GENERATIVE DEEP LEARNING WITH TENSORFLOW

Week 1

Style transfer

Wassily Kandinsky (wikipedia)
https://upload.wikimedia.org/wikipedia/commons/b/b4/Vassily_Kandinsky%2C_1913_-_Composition_7.jpg

https://cdn.pixabay.com/photo/2017/02/28/23/00/swan-2107052_1280.jpg

Style transfer is an image processing technique by which the style and textures from one image can be transferred onto another image. For example, if we take the image of the swan in the middle of the screen, we can combine it with the image of an impressionist painting by Wassily Kandinsky on the left. The results will be a stylized version of the swan as you can see on the right-hand side.

Approaches to Style Transfer

1. Supervised Learning.
2. Neural Style Transfer
3. Fast Neural Style Transfer

1. Supervised Learning

style transfer using deep learning. These are a
you build a network that's given pairs of images and is
natch the contents of the other. The second and
ansfer, and we'll dig into that this week. That's what
d how one image should map to another. Instead, we
se features as styles.

is we don't need lots and lots of training samples. We can do it with just a single pair of images.
Finally, there's an enhancement of style transfer that speeds things up, and it's called fast neural style transfer. We'll also take a look at how that works.

- Pairs: original & stylized image.
  - Need lots of pairs!

The supervised learning approach means that you need to get pairs of images with both the original and the desired style. In other words, you might need to brush up on your drawing skills to manually create those stylized images.

Imagine drawing various birds in the style of a
match one style to the other.

With supervised learning, you'll need many im
feasible.

2. Neural Style Transfer

- pre-trained model
- Inputs: a single pair of images
  - Extract style from image 1
  - Extract content from image 2
- Generate image to match style and content.
  - In a loop: minimize loss

Neural style transfer works differently and that you generally start with a pre-trained model. What's pretty convenient is that your model can train itself on a single pair of images. You use the model to extract the style from the first image and also to extract the content from the second image.

Then you create an image with the elements of both in a way that matches the style of image one and the content of image two.

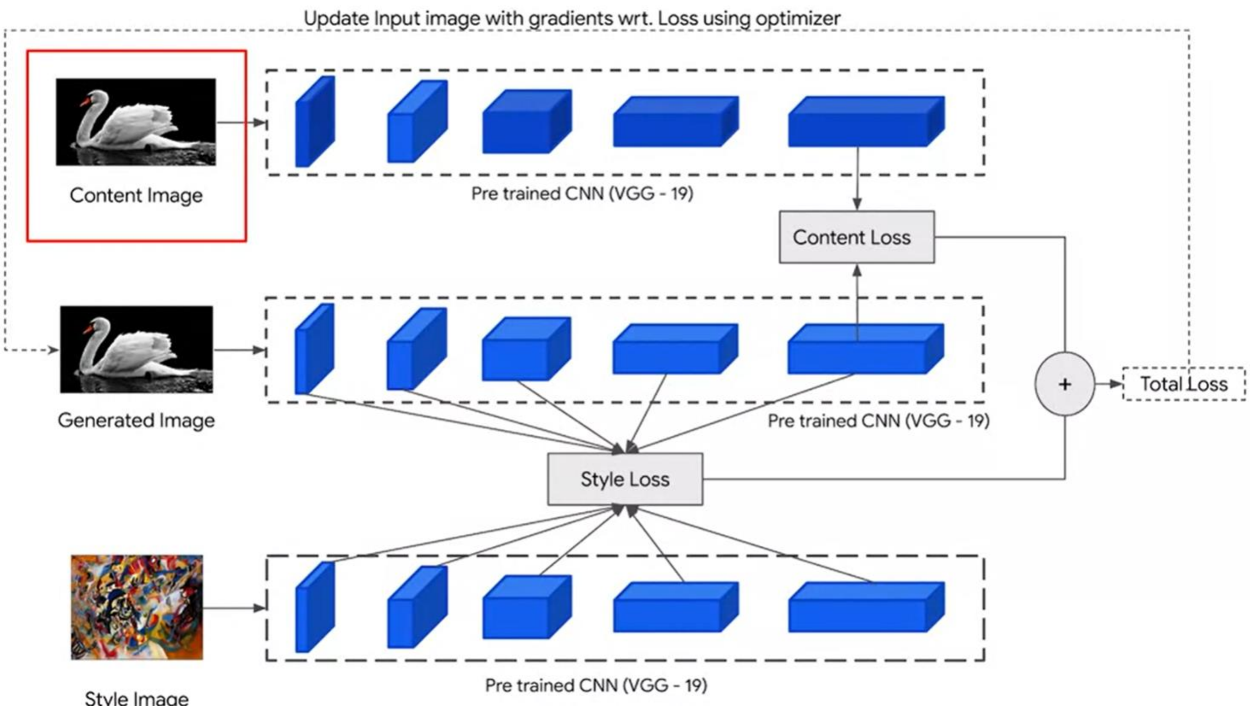
You'll do this iteratively and a loop by minimizing the loss of the generated image with respect to the style of image one, and the content of image two.

