Mason Lien

Homework

Try out the autoencoder architecture above and test out the relationship between the mean squared error and the size of the latent variable (e.g., the (7,7,1) shape used above) using the above architecture on MNIST after some training. Try 2 or 3 code sizes (or more if you like) and report the parameters of a best fit line

In [1]:
```
# We'll start with our library imports...
from __future__ import print_function

import numpy as np                    # to use numpy arrays
import tensorflow as tf               # to specify and run computation graphs
import tensorflow_datasets as tfds # to load training data
import matplotlib.pyplot as plt    # to visualize data and draw plots
from tqdm import tqdm                 # to track progress of loops
```

```
Matplotlib created a temporary config/cache directory at /tmp/matplotlib-
9qy6ts0o because the default path (/home/agrobinf/masonlien/.config/matpl
otlib) is not a writable directory; it is highly recommended to set the M
PLCONFIGDIR environment variable to a writable directory, in particular t
o speed up the import of Matplotlib and to better support multiprocessin
g.
```

In [2]:
```
# Let's use the code from Hack2 to load MNIST
ds = tfds.load('mnist', shuffle_files=True) # this loads a dict with the da

# We can create an iterator from each dataset
# This one iterates through the train data, shuffling and minibatching by 3
train_ds = ds['train'].shuffle(1024).batch(32)
```

In [73]:
```
def upscale_block(filters, kernel_size=3, scale=2, activation=tf.nn.elu):
    """[Sub-Pixel Convolution](https://arxiv.org/abs/1609.05158)"""
    # Increase the number of channels to the number of channels times the s
    conv = tf.keras.layers.Conv2D(filters * (scale**2),
                                  (kernel_size, kernel_size),
                                  activation=activation,
                                  padding='same')
    # Rearrange blocks of (1,1,scale**2) pixels into (scale,scale,1) pixels
    rearrange = tf.keras.layers.Lambda(
        lambda x: tf.nn.depth_to_space(x, scale))
    return tf.keras.Sequential([conv, rearrange])


class UpscaleBlock(tf.keras.layers.Layer):
    def __init__(self, number, kernel_size=3, activation=tf.nn.swish):
        super().__init__(name="UpscaleBlock" + str(number))
        self.activation = activation
        self.kernel_size = kernel_size
        self.is_built = False

    def build(self, x):
        channels = x.shape.as_list()[-1]
        filters = channels // 2

        bn1 = tf.keras.layers.BatchNormalization()
        conv1 = upscale_block(filters)
        bn2 = tf.keras.layers.BatchNormalization()
        conv2 = tf.keras.layers.Conv2D(filters,
                                       self.kernel_size,
                                       padding='same')
        self.main_network = [self.activation, bn1, conv1, self.activation,

        self.skip_connection = upscale_block(filters)
        self.se_activate = SqueezeExcite(filters)
        self.is_built = True

    def __call__(self, input_):
        if not self.is_built:
            self.build(input_)
        x = input_
        for layer in self.main_network:
            x = layer(x)
        output = x
        skip = self.skip_connection(input_)
        return skip + 0.1 * output


class FactorizedReduce(tf.Module):
    """Downscale version of the sub-pixel convolution which re-arranges pix
    def __init__(self, channels):
        super(FactorizedReduce, self).__init__()
        assert channels % 2 == 0
        self.conv_1 = tf.keras.layers.Conv2D(channels // 4, 1, strides=2)
        self.conv_2 = tf.keras.layers.Conv2D(channels // 4, 1, strides=2)
        self.conv_3 = tf.keras.layers.Conv2D(channels // 4, 1, strides=2)
        self.conv_4 = tf.keras.layers.Conv2D(channels - 3 * (channels // 4)
```

```
                                            1,
                                            strides=2)
        self.convs = [self.conv_1, self.conv_2, self.conv_3, self.conv_4]

    def __call__(self, x):
        """Assumes NHCW data"""
        assert x.shape[2] > 1
        assert x.shape[3] > 1
        out = tf.nn.swish(x)
        conv1 = self.conv_1(out)
        conv2 = self.conv_2(out[:, :, 1:, 1:])
        conv3 = self.conv_3(out[:, :, :, 1:])
        conv4 = self.conv_4(out[:, :, 1:, :])
        out = tf.concat([conv1, conv2, conv3, conv4], -1)
        return out


class SqueezeExcite(tf.Module):
    """Activation function that performs gating"""
    def __init__(self, out_channels):
        super().__init__()
        num_hidden = max(out_channels // 16, 4)
        self.net = tf.keras.Sequential([
            tf.keras.layers.Dense(num_hidden), tf.keras.layers.Lambda(tf.nn
            tf.keras.layers.Dense(out_channels), tf.keras.layers.Lambda(tf.
        ])

    def __call__(self, x):
        """The choice of axes assumes we're working with NHWC data"""
        ax = tf.math.reduce_mean(x, axis=[1, 2])
        # data should be flat at this po,int
        bx = self.net(ax)
        cx = tf.expand_dims(tf.expand_dims(bx, 1), 1)
        return cx * x


class DownscaleBlock(tf.keras.layers.Layer):
    def __init__(self, number, kernel_size=3, activation=tf.nn.swish):
        super().__init__(name="DownscaleBlock" + str(number))
        self.activation = activation
        self.kernel_size = kernel_size
        self.is_built = False

    def build(self, x):
        channels = x.shape.as_list()[-1]
        filters = channels * 2

        bn1 = tf.keras.layers.BatchNormalization()
        conv1 = tf.keras.layers.Conv2D(filters,
                                       self.kernel_size,
                                       strides=2,
                                       padding='same')
        bn2 = tf.keras.layers.BatchNormalization()
        conv2 = tf.keras.layers.Conv2D(filters,
                                       self.kernel_size,
                                       padding='same')
        self.main_network = [self.activation, bn1, conv1, self.activation,
```

```
        self.skip_connection = FactorizedReduce(filters)
        self.se_activate = SqueezeExcite(filters)
        self.is_built = True

    def __call__(self, input_):
        if not self.is_built:
            self.build(input_)
        x = input_
        for layer in self.main_network:
            x = layer(x)
        output = x
        skip = self.skip_connection(input_)
        return skip + 0.1 * output
```

