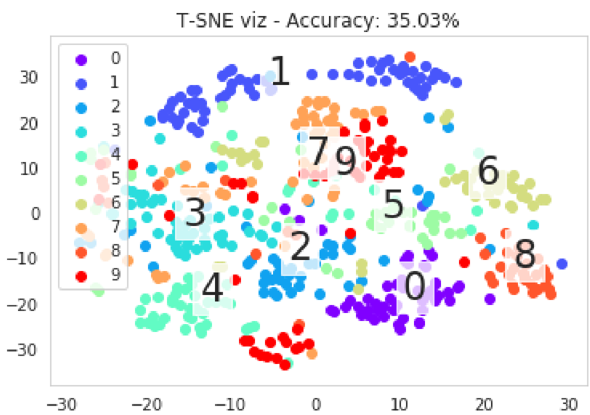
```
    W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
    W3 = 2*np.random.rand(128,  10).astype('float32').T - 1

    # reinitialize Monitoring variables to empty
    accuracies = []
    losses     = []
    mean_activ = []
    hw1, hw2, hw3 = [], [], []
    #progbar.update(i % 5)

    # optimize NN
    for i in range(200):
        L1, L2, L3     = forward_pass(X, W1, W2, W3)
        dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3)
        W1, W2, W3     = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)

        progbar.update(i % 200)
        calculate_loss()
        accuracy()
        hw_update()
        losses_dict[j] =losses #store losses for this round with the learning rate


        if i == 199:
            if checknan(L1) or checknan(L2) or checknan(L3):
                print('\nNaN encountered in activations. Try lowering the learning rate and/or correcting bugs in your code.')
                print('Optimization halted')
                break
            clear_output(wait=True)
            confusion_matrix()
            triple_plot()
            tsne_viz()
            lr_accuracy_rate[j] =accuracies #store losses for this round with the learning rate
```











In [ ]:

```

```

In [ ]:

```

```

Original Code

- with modifications made at each header corresponding to the problem number

In [252]:

```
import os
import numpy as np
import matplotlib.pyplot as plt
```

In [253]:

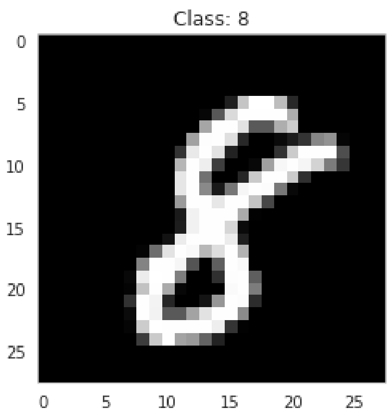
```
# Load MNIST dataset
from tensorflow.keras.datasets import mnist
# X is an array of 60,000 28x28 black and white images with pixel values from 0-255
# Y is an array of 60,000 labels with values from 0-9 denoting the image class
(X, Y), (_, _) = mnist.load_data()
```

In [254]:

```
# Show some random example images from X
random_digit = np.random.randint(0, len(X) - 1)
plt.imshow(X[random_digit], cmap='gray')
# Show image class
plt.title('Class: %01d' % Y[random_digit])
```

Out[254]:

```
Text(0.5, 1.0, 'Class: 8')
```



In [255]:

```
# Do some basic transforms of data

# We want the pixels in X to have values between 0-1. We'll need to cast to float32 to support the
range 0-1.
X = X.astype('float32') / 255.0

# As mentioned in lecture 3, we'll need to flatten X to input it into our basic neural network
X = X.reshape((len(X), -1)).T

# Y should be a 1-hot vector with a 1 indicating the digit class and 0 elsewhere
T = np.zeros((len(Y), 10), dtype='float32').T
for i in range(len(Y)):
    T[Y[i], i] = 1
```

Weight definitions

Let's define our basic 3-layer network.

$W1, W2, W3$ are the randomly initialized weight matrices.

We cast to float32 for efficiency as the default type for np.random.rand is 64 bit, and that's slower.

In [256]:

```
#%% Setup: 784 -> 256 -> 128 -> 10
W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
```

In [257]:

```
# Define basic sigmoid activation
```

```
def sigmoid(x): return 1.0/(1.0 + np.e**-x)

# Add other activation functions here
```

Core functions

The next 4 functions are the heart of a basic neural network

- Forward pass: calculates activations
- Backward pass: calculates the gradients
- Update weights: applies the gradient updates to the weight matrices
- Calculate loss: reports how we're doing

In [258]:

```
def forward_pass(X, W1, W2, W3):
    L1 = sigmoid(W1.dot(X))
    L2 = sigmoid(W2.dot(L1))
    L3 = sigmoid(W3.dot(L2))
    return L1, L2, L3
```

In [259]:

```
def backward_pass(L1, L2, L3, W1, W2, W3):
    dW3 = (L3 - T) * L3*(1 - L3)
    dW2 = W3.T.dot(dW3)*(L2*(1-L2))
    dW1 = W2.T.dot(dW2)*(L1*(1-L1))
    return dW1, dW2, dW3
```

In [260]:

```
def update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2):
    W3 -= lr*np.dot(dW3, L2.T)
    W2 -= lr*np.dot(dW2, L1.T)
    W1 -= lr*np.dot(dW1, X.T)
    return W1, W2, W3
```

In [261]:

```
# Monitoring variables
accuracies = []
losses     = []
mean_activ = []
hw1, hw2, hw3 = [], [], []
```

In [262]:

```
def calculate_loss():
    global losses
    loss = np.sum((L3 - T)**2)/len(T.T)
    losses.append(loss)
    #print("[%04d] MSE Loss: %0.6f" % (i, loss))
```

In [263]:

```
def accuracy():
    global L3, accpct, accuracies
    predictions = np.zeros(L3.shape, dtype='float32')
    for j, m in enumerate(np.argmax(L3.T, axis=1)): predictions[m,j] = 1
    acc = np.sum(predictions*T)
    accpct = 100*acc/X.shape[1]
    accuracies.append(accpct)
```

In [264]:

```
def hw_update():
```

```
    global L1, L2, L3, dW1, dW2, dW3, hw1, hw2, hw3
    uw1, uw2, uw3 = np.dot(dW1, X.T), np.dot(dW2, L1.T), np.dot(dW3, L2.T)
    hw1.append(lr*np.abs(uw1).mean()), hw2.append(lr*np.abs(uw2).mean()), hw3.append(lr*np.abs(uw3)
.mean())
```

In [265]:

```
# Helper function to draw confusion matrix
def confusion_matrix():
    global X, L1, L2, L3, W1, W2, W3, dW1, dW2, dW3, accuracies, losses, accpct
    import seaborn
    os.makedirs('train', exist_ok=True)
    predictions = np.zeros(L3.shape, dtype='float32')
    for j, m in enumerate(np.argmax(L3.T, axis=1)): predictions[m,j] = 1
    acc = np.sum(predictions*T)
    accpct = 100*acc/X.shape[1]
    # accuracies.append(accpct)

    data = np.zeros((10,10,))
    for z, c in enumerate(np.argmax(T.T, axis=1)): data[c][np.argmax(predictions.T[z])] += 1
    for z, s in enumerate(data.sum(axis=0)): data[:,z] /= (s + 1e-5)
    seaborn.set_style("whitegrid", {'axes.grid' : False})
    seaborn.heatmap(data, annot=data*100, fmt='0.0f', cmap='Wistia')
    plt.xlabel('Actual'), plt.ylabel('Predicted'), plt.title('Confusion matrix (ACC %0.2f%%)' % acc
pct)
    plt.savefig(os.path.join('train', 'confusion-%03d-%r.png' % (i,lr)))
# edited train function name to include learning rate
    #plt.savefig(os.path.join('train', 'tsne-%03d-%r-%r.png' % (i, k,lr)))
    plt.show(), plt.close()
```

In [266]:

```
# Helper function to visualize training
def triple_plot():
    global X, L1, L2, L3, W1, W2, W3, dW1, dW2, dW3, c, losses, accpct, mean_activ, hw1, hw2, hw3
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12,3))
    testi = np.random.choice(range(60000))
    ax1.imshow(X.T[testi].reshape(28,28), cmap='gray')
    ax1.set_xticks([]), ax1.set_yticks([])
    cls = np.argmax(L3.T[testi])
    ax1.set_title("Prediction: %d confidence=%0.2f" % (cls, L3.T[testi][cls]/np.sum(L3.T[testi])))

    ax2.plot(losses, color='blue')
    ax2.set_title("Loss"), ax2.set_yscale('log')
    ax3.plot(accuracies, color='blue')
    ax3.set_ylim([0, 100])
    ax3.axhline(90, color='red', linestyle=':')     # Aim for 90% accuracy in 200 epochs
    ax3.set_title("Accuracy: %0.2f%%-%03d-%r" % (accpct,i, lr))
    #ax3.set_title("Accuracy: %0.2f%%-%03d-%r" % (accpct,i, k))
    plt.show(), plt.close()

    fig, ((ax1, ax2, ax3, ax4), (ax5, ax6, ax7, ax8), (ax9, ax10, ax11, ax12)) = plt.subplots(3, 4,
figsize=(10,10))
    ax1.imshow(np.reshape(L1.mean(axis=1), (16, 16,)), cmap='gray', interpolation='none'), ax1.set_
title('L1 $\mu$=%0.2f $\sigma$=%0.2f' % (L1.mean(), L1.std()))
    ax2.imshow(np.reshape(L2.mean(axis=1), (16, 8,)),  cmap='gray', interpolation='none'), ax2.set_
title('L2 $\mu$=%0.2f $\sigma$=%0.2f' % (L2.mean(), L2.std()))
    ax3.imshow(np.reshape(L3.mean(axis=1), (10, 1,)),  cmap='gray', interpolation='none'), ax3.set_
title('L3 $\mu$=%0.2f $\sigma$=%0.2f' % (L3.mean(), L3.std())), ax3.set_xticks([])
    activations = np.concatenate((L1.flatten(), L2.flatten(), L3.flatten()))
    try:
        ax4.hist(activations)
    except ValueError:
        pass
    ax4.set_title('Activation histogram')

    ax5.imshow(np.reshape(W1.mean(axis=0), (28, 28,)), cmap='gray', interpolation='none'), ax5.set_
title('W1 $\mu$=%0.2f $\sigma$=%0.2f' % (W1.mean(), W1.std()))
    ax6.imshow(np.reshape(W2.mean(axis=0), (16, 16,)), cmap='gray', interpolation='none'), ax6.set_
title('W2 $\mu$=%0.2f $\sigma$=%0.2f' % (W2.mean(), W2.std()))
    ax7.imshow(np.reshape(W3.mean(axis=0), (16, 8, )), cmap='gray', interpolation='none'), ax7.set_
title('W3 $\mu$=%0.2f $\sigma$=%0.2f' % (W3.mean(), W3.std())), ax7.set_xticks([])
    ax8.plot(accuracies, color='blue'), ax8.set_title("Accuracy: %0.2f%%-%03d-%r" % (accpct,i,lr)),
ax8.set_ylim(0, 100)
```

```
    uw1, uw2, uw3 = np.dot(dW1, X.T), np.dot(dW2, L1.T), np.dot(dW3, L2.T)
    hw1.append(lr*np.abs(uw1).mean()), hw2.append(lr*np.abs(uw2).mean()), hw3.append(lr*np.abs(uw3)
.mean())
    ax9.imshow(np.reshape(uw1.sum(axis=0),  (28, 28,)), cmap='gray', interpolation='none'), ax9.set
_title ('$\Delta$W1: %0.2f E-5' % (1e5 * lr * np.abs(uw1).mean()), color='r')
    ax10.imshow(np.reshape(uw2.sum(axis=0), (16, 16,)), cmap='gray', interpolation='none'), ax10.se
t_title('$\Delta$W2: %0.2f E-5' % (1e5 * lr * np.abs(uw2).mean()), color='g')
    ax11.imshow(np.reshape(uw3.sum(axis=0), (16, 8, )), cmap='gray', interpolation='none'), ax11.se
t_title('$\Delta$W3: %0.2f E-5' % (1e5 * lr * np.abs(uw3).mean()), color='b'), ax11.set_xticks([])

    ax12.plot(hw1, color='r', label='dW1'), ax12.plot(hw2, color='g', label='dW2'), ax12.plot(hw3,
color='b', label='dW3'), ax12.set_title('Weight update magnitude')
    ax12.legend(loc='upper right'), ax12.set_yscale('log')

    plt.suptitle("Weight and update visualization ACC: %0.2f%% LR=%0.8f" % (accpct, lr))
    plt.savefig(os.path.join('train', 'train-%03d-%r.png' % (i,lr)))                    # edited t
ain function name to include learning rate
    #plt.savefig(os.path.join('train', 'tsne-%03d-%r-%r.png' % (i, k,lr)))
    plt.show(), plt.close()
```

In [267]:

```
# Helper function to visualize via T-SNE
def tsne_viz():
    global X, L1, L2, L3, W1, W2, W3, dW1, dW2, dW3, accuracies, losses
    from matplotlib import cm
    import sklearn.manifold
    colors = iter(cm.rainbow(np.linspace(0, 1, 10)))
    X_embedded = sklearn.manifold.TSNE(n_components=2).fit_transform(L2.T[:500])
    for digit in range(10):
        xx = X_embedded[Y[:500] == digit, 0]
        yy = X_embedded[Y[:500] == digit, 1]
        plt.scatter(xx, yy, c=[next(colors)], label=digit)
        t = plt.text(np.median(xx), np.median(yy), digit, fontsize=24)
        t.set_bbox({'facecolor': 'white', 'alpha': 0.75})
    plt.title('T-SNE viz - Accuracy: %0.2f%%' % accpct), plt.legend()
    plt.savefig(os.path.join('train', 'tsne-%03d-%r.png' % (i, lr)))                    # edited t
ain function name to include learning rate
    #plt.savefig(os.path.join('train', 'tsne-%03d-%r-%r.png' % (i, k,lr)))
    plt.show(), plt.close()
```

In [268]:

```
def checknan(x):
    # Checks if matrix contains NaN
    return np.any(np.isnan(x))
```

In [279]:

```
%matplotlib inline
```

In [278]:

```
#%%javascript
#IPython.OutputArea.prototype._should_scroll = function(lines) { return false; }
```

2.1 Main loop - Adjust Learning Rate

Iterate through the array of various learning rates and capure each of their outputs to validate which learning rate is optimal base don loss and accuracy charts

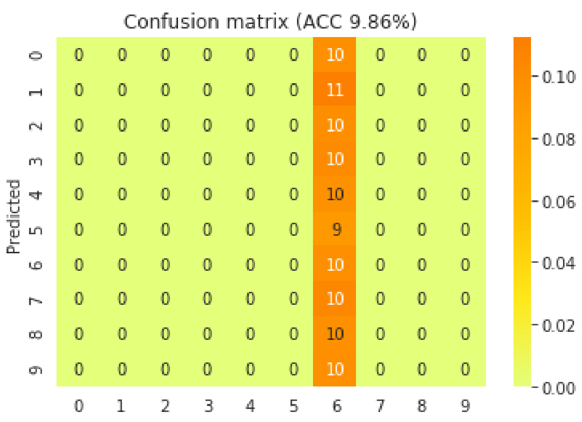
- Best iteration = 3e^5 with 71% accuracy

This is the main training loop. The three key lines are the forward pass, backward pass and weight updates. Hint: If your optimization

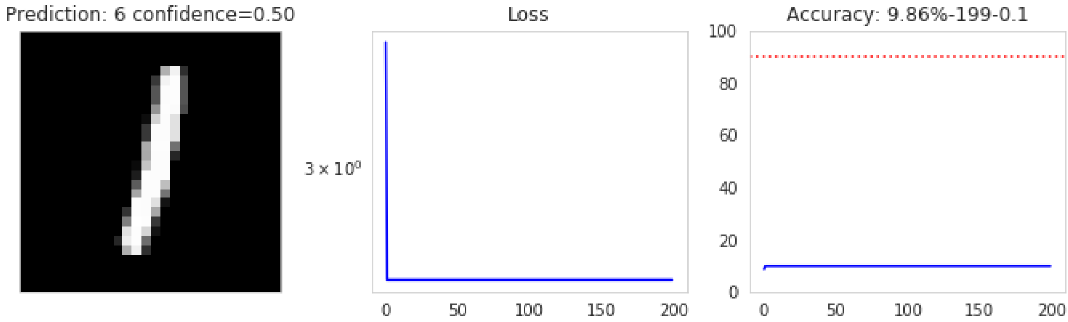
This is the main training loop. The three key lines are the forward pass, backward pass and weight updates. Hint: if your optimization is blowing up or getting NaNs, try decreasing the learning rate.