As such, you follow a loop a little bit like when you train a neural network, but you're only using two images and using loss minimization to merge the style and the content into the

Reference: A Neural Algorithm of Artistic Style

- A Neural Algorithm of Artistic Style (Gatys, Ecker & Bethge, 2015)

Style transfer conceptual overview

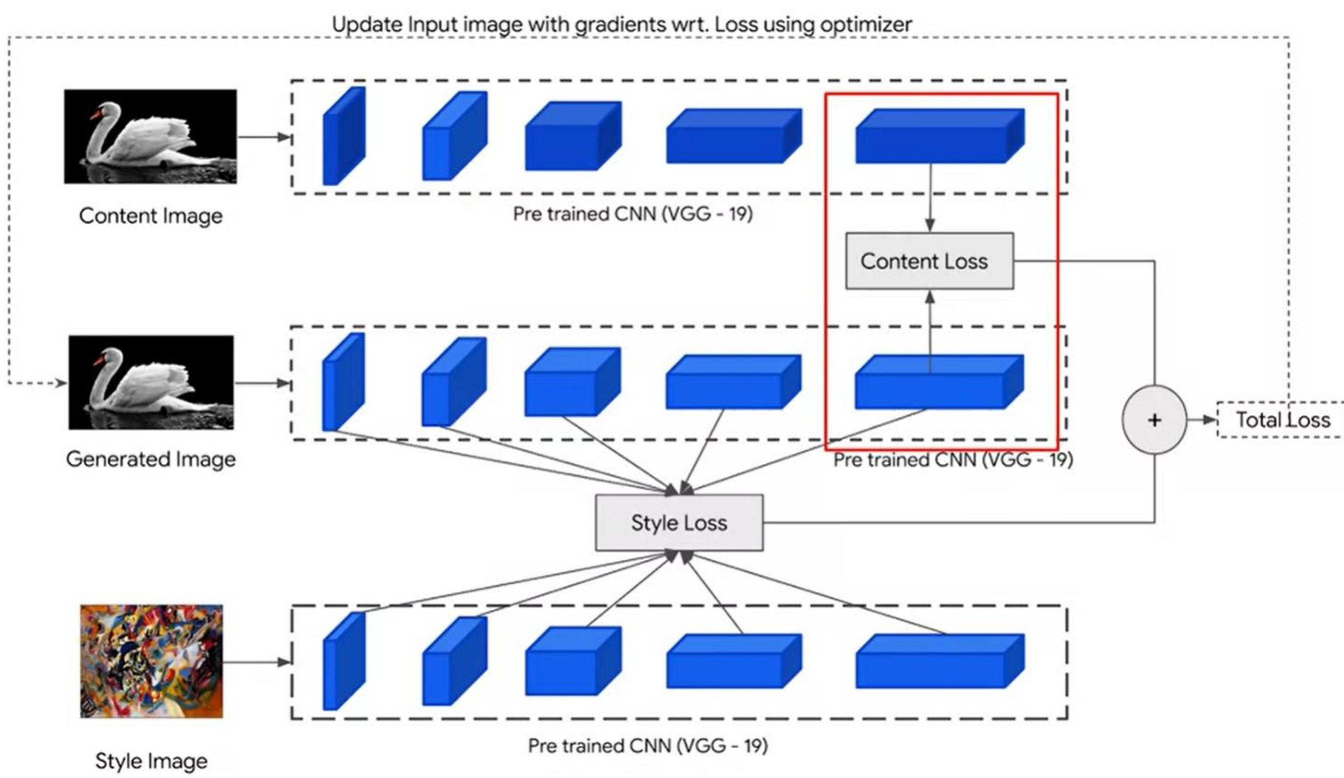


Now let's explore how the style transfer algorithm works, we'll start with the image that we want to transform and we'll call this the content image.

Here we can see it's a swan. Then there's the image that gives us the style that we want to convert our image into. We'll call that the style image. From these, we'll create a new image that we'll call the generated image. It would be created over many iterations initially looking very much like the content image but over time it will take on the attributes of the style image.

Using a pre-trained CNN, for example, VGG-19, we can extract the content features from the content image. We'll represent these features as the last block in the network here, and the style is also extracted by using a pre-trained CNN.

Style information can be extracted from every layer in the network, and this is represented here by the arrows coming out of each block in the network diagram.

