ASTR 21100/312

"*Computational Techniques in Astrophysics*"

a final project option

Using neural networks to generate images of galaxies

Lead instructor(s) for this project: Andrey Kravtsov, Aster Taylor

Undergraduate students: 60 points + up to 15 possible extra credit points

Part I is due Fri, May 12 by 9pm; Part II is due Fri, May 19, 9pm

Background

Galaxies are distant stellar systems and are one of the main subjects of astronomical research. We are located in one of them - the Milky Way. Often methods used to analyze galaxies are tested on artificially produced images of these systems that are constructed from an underlying model of their light distribution, taking into account observational effects, such as noise, background light, etc.

In the last two decades, several approaches have been developed to generate artificial images of scenes, objects, animals, or humans either from completely random input (noise) or starting from low-quality images. Over the past decade, some of these approaches based on neural networks have produced rapid and impressive progress in the quality and realism of the images. You can generate examples of images of human faces produced by such an algorithm (these images are artificial and do not correspond to any face of actual living people). And such algorithms can impressively enhance very dimly lit scenes (you can see examples if you scroll down to the end of this page). Such algorithms are also used for "style transfer" where an image (a photo) is transformed into a similar image but with some specific style, as though it was painted by a specific painter.

One of the most effective approaches for such applications is based on neural networks and is called Generative Adversarial Networks (GANs), proposed by Ian Goodfellow and collaborators in a 2014 paper. Generative is because the method produces something meaningful like an image. Adversarial Networks refer to the two neural networks used in such approach: Generator network is trained to produce an image from an input image and Discriminator network evaluates how probable it is that image is from a data set similar to the training data, but is much larger. The discriminator output is sent back to the generative network.

The training procedure for Generator network is to maximize the probability of Discriminator of making a mistake, while Discriminator is trained to maximize the probability it makes correct classification of input image. The procedure is thus adversarial: the generator algorithm learns to produce better "forgeries", while discriminator is trained to become better at recognizing what is a forgery and what is real.

In this project you will use such approach to generate artificial images of galaxies. Namely, in the standard option you need to set up and train adversarial neural networks to generate random artificial images of galaxies from random input. An example of a GAN network that can be used for this is available in Chapter 12 of Francois Chollet's book, which will be made available in the #proj05_gan_gal channel on Slack.

Using this example, you can experiment with different choices for the network architecture and data augmentation strategies.

The training and test data sets for this project are extracted from the Legacy survey online database, where in some areas of the sky images obtained with the Sloan Digital Sky Survey

(SDSS) on a relatively small telescope and images from the Hyper-Supreme Camera (HSC) survey obtained on an 8-meter telescope exist. So one can get lower and higher quality images of the same galaxy, as shown in a few examples below.

Which computational and analysis methods are used in this project

- Neural networks and deep learning for generative image models.
- Design of neural networks
- Validation of machine learning results and measuring their accuracy.
- Plotting in 2d (and possibly in 3D) and image gallery.

Preparatory steps

- Read background information about convolutional neural networks. This can be found in the first four chapters of the free online book "Neural Networks and Deep Learning" by Michael Nielsen. Or in Chapters 2 and 8 of Francois Chollet's book on deep learning (see PDF files of the chapters in the #ml-info channel on Slack).
- Install sci-kit image image package, which is useful for image manipulation.
- Install Tensorflow machine learning package (installation instructions). You will be using Tensorflow and keras package embedded in it.
- Review the code implementing a GAN for image generation described in Chapter 12 of Francois Chollet's book (see #proj05_gan_gal for the PDF). The code itself can be found here.
- Download zip file with ≈ 6500 128x128 galaxy images from the Hyper-Supreme Camera survey that will be used to train GAN. Unpack the zip file in the directory, where you expect to work on your project notebook.

