portfolio_8

May 25, 2023

1 The Auto Encoder

In this extended portfolio we created and trained an Auto Encoder (AE) on the CIFAR100 dataset. The aim is to learn a good low-dimensional representation of the input, which in this case is 32*32 pixels RGB images. This is useful in the same way that dimensionality reduction techniques such as PCA are, the latent representation we learn could tell us something useful about the data or we could use this latent representation as an input to another model, the benefit of this being that it should make the downstream task the model is performing simpler as it's working on a much simpler and more useful input (if our encoding is good!).

The AE is a neural network that is made up of two "sub-networks" an encoder and decoder. The encoder is a neural architecture where the dimensionality of the layers is decreasing as we go deeper in the encoder, the decoder take as input the output of the encoder and its structure does the opposite of the encoder with it's layers increasing in dimension with its output layer having the same dimensionality as the input layer of the input to the encoder.

We then train the AE to reconstruct its input, ie. the output of the network should be as close as possible to the input to the network. The idea being that the encoder layers have to extract useful features from the image in order to create a low-dimensional representation of the image that has as much useful information as possible such that the decoder can then reconstruct the original image from this latent representation created by the encoder. The lowest-dimensional layer in the AE (the output layer of the encoder) is called the bottleneck layer.

We used the following architecture for our AE network:

[ ]:
```
import VAE
model = VAE.AutoEncoder.load_from_checkpoint("saved_models_CIFAR10/
  ↪autoencoder_512.ckpt")
print(model)
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models_CIFAR10/autoencoder_512.ckpt`
```

```
AutoEncoder(
  (encoder): Encoder(
    (net): Sequential(
      (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU()
```

```
      (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU()
      (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (7): ReLU()
      (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      (9): Flatten(start_dim=1, end_dim=-1)
      (10): Linear(in_features=1024, out_features=512, bias=True)
    )
  )
  (decoder): Decoder(
    (linear): Sequential(
      (0): Linear(in_features=512, out_features=1024, bias=True)
      (1): ReLU()
    )
    (net): Sequential(
      (0): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): ReLU()
      (2): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (3): ReLU()
      (4): ConvTranspose2d(16, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
      (5): Tanh()
    )
  )
)
```

As you can see the encoder and decoder are convolutional and make use of the ReLU activation function. In the encoder we also use maxpooling layers which help reduce dimensionality and we found they also greatly boosted performance. At the end of the encoder we also had a fully connected layer which allowed us to adjust the dimensionality of the latent space at the bottleneck layer by adjusting its size, here the size is 512. The number of layers, activation function, kernel sizes etc. were chosen through experimentation with different architectures and choosing the architecture that performed best.

We used pytorch and lightning to write the code for the architectures and for training them. Lightning was specifically useful as it carried out automatic checkpointing, learning rate scheduling, kept track of and helped visualize (through tensorboard) training and testing loss, could parallelize work over multiple GPU's and can reschedule training on SLURM if time for the job is coming to an end. We trained our AE models on BluePebble which greatly speed up training of the models and allowed for many variations of the network to be trained in parallel greatly reducing the amount of time needed to compare performance of different network architectures and selecting hyperparameters.

Let's now investigate our results, first, by exploring what the dataset looks like, we will now run some code chunks which will print out a grid of 4 of the images in CIFAR100 (training set) along with their labels:

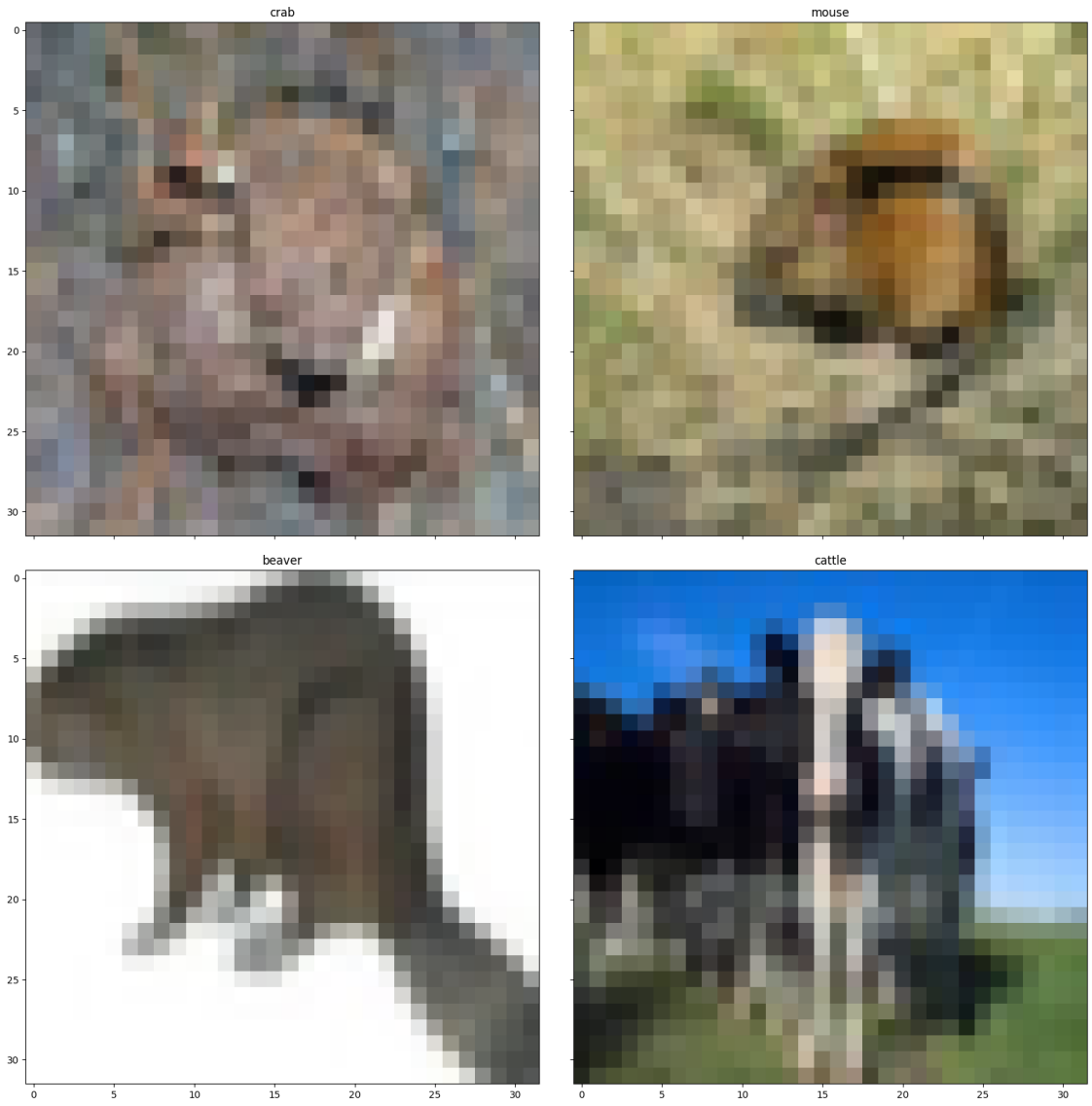
[ ]:
```
# Do some imports from libraries and custom python scripts
import data
import matplotlib.pyplot as plt
import torch.cuda as cuda
import numpy as np
import torch
import torchvision

# We check the device
device = "cuda" if cuda.is_available() else "cpu"
print("Using device: ", device)

# Create the training and testing dataloaders and extract some images and their␣
 ↪labels from the training set and test set
training_loader, test_loader = data.get_data_loader("CIFAR100", 64, device)
train_images, train_targets = next(iter(training_loader))
test_images, test_targets = next(iter(test_loader))
```

```
Using device:  cpu
Files already downloaded and verified
Files already downloaded and verified
```

[ ]:
```
# We will now create a function which given the targets will return the␣
 ↪corresponding labels
def cifar10_label(target):
    """
    Returns the label for a given target value in the CIFAR10 dataset.
    """
    cifar10_labels = [
        'airplane', 'automobile', 'bird', 'cat', 'deer',
        'dog', 'frog', 'horse', 'ship', 'truck'
    ]
    return [cifar10_labels[t] for t in target]

def cifar100_label(target):
    """
    Returns the label for a given target value in the CIFAR100 dataset.
    """
    cifar100_labels = [
        'apple', 'aquarium_fish', 'baby', 'bear', 'beaver',
        'bed', 'bee', 'beetle', 'bicycle', 'bottle', 'bowl',
        'boy', 'bridge', 'bus', 'butterfly', 'camel', 'can',
        'castle', 'caterpillar', 'cattle', 'chair', 'chimpanzee',
        'clock', 'cloud', 'cockroach', 'couch', 'crab', 'crocodile',
```

```
        'cup', 'dinosaur', 'dolphin', 'elephant', 'flatfish',
        'forest', 'fox', 'girl', 'hamster', 'house', 'kangaroo',
        'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
        'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle',
        'mountain', 'mouse', 'mushroom', 'oak_tree', 'orange',
        'orchid', 'otter', 'palm_tree', 'pear', 'pickup_truck',
        'pine_tree', 'plain', 'plate', 'poppy', 'porcupine',
        'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket',
        'rose', 'sea', 'seal', 'shark', 'shrew', 'skunk', 'skyscraper',
        'snail', 'snake', 'spider', 'squirrel', 'streetcar', 'sunflower',
        'sweet_pepper', 'table', 'tank', 'telephone', 'television',
        'tiger', 'tractor', 'train', 'trout', 'tulip', 'turtle',
        'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman',
        'worm'
    ]
    return [cifar100_labels[t] for t in target]
```