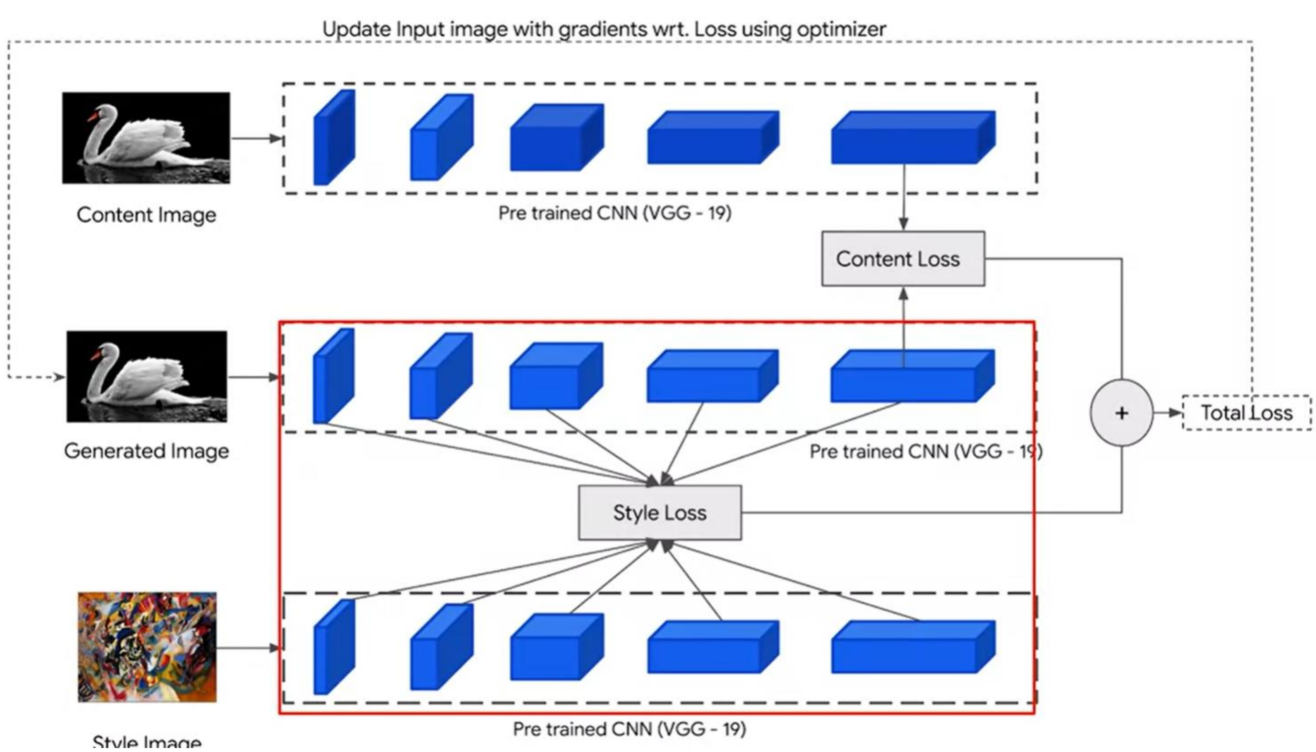


We'll then initialize our generated image from the content image. The model can compare the generated image with the content image by calculating what's called a content loss.

On every iteration, it checks how much of the original content is present in the generated image. As you can imagine on the first iteration, the content loss will be close or equal to zero depending on the initialization of the pre-trained CNN.