Project description (minimal steps required)

Stage I (25 points) your preliminary work on the steps in these stages is due Friday, May 12, 9pm)

- Read images in the directory to which you unzipped the file with images linked above. This can be done easily with keras, as described in Ch. 12 of Francois Chollet's book and shown in his code (in this example 128x128 images are downsampled to 64x64 images):

```
from tensorflow import keras
dataset = keras.utils.image_dataset_from_directory(data_path,
                                                   label_mode=None,
                                                   image_size=(64, 64),
                                                   batch_size=1,
                                                   smart_resize=True)
```

- RGB images consist of a 3D array with shape (npix, npix, 3) where each pixel holds integer representing intensity in a given color channel from 0 (nothing) to 255 (highest intensity). For training rescale pixels to the floating point numbers in the range [0,1].

- Copy the GAN code from the Francois Chollet's notebook, review it and set it up to run on these galaxy images. Try to do trial runs for a few epochs (3-5 epochs) and examine the images generated by the GAN at each epoch. (5 points)

- Vary the learning rate of the discriminator and generator networks in the range 0.0001 to 0.01 and examine effect it has on the dloss and gloss function values output during training and on the images that GAN generates. Note values of the learning rate for which you see the fastest improvement in generated images (5 points)

- Vary the batch size for images used in training and examine effect it has on improvements in generated images obtained after a given number of training epochs. (5 points)

- Write code that can generate and save images that are transformations of the original images in the training sample using some augmentation strategy - such as image rotation. Try to double the training sample with such modified images. Galaxy images chosen for rotation and the rotation angle should be random. (10 points)

Stage II (35 points, due Fri, May 19, 9pm)

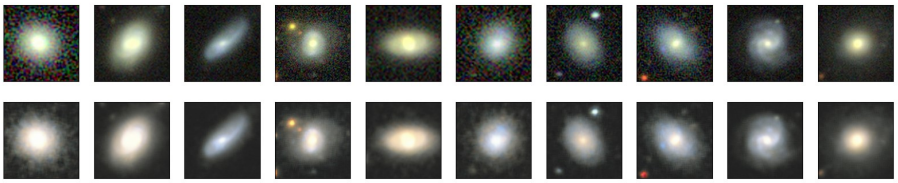
- Test effect of image sample augmentation by running GAN network on the augmented image samples and comparing the speed and quality of the generated images after a fixed number of epochs (5 points).

- For the GAN with the best parameters you could identify, train the model for as long as you can recording the progression of images it generates and present image collages at a representative set of epochs to demonstrate improvements in the generated images

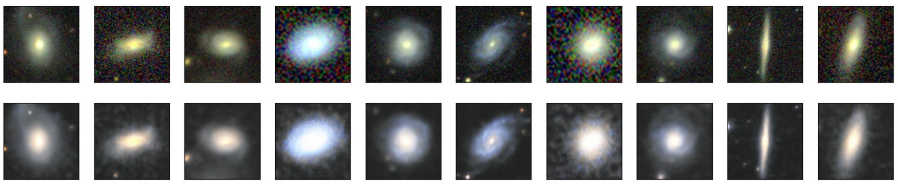
as training progresses. (30 points)

While building the convolutional auto encoder, we tried different numbers of Max Pooling in the encoder. Each Max Pooling effectively "compresses" the image and makes the latent space smaller. This helps with noise reduction but also makes us lose small scale features. We tried this at 50, 100, 300 and 600 epochs (500 in the case of 2 Max Pooling layers). In the case with 3 rows of images, the top row is the images from SDSS, the middle row is enhanced by the autoencoder, and the last row is the original high quality images. In the case with 2 rows of images, the first row is the images from SDSS, and the second row is the enhanced images.