In [280]:

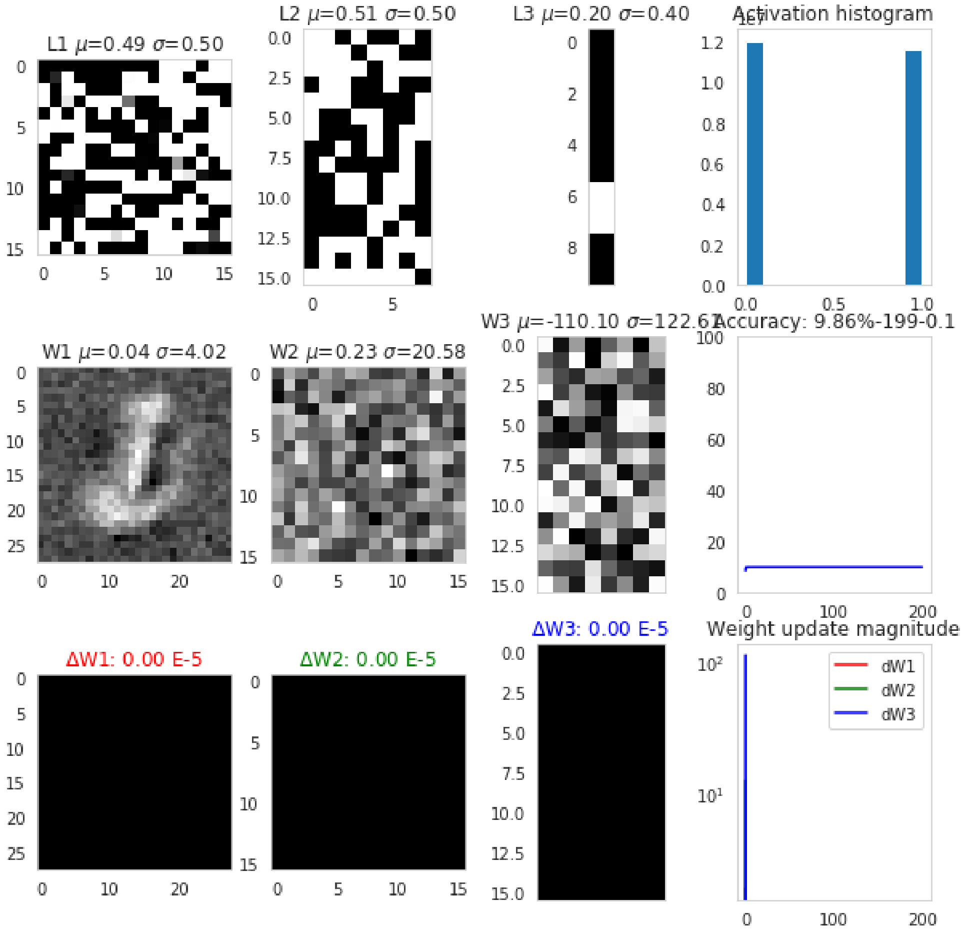
```
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# # Learning rate, decrease if optimization isn't working
#
# learning_rate_array = [3e-5, 2e-5, 1e-5, 1e-3, 1e-1] # created list of learning rates to try
# losses_dict = {} # create dictionary to store resulting loss for learning rate trials to plot wi
th later
# lr_accuracy_rate = {}
#
#
# for j in range(len(learning_rate_array)):
#
#     # set updated learning rate each loop
#     lr = learning_rate_array[j]
#
#     # reinitialize weights to random %% Setup: 784 -> 256 -> 128 -> 10
#     W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
#     W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
#     W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
#
#     # reinitialize Monitoring variables to empty
#     accuracies = []
#     losses     = []
#     mean_activ = []
#     hw1, hw2, hw3 = [], [], []
#     #progbar.update(i % 5)
#
#     # optimize NN
#     for i in range(200):
#         L1, L2, L3     = forward_pass(X, W1, W2, W3)
#         dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3)
#         W1, W2, W3     = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#         progbar.update(i % 200)
#         calculate_loss()
#         accuracy()
#         hw_update()
#         losses_dict[j] =losses #store losses for this round with the learning rate
#
#
#         if i == 199:
#             if checknan(L1) or checknan(L2) or checknan(L3):
#                 print('\nNaN encountered in activations. Try lowering the learning rate and/or co
rrecting bugs in your code.')
#                 print('Optimization halted')
#                 break
#             clear_output(wait=True)
#             confusion_matrix()
#             triple_plot()
#             tsne_viz()
#             lr_accuracy_rate[j] =accuracies #store losses for this round with the learning rate
#
```

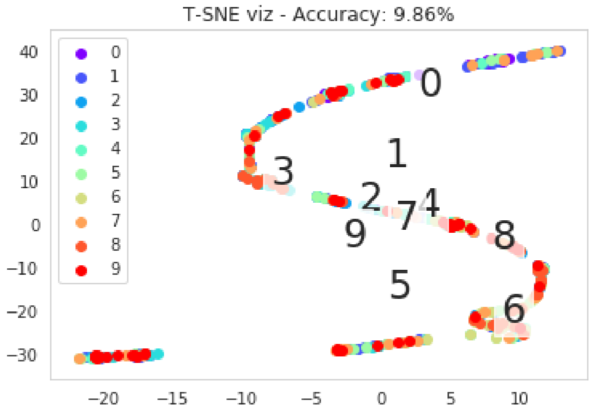


Actual



Weight and update visualization ACC: 9.86% LR=0.10000000





Problem 2.1: Learning Rate Loss Line Chart

In [281]:

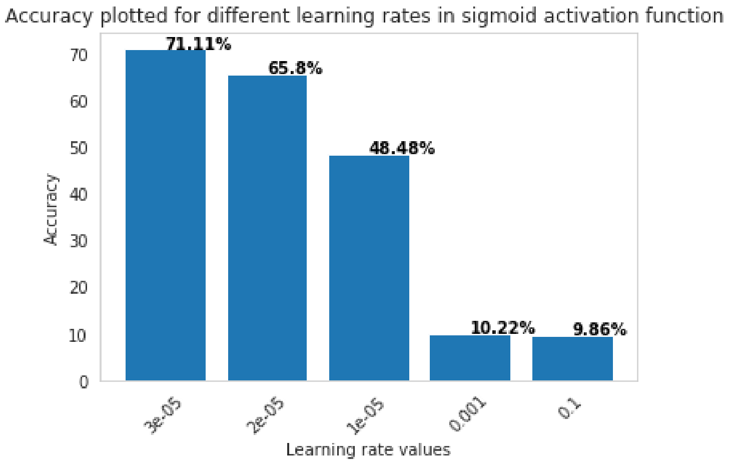
```
# plot all the different learning rates on 1 chart
# for k in range(len(losses_dict)):
#     line = plt.plot(losses_dict[k], label = str(learning_rate_array[k]))
#     plt.title("Losses plotted for various learning rates ")
#     plt.legend()
#     plt.ylim(0,5)
# plt.show()
#
# View Individual Charts
#for k in range(len(losses_dict)):
#    line = plt.plot(losses_dict[k], label = str(learning_rate_array[k]))
#    plt.title("Losses plotted for various learning rates ")
#    plt.legend()
#    plt.show()
```

Problem 2.1: Learning Rate Accuracy Bar Plot

In [305]:

```
# a_list = []
# for i in range(len(learning_rate_array)):
#     a_list.append(lr_accuracy_rate[i][len(lr_accuracy_rate[i])-1])
#
# # bar plot of accuracy
#
# import numpy as np
# import matplotlib.pyplot as plt
# from itertools import chain # library required to unchain
#
# height = list(a_list) # unchain the dictionary accuracy value
# bars =
learning_rate_array[0],learning_rate_array[1],learning_rate_array[2],learning_rate_array[3],learni
te_array[4]
# y_pos = np.arange(len(bars))
#
# x_labels = list(a_list)
#
# plt.bar(y_pos, height)# Create bars
# #ax = accuracy_dict.plot(kind='bar')
# plt.xticks(y_pos, bars)# Create names on the x-axis
# plt.xlabel('Learning rate values')
# #plt.set_xticklabels(x_labels)
# plt.ylabel('Accuracy')
```

```
# plt.title("Accuracy plotted for different learning rates in sigmoid activation function ")
# plt.xticks(rotation=45)
#
# for i, v in enumerate(x_labels):
#     plt.text(i, v, str(round(v,2))+"%", color='black', fontweight='bold')
#
#
# plt.show() # Show graphic
```



2.2 Main loop - Activation Function

----

2 - Activation function: Try changing the sigmoid function to “1.0/(1.0 + np.e* -(*k*x))”, where k is another training parameter. Explain what k does.

What is the effect of a small k on training versus a larger value for k? Is there an optimal k for a given learning rate? Use the default learning rate “1e-5” for your experiments.

Justify your position in words and show up to 5 plots.

Updating sigmoid funciton to take k parameter & updating helperfunction to send in k parameter to sigmoid function

In [310]:

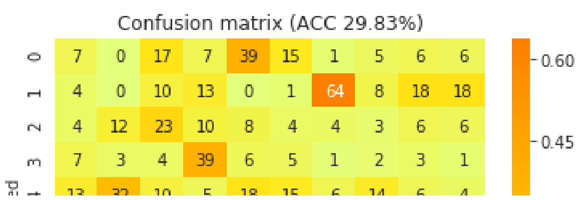
```
# # Define basic sigmoid activation
# def sigmoid(x): return 1.0/(1.0 + np.e**-x)
#
# def sigmoid(x,k): return 1.0/(1.0 + np.e**-(k*x)) #sigmoid with k parameter
#
# # update forward pass as it is using sigmoid
# def forward_pass(X, W1, W2, W3, k):
#     L1 = sigmoid(W1.dot(X), k)
#     L2 = sigmoid(W2.dot(L1), k)
#     L3 = sigmoid(W3.dot(L2), k)
#     return L1, L2, L3
# # update backward pass as it needs to include * by k because e^kx = ke^kx
# def backward_pass(L1, L2, L3, W1, W2, W3, k):
#     dW3 = (L3 - T) * L3*(1 - L3) * k
#     dW2 = W3.T.dot(dW3)*(L2*(1-L2)) * k
#     dW1 = W2.T.dot(dW2)*(L1*(1-L1)) * k
#     return dW1, dW2, dW3
```

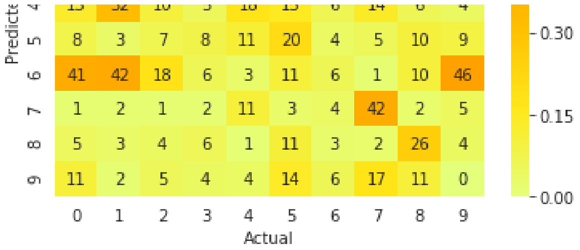
2.2 Main loop - Activation Function

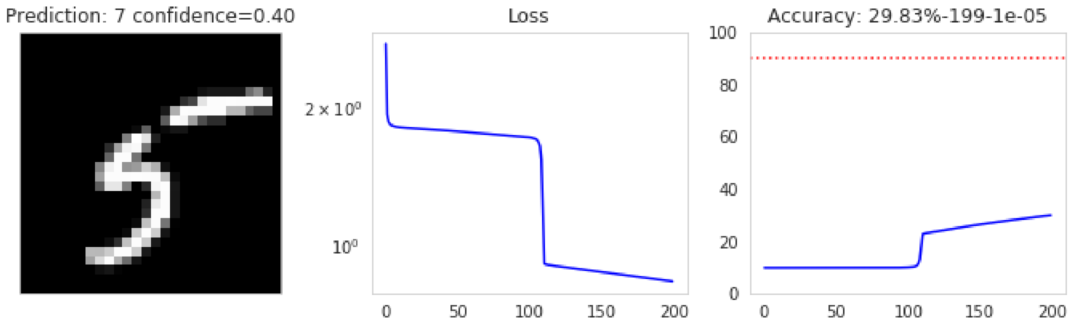
- Best iteration = 1e^5 with 57% accuracy

In [312]:

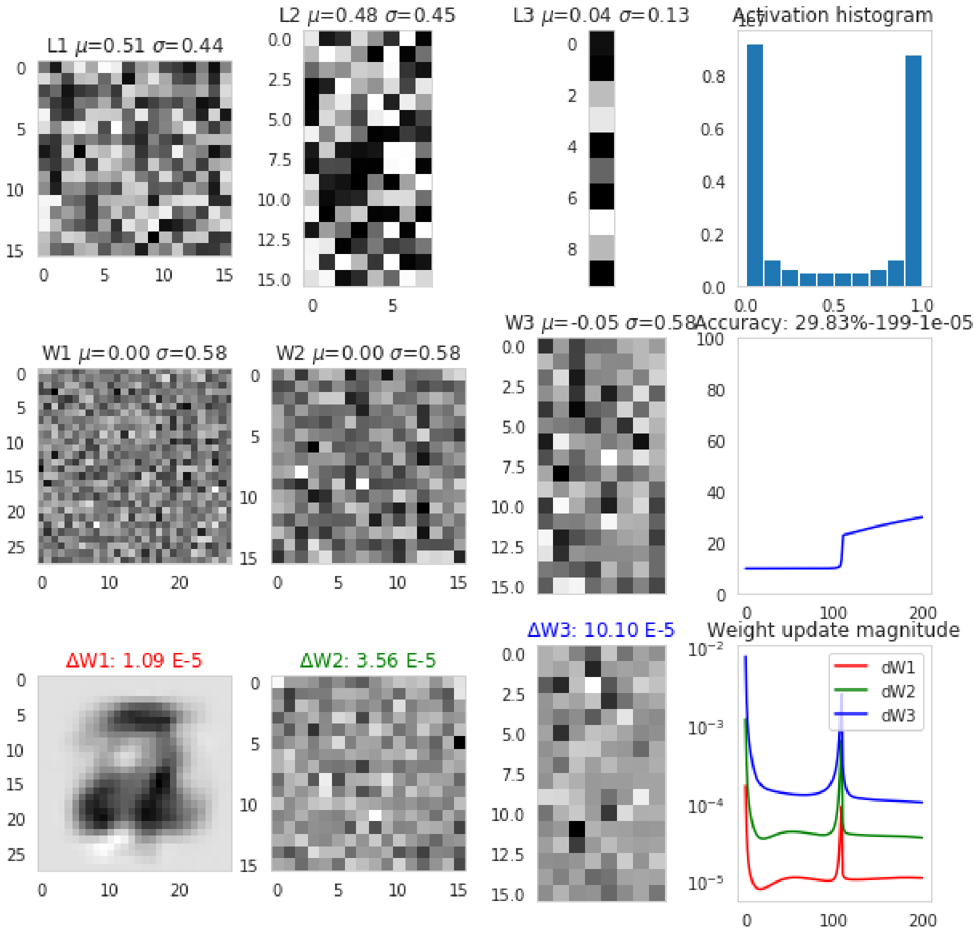
```
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# # Activation functions k parameter tuning
# k_test_values = [.1, .4, .5, 1, 1.2]
#
# # default learning rate = 1e-5
# lr = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
#
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# for j in range(len(k_test_values)):
#
#     # set k in each loop
#     k = k_test_values[j]
#
#     # reinitialize weights to random %% Setup: 784 -> 256 -> 128 -> 10
#     W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
#     W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
#     W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
#
#     # reinitialize Monitoring variables to empty
#     accuracies = []
#     losses     = []
#     mean_activ = []
#     hw1, hw2, hw3 = [], [], []
#     # progbar.update(i % 5)
#
#     # optimize NN
#     for i in range(200):
#         L1, L2, L3    = forward_pass(X, W1, W2, W3, k)
#         dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3, k)
#         W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#         progbar.update(i % 200)
#         calculate_loss()
#         accuracy()
#         hw_update()
#
#         if i == 199:
#             if checknan(L1) or checknan(L2) or checknan(L3):
#                 print('\nNaN encountered in activations. Try lowering the learning rate and/or co
rrecting bugs in your code.')
#                 print('Optimization halted')
#                 break
#             clear_output(wait=True)
#             confusion_matrix()
#             triple_plot()
#             tsne_viz()
#
#         losses_dict[j] =losses #store losses for this round with the learning rate
#         accuracy_dict[j] =accuracies #store losses for this round with the learning rate
```

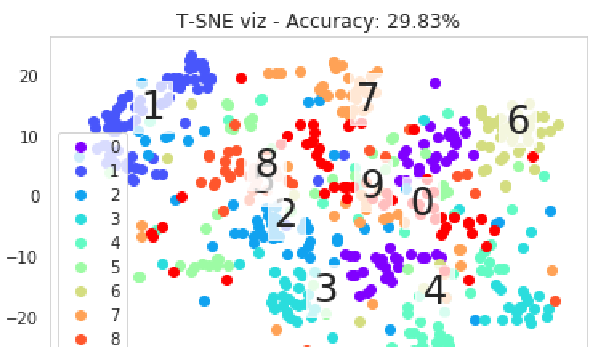


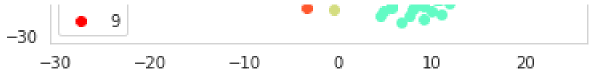




Weight and update visualization ACC: 29.83% LR=0.00001000



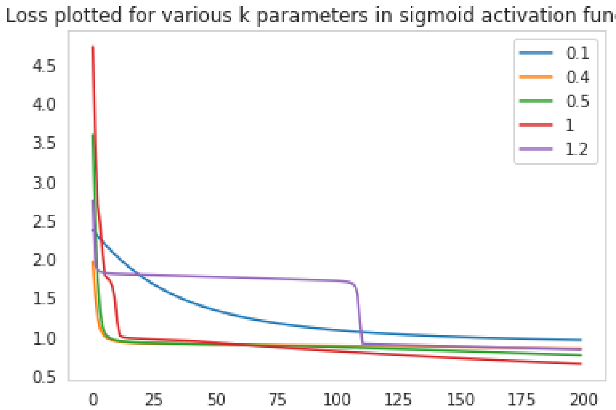




Problem 2.2: Line Chart to determine which k parameter has the best loss

In [313]:

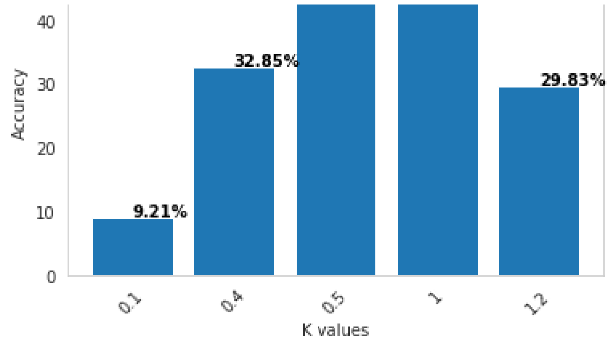
```
# #plot all the different k parameters : k=.5 looks the best!
# for m in range(len(k_test_values)):
#     plt.plot(losses_dict[m], label = str(k_test_values[m]))
#     plt.title("Loss plotted for various k parameters in sigmoid activation function ")
#     plt.legend()
# plt.show()
#
```



Problem 2.2: Bar Plot to determine which k parameter has the highest accuracy

In [315]:

```
# a_list = []
# for i in range(len(k_test_values)):
#     a_list.append(accuracy_dict[i][len(lr_accuracy_rate[i])-1])
#
# # bar plot of accuracy
#
# import numpy as np
# import matplotlib.pyplot as plt
# from itertools import chain # library required to unchain
#
# height = list(a_list) # unchain the dictionary accuracy value
# bars =  k_test_values[0],k_test_values[1],k_test_values[2],k_test_values[3],k_test_values[4]
# y_pos = np.arange(len(bars))
#
# x_labels = list(a_list)
#
# plt.bar(y_pos, height)# Create bars
# #ax = accuracy_dict.plot(kind='bar')
# plt.xticks(y_pos, bars)# Create names on the x-axis
# plt.xlabel('K values')
# #plt.set_xticklabels(x_labels)
# plt.ylabel('Accuracy')
# plt.title("Accuracy plotted for different k values for sigmoid activation function ")
# plt.xticks(rotation=45)
#
# for i, v in enumerate(x_labels):
#     plt.text(i, v, str(round(v,2))+"%", color='black', fontweight='bold')
#
#
# plt.show() # Show graphic
```



In [ ]:

Problem 2.3: Initialization with sigmoid activation function

In [317]:

```
# # Define basic sigmoid activation
# def sigmoid(x): return 1.0/(1.0 + np.e**-x)
#
# def sigmoid(x,k): return 1.0/(1.0 + np.e**-(k*x)) #sigmoid with k parameter
#
# # update forward pass as it is using sigmoid
# def forward_pass(X, W1, W2, W3, k):
#     L1 = sigmoid(W1.dot(X), k)
#     L2 = sigmoid(W2.dot(L1), k)
#     L3 = sigmoid(W3.dot(L2), k)
#     return L1, L2, L3
#
#
# # update backward pass as it needs to include * by k because e^kx = ke^kx
# def backward_pass(L1, L2, L3, W1, W2, W3, k):
#     dW3 = (L3 - T) * L3*(1 - L3) * k
#     dW2 = W3.T.dot(dW3)*(L2*(1-L2)) * k
#     dW1 = W2.T.dot(dW2)*(L1*(1-L1)) * k
#     return dW1, dW2, dW3
```

Problem 2.3 Main loop - Initialize with Random, normal, uniform, and poisson

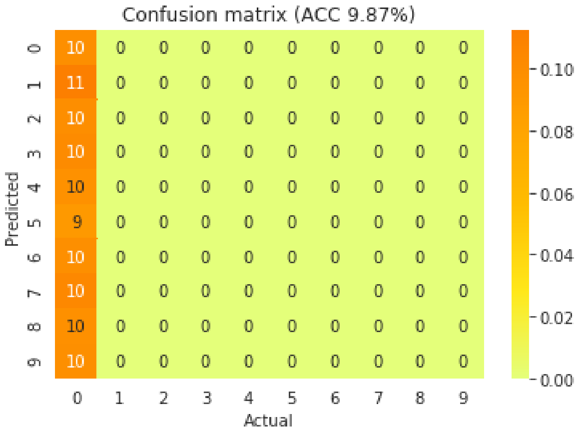
- The best is normal distribution with 78% accuracy

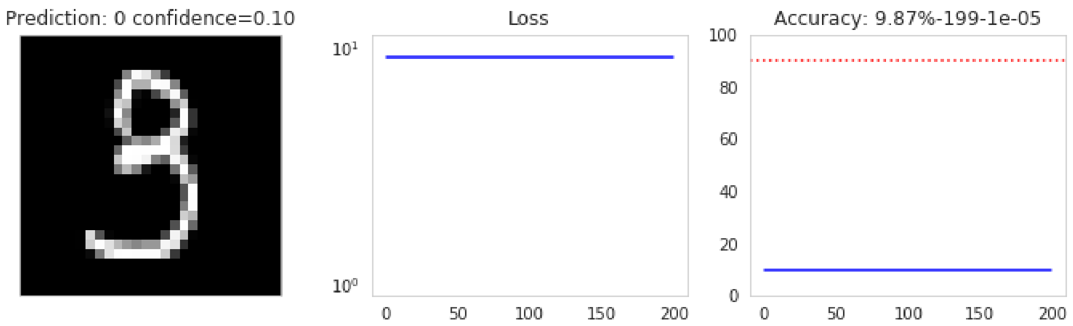
In [318]:

```
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# k=1 # optimal k
# lr = 1e-5 # default learning rate = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# # Activation functions k parameter tuning
# weights = ['normal', 'default', 'normal_sqrt', 'normal_std25', 'normal_std05', 'normal_std01', '
```

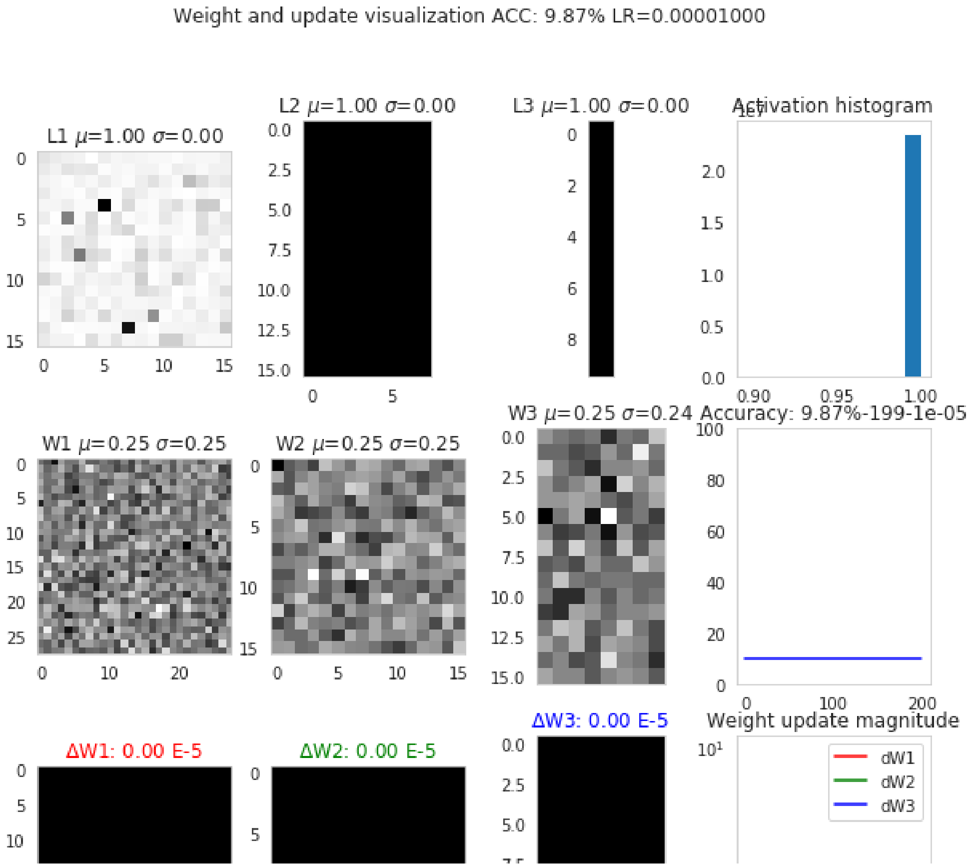
```
normal_std1', 'uniform', 'poisson']
#
#
# for j in range(len(weights)):
#
#     # reinitialize weights to random %% Setup: 784 -> 256 -> 128 -> 10
#     if weights[j] == "default":
#         W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
#         W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
#         W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
#     if weights[j] == "normal":
#         W1 = (np.random.normal(loc=0, scale=1, size=(784, 256)).astype('float32').T)/3
#         W2 = (np.random.normal(loc=0, scale=1, size=(256, 128)).astype('float32').T)/3
#         W3 = (np.random.normal(loc=0, scale=1, size=(128,  10)).astype('float32').T)/3
#     if weights[j] == "normal_sqrt":
#         # Best distribution for Relu activation functions
#         # Normal distribution std = 1/sqrt(Layer size)
#         W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
#         W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
#         W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#     if weights[j] == "normal_std25":
#         # normal distribution (std = 0.25)
#         W1 = np.random.normal(0,.1,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,.1,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,.1,(128,10)).astype('float32').T
#     if weights[j] == "normal_std05":
#         # normal distribution (std = 0.5)
#         W1 = np.random.normal(0,.5,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,.5,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,.5,(128,10)).astype('float32').T
#     if weights[j] == "normal_std01":
#         # normal distribution (std = .1)
#         W1 = np.random.normal(0,.1,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,.1,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,.1,(128,10)).astype('float32').T
#     if weights[j] == "normal_std1":
#         # normal distribution (std = 1)
#         W1 = np.random.normal(0,1,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,1,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,1,(128,10)).astype('float32').T
#     if weights[j] == "uniform":
#         W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
#         W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
#         W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
#     if weights[j] == "poisson":
#         W1 = (np.random.poisson(lam=1, size=(784, 256)).astype('float32').T)/4
#         W2 = (np.random.poisson(lam=1, size=(256, 128)).astype('float32').T)/4
#         W3 = (np.random.poisson(lam=1, size=(128,  10)).astype('float32').T)/4
#
#
#
#     # reinitialize Monitoring variables to empty
#     accuracies = []
#     losses     = []
#     mean_activ = []
#     hw1, hw2, hw3 = [], [], []
#     # progbar.update(i % 5)
#
#     # optimize NN
#     for i in range(200):
#         L1, L2, L3    = forward_pass(X, W1, W2, W3, k)
#         dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3, k)
#         W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#         progbar.update(i % 200)
#         calculate_loss()
#         accuracy()
#         hw_update()
#
#         if i == 199:
#             if checknan(L1) or checknan(L2) or checknan(L3):
#                 print('\nNaN encountered in activations. Try lowering the learning rate and/or co
rrecting bugs in your code.')
#                 print('Optimization halted')
#                 break
#             clear_output(wait=True)
#             confusion_matrix()
```