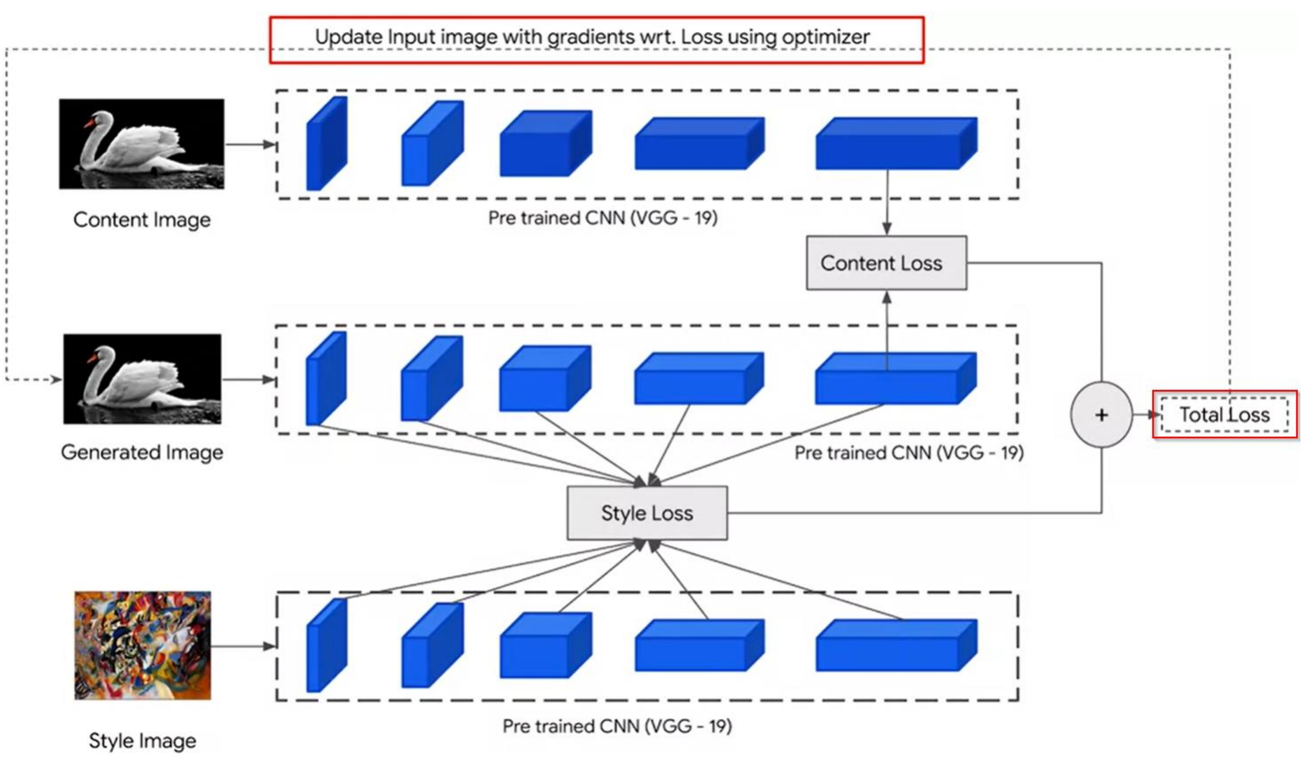
Over several iterations, as you will see, the content loss may increase as
the generated image is modified to incorporate the style information.
Now, this might seem counterintuitive, but increasing the content loss is
actually a good thing because that would indicate that there are differences between the generated image and the original content image. The loss would be zero if they were identical.

But of course, you don't want the loss to get too high or you'd lose the features and attributes of the original image. Similarly, you want to reduce the style loss because we'd like some of its attributes to be in the generated image. It's ultimately a balancing game.



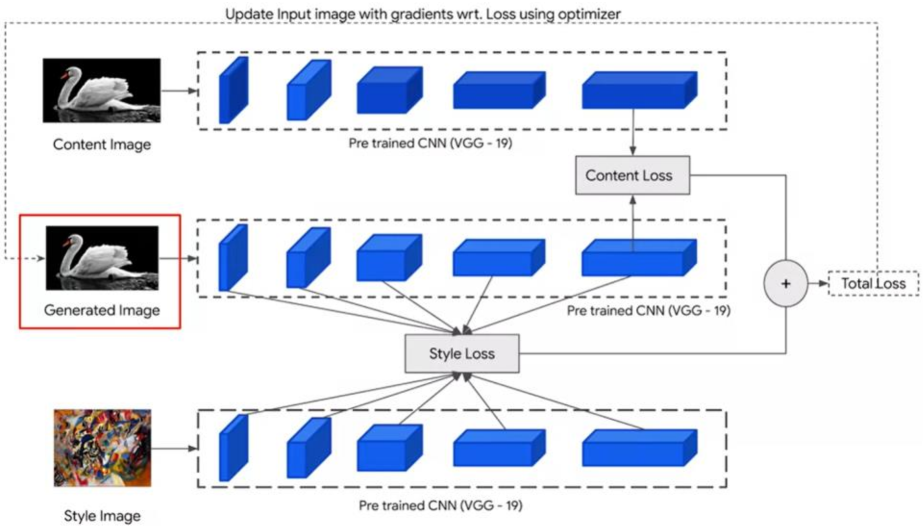
The algorithm also compares the generated image with the style image using a metric called the style loss and you can see that here. The style loss is calculated by comparing each layer of the CNNs for both the generated image and the style image.

The style loss ends up being an average loss across the multiple layers that are being compared. In early iterations, as the images are very different, the style loss will be very high, but the goal is to reduce it over time as the generated image incorporates
more of the style information.



You can imagine that you'd want a generated image to keep the overall loss low. You can combine the style loss with the content loss to get a total loss. We'll see in a bit how the two losses are added together to get that total loss.

Given this information, we can use an optimizer to update the generated image to get a new generated image whose total loss is lower. If you recall how you can use gradients in an optimizer to update each layer of a neural network, you can apply a very similar concept here to updating the input image.