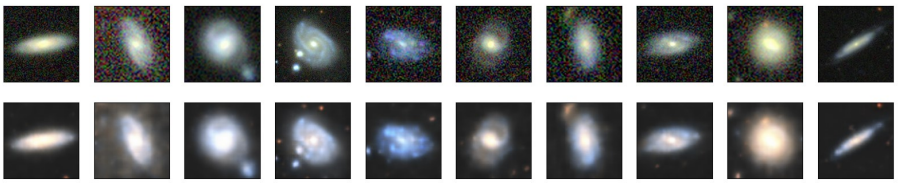
50 Epochs: No Max Pooling:



One Max Pooling Layer:

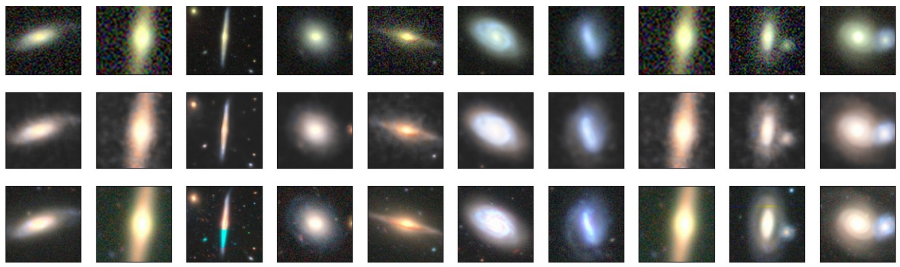


Two Max Pooling Layers:

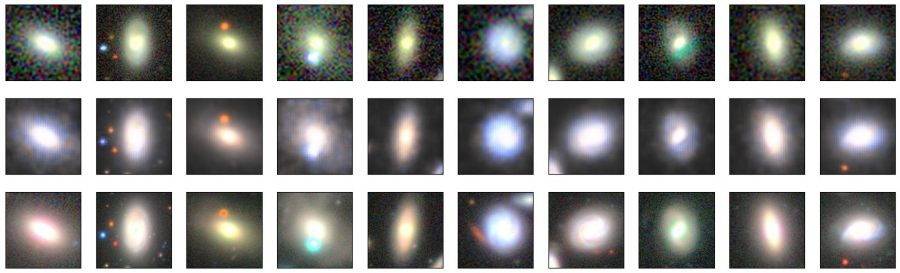


100 Epochs: No Max Pooling:

One Max Pooling:

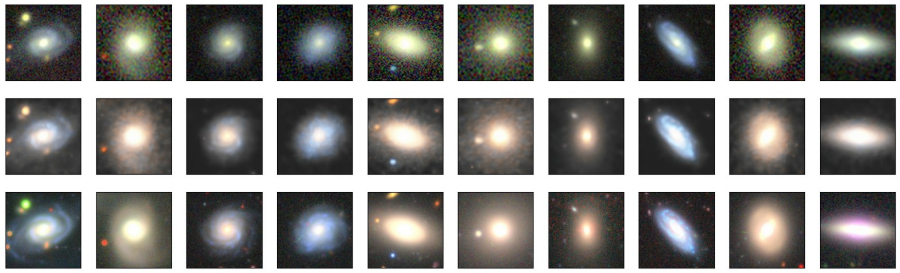


Two Max Pooling:



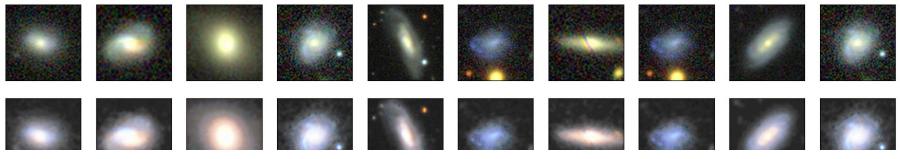
300 Epochs: No Max Pooling:

One Max Pooling:



Two Max Pooling: (currently being run)

500/600 Epochs: No Max Pooling:



One Max Pooling:

Two Max Pooling (500 epochs):

Having no Max Pooling layer seems to keep a lot of noise in the image, while having 2 Max Pooling layers makes the images very soft. It seems like 1 Max Pooling layer is the best option. There is also an UpSampling layer option in keras that might be worth testing. We also switched to using online learning, which yielded slightly improved results. The code below is the python script I used to generate images:

In [ ]:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras import utils
from tensorflow.keras.models import Model

data_path = r'/home/astr211/gal_hsc_128x128'
clean_data = utils.image_dataset_from_directory(data_path, shuffle=False, image_siz

data_path = r'/home/astr211/gal_sdss_128x128'
noisy_data = utils.image_dataset_from_directory(data_path, shuffle=False, image_siz

def generator(array):
    for el in array:
        yield el

def get_img_arrs(gen1, gen2, bound1 = 0.8, bound2 = 0.9):
    g1 = generator(gen1)
    g2 = generator(gen2)

    test_arr1, test_arr2 = [], []
    train_arr1, train_arr2 = [], []
    val_arr1, val_arr2 = [], []
    while True:
        try: # try to read an image
            image1 = next(g1)
            image1 = ((image1[0])[0].numpy()).astype('int32')
            r = np.random.uniform(0., 1.)
            if r <= bound1:
                train_arr1.append(image1/255)
            elif r <= bound2:
                val_arr1.append(image1/255)
            else:
                test_arr1.append(image1/255)
        except: # if file ended, above will generate error
            break # exit the loop

        image2 = next(g2)
        image2 = ((image2[0])[0].numpy()).astype('int32')
        if r <= bound1:
            train_arr2.append(image2/255)
        elif r <= bound2:
            val_arr2.append(image2/255)
        else:
            test_arr2.append(image2/255)
    return np.array(train_arr1), np.array(test_arr1), np.array(val_arr1), np.array(

clean_train, clean_test, clean_val, noisy_train, noisy_test, noisy_val = get_img_ar

def display3(array1, array2, array3):
    """
    Displays 10 random images from 3 arrays
    """
    n=10

    indices = np.random.randint(len(array1), size=n)
    images1 = array1[indices, :]
```

```
    images2 = array2[indices, :]
    images3 = array3[indices, :]
    plt.figure(figsize=(20,6))
    for i, (image1, image2, image3) in enumerate(zip(images1, images2, images3)):
        ax = plt.subplot(3, n, i+1)
        plt.imshow(image1)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(3, n, i + 1 + n)
        plt.imshow(image2)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(3, n, i + 1 + 2*n)
        plt.imshow(image3)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    plt.savefig("image.png")

def display(array1, array2):
    """
    Displays ten random images from each one of the supplied arrays.
    """

    n = 10

    indices = np.random.randint(len(array1), size=n)
    images1 = array1[indices, :]
    images2 = array2[indices, :]
    plt.figure(figsize=(20, 4))
    for i, (image1, image2) in enumerate(zip(images1, images2)):
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(image1)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(image2)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    plt.show()

# Display the original and noisy images
display(clean_train, noisy_train)

input = layers.Input(shape=(128, 128, 3))

# Encoder
x = layers.Conv2D(256, (3, 3), activation="relu", padding="same")(input)
x = layers.MaxPooling2D((2, 2), padding="same")(x)
x = layers.Conv2D(256, (3, 3), activation="relu", padding="same")(x)
x = layers.MaxPooling2D((2, 2), padding="same")(x)
```

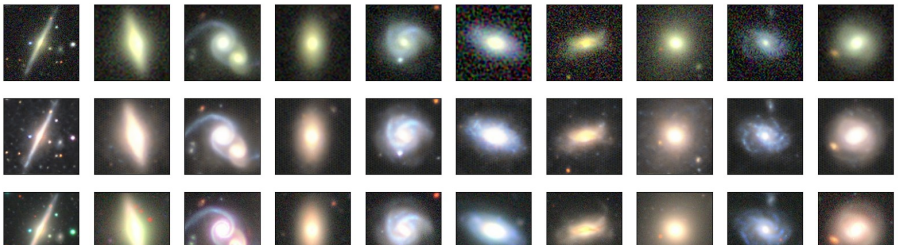
```
x = layers.MaxPooling2D((2, 2), padding="same")(x)
x = layers.Conv2D(256, (3, 3), activation="relu", padding="same")(x)
#x = layers.MaxPooling2D((2, 2), padding="same")(x)