In [241]:

```
#code 2
encoder_network = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, padding='same',
                           activation=tf.nn.swish),  #28,28,16
    DownscaleBlock(1),  # 14,14,32
    DownscaleBlock(2),  # 7,7,64
    tf.keras.layers.Conv2D(64, 3, padding='same',
                           activation=tf.nn.swish),  # 7,7,64
    tf.keras.layers.Conv2D(16, 3, padding='same',
                           activation=tf.nn.swish),  # 7,7,16
    tf.keras.layers.Conv2D(1, 3, padding='same'),  # 7,7,1
])

decoder_network = tf.keras.Sequential([
    tf.keras.layers.Conv2D(4, 3, padding='same',
                           activation=tf.nn.swish),  # 7,7,4
    tf.keras.layers.Conv2D(16, 3, padding='same',
                           activation=tf.nn.swish),  # 7,7,16
    tf.keras.layers.Conv2D(64, 3, padding='same',
                           activation=tf.nn.swish),  # 7,7,64
    UpscaleBlock(1),  # 14,14,32
    UpscaleBlock(2),  # 28,28,16
    tf.keras.layers.Conv2D(4, 3, padding='same',
                           activation=tf.nn.swish),  #28,28,4
    tf.keras.layers.Conv2D(1, 3, padding='same'),  #28,28,1
])
```

In [208]:

```
for batch in train_ds:
    x = tf.cast(batch['image'], tf.float32)
    code = encoder_network(x)
    output = decoder_network(code)
    break
print(x.shape, code.shape, output.shape)
```

```
(32, 28, 28, 1) (32, 7, 7, 16) (32, 28, 28, 1)
```

In [226]:
```
loss_results = []

NOISE_COEFF = 10.

def autoencoder_loss(x, x_hat):
    reconstruction_loss = tf.reduce_mean(tf.square(x_hat - x)) # Mean Squar
    total_loss = reconstruction_loss
    return total_loss

max_steps = 250
learning_rate = 1e-3
step = 0
optimizer = tf.keras.optimizers.Adam()
for batch in tqdm(train_ds):
    with tf.GradientTape() as tape:
        x = tf.cast(batch['image'], tf.float32)
        code = encoder_network(x + tf.random.normal(x.shape))
        output = decoder_network(code)
        loss = autoencoder_loss(x, output)
    gradient = tape.gradient(loss, encoder_network.trainable_variables + de
    optimizer.apply_gradients(zip(gradient, encoder_network.trainable_varia

    loss_results.append(np.mean(loss))

    step += 1
    if step > max_steps:
        break
```

```
13%|█         | 250/1875 [00:11<01:14, 21.85it/s]
```

In [177]:
```
#used UpscaleBlock & DownscaleBlock - tf.keras.layers.Conv2D(1, 3, padding=
loss_array1 = np.array(loss_results)
```

In [181]:
```
#used UpscaleBlock & DownscaleBlock - tf.keras.layers.Conv2D(7, 3, padding=
loss_array2 = np.array(loss_results)
```

In [185]:
```
#used UpscaleBlock & DownscaleBlock - tf.keras.layers.Conv2D(16, 3, padding
loss_array3 = np.array(loss_results)
```

In [211]:
```
#used upscale_block & DownscaleBlock - tf.keras.layers.Conv2D(16, 3, paddin
loss_array4 = np.array(loss_results)
```