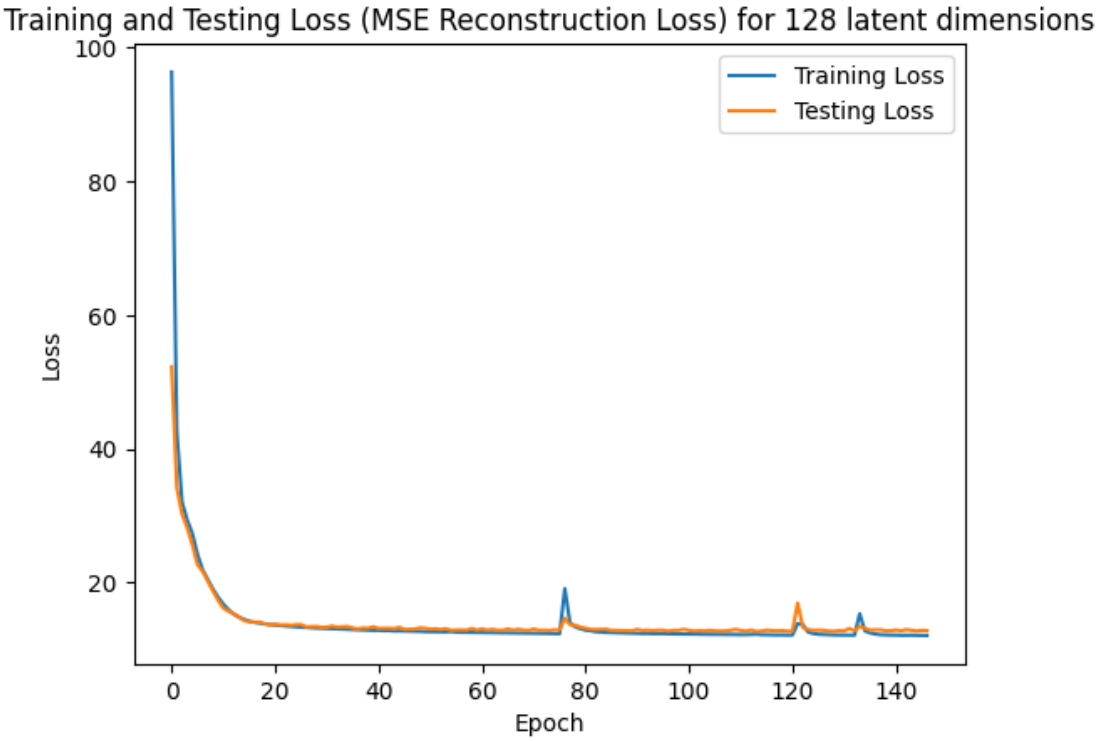
[ ]:
```
# Let's now print the image grid
from mpl_toolkits.axes_grid1 import ImageGrid

fig = plt.figure(figsize=(20., 20.))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2, 2),  # creates 2x2 grid of axes
                 axes_pad=0.5,  # pad between axes
                 )
labels = cifar100_label(train_targets[:4])
for i, (ax, im) in enumerate(zip(grid, train_images[:4])):
    ax.imshow(np.transpose(im.numpy(), (1, 2, 0)))
    ax.set_title('{0}'.format(labels[i]))

plt.show()
```



Above we can see some of the images from CIFAR100 with their class label written above. First we will look at how the AE performs after being trained on the CIFAR100 training set with a latent dimension size of 128, as we mentioned earlier we trained the AE on the HPC and made use of parallelization built into the lightning package to speed up the training process. Let's first look at what the training process looked like by plotting the mean reconstruction loss on the testing and training sets for each epoch.

[ ]:
```
# Create plot of the training and testing loss from csv file
import pandas as pd
import matplotlib.pyplot as plt

# Create plot of the training and testing accuracy from csv file
def plot_loss(l_dim):
```

```
    # Read the csv file
    df_train = pd.read_csv("results/ae_" + str(l_dim) +"_train.csv")
    df_test = pd.read_csv("results/ae_" + str(l_dim) + "_test.csv")

    # Create the plot
    plt.plot(df_train["Value"], label="Training Loss")
    plt.plot(df_test["Value"], label="Testing Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training and Testing Loss (MSE Reconstruction Loss) for " +␣
 ↪str(l_dim) + " latent dimensions")
    plt.legend()
    plt.show()

# Use the function we just created to plot the loss
plot_loss(128)
```



Let's now run some images through the encoder part of the autoencoder and then through the decoder:

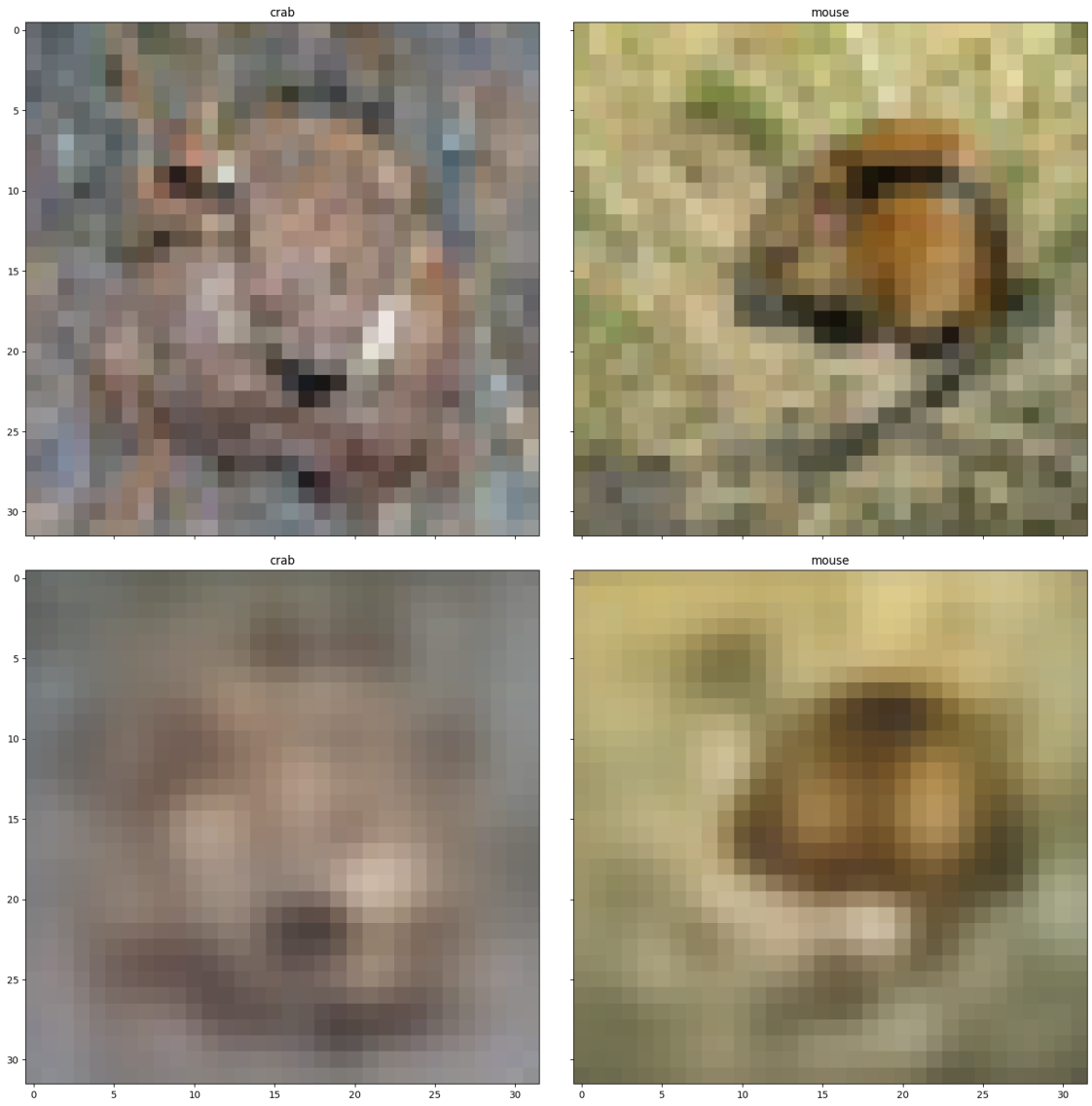
[ ]:
```
import VAE
# Load in our model that we trained on BluePebble
```

```
model = VAE.AutoEncoder.load_from_checkpoint("saved_models/autoencoder_128.
  ↪ckpt")

# disable randomness, dropout, etc...
model.eval()

# Let's encode two of the images
encoded_images = model.encoder(train_images[:2])

# Let's now decode two of the images
decoded_images = model.decoder(encoded_images)
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_128.ckpt`
```

We've now encoded and decoded two of our images, let's plot our decoded images from the autoencoder against the original images input to the network:

[ ]:
```
fig = plt.figure(figsize=(20., 20.))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2, 2),  # creates 2x2 grid of axes
                 axes_pad=0.5,  # pad between axes
                 )
labels = cifar100_label(train_targets[:2])
for i, ax in enumerate(grid):
    if i==0 or i==1 :
        ax.imshow(np.transpose(train_images[i].numpy(), (1, 2, 0)))
        ax.set_title('{0}'.format(labels[i]))
    if i==2 or i==3:
        ax.imshow(np.transpose(decoded_images[i-2].detach().numpy(), (1, 2, 0)))
        ax.set_title('{0}'.format(labels[i-2]))

plt.show()
```