Which should make the generated image into something closer to a merged image between the two input images. We'll repeat the process comparing the generated image against the style image to get a style loss and also comparing the generated image against the content image to get a content loss.

Adding the two losses into a total loss, and then optimizing to reduce the overall total loss.
Now notice that we're not training this network. It's being used to extract the content and style information from the images. But the process is quite similar to network training in that you iteratively calculate and minimize a loss through a loss function and then apply what you've learned using an optimizer to get new data.

Note that while I'm showing three networks here, there's really only one network. I'm just splitting into three like this to make it easier to illustrate.

2. Neural Style Transfer

Advantages

- Requires only single pair of content image and style image.

Drawbacks

- Requires many passes for stylizing each image.
- Takes time in stylizing every time an input is given.

Reference: Perceptual Losses for Real-Time Style Transfer and Super-Resolution

---

- Perceptual Losses for Real-Time Style Transfer and Super-Resolution (Johnson, Alahi & Li, 2016)

Pre-processing

Steps to Develop Neural Style Transfer

1. Preprocess, content & style images
2. Load pre trained model, define loss functions
3. Loop:
    a. Optimize and generate new image
    b. Visualize Outputs

The algorithm goes through these steps. First, we'll load and pre-process the content and style images into tensors so that they can be used within the CNN.

Then we'll set up the model infrastructure by loading a pre-trained model and defining the loss functions for both content and style loss.

Next, we'll define the style transfer loop that applies an optimizer to the total loss and generates a new image. As each new image is generated, we can visualize the outputs just to see how the style transfer is going along.

Load and Preprocess Images

- VGG-19 pre trained on 'imagenet'
- Accepts pixels values [0...255]
    - Don't normalize to [0 ... 1]
- Expects centered pixel values

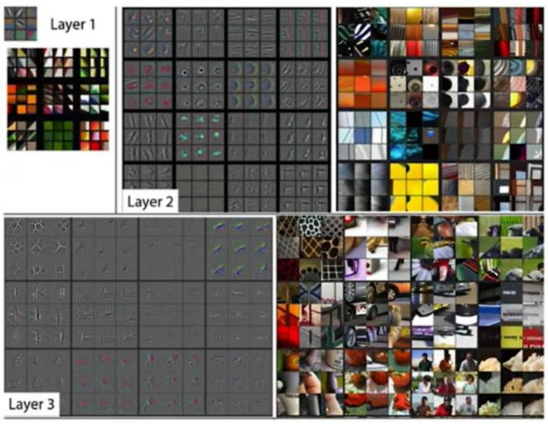
You'll want to decide which pre-trained network to use so that the images are formatted to be fed into the selected model. In this example, I'm going to use VGG 19, which is a little unique in how it expects images to be formatted.

You don't need to normalize the pixel values to arrange them between zero and one, but you do need to center them so that the average of the pixel values is zero.

This involves shifting the distribution of pixel values so that the distribution is centered around zero.

```
def preprocess_image(image):
    ...
    image = tf.keras.applications.vgg19.preprocess_input(image)

    return image
```

How CNN visualizes images?



https://arxiv.org/pdf/1311.2901.pdf

›n, the code for centering these pixels is provided for you. You'll see
:h calls the Keras function pre-process input. This function centers
the image will be ready to be fed into the VGG 19 network.

To understand style transfer, one has to understand a little bit about how CNN see images. Typically, a CNN will learn general low-level features like lines, edges, and basic shapes in the layers that are closer to the input. You can see examples here in the visualizations labeled layer one, layer two, and layer three.

The CNN builds on these low-level features to learn complex features like faces, noses, wheels, et cetera, and the layers that are deeper into the neural network.

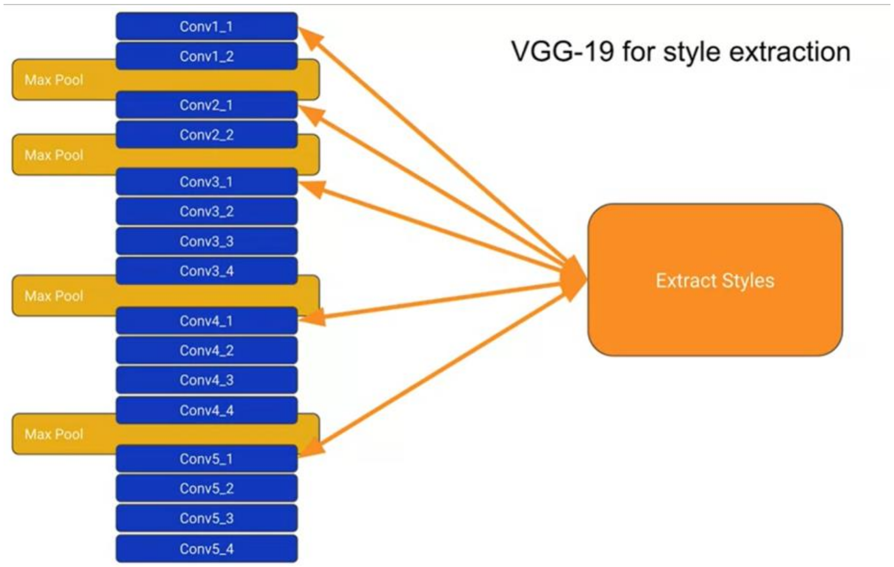
You can see some sample visualizations on the right here. Here's the question I'd like you to think about, between the lower level features and complex features, which do you think represents the style of the painting versus the content of the painting?