# Decoder
#x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same
x = layers.Conv2DTranspose(256, (3, 3), strides=2, activation="relu", padding="same
x = layers.Conv2DTranspose(256, (3, 3), strides=2, activation="relu", padding="same
x = layers.Conv2D(3, (3, 3), activation="sigmoid", padding="same")(x)

# Autoencoder
autoencoder = Model(input, x)
autoencoder.compile(tf.keras.optimizers.Adam(lr = 0.0000001, beta_1 = 0, beta_2 = 0
autoencoder.summary()

autoencoder.fit(
    x=noisy_train,
    y=clean_train,
    epochs=300,
    batch_size=1,
    shuffle=True,
    validation_data=(noisy_val, clean_val),
)

autoencoder.save('my_model_two_pool_300')

predictions = autoencoder.predict(noisy_test)
display3(noisy_test, predictions, clean_test)
```

Following experiments with above code, I found an improved autoencoder example online which provided much better results than the code used above. This example uses kernel regularizers and a custom loss function. The decoder is significantly more advanced too.

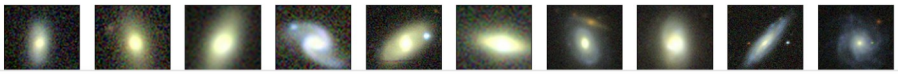
This method gets around the problem of a small latent space by adding encoded and decoded layers at different points. These are known as skip connections, and seem to be the main way that autoencoders are able to combat lossy image compression. In addition, the custom loss function is capable of both capturing large-scale (larger than a pixel), a concept that is defined as perceptual loss, and small-scale pixel differences. I've put a few links regarding this in my conclusion below the code block, including where I got my code from.

I tested this code for 10, 200 and 400 epochs, and am currently testing it for 600 epochs. Each epoch for this code takes much longer (around 4 minutes compared to sub 1 minute for the previous code); however, the results produced are very good. Below the code block, I have summarized my findings.

10 Epochs:



200 Epochs:



In [ ]:
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras import utils
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
from tensorflow.keras.applications.vgg16 import VGG16
mod = VGG16(include_top=False, weights='imagenet')
mod.trainable = False
from tensorflow.keras import backend as K

data_path = r'/home/astr211/gal_hsc_128x128'
clean_data = utils.image_dataset_from_directory(data_path, shuffle=False, image_siz

data_path = r'/home/astr211/gal_sdss_128x128'
noisy_data = utils.image_dataset_from_directory(data_path, shuffle=False, image_siz

def generator(array):
    for el in array:
        yield el

def get_img_arrs(gen1, gen2, bound1 = 0.8, bound2 = 0.9):
    g1 = generator(gen1)
    g2 = generator(gen2)

    test_arr1, test_arr2 = [], []
    train_arr1, train_arr2 = [], []
    val_arr1, val_arr2 = [], []
    while True:
        try: # try to read an image
            image1 = next(g1)
            image1 = ((image1[0])[0].numpy()).astype('int32')
            r = np.random.uniform(0., 1.)
            if r <= bound1:
                train_arr1.append(image1/255)
            elif r <= bound2:
                val_arr1.append(image1/255)
            else:
                test_arr1.append(image1/255)
        except: # if file ended, above will generate error
            break # exit the loop

        image2 = next(g2)
        image2 = ((image2[0])[0].numpy()).astype('int32')
        if r <= bound1:
            train_arr2.append(image2/255)
        elif r <= bound2:
            val_arr2.append(image2/255)
        else:
            test_arr2.append(image2/255)
```

```
    return np.array(train_arr1), np.array(test_arr1), np.array(val_arr1), np.array(

clean_train, clean_test, clean_val, noisy_train, noisy_test, noisy_val = get_img_ar

def display3(array1, array2, array3):
    """
    Displays 10 random images from 3 arrays
    """
    n=10

    indices = np.random.randint(len(array1), size=n)
    images1 = array1[indices, :]
    images2 = array2[indices, :]
    images3 = array3[indices, :]
    plt.figure(figsize=(20,6))
    for i, (image1, image2, image3) in enumerate(zip(images1, images2, images3)):
        ax = plt.subplot(3, n, i+1)
        plt.imshow(image1)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(3, n, i + 1 + n)
        plt.imshow(image2)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(3, n, i + 1 + 2*n)
        plt.imshow(image3)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    plt.savefig("Best_Model_200.png")

def display(array1, array2):
    """
    Displays ten random images from each one of the supplied arrays.
    """

    n = 10

    indices = np.random.randint(len(array1), size=n)
    images1 = array1[indices, :]
    images2 = array2[indices, :]
    plt.figure(figsize=(20, 4))
    for i, (image1, image2) in enumerate(zip(images1, images2)):
        ax = plt.subplot(2, n, i + 1)
        plt.imshow(image1)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

        ax = plt.subplot(2, n, i + 1 + n)
        plt.imshow(image2)
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```