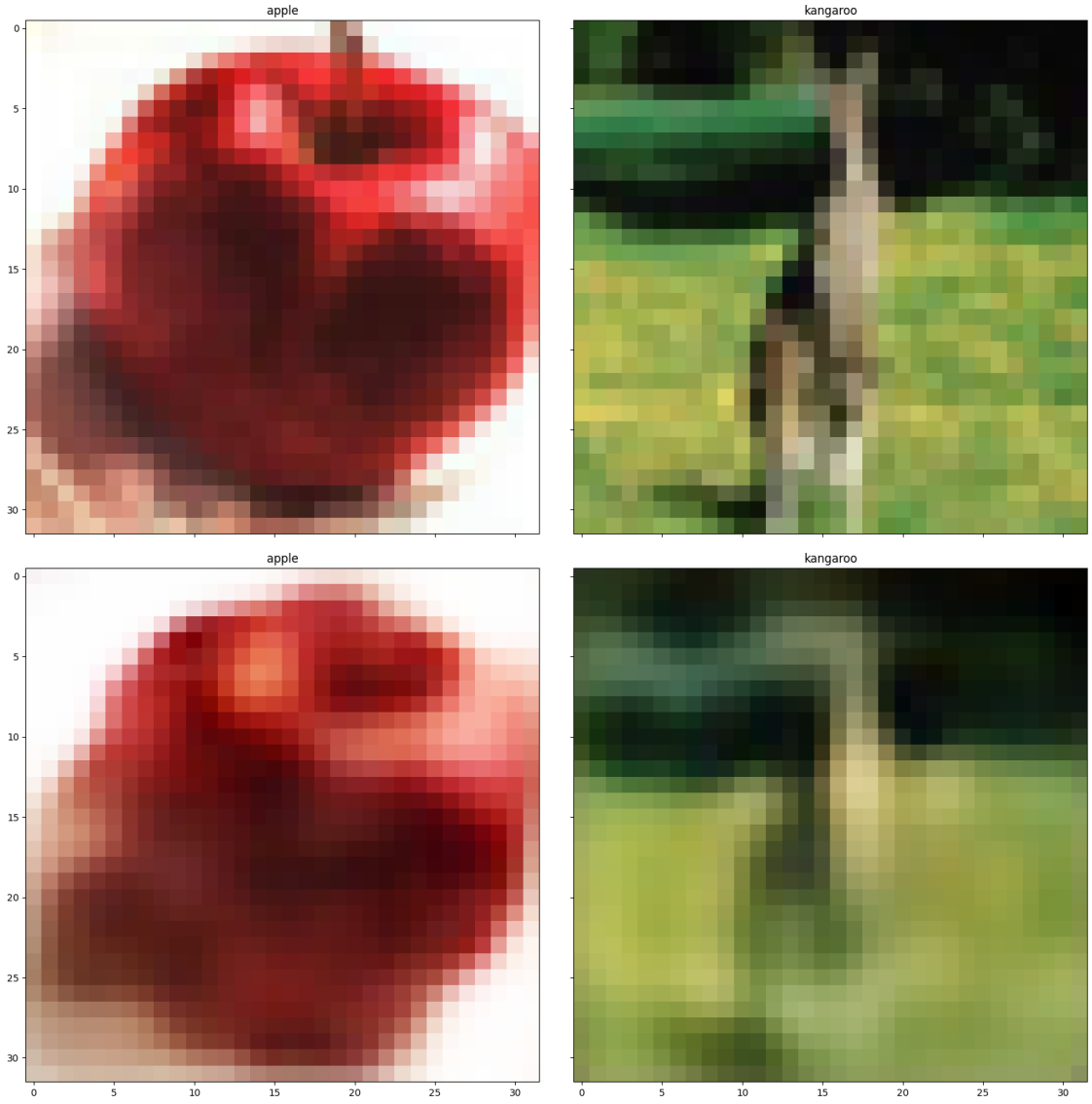


We see that the decoded images have a strong resemblance to the original images, but note that these images are from the training set, meaning our model was optimized to work on these images. Let's now take a look at what our reconstructed images look like when we use images from the testing set:

[ ]:
```
# Let's encode two of the images (from the test set)
encoded_images = model.encoder(test_images[:2])

# Let's now decode two of the images
decoded_images = model.decoder(encoded_images)

# Let's now print the image grid
fig = plt.figure(figsize=(20., 20.))
```

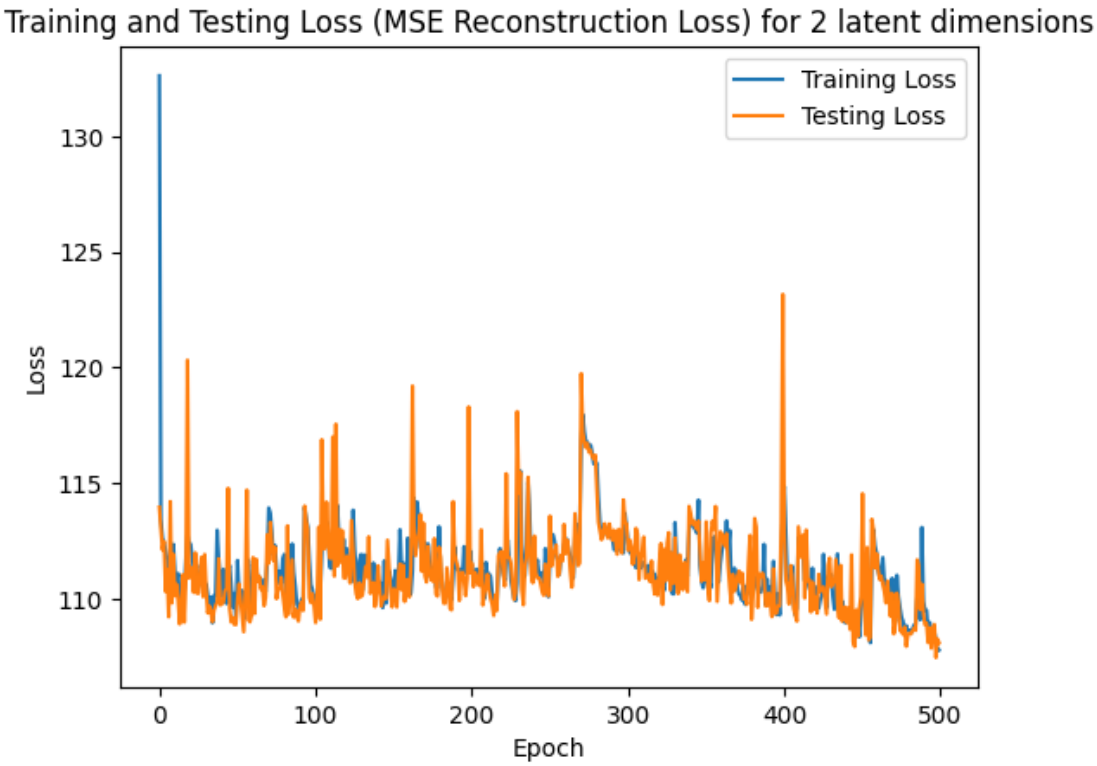
```
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2, 2),  # creates 2x2 grid of axes
                 axes_pad=0.5,  # pad between axes
                 )
labels = cifar100_label(test_targets[:2])
for i, ax in enumerate(grid):
    if i==0 or i==1 :
        ax.imshow(np.transpose(test_images[i].numpy(), (1, 2, 0)))
        ax.set_title('{0}'.format(labels[i]))
    if i==2 or i==3:
        ax.imshow(np.transpose(decoded_images[i-2].detach().numpy(), (1, 2, 0)))
        ax.set_title('{0}'.format(labels[i-2]))

plt.show()
```

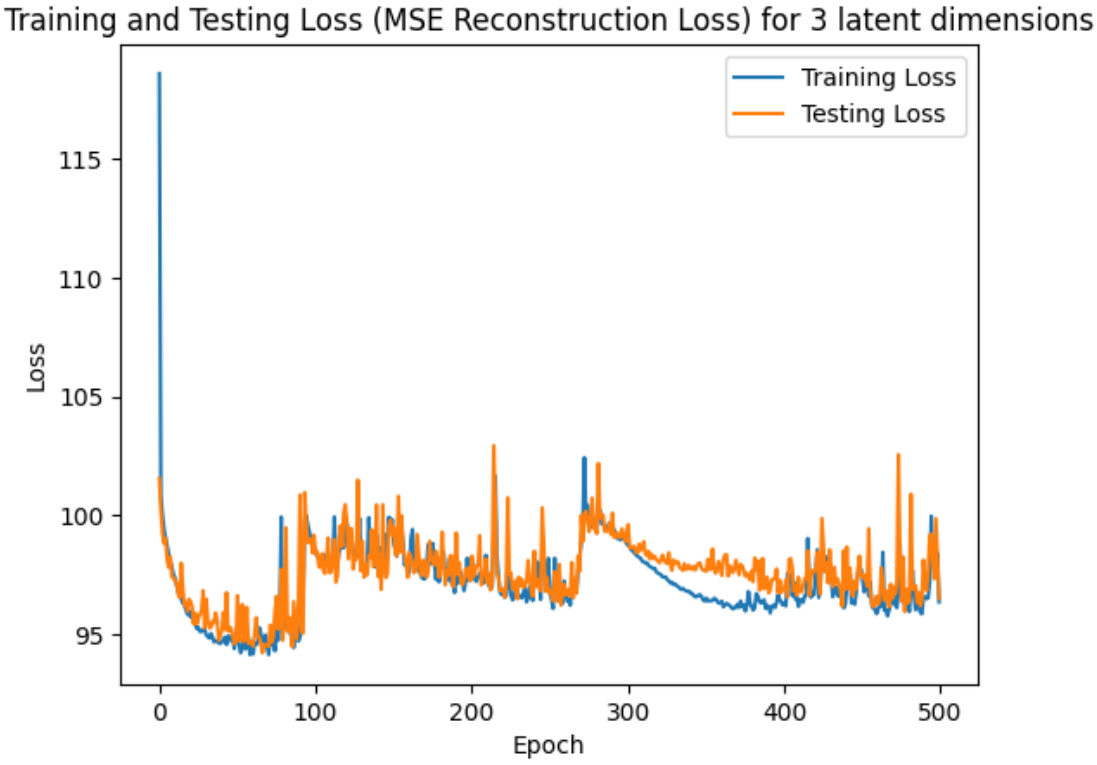
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