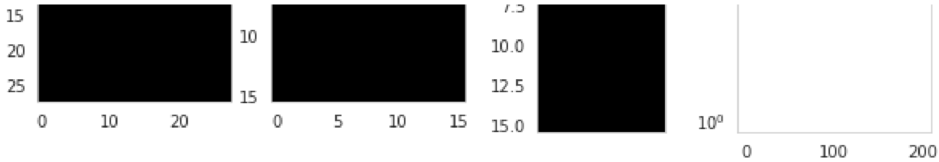
```
#              triple_plot()
#              tsne_viz()
#
#              losses_dict[j] =losses #store losses for this round with the learning rate
#              accuracy_dict[j] =accuracies #store losses for this round with the learning rate
#
#
```

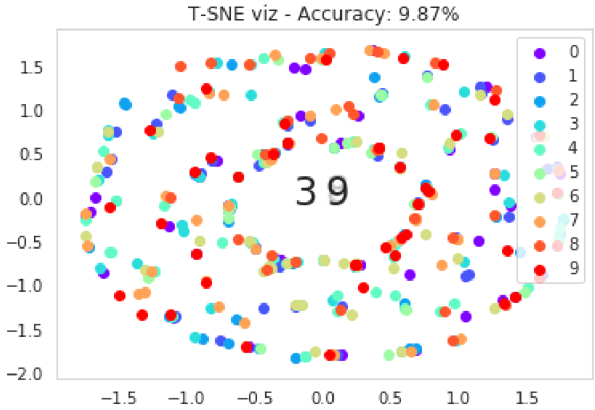





```
/opt/anaconda3/lib/python3.7/site-packages/matplotlib/ticker.py:2241: UserWarning: Data has no pos
itive values, and therefore cannot be log-scaled.
  "Data has no positive values, and therefore cannot be "
```



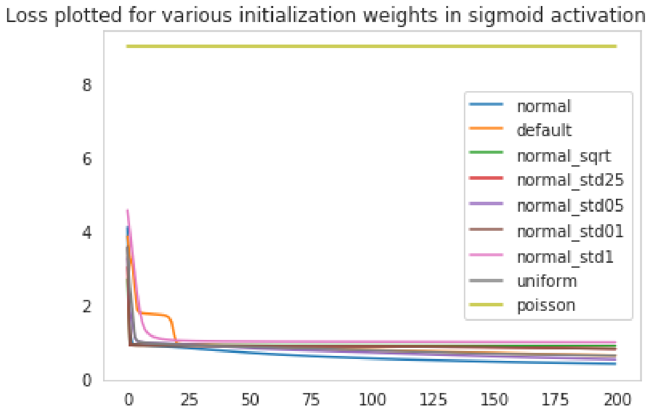




2.3 Initialization Functions Line Plots for best weights

In [319]:

```
# #plot all the different k parameters : k=.5 looks the best!
# for m in range(len(weights)):
#     plt.plot(losses_dict[m], label = str(weights[m]))
#     plt.title("Loss plotted for various initialization weights in sigmoid activation function ")
#     plt.legend()
# plt.show()
```

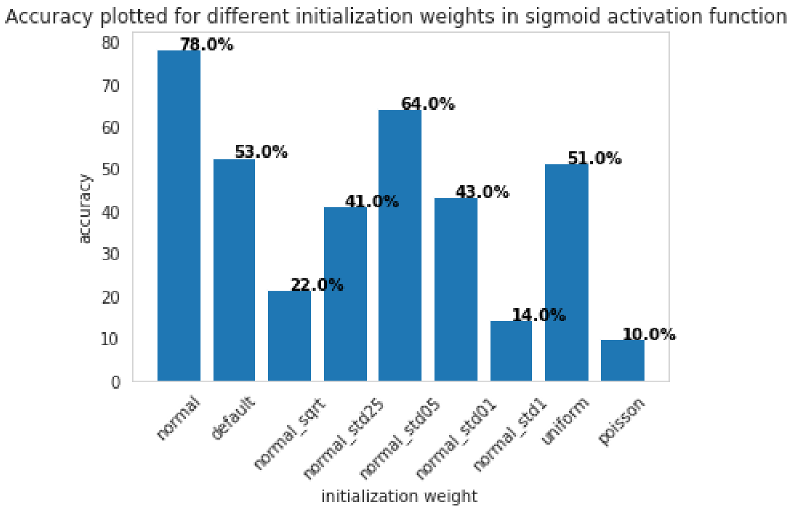


2.3 Initialization Functions Accuracy Bar Plot

In [320]:

```
#
# #a_list
# a_list = []
# for i in range(len(weights)):
#     a_list.append(accuracy_dict[i][len(accuracy_dict[i])-1])
#
# # bar plot of accuracy
#
# import numpy as np
# import matplotlib.pyplot as plt
# from itertools import chain # library required to unchain
#
# height = list(a_list) # unchain the dictionary accuracy value
# bars =
(weights[0],weights[1],weights[2],weights[3],weights[4],weights[5],weights[6],weights[7],weights[8…
# y_pos = np.arange(len(bars))
```

```
# y_pos = np.arange(len(bars))
#
# x_labels = list(a_list)
#
# plt.bar(y_pos, height)# Create bars
# #ax = accuracy_dict.plot(kind='bar')
# plt.xticks(y_pos, bars)# Create names on the x-axis
# plt.xlabel('initialization weight')
# #plt.set_xticklabels(x_labels)
# plt.ylabel('accuracy')
# plt.title("Accuracy plotted for different initialization weights in sigmoid activation function
")
# plt.xticks(rotation=45)
#
# for i, v in enumerate(x_labels):
#     plt.text(i, v, str(round(v))+"%", color='black', fontweight='bold')
#
#
# plt.show() # Show graphic
```



In [58]:

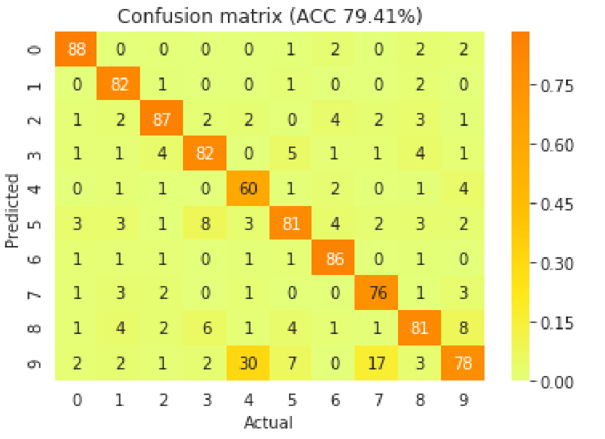
Problem 2.3 Main loop - Another attempt after the above, but with random initialization

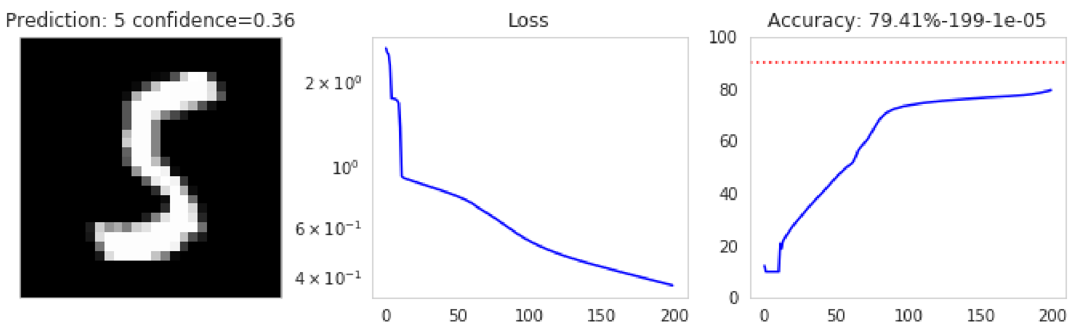
- The best is randomSqrt with 79.41% accuracy (even better than the previous runs & normal before!)

In [327]:

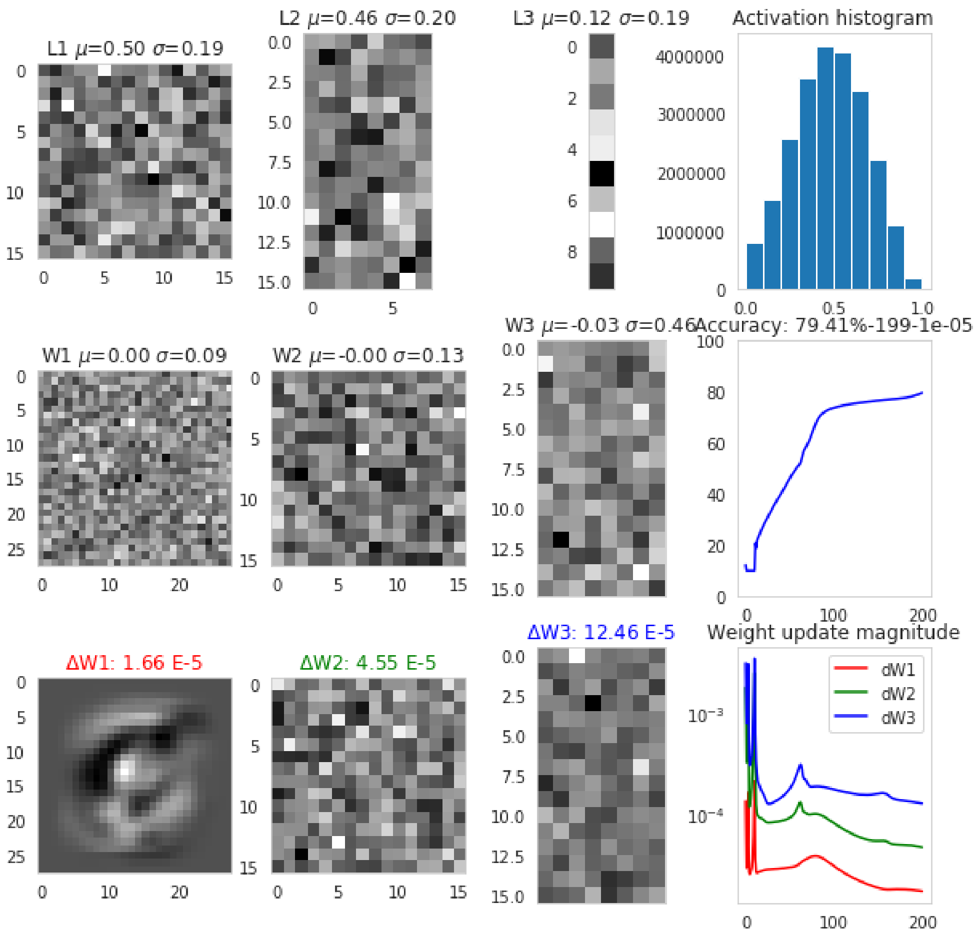
```
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# # Activation functions k parameter tuning
# k= 1
#
# # default learning rate = 1e-5
# lr = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# sqrt_initialization_losses_dict = []
#
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# sqrt_initialization_accuracy_dict = []
#
# # Activation functions k parameter tuning
# weights = ['randomSqrt']
#
#
# for i in range(len(weights)):
```

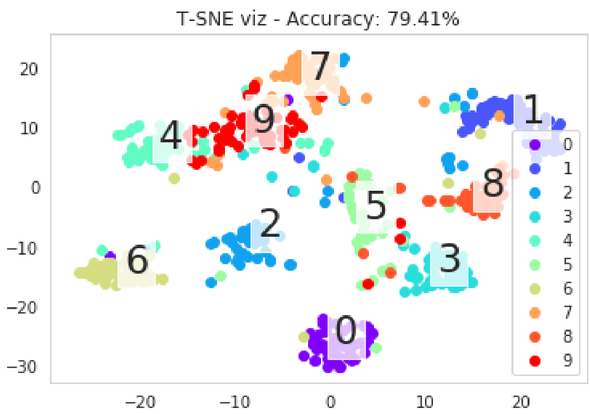
```
# for j in range(len(weights)):
#
#     # reinitialize weights to random %% Setup: 784 -> 256 -> 128 -> 10
#     if weights[j] == "randomSqrt":
#         # initialize with random - all the weights randomly from a univariate "Gaussian"
(Normal) distribution having mean 0 and variance 1
#         # and multiply them by a negative power of 10 to make them small.
#         W1 = np.random.randn(784, 256).T*np.sqrt(2/256)
#         W2 = np.random.randn(256, 128).T*np.sqrt(2/128)
#         W3 = np.random.randn(128, 10).T*np.sqrt(2/10)
#
#     # reinitialize Monitoring variables to empty
#     accuracies = []
#     losses     = []
#     mean_activ = []
#     hw1, hw2, hw3 = [], [], []
#     # progbar.update(i % 5)
#
#     # optimize NN
#     for i in range(200):
#         L1, L2, L3    = forward_pass(X, W1, W2, W3, k)
#         dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3, k)
#         W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#         progbar.update(i % 200)
#         calculate_loss()
#         accuracy()
#         hw_update()
#
#         if i == 199:
#             if checknan(L1) or checknan(L2) or checknan(L3):
#                 print('\nNaN encountered in activations. Try lowering the learning rate and/or co
rrecting bugs in your code.')
#                 print('Optimization halted')
#                 break
#             clear_output(wait=True)
#             confusion_matrix()
#             triple_plot()
#             tsne_viz()
#
#
#             sqrt_initialization_losses_dict =losses #store losses for this round with the
learning rate
#             sqrt_initialization_accuracy_dict =accuracies #store losses for this round with the
learning rate
#
```