```
    plt.show()

# Display the original and noisy images
display(clean_train, noisy_train)

input = layers.Input(shape=(128, 128, 3))

# Encoder
x1 = layers.Conv2D(256, (3, 3), activation="relu", padding="same", kernel_regulariz
x2 = layers.Conv2D(256, (3, 3), activation ="relu", padding="same", kernel_regulari
x3 = layers.MaxPooling2D((2, 2), padding="same")(x2)

x4 = layers.Conv2D(256, (3, 3), activation="relu", padding="same", kernel_regulariz
x5 = layers.Conv2D(256, (3, 3), activation="relu", padding="same", kernel_regulariz
x6 = layers.MaxPooling2D((2, 2), padding="same")(x5)

encoded = layers.Conv2D(256, (3, 3), activation = "relu", padding = "same", kernel_

# Decoder
x7 = layers.Conv2DTranspose(256, (3, 3), activation="relu", padding="same", strides
x8 = layers.Conv2D(256, (3, 3), activation="relu", padding="same", kernel_regulariz
x9 = layers.Conv2D(256, (3, 3), activation="relu", padding="same", kernel_regulariz
x10 = layers.Add()([x5, x9])

x11 = layers.Conv2DTranspose(256, (3, 3), padding = "same", strides = 2, activation
x12 = layers.Conv2D(256, (3, 3), activation = "relu", padding = "same", kernel_regu
x13 = layers.Conv2D(256, (3, 3), activation = "relu", padding = "same", kernel_regu
x14 = layers.Add()([x2, x13])

decoded = layers.Conv2D(3, (3, 3), activation="sigmoid", padding="same", kernel_reg

def VGGloss(y_true, y_pred):
    pred = K.concatenate([y_pred])
    true = K.concatenate([y_true])
    vggmodel = mod
    f_p = vggmodel(pred)
    f_t = vggmodel(true)
    return K.mean(K.square(f_p - f_t))

def MSE_loss(y_true, y_pred):
    return K.mean(K.square(y_pred - y_true), axis=-1)

def total_loss(y_true, y_pred):
    return VGGloss(y_true, y_pred) + MSE_loss(y_true, y_pred)

# Autoencoder
autoencoder = Model(input, decoded)
autoencoder.compile("Adam", loss=total_loss)
autoencoder.summary()

autoencoder.fit(
    x=noisy_train,
    y=clean_train,
    epochs=200,
    batch_size=1,
    shuffle=True,
    validation_data=(noisy_val, clean_val),
)
```