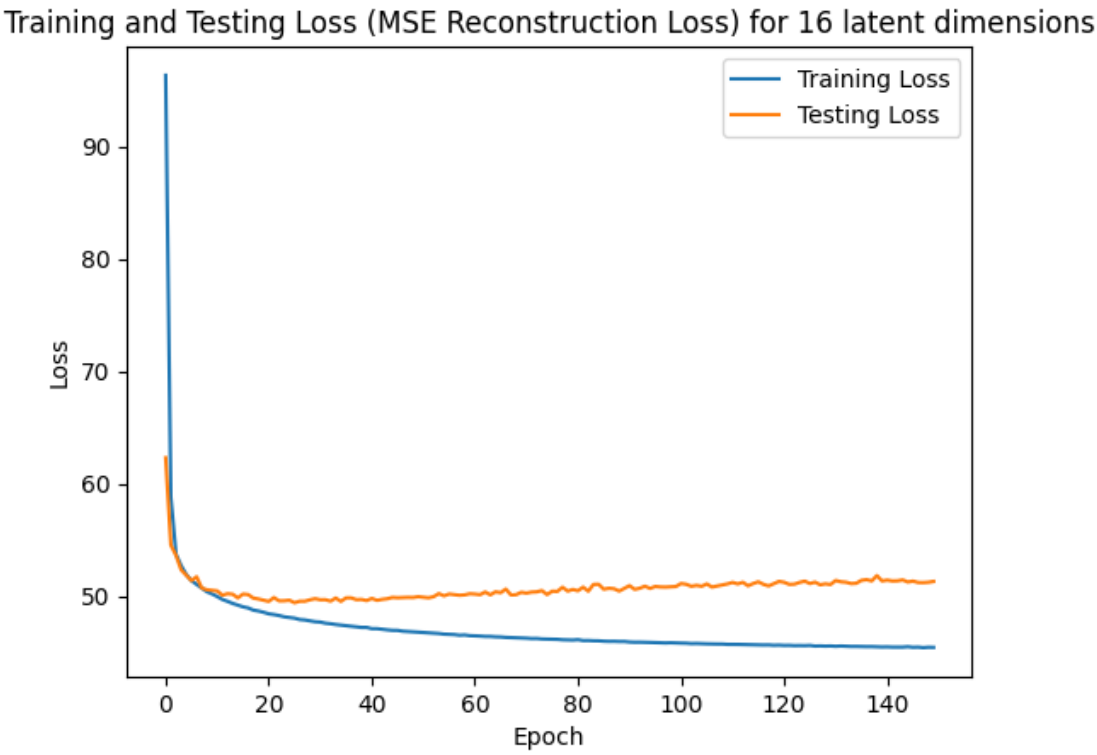
Again it appears qualitatively that our autoencoder is successfully reconstructing these images very well! and this time on data from the test set, that it hasn't seen before, so it appears the autoencoder is successfully encoding the images into a small latent representation and then is successfully able to reconstruct the original image from this latent representation (not pixel for pixel, but close in appearance!). So far we have explored the performance of the autoencoder with a latent dimension of 128, but how would the autoencoder perform if we change the size of the latent dimension? This is what we will investigate now. On BluePebble I have trained a variety of autoencoders with differing latent dimensions, let's now observe how they did during training:

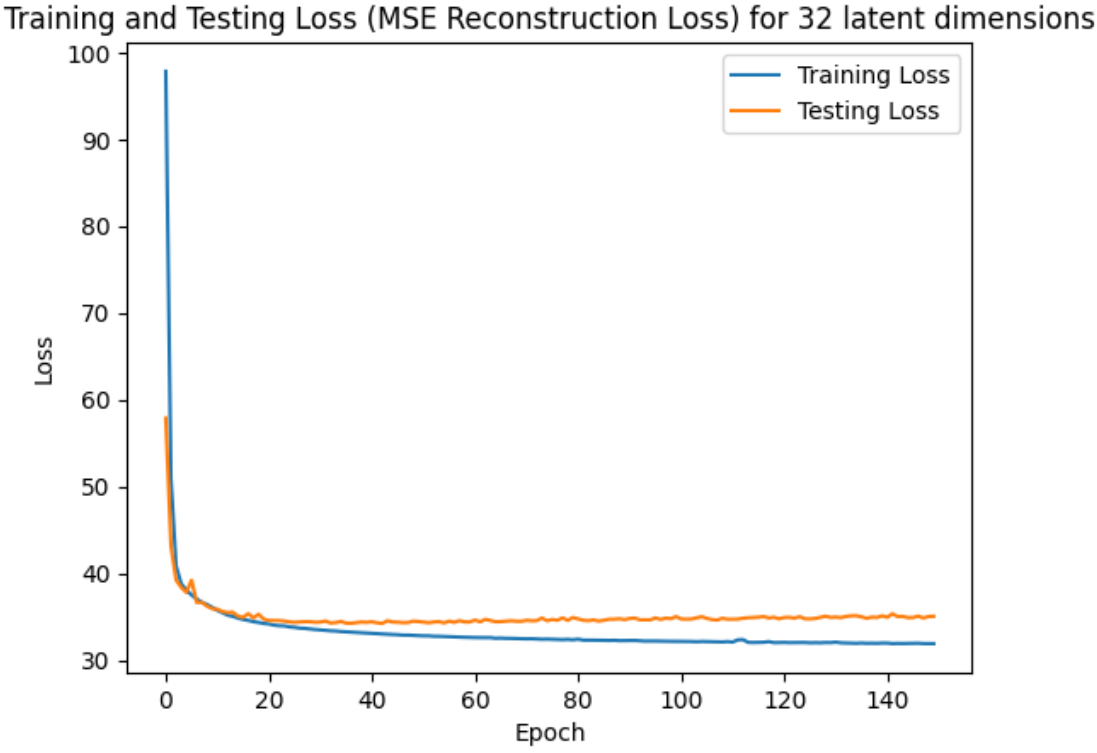
[ ]:
```
lat_dims = [2, 3, 8, 16, 32, 64, 128, 256, 512]
for i in lat_dims:
    plot_loss(i)
```

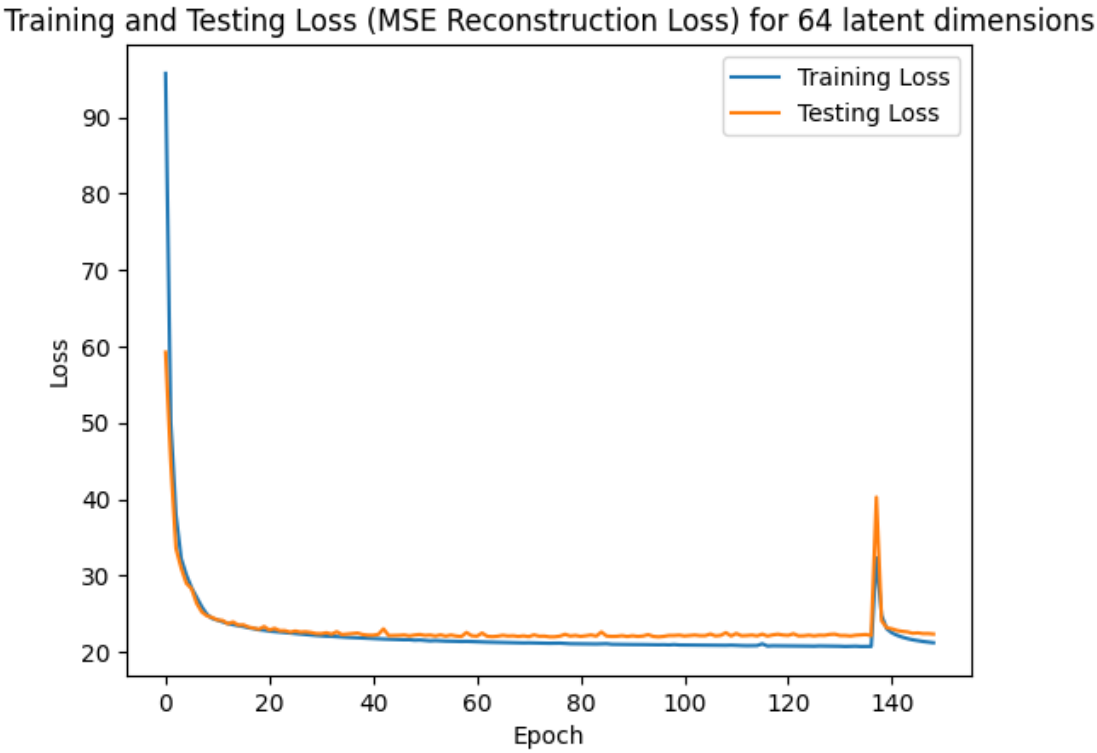

Training and Testing Loss (MSE Reconstruction Loss) for 2 latent dimensions
Training Loss
Testing Loss
Loss
130
125
120
115
110
0
100
200
300
400
500
Epoch