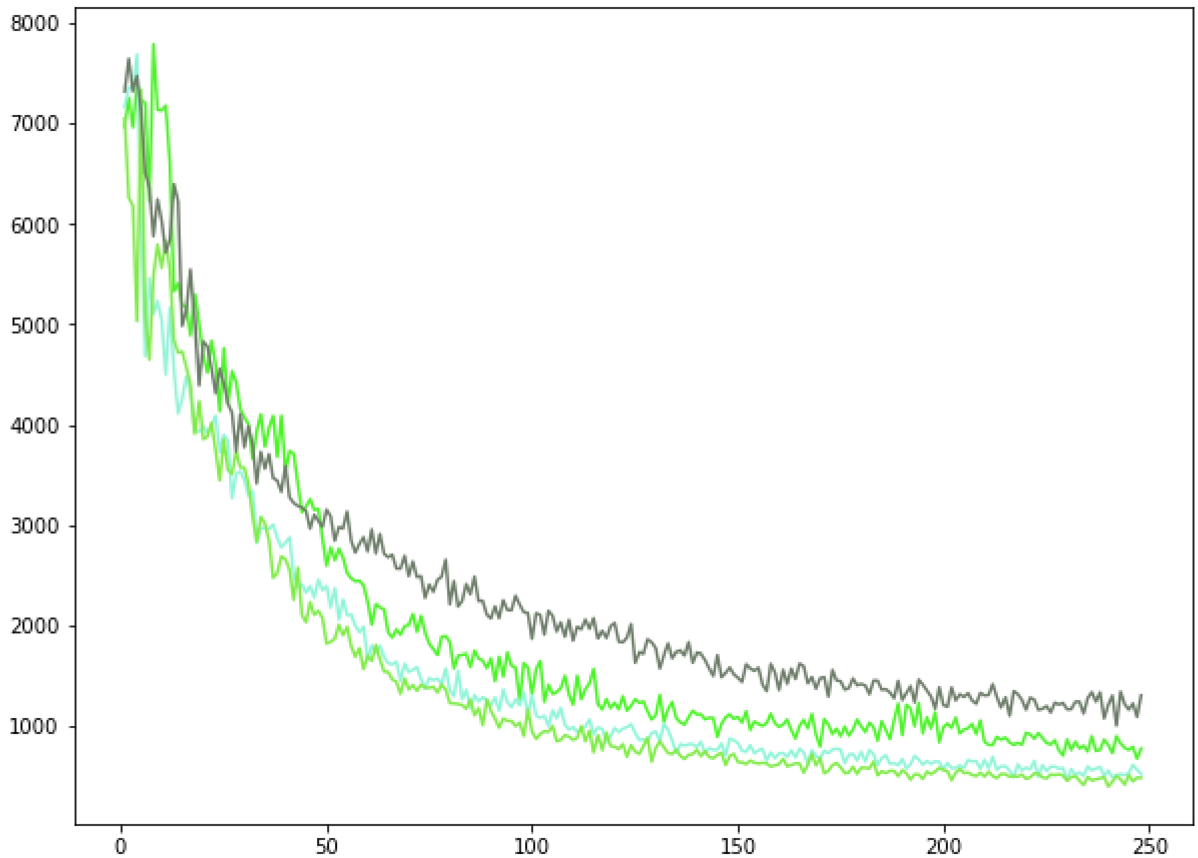
In [217]:
```
a = loss_array1, loss_array2, loss_array3, loss_array4
```

In [220]:
```
b = [x for x in range(1,249)]
```

In [221]:
```
import random
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

# define a and b here

# Helps pick a random color for each plot, used for readability
rand = lambda: random.randint(0, 255)
fig = plt.figure(figsize=(10,7.5))
ax = fig.add_subplot(111)
for ydata in a:
    clr = '#%02X%02X%02X' % (rand(),rand(),rand())
    plot, = ax.plot(b, ydata, color=clr)
```



In [189]:
```
def best_fit_line(x_values, y_values):
    """Returns slope and y-intercept of the best fit line of the values"""

    mean = lambda l: sum(l)/len(l)
    multiply = lambda l1, l2: [a*b for a, b in zip(l1, l2)]

    m = ( (mean(x_values)*mean(y_values) - mean(multiply(x_values, y_values
          (mean(x_values)**2              - mean(multiply(x_values, x_values

    b = mean(y_values) - m*mean(x_values)

    return m, b
```

In [224]:
```
for i in a:
    print(best_fit_line(b,i))
```

```
(-18.314547752464488, 4251.1073392858325)
(-15.635452206894612, 3477.724421672166)
(-15.60919271338766, 3364.6420472020413)
(-16.340339041643293, 4360.947656044343)
```

autoencoders are neural networks that learn datarepresentations in an unsupervised manner. The encoder learns the compact representation of the input data, and the decoder decompresses to reconstruct the input data. Since we are working with image data, we are using a convolutional autoencoder which consists of convolutional layers and pooling layers which downsample. the decoder upsamples the image. When changing the code.shape in the last layer of the encoder (tf.keras.layers.Conv2D(n, 3, padding='same')), there is a change in the loss or MSE. when the filter number is changed from 1 -> 7 -> 16, the slope of the loss becomes less negative, meaning that the change of loss over steps reduces, and also the y-intercept is reduced. This has to do with the reduced downscaling of the final layer that feeds into the decoder.