Weight and update visualization ACC: 79.41% LR=0.00001000





In [ ]:

Problem 3.1: Activations and Gradients:

- Screenshots for 3.1 taken from the above runs of sigmoid using poisson and normal in 2.3 problem

In [ ]:

```

```

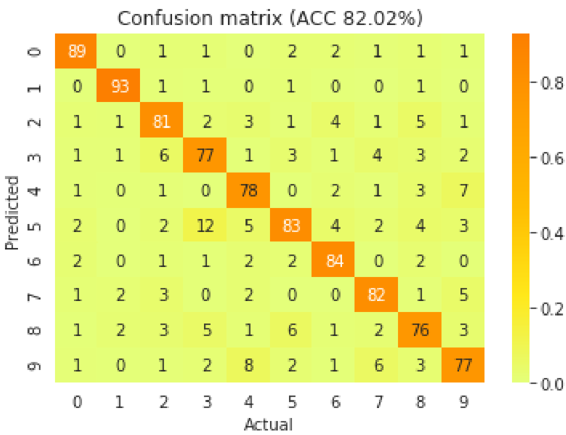
Problem 3.2 Main - Tanh

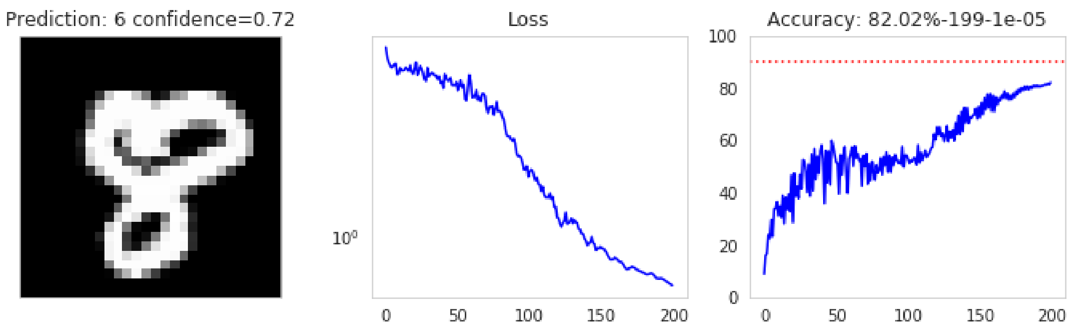
- Tanh with k = 1, learning rate = 1e05, & normal distribution = 66.96%
- Tanh with k = 1, learning rate = 1e05, & random distribution = 82.02%

In [330]:

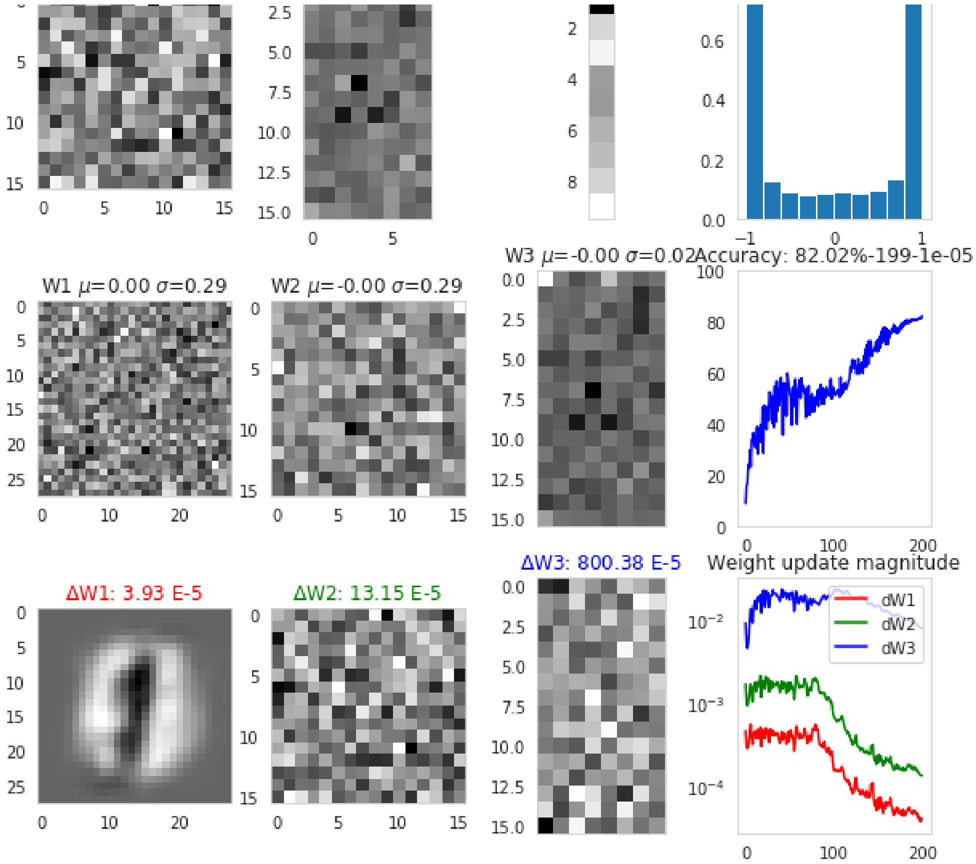
```
# # tanh
# # Define new activation functions
# #def than(x): return np.tanh(x)
#
# def forward_pass(X, W1, W2, W3):
#     L1 = np.tanh(W1.dot(X))
#     L2 = np.tanh(W2.dot(L1))
#     L3 = np.tanh(W3.dot(L2))
#     return L1, L2, L3
#
# # update backward pass
# def backward_pass(L1, L2, L3, W1, W2, W3):
#     dW3 = (L3 - T)  *  (1 - L3**2)
#     dW2 = W3.T.dot(dW3)*(1 - L2**2)
#     dW1 = W2.T.dot(dW2)*(1 - L1**2)
#     return dW1, dW2, dW3
#
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# k=1
#
# # default learning rate = 1e-5
# lr = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
#
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# # LOW TANH WEIGHTS
#
# # Normal distribution std = 1/sqrt(Layer size)
# #W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
# #W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
# #W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#
# #W1 = np.random.randn(784, 256).T*np.sqrt(1/(256+784))
# #W2 = np.random.randn(256, 128).T*np.sqrt(1/(128+256))
# #W3 = np.random.randn(128, 10).T*np.sqrt(1/(10+128))
#
# #W1 = np.random.randn(784, 256).T*np.sqrt(2/256)
# #W2 = np.random.randn(256, 128).T*np.sqrt(2/128)
# #W3 = np.random.randn(128, 10).T*np.sqrt(2/10)
#
#
# # BEST TWO TANH WEIGHTS
#
# #best normal weights for sigmoid
# #W1 = (np.random.normal(loc=0, scale=1, size=(784, 256)).astype('float32').T)/3
# #W2 = (np.random.normal(loc=0, scale=1, size=(256, 128)).astype('float32').T)/3
# #W3 = (np.random.normal(loc=0, scale=1, size=(128,  10)).astype('float32').T)/3
#
# #best weights for tanh
```

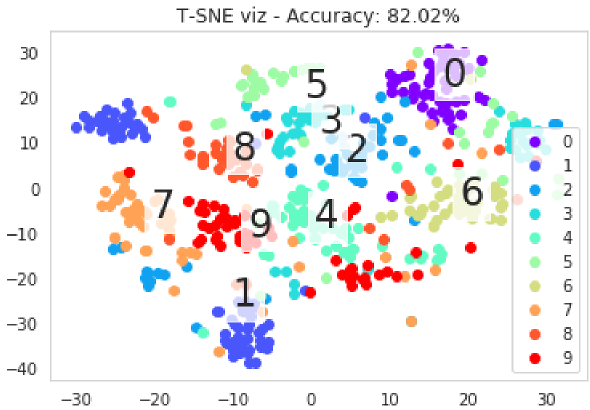
```
# #best weights for tanh
# W1 = np.random.rand(784, 256).astype('float32').T - .5
# W2 = np.random.rand(256, 128).astype('float32').T - .5
# W3 = np.random.rand(128,  10).astype('float32').T - .5
#
# # reinitialize Monitoring variables to empty
# accuracies = []
# losses     = []
# mean_activ = []
# hw1, hw2, hw3 = [], [], []
# # progbar.update(i % 5)
#
# # optimize NN
# for i in range(200):
#     L1, L2, L3    = forward_pass(X, W1, W2, W3)
#     dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3)
#     W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#     progbar.update(i % 200)
#     calculate_loss()
#     accuracy()
#     hw_update()
#
#
#     if i == 199:
#         if checknan(L1) or checknan(L2) or checknan(L3):
#             print('\nNaN encountered in activations. Try lowering the learning rate and/or
correcting bugs in your code.')
#             print('Optimization halted')
#             break
#         clear_output(wait=True)
#         confusion_matrix()
#         triple_plot()
#         tsne_viz()
#
#         #losses_dict[j] =losses #store losses for this round with the learning rate
#         #accuracy_dict[j] =accuracies #store losses for this round with the learning rate
#
```





Weight and update visualization ACC: 82.02% LR=0.00001000





In [ ]:

```

```

Problem 3.3 Main - Cross entropy implementation

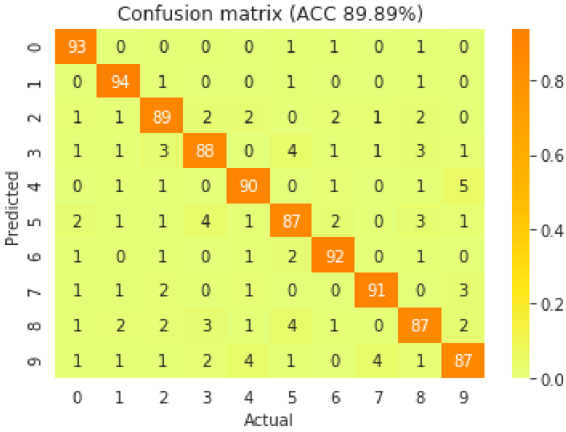
- Sigmoid with cross entropy, accuracy = 89.89%

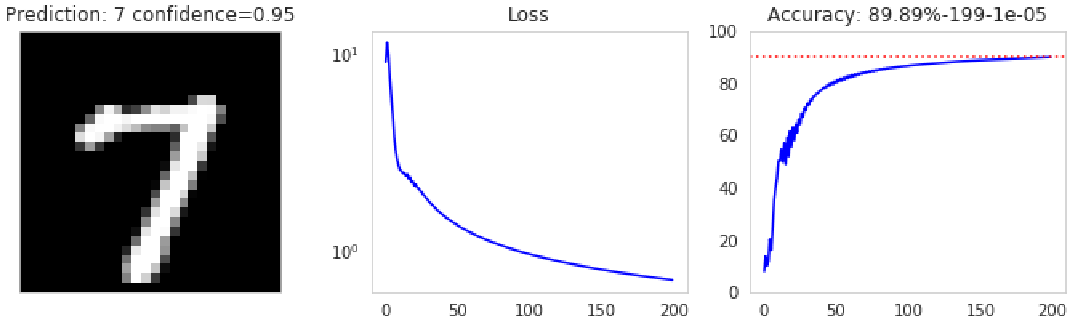
In [124]:

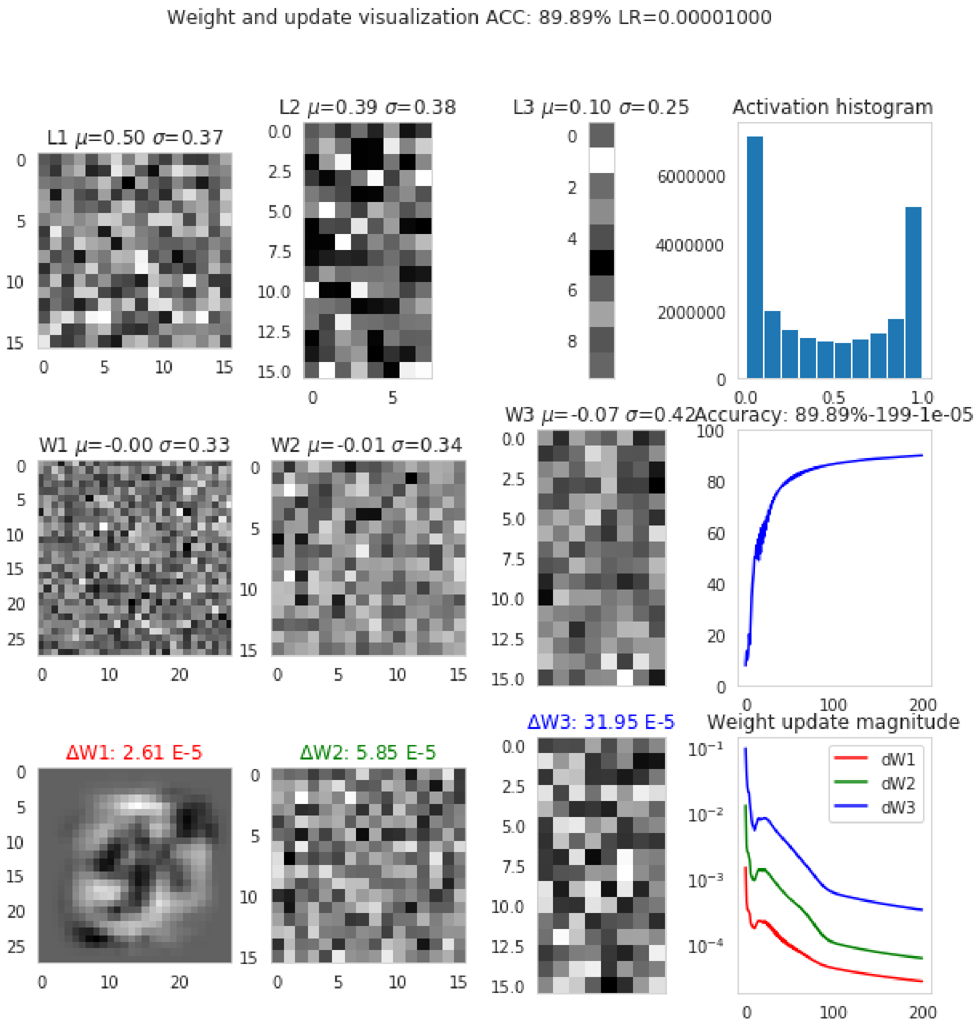
```
# # sigmoid with cross entropy
#
# # Define basic sigmoid activation
# def sigmoid(x): return 1.0/(1.0 + np.e**-x)
# #def sigmoid(x,k): return 1.0/(1.0 + np.e**-(k*x)) #sigmoid with k parameter
#
# def sigmoid_cross_entrophy_forward(X, W1, W2, W3):
```