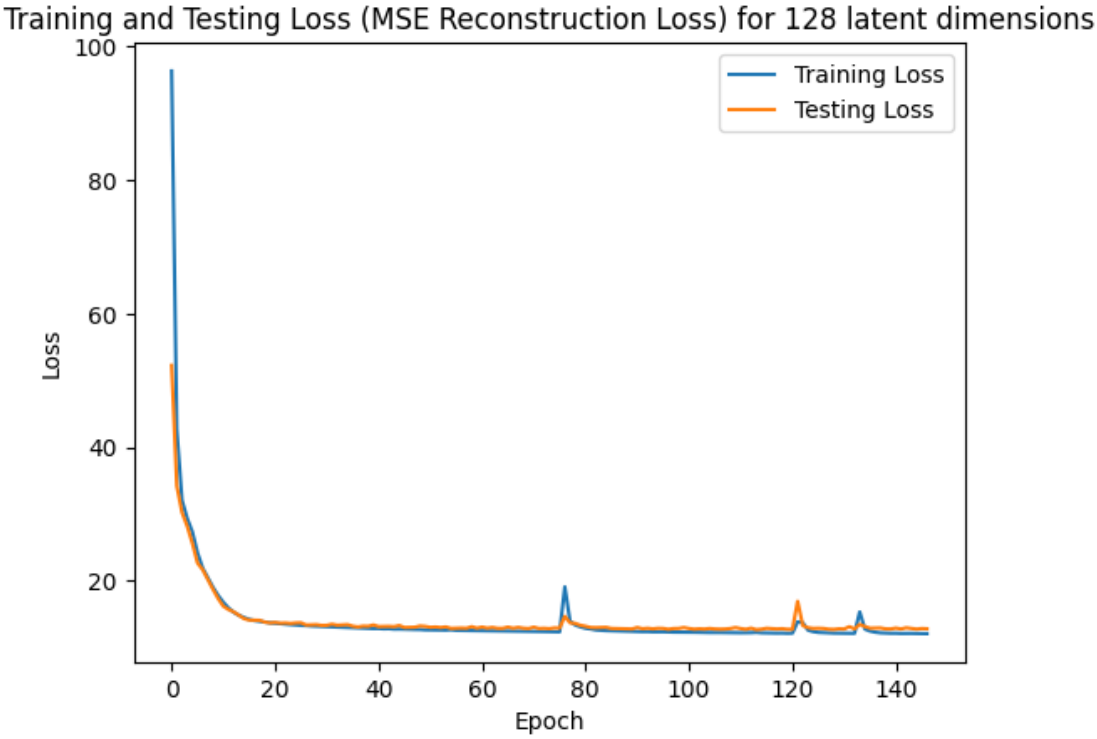

Training and Testing Loss (MSE Reconstruction Loss) for 3 latent dimensions
Training Loss
Testing Loss
Loss
115
110
105
100
95
0
100
200
300
400
500
Epoch

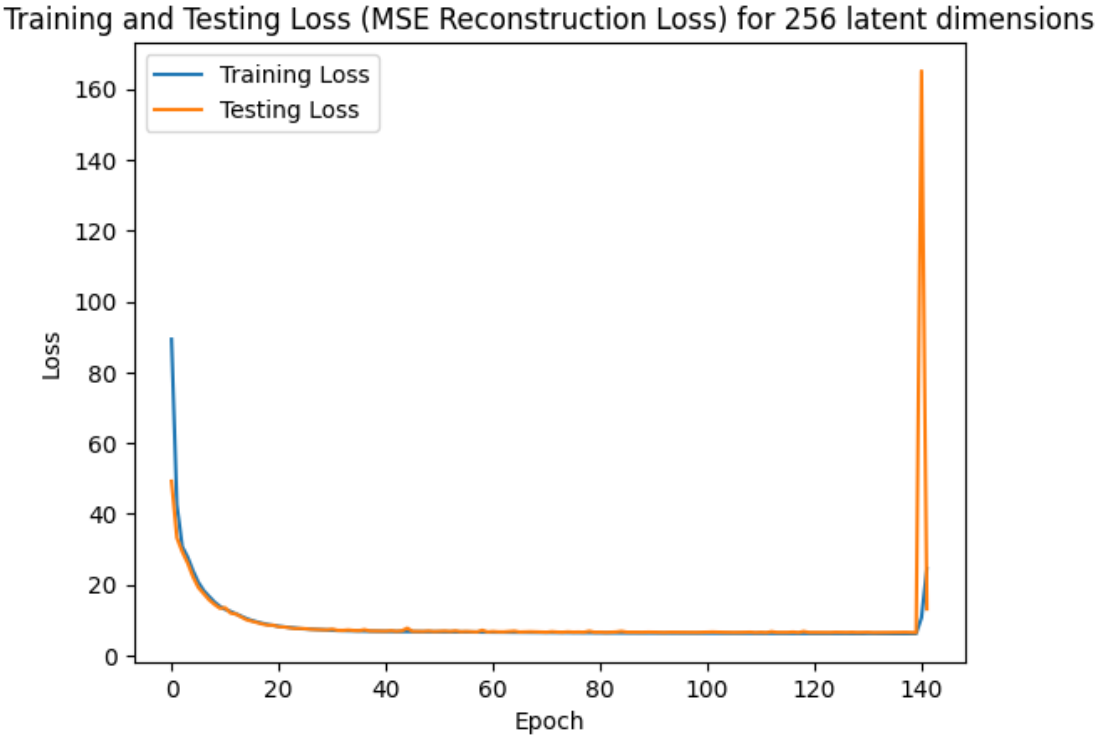

Training and Testing Loss (MSE Reconstruction Loss) for 8 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
95
90
85
80
75
70
65
60
0
20
40
60
80
100
120
140

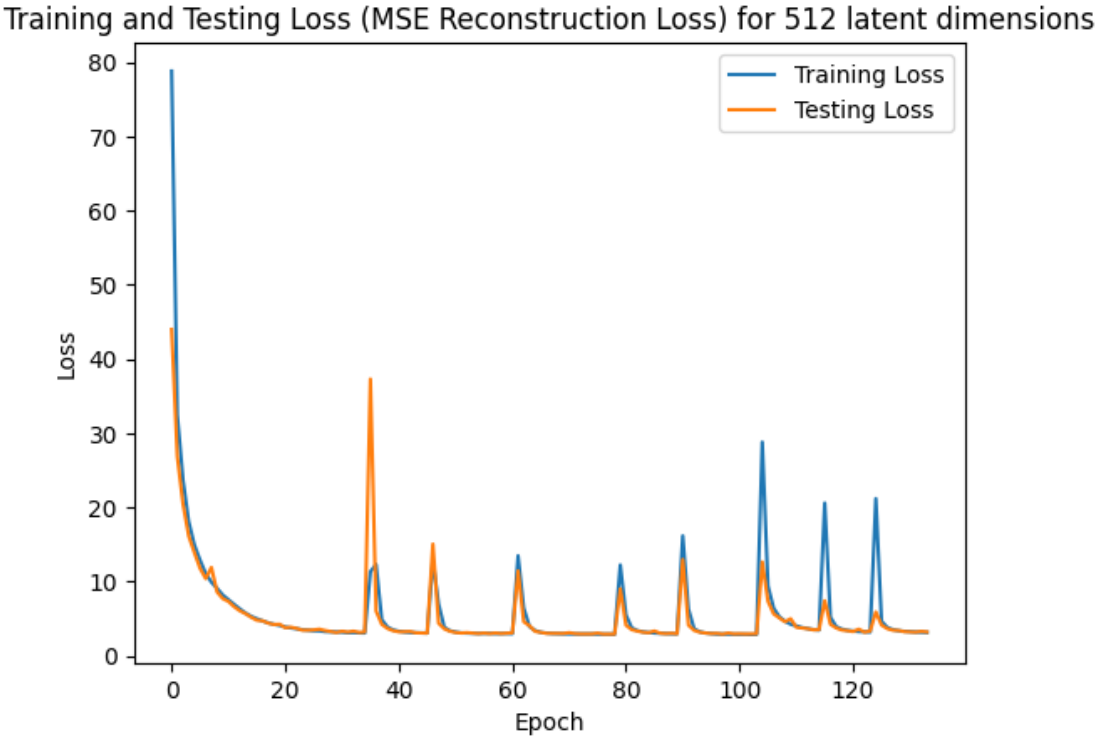

Training and Testing Loss (MSE Reconstruction Loss) for 16 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
90
80
70
60
50
0
20
40
60
80
100
120
140


Training and Testing Loss (MSE Reconstruction Loss) for 32 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
100
90
80
70
60
50
40
30
0
20
40
60
80
100
120
140


Training and Testing Loss (MSE Reconstruction Loss) for 64 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
90
80
70
60
50
40
30
20
0
20
40
60
80
100
120
140


Training and Testing Loss (MSE Reconstruction Loss) for 128 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
100
80
60
40
20
0
20
40
60
80
100
120
140


Training and Testing Loss (MSE Reconstruction Loss) for 256 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
0
20
40
60
80
100
120
140
160


Training and Testing Loss (MSE Reconstruction Loss) for 512 latent dimensions
Training Loss
Testing Loss
Loss
Epoch
0
10
20
30
40
50
60
70
80
100
120

The first thing we notice is that for small latent dimensions (namely 2 and 3), the training loss is extremely noisy, ie. it has high variability especially in contrast to our other plots, we also notice that the training loss goes through periods of increase before decreasing again and that the number of epochs we have to train for is much larger (it hits the max of 500 epochs). It is worth noting that we train the AEs until either the loss is no longer improving or we've reached 500 epochs of training. For latent dimensions 8, 16 and 32 we see less noise in the loss and it takes many fewer epochs for training to finish, we do see the testing loss starting to slowly rise in all of these plots though showing the beginnings of our AEs over fitting. For latent dimensions 64, 128, 256 and 512 we see again the testing and training loss initially decreasing, but this time without the testing loss starting to increase later on, we do however now see spikes where the training and testing loss increase. Overall these plots are what we might expect from a model that is succeeding in learning something from the data, except for with latent dimensions 2 and 3 where the loss is extremely variable and it struggles to reduce the loss to small values. This indicates that perhaps for very small latent dimensions the AEs struggle to reduce reconstruction loss on the data as the latent dimension is too small to encode enough information about the input image.