```
autoencoder.save('Best_Model_200')

predictions = autoencoder.predict(noisy_test)
display3(noisy_test, predictions, clean_test)
```

interpretation of data, and will be inefficient at removing noise. On the other hand, the lossy compression required to shrink latent vector space size causes the image to become blurry and suffer from larger scale loss, due to the fact that we are effectively trying to express a very complicated function with far fewer parameters, resulting in inevitable loss of data. For traditional autoencoders, this means accepting some compromise between noise reduction and loss in image quality, and adjustment of depth of network to achieve results that are acceptable. For this reason, autoencoders tend to be quite shallow networks, with not many layers.

However, skip connections allow for an improvement in image quality while also reducing noise, by allowing us to send feature maps directly from an earlier stage of the encoder to a later of the decoder. Earlier convolutions, which retain sharpness, are then mapped to later stages of the decoder, allowing us to achive the "best of both worlds" and retain sharpness aand reduce noise. Skip connections are something that are not perfectly understood, but empirically have produced significantly better results, and are something that are being experimented with actively. Most of the best autoencoders are reliant on skip/ residual connections, such as the ResNet Autoencoder, and its convolutional version C-RAE. While we did not make our network any deeper when we introduced skip connections, this is something that could be experimented with, as our images are now significantly sharper. However, we should also consider using a larger dataset if we were to do this. Our current network already has over 5 million parameters and takes over 3 minutes per epoch, and a deeper network would only increase this issue. I am therefore doubtful that a deeper network on a dataset of this size would show significant improvement.

Our earlier autoencoders that used binary crossentropy as a loss function suffered from a plateau in the loss function after very few epochs. This is also something that we clearly see in our images, which do not improve with an increase in training time. Skip connections, and a change in loss function also aid in solving this issue. Skip connections have been shown to help prevent exploding gradients as well as vanishing gradients, while a custom loss function based on the VGG16 autoencoder is capable of capturing both small-scale and large-scale loss. As a result, we noticed a plateau far later than we did in our earlier autoencoders. The more advanced autoencoder showed a decreasing loss for both validation and data and training data till around the 100th epoch- a massive improvement over the 5-10 epochs it took for previous models to plateau. We noticed this plateauing of loss function for earlier versions of our autoencoders for even with drastically lowered learning rates, and for increased learning rates too, indicating that the gradient was actually near 0.

We only experimented very superficially with regularizers. Imposing a kernel regularizer with the in built parameters on each layer of the earlier version of the autoencoder led to very sub par results, leading us to not try experimenting too much with kernel regularizers. However, our final version does use kernel regularizers with modified parameters. Further

However, our final version does use kernel regularizers with modified parameters. Further testing should see what happens when regularization is removed, and what happens when regularization paramters are changed. Perhaps a regularization with better parameters on our earlier autoencoders would have also prevented a vanishing gradient.

Another experiment that was carried out was utilizing ConvTranspose with stride = 2 vs UpSampling2D. UpSampling2D is not a machine learning layer, instead just doing the opposite of MaxPooling- it takes a pixel and makes 4 pixels of the same value. Using this in combination with a convolution layer seems to do a similar job to using Conv2DTranspose, and most runs just used Conv2DTranspose. However, using UpSampling2D without a convolutional layer may be beneficial in reducing hyperparameters and decrease training time, at the expense of degraded image quality.

Links have been provided below for further reading about some of the topics that I ended up exploring: Code for the version of autoencoder that produced best results. This paper uses elu as opposed to relu, which is something I did not experiment with. It also uses a learning rate that is a function of epoch, as opposed to constant I did not experiment with this either: https://ishih.medium.com/image-super-resolution-and-denoising-using-autoencoder-2839eb4f02

Paper that proposes skip/ residual connections. This paper came out of the development of the Res-VGAE autoencoder: https://arxiv.org/pdf/2105.00695v1.pdf

towardsDataScience article on medium that does avery good job of explaining skip connectiosn in a layman's language, as opposed to a highly mathematical look at it. Unfortunately the paper is behind a paywall, but creating a free account on medium gets 3 free articles: https://medium.com/towards-data-science/using-skip-connections-to-enhance-denoising-autoencoder-algorithms-849e049c0ac9

Paper that proposes using perceptual loss instead of the previous standard loss function for autoencoders (ELBO): https://arxiv.org/abs/1603.08155