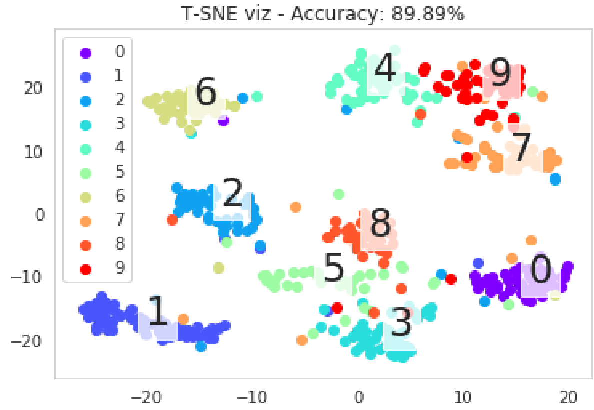
```
# def sigmoid_cross_entrophy_forward(X, W1, W2, W3):
#     # Forward pass
#     L1 = sigmoid(W1.dot(X))
#     L2 = sigmoid(W2.dot(L1))
#     L3 = sigmoid(W3.dot(L2))
#     return L1, L2, L3
#
# def sigmoid_cross_entrophy_backward(L1, L2, L3, W1, W2, W3):
#     # Backward pass
#     dW3 = (L3 - T)
#     dW2 = W3.T.dot(dW3)*(L2*(1-L2))
#     dW1 = W2.T.dot(dW2)*(L1*(1-L1))
#     return dW1, dW2, dW3
#
# def calculate_loss_ce():
#     global losses
#     loss = np.sum(np.nan_to_num(-T*np.log(L3)-(1-T)*np.log(1-L3)))/len(T.T) # cross entropy
#     losses.append(loss)
#     #print("[%04d] MSE Loss: %0.6f" % (i, loss))
#
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# k=1
#
# # default learning rate = 1e-5
# lr = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
#
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# # Best distribution for Relu activation functions
# # Normal distribution std = 1/sqrt(Layer size)
# #W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
# #W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
# #W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#
# #best normal weights for sigmoid
# W1 = (np.random.normal(loc=0, scale=1, size=(784, 256)).astype('float32').T)/3
# W2 = (np.random.normal(loc=0, scale=1, size=(256, 128)).astype('float32').T)/3
# W3 = (np.random.normal(loc=0, scale=1, size=(128,  10)).astype('float32').T)/3
#
#
# # reinitialize Monitoring variables to empty
# accuracies = []
# losses     = []
# mean_activ = []
# hw1, hw2, hw3 = [], [], []
# # progbar.update(i % 5)
# temp_accuracies = []
#
# # optimize NN
# for i in range(200):
#     L1, L2, L3    = sigmoid_cross_entrophy_forward(X, W1, W2, W3)
#     dW1, dW2, dW3 = sigmoid_cross_entrophy_backward(L1, L2, L3, W1, W2, W3)
#     W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#     progbar.update(i % 200)
#     #calculate_loss()
#     calculate_loss_ce()
#     accuracy()
#     hw_update()
#
#
#     if i == 199:
#         if checknan(L1) or checknan(L2) or checknan(L3):
#             print('\nNaN encountered in activations. Try lowering the learning rate and/or
correcting bugs in your code.')
#             print('Optimization halted')
#             break
#         clear_output(wait=True)
#         confusion_matrix()
#         triple_plot()
```

```
#          triple_plot()
#          tsne_viz()
#
#          losses_dict =losses #store losses for this round with the learning rate
#          accuracy_dict =accuracies #store losses for this round with the learning rate
```









In [ ]:

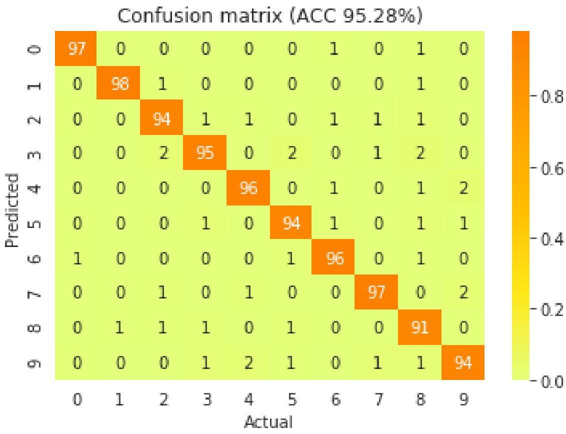
Problem 3.4 Relu / Relu6

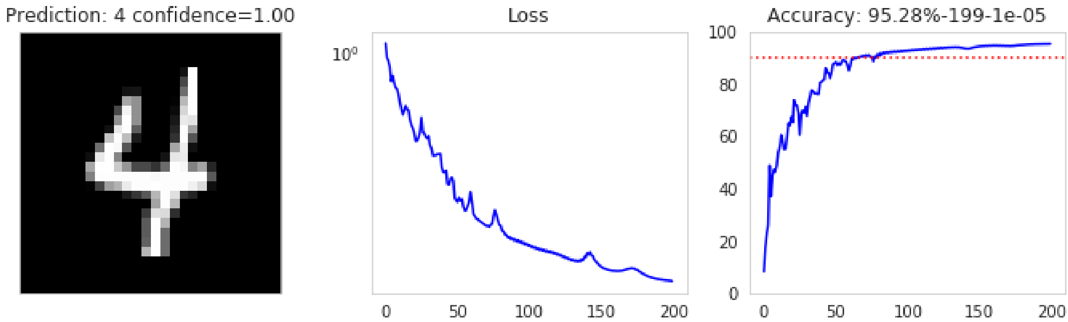
Problem 3.4 Main - relu

In [333]:

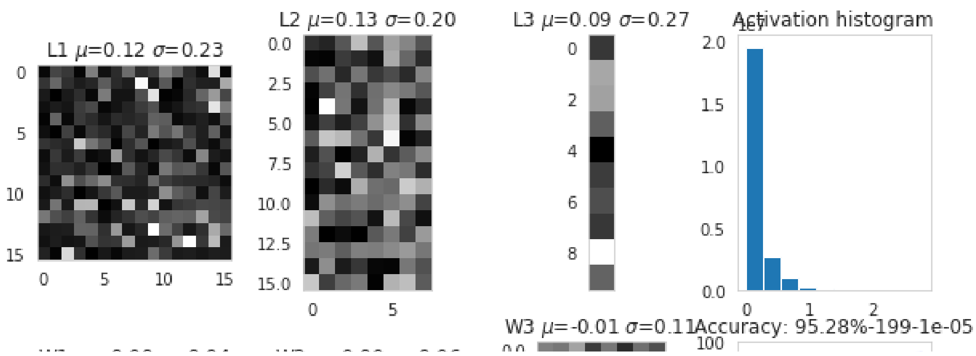
```
# def relu(x): return np.maximum(x, 0)
# def drelu(x): return 1. * (x>0)
#
# # Define forward pass
# def forward_pass_relu(X, W1, W2, W3):
#     L1 = relu(W1.dot(X))
#     L2 = relu(W2.dot(L1))
#     L3 = relu(W3.dot(L2))
#     return L1, L2, L3
#
# # Define backward pass
# def backward_pass_relu(L1, L2, L3, W1, W2, W3):
#     dW3 = (L3 - T) * drelu(L3)
#     dW2 = W3.T.dot(dW3) * drelu(L2)
#     dW1 = W2.T.dot(dW2) * drelu(L1)
#     return dW1, dW2, dW3
#
# def update_weights_relu(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2):
#     W3 -= lr*np.dot(dW3, L2.T)
#     W2 -= lr*np.dot(dW2, L1.T)
#     W1 -= lr*np.dot(dW1, X.T)
#     return W1, W2, W3
#
# # Monitoring variables
# accuracies = []
# losses     = []
# mean_activ = []
# hw1, hw2, hw3 = [], [], []
#
# #%% Setup: 784 -> 256 -> 128 -> 10
# W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
# W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
# W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
```

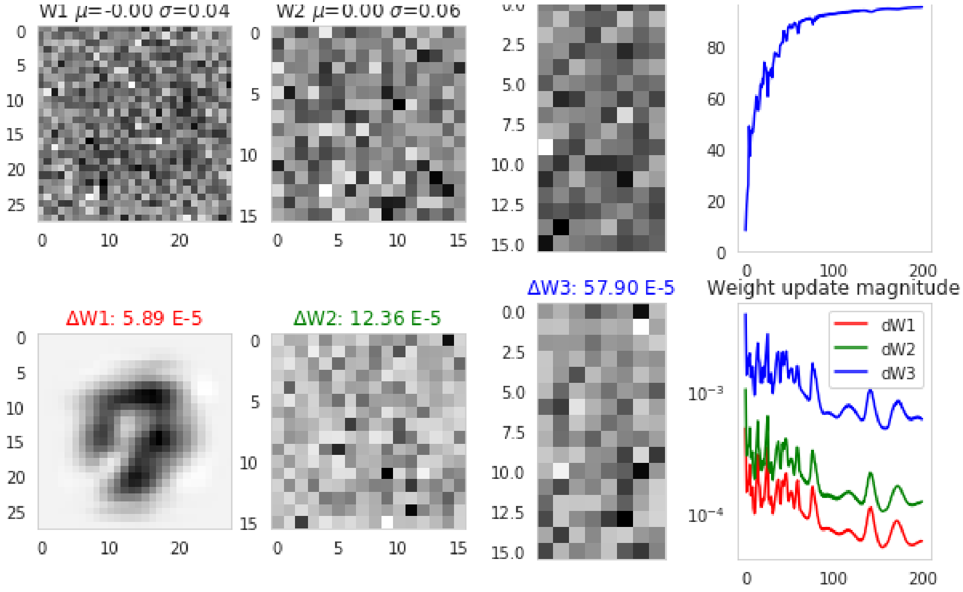
```
# # Learning rate, decrease if optimization isn't working
# lr = 1e-5
#
# for i in range(200):
#     L1, L2, L3     = forward_pass_relu(X, W1, W2, W3)
#     dW1, dW2, dW3 = backward_pass_relu(L1, L2, L3, W1, W2, W3)
#     W1, W2, W3     = update_weights_relu(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#     progbar.update(i % 200)
#     calculate_loss()
#     accuracy()
#     hw_update()
#
#     if i == 199:
#         if checknan(L1) or checknan(L2) or checknan(L3):
#             print('\nNaN encountered in activations. Try lowering the learning rate and/or
correcting bugs in your code.')
#             print('Optimization halted')
#             break
#         clear_output(wait=True)
#         confusion_matrix()
#         triple_plot()
#         tsne_viz()
#
#
#         #losses_dict[i] =losses #store losses for this round with the learning rate
#         #accuracy_dict[i] =accuracies #store losses for this round with the learning rate
```

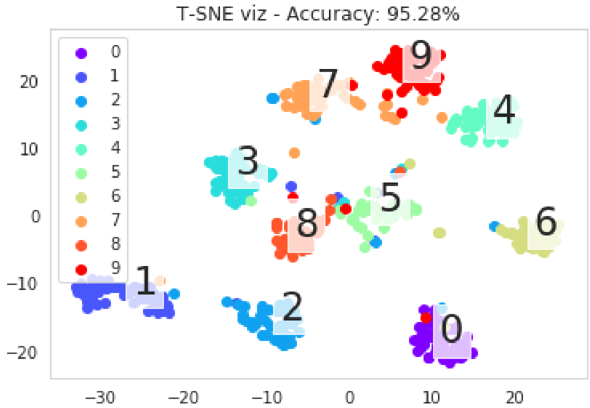




Weight and update visualization ACC: 95.28% LR=0.00001000







In [ ]:

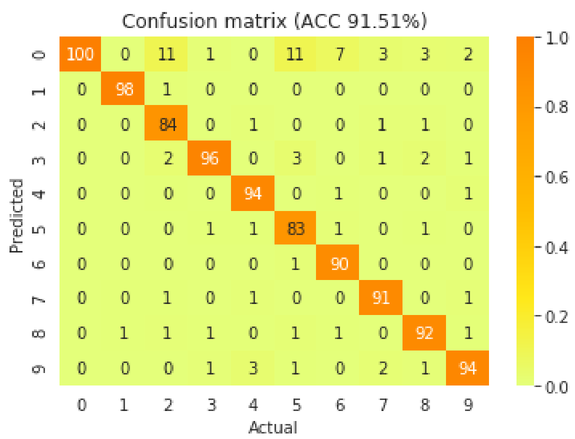
Problem 3.4 Main - relu6

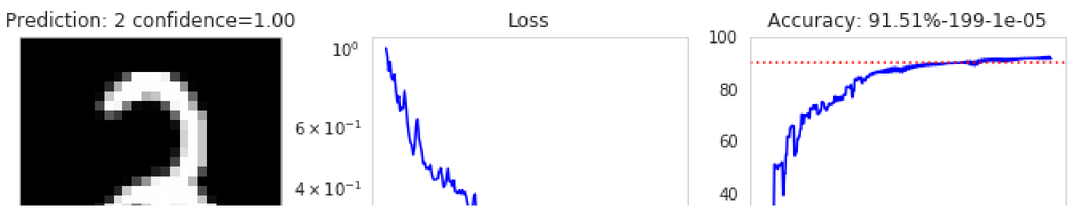
- Relu6 with normal dist has accuracy = 91%

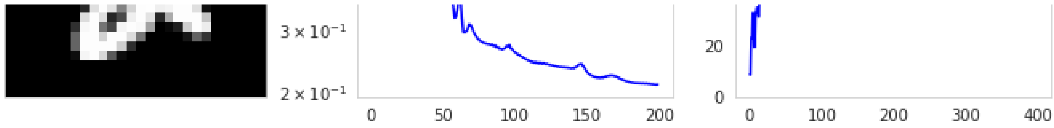
In [104]:

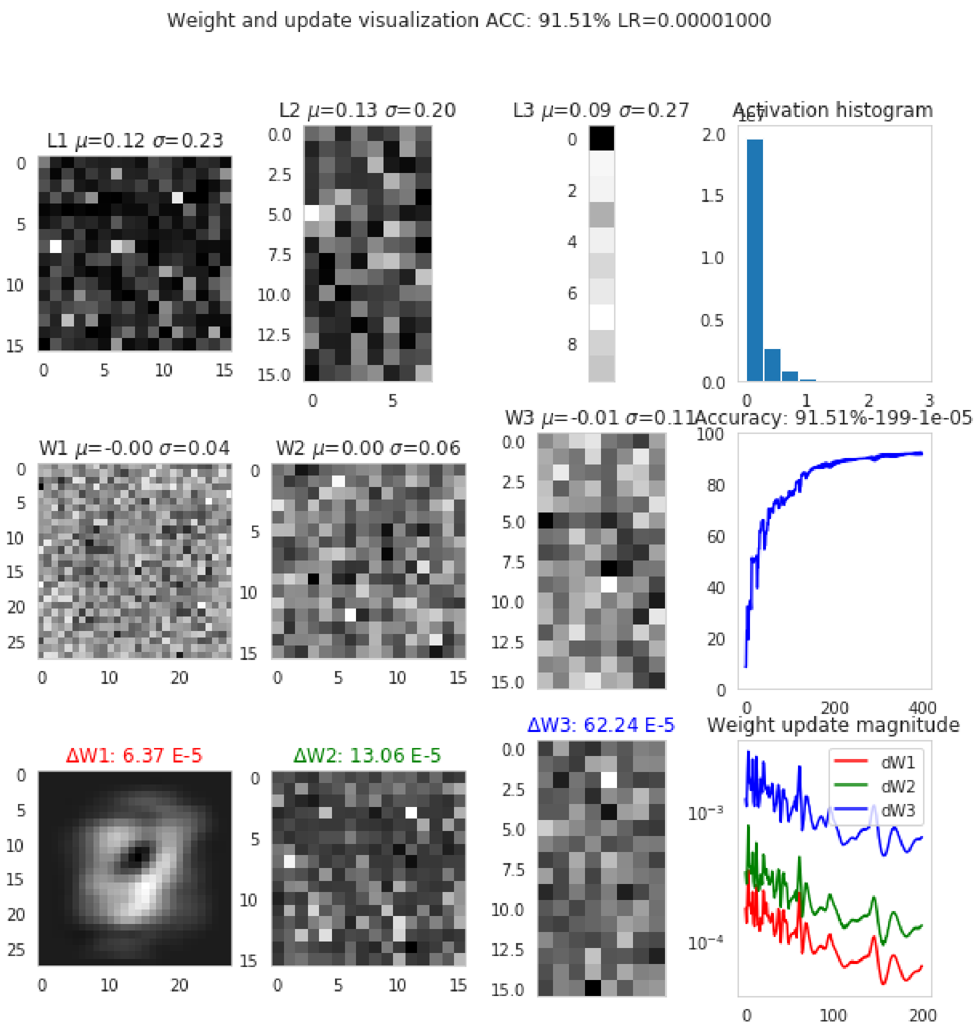
```
# def relu6(x):
#     maxVal = x * (x > 0)
#     maxVal[maxVal>=6]=6
#     return maxVal
#
# def forward_pass_relu6(X, W1, W2, W3):
#     L1 = relu6(W1.dot(X))
#     L2 = relu6(W2.dot(L1))
#     L3 = relu6(W3.dot(L2))
#     return L1, L2, L3
#
# def backward_pass_relu6(L1, L2, L3, W1, W2, W3):
#     dW3 = (L3 - T) * 1. * ((L3 > 0) * (L3 < 6))
#     dW2 = W3.T.dot(dW3) * 1. * ((L2 > 0) * (L2 < 6))
#     dW1 = W2.T.dot(dW2) * 1. * ((L1 > 0) * (L1 < 6))
#     return dW1, dW2, dW3
#
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# k=1
```

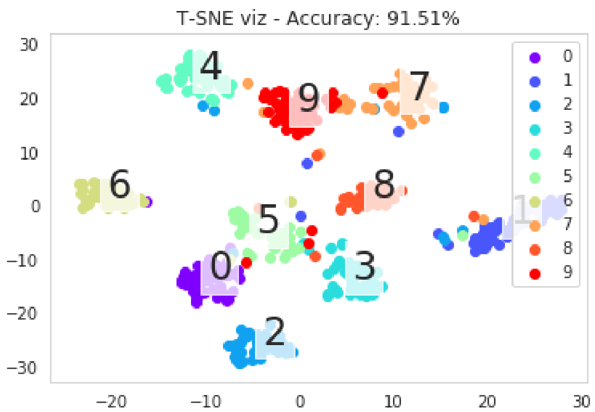
```
#
# # default learning rate = 1e-5
# lr = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
#
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# # Normal distribution std = 1/sqrt(Layer size)
# W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
# W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
# W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#
# # reinitialize Monitoring variables to empty
# accuracies = []
# losses     = []
# mean_activ = []
# hw1, hw2, hw3 = [], [], []
# # progbar.update(i % 5)
# temp_accuracies = []
#
# # optimize NN
# for i in range(200):
#     L1, L2, L3    = forward_pass_relu6(X, W1, W2, W3)
#     dW1, dW2, dW3 = backward_pass_relu6(L1, L2, L3, W1, W2, W3)
#     W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#     progbar.update(i % 200)
#     calculate_loss()
#     #cross_entropy()
#     accuracy()
#     hw_update()
#
#     losses_dict[i] =losses #store losses for this round with the learning rate
#     accuracy_dict[i] =accuracies #store losses for this round with the learning rate
#     accuracy()
#
#
#     if i == 199:
#         if checknan(L1) or checknan(L2) or checknan(L3):
#             print('\nNaN encountered in activations. Try lowering the learning rate and/or
correcting bugs in your code.')
#             print('Optimization halted')
#             break
#         clear_output(wait=True)
#         confusion_matrix()
#         triple_plot()
#         tsne_viz()
```





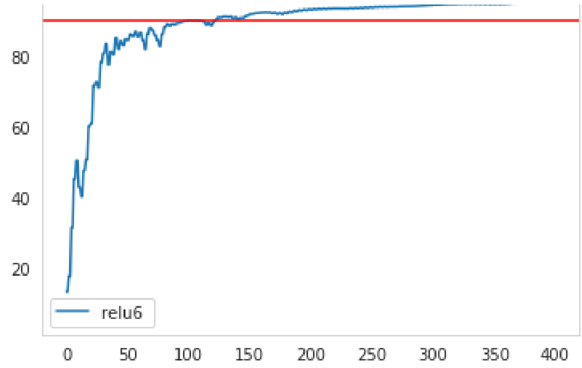






In [89]:

```
# #plot all the different k parameters : k=.5 looks the best!
# plt.plot(accuracies, label = "relu6 ")
# plt.title("Accuracy plotted for relu6 ")
# plt.legend()
# plt.axhline(90, color="red")
# #plt.abs(90)
# plt.ylim(1,100)
# plt.show()
```



Problem 4:

- Screenshots taken from previous iteration runs

In [ ]:

```

```

In [ ]:

```

```

APPENDIX: Other models attempted

Weight Initialization testing for Lrelu

In [357]:

```
# #leaky relu
# def lrelu(x, alpha=0.01):
#     return np.maximum(alpha*x, x)
#
# def dlrelu(x, alpha=0.01):
#     dx = np.ones_like(x)
#     dx[x < 0] = alpha
#     return dx
#
# def softmax(X):
#     exps = np.exp(X)
#     return exps / np.sum(exps, axis=0)
#
# def dsoftmax(x):
#     s = x.reshape(-1,1)
#     return np.diagflat(s) - np.dot(s, s.T)
#
# def calculate_loss_ce():
#     global losses
#     loss = np.sum(np.nan_to_num(-T*np.log(L3)-(1-T)*np.log(1-L3)))/len(T.T) # cross entropy
#     losses.append(loss)
#     #print("[%04d] MSE Loss: %0.6f" % (i, loss))
#
#
# # hybrid: Leaky Relu with Softmax cross entropy
```

```
# # Forward pass
# def forward_pass_lrelu(X, W1, W2, W3):
#     L1 = lrelu(W1.dot(X))
#     L2 = lrelu(W2.dot(L1))
#     L3 = softmax(W3.dot(L2))
#     return L1, L2, L3
#
# # Backward pass
# def backward_pass_lrelu(L1, L2, L3, W1, W2, W3):
#     dW3 = (L3 - T)
#     dW2 = W3.T.dot(dW3) * dlrelu(L2)
#     dW1 = W2.T.dot(dW2) * dlrelu(L1)
#     return dW1, dW2, dW3
#
# #update weights
# def update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2):
#     W3 -= lr*np.dot(dW3, L2.T)
#     W2 -= lr*np.dot(dW2, L1.T)
#     W1 -= lr*np.dot(dW1, X.T)
#     return W1, W2, W3
#
```

Test for best weights for Lrelu

In [369]:

```
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# #k=1 # optimal k
# lr = 1e-4 # default learning rate = 1e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# # Activation functions k parameter tuning
# #weights = ['normal', 'default', 'normal_sqrt', 'normal_std25', 'normal_std05', 'normal_std01',
'normal_std1', 'uniform', 'poisson']
# weights = 'default'
#
# for j in range(len(weights)):
#
#     # reinitialize weights to random %% Setup: 784 -> 256 -> 128 -> 10
#     if weights[j] == "default":
#         W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
#         W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
#         W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
#     if weights[j] == "normal":
#         W1 = (np.random.normal(loc=0, scale=1, size=(784, 256)).astype('float32').T)/3
#         W2 = (np.random.normal(loc=0, scale=1, size=(256, 128)).astype('float32').T)/3
#         W3 = (np.random.normal(loc=0, scale=1, size=(128,  10)).astype('float32').T)/3
#     if weights[j] == "normal_sqrt":
#         # Best distribution for Relu activation functions
#         # Normal distribution std = 1/sqrt(Layer size)
#         W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
#         W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
#         W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#     if weights[j] == "normal_std25":
#         # normal distribution (std = 0.25)
#         W1 = np.random.normal(0,.1,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,.1,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,.1,(128,10)).astype('float32').T
#     if weights[j] == "normal_std05":
#         # normal distribution (std = 0.5)
#         W1 = np.random.normal(0,.5,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,.5,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,.5,(128,10)).astype('float32').T
#     if weights[j] == "normal_std01":
#         # normal distribution (std = .1)
#         W1 = np.random.normal(0,.1,(784,256)).astype('float32').T
```

```
#         W2 = np.random.normal(0,.1,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,.1,(128,10)).astype('float32').T
#     if weights[j] == "normal_std1":
#         # normal distribution (std = 1)
#         W1 = np.random.normal(0,1,(784,256)).astype('float32').T
#         W2 = np.random.normal(0,1,(256,128)).astype('float32').T
#         W3 = np.random.normal(0,1,(128,10)).astype('float32').T
#     if weights[j] == "uniform":
#         W1 = 2*np.random.rand(784, 256).astype('float32').T - 1
#         W2 = 2*np.random.rand(256, 128).astype('float32').T - 1
#         W3 = 2*np.random.rand(128,  10).astype('float32').T - 1
#     if weights[j] == "poisson":
#         W1 = (np.random.poisson(lam=1, size=(784, 256)).astype('float32').T)/4
#         W2 = (np.random.poisson(lam=1, size=(256, 128)).astype('float32').T)/4
#         W3 = (np.random.poisson(lam=1, size=(128,  10)).astype('float32').T)/4
#
#
#
#     # reinitialize Monitoring variables to empty
#     accuracies = []
#     losses     = []
#     mean_activ = []
#     hw1, hw2, hw3 = [], [], []
#     # progbar.update(i % 5)
#
#
#     # optimize NN
#     for i in range(200):
#         L1, L2, L3    = forward_pass_lrelu(X, W1, W2, W3)
#         dW1, dW2, dW3 = backward_pass_lrelu(L1, L2, L3, W1, W2, W3)
#         W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#         progbar.update(i % 200)
#         #calculate_loss()
#         calculate_loss_ce()
#         accuracy()
#         hw_update()
#
#         if i == 199:
#             if checknan(L1) or checknan(L2) or checknan(L3):
#                 print('\nNaN encountered in activations. Try lowering the learning rate and/or co
rrecting bugs in your code.')
#                 print('Optimization halted')
#                 break
#             clear_output(wait=True)
#             confusion_matrix()
#             triple_plot()
#             tsne_viz()
#
#
#             losses_dict[j] =losses #store losses for this round with the learning rate
#             accuracy_dict[j] = max(accuracies) #store losses for this round with the learning
rate
```

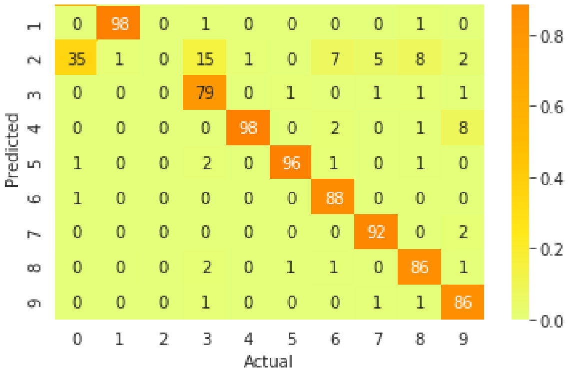
In [ ]:

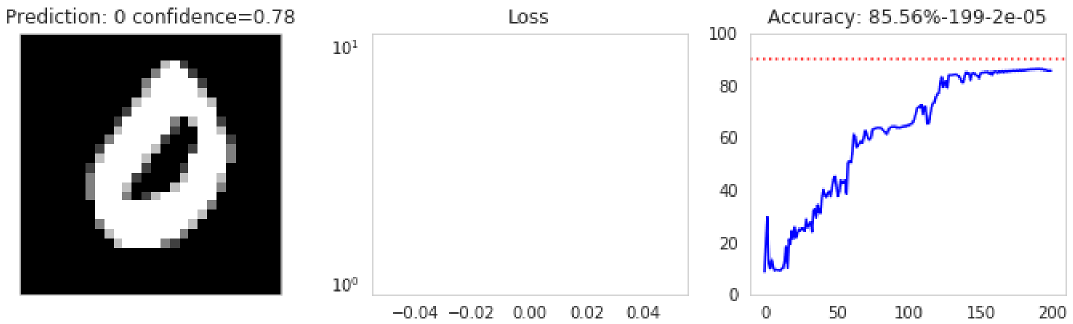
Relu with Cross Entropy: 85.56%

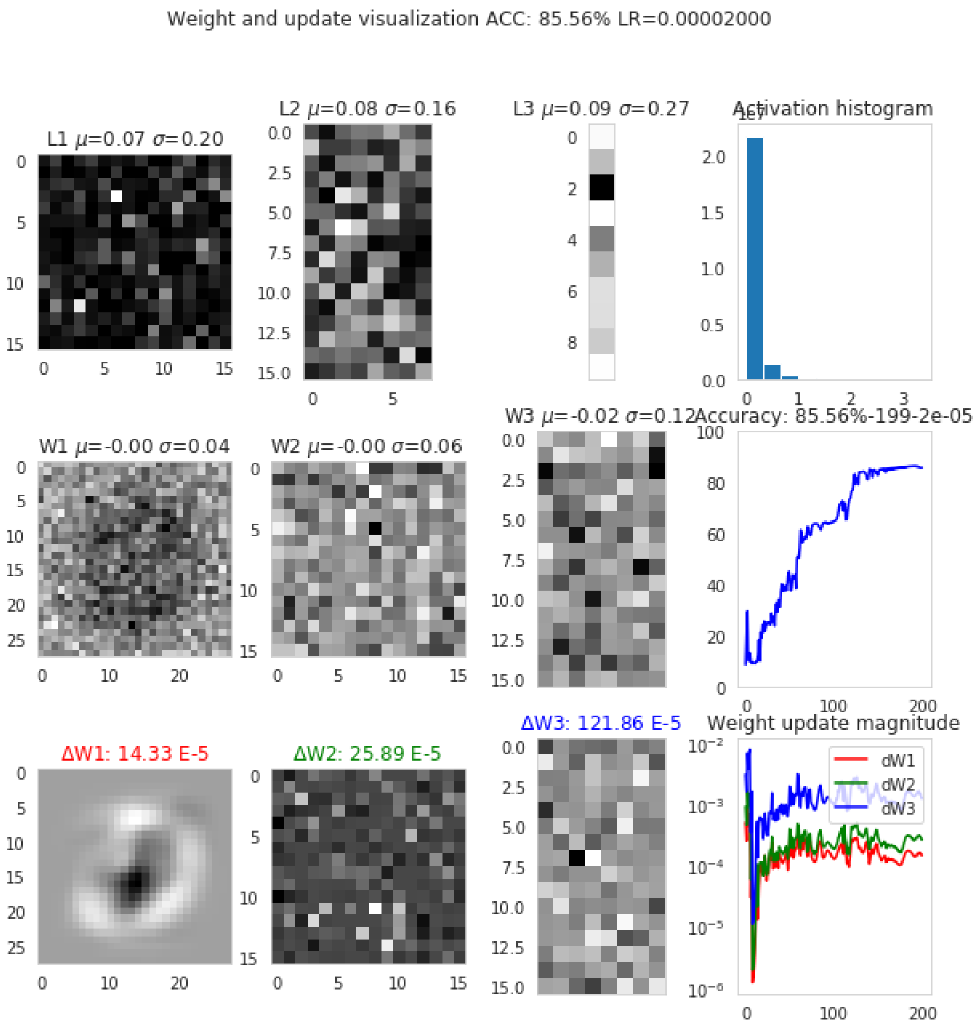
In [244]:

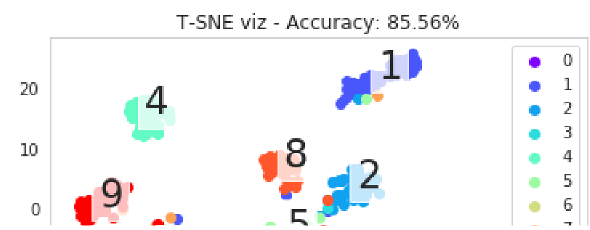
```
#
# # relu
# def relu(x): return np.maximum(x, 0)
# def drelu(x): return 1. * (x>0)
#
#
# def forward_pass_relu(X, W1, W2, W3):
#     # Forward pass
#     L1 = relu(W1.dot(X))
#     L2 = relu(W2.dot(L1))
#     L3 = relu(W3.dot(L2))
#     return L1, L2, L3
```

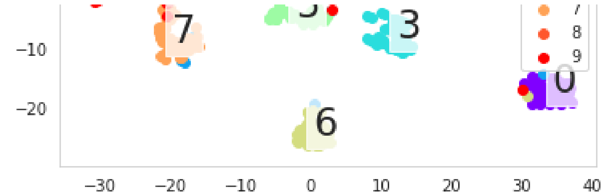
```
#
# def backward_pass_relu(L1, L2, L3, W1, W2, W3):
#     # Backward pass
#     dW3 = (L3 - T) * drelu(L3)
#     dW2 = W3.T.dot(dW3) * drelu(L2)
#     dW1 = W2.T.dot(dW2) * drelu(L1)
#     return dW1, dW2, dW3
#
#
# def calculate_loss_ce():
#     global losses
#     loss = np.sum(np.nan_to_num(-T*np.log(L3)-(1-T)*np.log(1-L3)))/len(T.T) # cross entropy
#     losses.append(loss)
#     #print("[%04d] MSE Loss: %0.6f" % (i, loss))
#
#
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# k=1
#
# # default learning rate = 1e-5
# lr = 2e-5
#
# # create dictionary to store resulting loss for activation rate trials to plot with later
# losses_dict = {}
#
# # create dictionary to store resulting accuracy for activation rate trials to plot with later
# accuracy_dict = {}
#
# # Normal distribution std = 1/sqrt(Layer size)
# W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
# W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
# W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#
#
# # reinitialize Monitoring variables to empty
# accuracies = []
# losses     = []
# mean_activ = []
# hw1, hw2, hw3 = [], [], []
# # progbar.update(i % 5)
# temp_accuracies = []
#
# # optimize NN
# for i in range(200):
#     L1, L2, L3    = forward_pass_relu(X, W1, W2, W3)
#     dW1, dW2, dW3 = backward_pass_relu(L1, L2, L3, W1, W2, W3)
#     W1, W2, W3    = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#     progbar.update(i % 200)
#     #calculate_loss()
#     calculate_loss_ce()
#     accuracy()
#     hw_update()
#
#
#     if i == 199:
#         if checknan(L1) or checknan(L2) or checknan(L3):
#             print('\nNaN encountered in activations. Try lowering the learning rate and/or
correcting bugs in your code.')
#             print('Optimization halted')
#             break
#         clear_output(wait=True)
#         confusion_matrix()
#         triple_plot()
#         tsne_viz()
#
#
#         losses_dict[i] =losses #store losses for this round with the learning rate
#         accuracy_dict[i] =accuracies #store losses for this round with the learning rate
```


Predicted
Actual


Prediction: 0 confidence=0.78
Loss
Accuracy: 85.56%-199-2e-05


Weight and update visualization ACC: 85.56% LR=0.00002000
L1 μ=0.07 σ=0.20
L2 μ=0.08 σ=0.16
L3 μ=0.09 σ=0.27
Activation histogram
W1 μ=-0.00 σ=0.04
W2 μ=-0.00 σ=0.06
W3 μ=-0.02 σ=0.12
Accuracy: 85.56%-199-2e-05
ΔW1: 14.33 E-5
ΔW2: 25.89 E-5
ΔW3: 121.86 E-5
Weight update magnitude
dW1
dW2
dW3


T-SNE viz - Accuracy: 85.56%



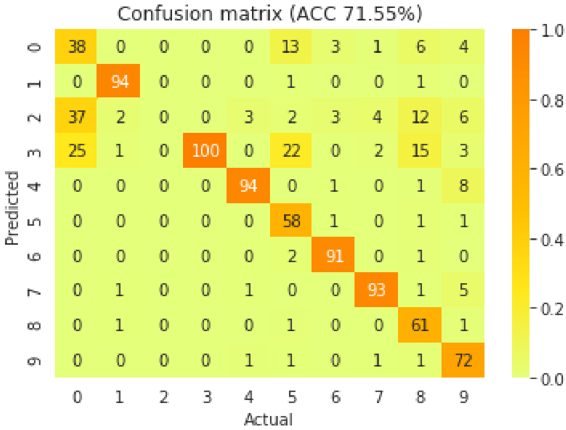
In [ ]:

Relu - 71.55%

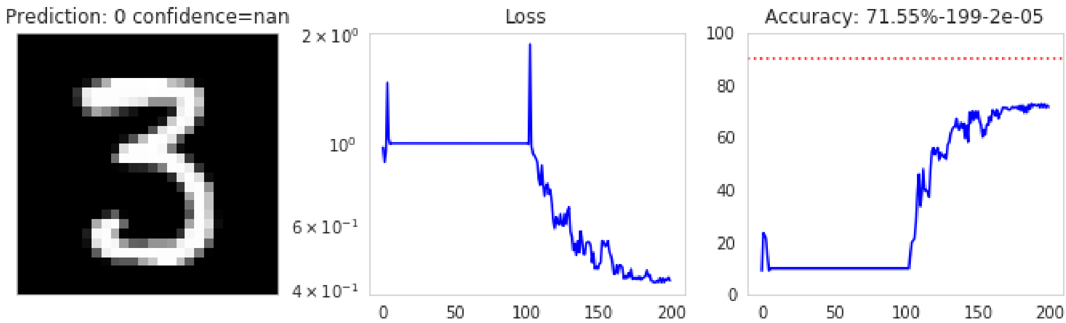
In [365]:

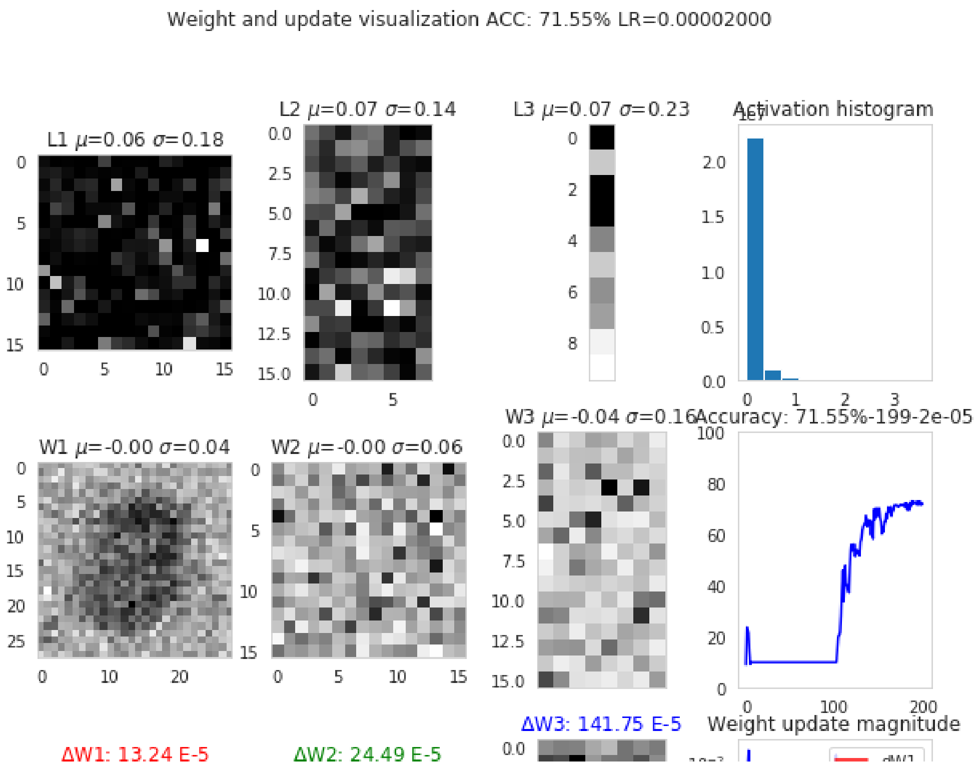
```
# def relu(x): return np.maximum(x, 0)
# def drelu(x): return 1. * (x>0)
#
# # Define forward pass
# def forward_pass_relu(X, W1, W2, W3):
#     L1 = relu(W1.dot(X))
#     L2 = relu(W2.dot(L1))
#     L3 = relu(W3.dot(L2))
#     return L1, L2, L3
#
# # Define backward pass
# def backward_pass_relu(L1, L2, L3, W1, W2, W3):
#     dW3 = (L3 - T) * drelu(L3)
#     dW2 = W3.T.dot(dW3) * drelu(L2)
#     dW1 = W2.T.dot(dW2) * drelu(L1)
#     return dW1, dW2, dW3
#
# def update_weights_relu(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2):
#     W3 -= lr*np.dot(dW3, L2.T)
#     W2 -= lr*np.dot(dW2, L1.T)
#     W1 -= lr*np.dot(dW1, X.T)
#     return W1, W2, W3
#
#
# # Main loop
# from IPython.display import clear_output
# from tensorflow.keras.utils import Progbar
# progbar = Progbar(200)
#
# # Learning rate, decrease if optimization isn't working
# lr = 2e-5
#
#
# # Monitoring variables
# accuracies = []
# losses     = []
# mean_activ = []
# hw1, hw2, hw3 = [], [], []
#
# #%% Setup: 784 -> 256 -> 128 -> 10
# W1 = np.random.normal(0, 1, [784, 256]).astype('float32').T /np.sqrt(784)
# W2 = np.random.normal(0, 1, [256, 128]).astype('float32').T /np.sqrt(256)
# W3 = np.random.normal(0, 1, [128,  10]).astype('float32').T /np.sqrt(128)
#
# for i in range(200):
#     L1, L2, L3     = forward_pass_relu(X, W1, W2, W3)
#     dW1, dW2, dW3 = backward_pass_relu(L1, L2, L3, W1, W2, W3)
#     W1, W2, W3     = update_weights_relu(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
#
#     progbar.update(i % 200)
#     calculate_loss()
#     accuracy()
#     hw_update()
#
#     if i == 199:
#         if checknan(L1) or checknan(L2) or checknan(L3):
#             print('\nNaN encountered in activations. Try lowering the learning rate and/or
```

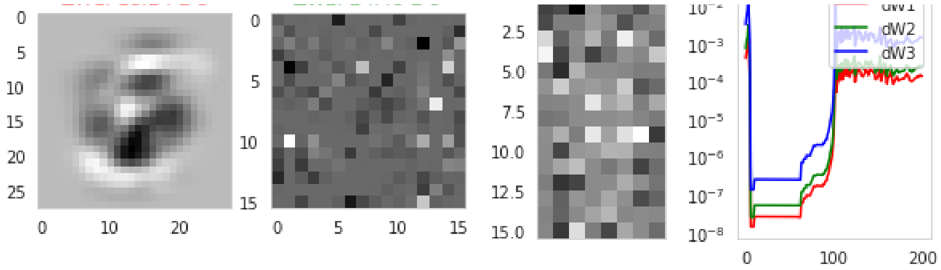
```
correcting bugs in your code.')
#                 print('Optimization halted')
#                 break
#             clear_output(wait=True)
#             confusion_matrix()
#             triple_plot()
#             tsne_viz()
#
#
#             losses_dict[i] =losses #store losses for this round with the learning rate
#             accuracy_dict[i] =max(accuracies) #store losses for this round with the learning rate
```



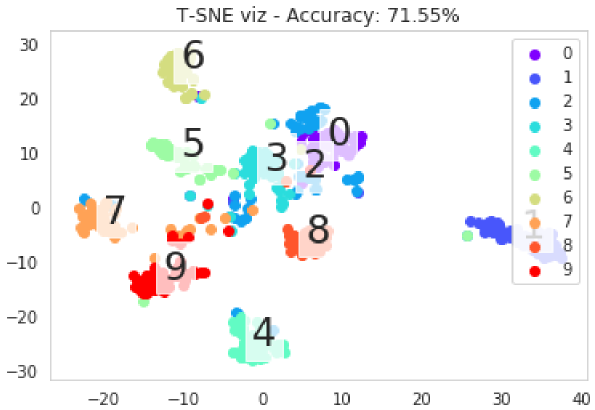

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:9: RuntimeWarning: invalid value encountered in float_scalars
  if __name__ == '__main__':
```







dW1
dW2
dW3



T-SNE viz - Accuracy: 71.55%
0
1
2
3
4
5
6
7
8
9