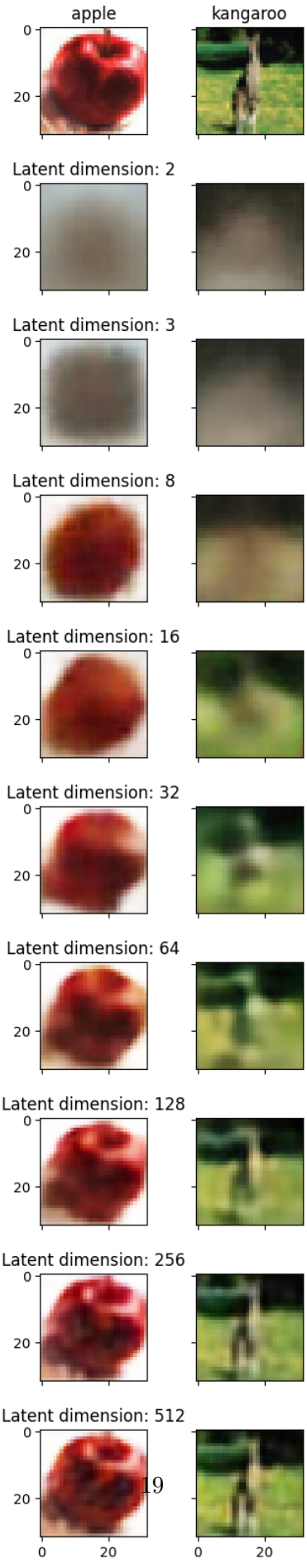
Now let's plot the reconstruction of one of the images (from the test set) done by each model so that we can qualitatively compare the differences between the models:

[ ]:
```
encoded_images = []
decoded_images = []
for i in lat_dims:
    # Load in our model that we trained on BluePebble
    model = VAE.AutoEncoder.load_from_checkpoint("saved_models/
 ↪autoencoder_"+str(i)+".ckpt")

    # disable randomness, dropout, etc...
    model.eval()
    # Let's encode two of the images (from the test set)
    enc = model.encoder(test_images[:2])
    encoded_images.append(enc)

    # Let's now decode two of the images
    decoded_images.append(model.decoder(enc))

# Let's now print the image grid
fig = plt.figure(figsize=(20., 20.))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(10, 2),
                 axes_pad=0.5,
                 )
labels = cifar100_label(test_targets[:2])
c=0
for i, ax in enumerate(grid):
    if i < 2:
```

```
        ax.imshow(np.transpose(test_images[i].numpy(), (1, 2, 0)))
        ax.set_title('{0}'.format(labels[i]))
    else:
        if i%2 == 0 and i != 2:
            c +=1
        ax.imshow(np.transpose(decoded_images[c][i%2].detach().numpy(), (1, 2,␣
↪0)))
        if i%2 == 0:
            ax.set_title('Latent dimension: {0}'.format(lat_dims[c]))

plt.show()
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_2.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_3.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_8.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_16.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_32.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_64.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_128.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_256.ckpt`
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
```

```
saved_models/autoencoder_512.ckpt`
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```


apple
kangaroo
Latent dimension: 2
Latent dimension: 3
Latent dimension: 8
Latent dimension: 16
Latent dimension: 32
Latent dimension: 64
Latent dimension: 128
Latent dimension: 256
Latent dimension: 512
0
20

Looking at the reconstructed images above we see a very definite story, as the number of latent dimensions increases so does the resemblance between it and the input image, ie. as the amount of information that the autoencoder is able to encode with increases, it's ability to reconstruct the original image increases, which makes sense as it can encode more information about the original image in its latent representation in bottleneck of the AE.

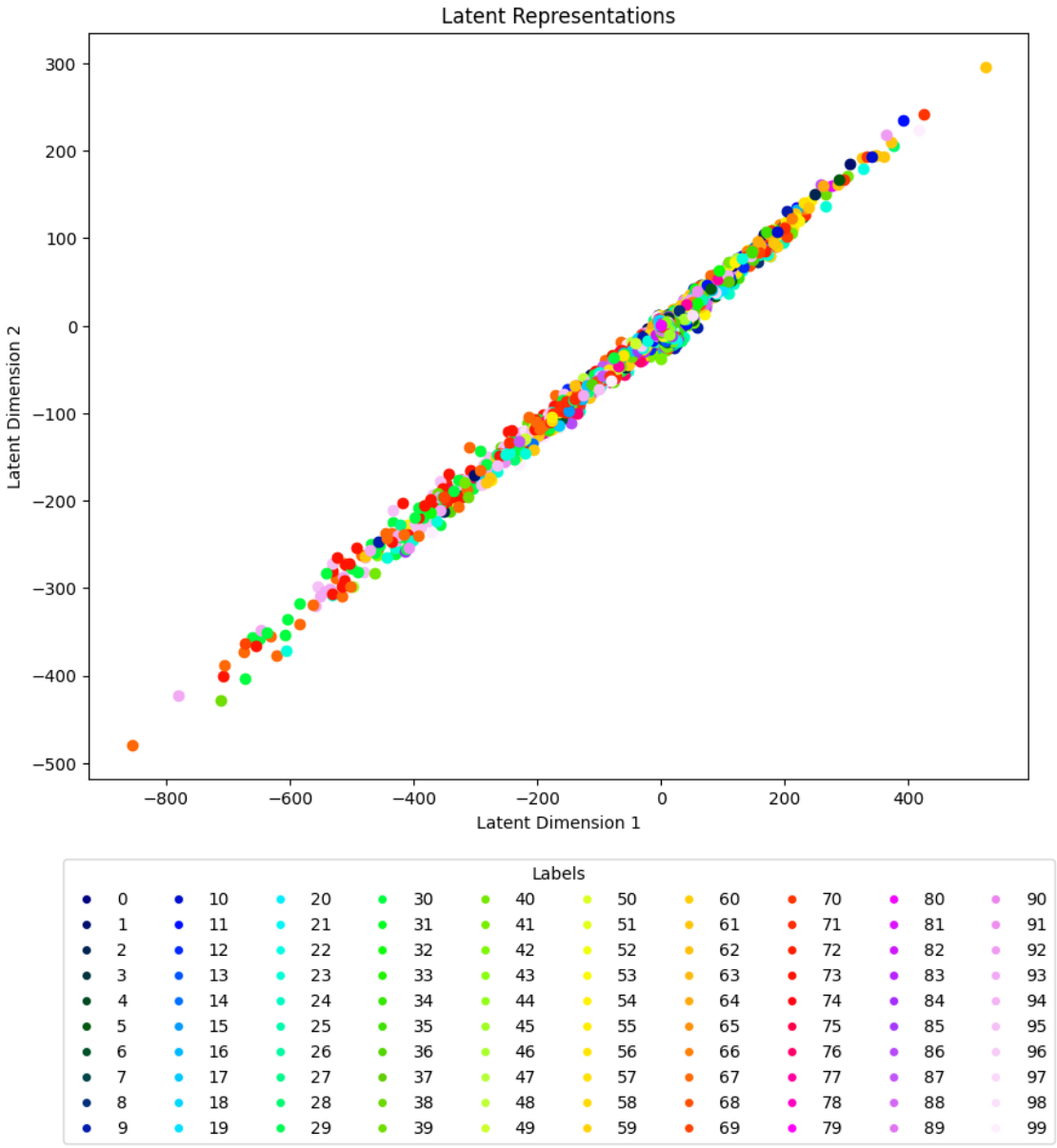
Let's now examine what the latent space of the auto encoders look like. We will start by plotting the images from the test set in the latent space for the model we trained with a bottleneck layer of size 2, as we can directly visualize this latent space:

[ ]:
```
def get_coarse_label(cifar_label):
    coarse_labels = {
        0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9,
        10: 10, 11: 10, 12: 11, 13: 12, 14: 13, 15: 14, 16: 15, 17: 16, 18: 17,␣
 ↪19: 18,
        20: 19, 21: 19, 22: 11, 23: 16, 24: 14, 25: 9, 26: 13, 27: 16, 28: 8,␣
 ↪29: 14,
        30: 19, 31: 7, 32: 17, 33: 14, 34: 9, 35: 11, 36: 19, 37: 12, 38: 14,␣
 ↪39: 8,
        40: 14, 41: 10, 42: 13, 43: 17, 44: 14, 45: 19, 46: 7, 47: 15, 48: 9,␣
 ↪49: 15,
        50: 8, 51: 13, 52: 19, 53: 19, 54: 11, 55: 17, 56: 11, 57: 0, 58: 8, 59:
 ↪ 11,
        60: 10, 61: 8, 62: 13, 63: 19, 64: 13, 65: 18, 66: 14, 67: 17, 68: 15,␣
 ↪69: 17,
        70: 13, 71: 15, 72: 12, 73: 15, 74: 14, 75: 15, 76: 16, 77: 13, 78: 14,␣
 ↪79: 13,
        80: 14, 81: 8, 82: 19, 83: 10, 84: 15, 85: 17, 86: 8, 87: 8, 88: 17, 89:
 ↪ 17,
        90: 13, 91: 19, 92: 13, 93: 12, 94: 15, 95: 19, 96: 11, 97: 19, 98: 11,␣
 ↪99: 17
    }
    return [coarse_labels.get(cifar_label[i], None) for i in␣
 ↪range(len(cifar_label))]
```

[ ]:
```
model = VAE.AutoEncoder.load_from_checkpoint("saved_models/
 ↪autoencoder_"+str(2)+".ckpt")
model.eval()
# Let's encode the entire test set
enc = np.empty((0, 2))
enc_labels = []
with torch.no_grad():
    for i, (images, targets) in enumerate(test_loader):
        latent_vectors = model.encoder(images).detach().numpy()
        enc = np.vstack((enc, latent_vectors))
```

```
        enc_labels.extend(targets.numpy())

cmap = plt.get_cmap('gist_ncar')
plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111)
scatter_list = []
for i in range(len(enc)):
    label = enc_labels[i]  # Labels corresponding to the latent vectors

    # Normalize the labels to the range [0, 1]
    normalized_label = label / 100

    # Get the colors from the colormap
    color = cmap(normalized_label)
    # plt.scatter(enc[i, 0], enc[i, 1], marker='o', color=color)
    scatter = plt.scatter(enc[i, 0], enc[i, 1], marker='o', color=color)
    scatter_list.append(scatter)

plt.xlabel('Latent Dimension 1')
plt.ylabel('Latent Dimension 2')
plt.title('Latent Representations')

labels = set(enc_labels)
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label=label,␣
 ↪markerfacecolor=cmap(label/100)) for label in labels]
ncol = 10  # Number of columns in the legend
legend = plt.legend(handles=legend_elements, loc='lower center',␣
 ↪bbox_to_anchor=(0.5, -0.5), ncol=ncol, title='Labels')