Well, imagine you are a painter and you can choose between using a paint brush or a pencil, and you're asked to draw a picture of a bird. Your choice of paint brush or pencil, will affect how you draw the lines and textures that define the style of your artwork.

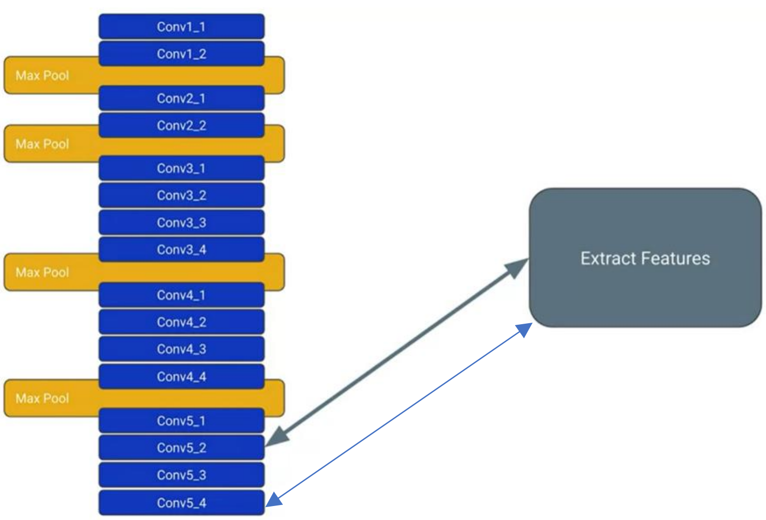
But whichever drawing device you choose, you'll still draw the high-level features that define a bird, such as wings, feathers, and a beak.

In other words, the lower level features define the style of your image and the higher level features define the content of your image.

Looking at the layers of the CNN, you can get the style from the lower numbered layers closer to the input, and it can retrieve the content of the image from the higher number layers deeper in the network.



Recall that you're going to use a copy of the VGG-19 network to extract the style features from the style image. The VGG-19 architecture consists of 19 convolutional layers, we show 16 of them here in blue. These are followed by three fully connected layers that aren't shown here for a total of 19 layers, hence the name VGG-19. The convolutional layers are grouped into five blocks. Note how they are numbered. The first layer in the first block is conv1_1, the second layer in the first block is conv1_2, the first layer in the second block is conv2_1 and so on. You can also see in this diagram that each block of convolutions is connected by a max pooling layer shown in yellow. Recall that the style can potentially be extracted from any of the convolutional layers of the network. From experimentation, it was discovered that the early layers in each block can be used effectively by taking the first convolutional layer for each block, which are conv1_1, conv2_1, 3_1, 4_1 and 5_1, you can use the pre-trained filters to extract the styles from style image.



Reference: Visualizing and Understanding Convolutional Networks - [Visualizing and Understanding Convolutional Networks](#) (Zeiler & Fergus, 2013)

Recall that you can use another copy of VGG-19 to extract the content from the content image based on the earlier discussion of which layers extract low-level features versus complex features. Well, which layers do you think you could then use to extract the content of the image? One good candidate is the last layer of the last convolutional block, which is conv5_4. It is good to experiment with a couple of layers yourself though, as oftentimes, the very last layer isn't the best one to use. This is because by the very last layer, the filters have reduced the image to very small images containing extracted features.

From my experience, choosing a layer that's positioned a little before that last layer will work well. For this example, let's take a look at using the second layer in the final block. There's no hard and fast rule here, experiment as you see fit using the Notebook and see what works for your particular task of dataset.

```python
# Content layer where we will pull our feature maps
content_layers = ['block5_conv2']

# Style layer of interest
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

layer_names = content_layers + style_layers
```

In code, we can just initialize a couple of arrays with the names of the layers that we want to use. For the VGG-19 model that we're using, the layer names are formatted with block number and conv number. For example, block5 conv2 is the second convolutional layer in the fifth block. Block5 conv2 is the layer that you'll choose to extract the content features. You'll store this in a list so that you can easily combine two lists into one. You'll also choose the first layer of each of the five blocks and store them in a list of style layers. You can then append the content layers and style layers into this list that we call layer names.

```python
def vgg_model(layer_names):
  """ Creates a vgg model that returns a list of intermediate output values."""
  vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
  vgg.trainable = False

  outputs = [vgg.get_layer(name).output for name in layer_names]

  model = tf.keras.Model(inputs=vgg.input, outputs=outputs)

  return model
```

Then you can define a function called VGG model that loads the VGG-19 model and creates a custom model. From that, it'll take a parame … performs earlier. T … classificat … et to initialize … Flow Keras. Yo … l way here.