# Set the legend label text to be split over multiple lines
legend_texts = legend.get_texts()
for text in legend_texts:
    text.set_text('\n'.join(text.get_text().split()))

plt.show()
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_2.ckpt`
```



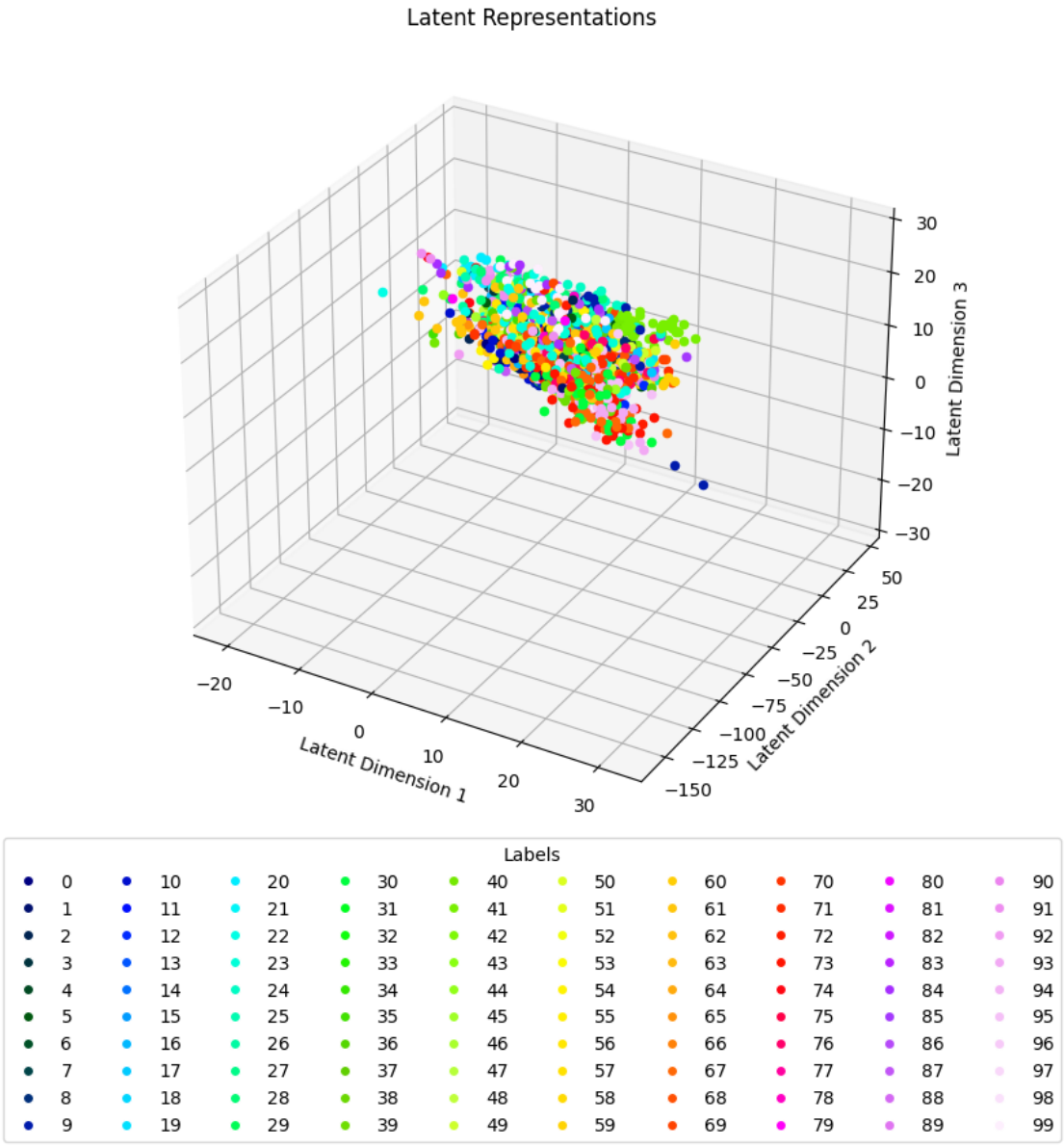
Interestingly all the encodings of the images fall roughly upon the y=x line, we also note that classes from different images appear all jumbled together (each color corresponds to a different class in CIFAR100) in the latent space. There's not a whole lot we can gauge from this visualization of the latent space and this may be because it's not a good encoding! As the decoder struggled to recreate anything like the input image (for the AE with latent space size 2) and that means this encoding could simply be too small and therefore not particularly useful.

Let's now look at the latent space of the AE with latent space size 3 as we can again plot this latent space:

[ ]:
```
from mpl_toolkits.mplot3d import Axes3D

model = VAE.AutoEncoder.load_from_checkpoint("saved_models/
 ↪autoencoder_"+str(3)+".ckpt")
model.eval()
# Let's encode the entire test set
enc = np.empty((0, 3))
enc_labels = []
with torch.no_grad():
    for i, (images, targets) in enumerate(test_loader):
        latent_vectors = model.encoder(images).detach().numpy()
        enc = np.vstack((enc, latent_vectors))
        enc_labels.extend(targets.numpy())

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter_list = []
for i in range(len(enc)):
    x = enc[i, 0]  # X-coordinate of the latent vector
    y = enc[i, 1]  # Y-coordinate of the latent vector
    z = enc[i, 2]  # Z-coordinate of the latent vector
    label = enc_labels[i]  # Label corresponding to the latent vector

    # Normalize the label to the range [0, 1]
    normalized_label = label / 100

    # Get the color from the colormap
    color = cmap(normalized_label)

    # Plot the point in 3D with color based on the label
    scatter = ax.scatter(x, y, z, color=color, marker='o')
    scatter_list.append(scatter)

ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.set_zlabel('Latent Dimension 3')
ax.set_title('Latent Representations')

labels = set(enc_labels)
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label=label,␣
 ↪markerfacecolor=cmap(label/100)) for label in labels]
ncol = 10  # Number of columns in the legend
legend = ax.legend(handles=legend_elements, loc='lower center',␣
 ↪bbox_to_anchor=(0.5, -0.4), ncol=ncol, title='Labels')

# Set the legend label text to be split over multiple lines
for handle in legend.legendHandles:
```

```
    handle.set_label('\n'.join(handle.get_label().split()))


plt.show()
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_3.ckpt`
/tmp/ipykernel_109221/1646310964.py:44: MatplotlibDeprecationWarning: The
legendHandles attribute was deprecated in Matplotlib 3.7 and will be removed two
minor releases later. Use legend_handles instead.
  for handle in legend.legendHandles:
```

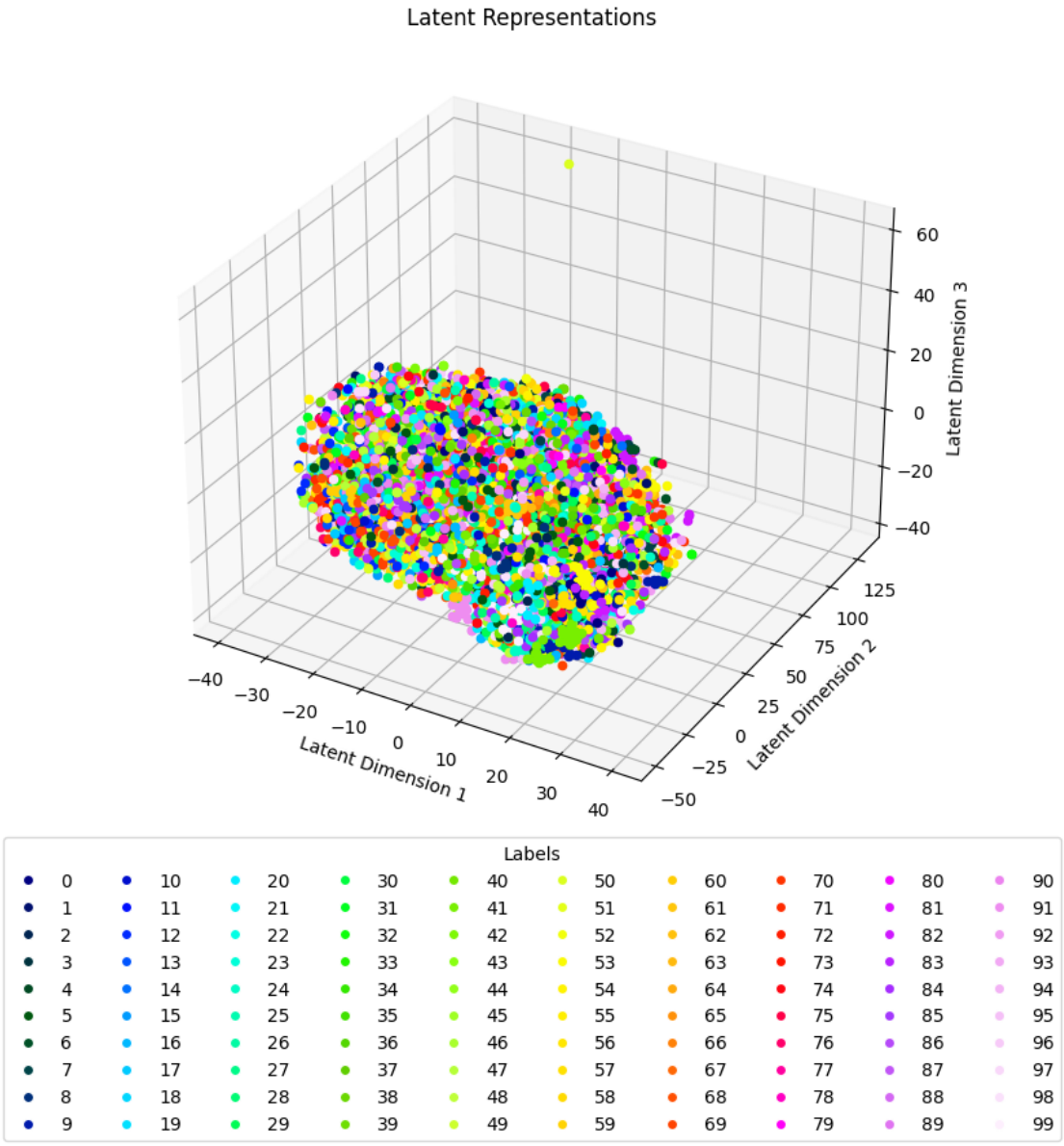


The latent space here seems to have slightly more structure as not all the points group around a line, however, it's still hard to discern anything about the structure of the latent space and points from different classes still appear mostly jumbled. Let's now skip to visualizing some of the latent spaces of the AE with larger dimensions for the bottleneck layer. We will have to use t-SNE in order to reduce the dimension of them, so we can plot them.

[ ]:
```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=3, random_state=42)

model = VAE.AutoEncoder.load_from_checkpoint("saved_models/
 ↪autoencoder_"+str(256)+".ckpt")
model.eval()
# Let's encode the entire test set
enc = np.empty((0, 256))
enc_labels = []
with torch.no_grad():
    for i, (images, targets) in enumerate(test_loader):
        latent_vectors = model.encoder(images).detach().numpy()
        enc = np.vstack((enc, latent_vectors))
        enc_labels.extend(targets.numpy())

# Transform the latent vectors into 3D using t-SNE
enc = tsne.fit_transform(enc)

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter_list = []
for i in range(len(enc)):
    x = enc[i, 0]  # X-coordinate of the latent vector
    y = enc[i, 1]  # Y-coordinate of the latent vector
    z = enc[i, 2]  # Z-coordinate of the latent vector
    label = enc_labels[i]  # Label corresponding to the latent vector

    # Normalize the label to the range [0, 1]
    normalized_label = label / 100

    # Get the color from the colormap
    color = cmap(normalized_label)

    # Plot the point in 3D with color based on the label
    scatter = ax.scatter(x, y, z, color=color, marker='o')
    scatter_list.append(scatter)

ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
```

```
ax.set_zlabel('Latent Dimension 3')
ax.set_title('Latent Representations')

labels = set(enc_labels)
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label=label,␣
 ↪markerfacecolor=cmap(label/100)) for label in labels]
ncol = 10  # Number of columns in the legend
legend = ax.legend(handles=legend_elements, loc='lower center',␣
 ↪bbox_to_anchor=(0.5, -0.4), ncol=ncol, title='Labels')

# Set the legend label text to be split over multiple lines
for handle in legend.legendHandles:
    handle.set_label('\n'.join(handle.get_label().split()))


plt.show()
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_256.ckpt`
/tmp/ipykernel_109221/1198158666.py:48: MatplotlibDeprecationWarning: The
legendHandles attribute was deprecated in Matplotlib 3.7 and will be removed two
minor releases later. Use legend_handles instead.
  for handle in legend.legendHandles:
```

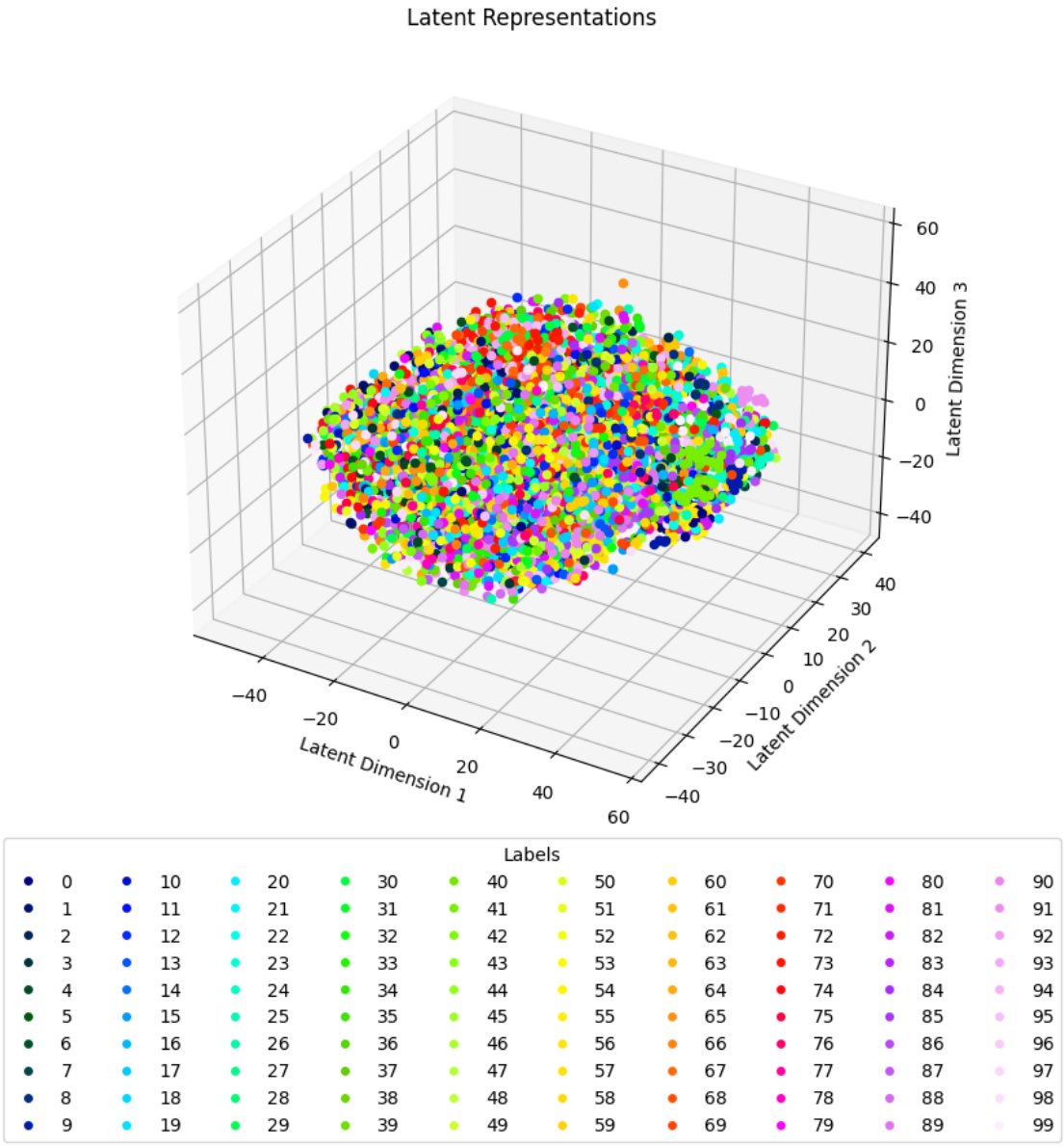


Again it's hard to discern anything about this latent space! But this doesn't mean the latent space is not useful, as the decoder can create a very good reconstruction from these encodings. Let's try with an even larger latent dimension:

[ ]:
```
tsne = TSNE(n_components=3, random_state=42)

model = VAE.AutoEncoder.load_from_checkpoint("saved_models/
  ↪autoencoder_"+str(512)+".ckpt")
model.eval()
# Let's encode the entire test set
enc = np.empty((0, 512))
```

```
enc_labels = []
with torch.no_grad():
    for i, (images, targets) in enumerate(test_loader):
        latent_vectors = model.encoder(images).detach().numpy()
        enc = np.vstack((enc, latent_vectors))
        enc_labels.extend(targets.numpy())

# Transform the latent vectors into 3D using t-SNE
enc = tsne.fit_transform(enc)

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter_list = []
for i in range(len(enc)):
    x = enc[i, 0]  # X-coordinate of the latent vector
    y = enc[i, 1]  # Y-coordinate of the latent vector
    z = enc[i, 2]  # Z-coordinate of the latent vector
    label = enc_labels[i]  # Label corresponding to the latent vector

    # Normalize the label to the range [0, 1]
    normalized_label = label / 100

    # Get the color from the colormap
    color = cmap(normalized_label)

    # Plot the point in 3D with color based on the label
    scatter = ax.scatter(x, y, z, color=color, marker='o')
    scatter_list.append(scatter)

ax.set_xlabel('Latent Dimension 1')
ax.set_ylabel('Latent Dimension 2')
ax.set_zlabel('Latent Dimension 3')
ax.set_title('Latent Representations')

labels = set(enc_labels)
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label=label,␣
 ↪markerfacecolor=cmap(label/100)) for label in labels]
ncol = 10  # Number of columns in the legend
legend = ax.legend(handles=legend_elements, loc='lower center',␣
 ↪bbox_to_anchor=(0.5, -0.4), ncol=ncol, title='Labels')

# Set the legend label text to be split over multiple lines
for handle in legend.legendHandles:
    handle.set_label('\n'.join(handle.get_label().split()))


plt.show()
```

```
Lightning automatically upgraded your loaded checkpoint from v1.9.4 to
v2.0.1.post0. To apply the upgrade to your files permanently, run `python -m
lightning.pytorch.utilities.upgrade_checkpoint --file
saved_models/autoencoder_512.ckpt`
/tmp/ipykernel_109221/1009648917.py:47: MatplotlibDeprecationWarning: The
legendHandles attribute was deprecated in Matplotlib 3.7 and will be removed two
minor releases later. Use legend_handles instead.
  for handle in legend.legendHandles:
```



Once again everything seems jumbled together and there is no structure that we can see. We also tried plotting the points colors according to their coarse-grained labels, however, everything

still appeared jumbled. There could be a few reasons for this: one is that we are carrying out dimensionality reduction on the latent representations another is it could be that the encoder isn't trying to create representations that allow for a good classifiation and therefore trying to visualize the space by colouring according to class may not be helpful, perhaps grouping according the amount of red, blue and green or other features could let us explore this latent space in a better way. Also, we are trying to visualize the latent space using images from the test set, images the encoder has never seen before and it could be that the encoding isn't very smooth, this is where it comes in handy using a Variational Auto Encoder (VAE) which is capable of producing more robust latent and smooth latent spaces.