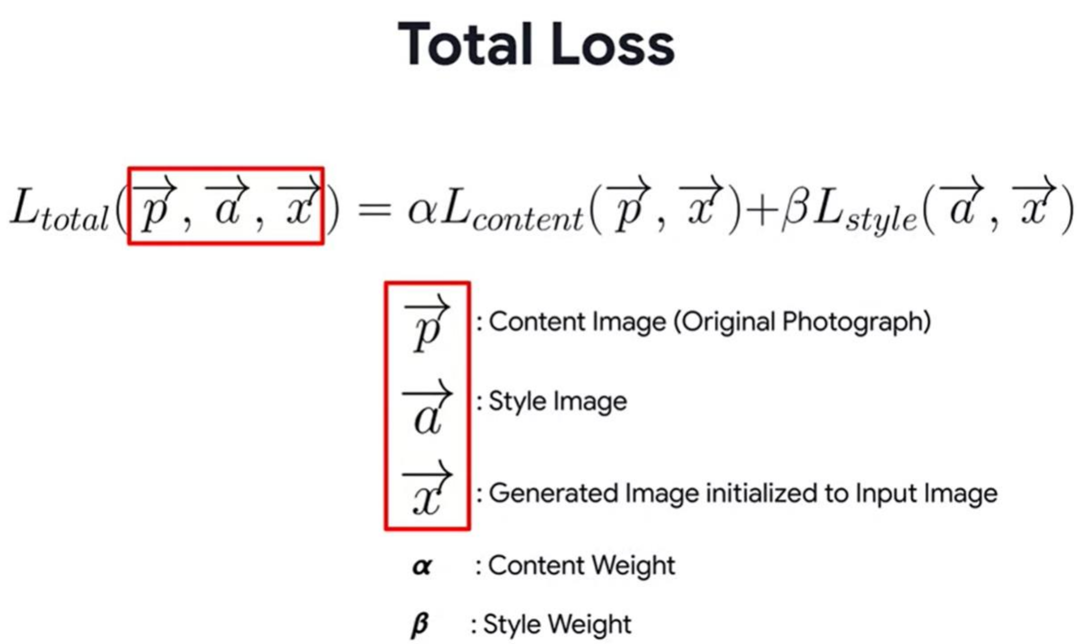
VGG-19 v … the mode … As a resu … with the model ty … get the outputs … get

A large part of the algorithm is in handling the loss calculations, so you'll explore that next. Recapping our architecture, recall that there are two sets of loss. There's the content loss that comes from comparing the content features between the content image and the generated image. You can get the content features from each image by using one of the layers in the fifth and last convolutional block of the VGG19.

Then there's the style loss which compares the style features between the style image and the generated image. You can get the style features using the first layer of each convolutional block. Then you can combine these two losses to get the total loss. Recall that the total loss is a combination of the content loss and the style loss. You can simply add the content loss to the style loss like this.

layer, passing in each layer name and then calling dot output. As you can see here, outputs is vgg get layer of name.output for the name in each layer names list.
The list of outputs from the feature extracting layers will then be passed into the call to tf.keras.Model. For the inputs, you'll specify the original VGG input and set the outputs list created by the previous line of code. Specifying the list of layers that you want to output, allows you to easily experiment with the resulting model. Now you've defined the model and you've pre-processed the images, next up, you'll look at the loss functions for the styles and for the content,
as well as the total loss, which combines the style and content losses. You'll see that in the next video.

Total loss and content loss

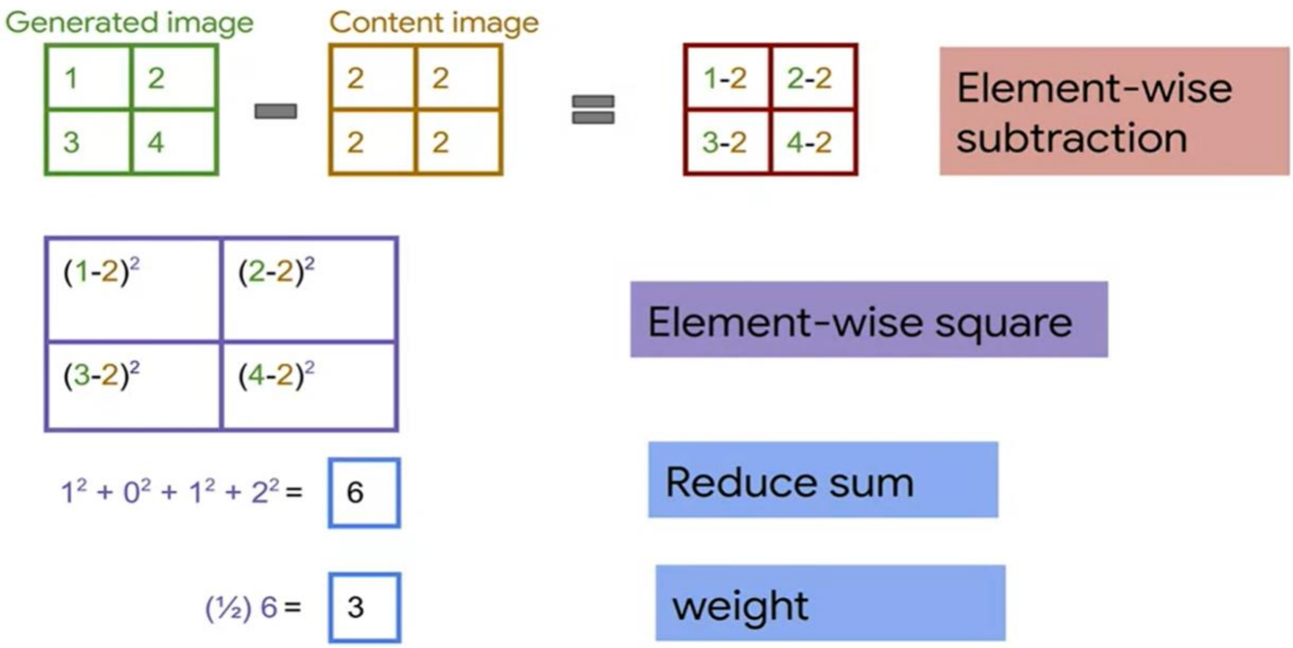


But you'd have much more flexibility in how much importance you want to give to keeping the content versus how much you want to transferring the style. If you can multiply each by a scaler representing the weights that you'll give to each loss.

In this case, Alpha is the weight that you'll give to the content loss, and Beta is the weight that you'll give to the style loss.

This is the same equation for total loss, but with some details on what inputs go into the loss functions. The content loss function takes as input the content image represented by the variable p and the generated image represented by the variable x. The style loss function takes as input the style image represented by the variable a and the generated image represented by the variable x. The total loss is therefore a function of all three images, the content image p, the style image a, and the generated image x.

Content Loss



$$L_{content}(\overrightarrow{p}, \overrightarrow{x}, l) = \frac{1}{2}\sum_{i,j}(F^l_{i,j} - P^l_{i,j})^2$$

l: layer *l*

F^l_{ij}: Content representation of generated image x in layer l (Activation of *i*th feature map at position *j* in layer *l of generated image).*

P^l_{ij}: Content representation of content image p in layer l. (Activation of the *i* th feature map at position *j* in layer *l* of content image.)

```
def get_content_loss(features, targets):
  return 0.5 * tf.reduce_sum(tf.square(features - targets))
```

Style Loss

$$\overset{\text{Generated image}}{\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}} - \overset{\text{Style image}}{\begin{bmatrix}2 & 2\\ 2 & 2\end{bmatrix}} = \begin{bmatrix}1-2 & 2-2\\ 3-2 & 4-2\end{bmatrix}$$

Element-wise subtraction

$$\begin{bmatrix}(1-2)^2 & (2-2)^2\\ (3-2)^2 & (4-2)^2\end{bmatrix}$$

Element-wise square

$$1^2 + 0^2 + 1^2 + 2^2 = \boxed{6}$$

Reduce sum

$$({}^{?}/_{?})\ 6 = \boxed{?}$$

weight

Style Loss

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{i,j}^l - A_{ij}^l)^2$$

l: layer *l*

A^l_{ij}: Style Representation(Gram Matrix) of style image a.

G^l_{ij}: Style Representation(Gram Matrix) of generated image x

Gram Matrix

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l$$

G_{ij}^l: inner product between vectorized feature maps i and j in layer l.

Gram Matrix - In Practice

```
def gram_matrix(input_tensor):
  result = tf.linalg.einsum('bijc,bijd->bcd', input_tensorT, input_tensor)
  input_shape = tf.shape(input_tensor)
  num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
  return result/(num_locations)
```

https://numpy.org/doc/stable/reference/generated/numpy.einsum.html

Get Style Feature Representation at Layers

```
def get_style_image_features(image):
  preprocessed_style_image = preprocess_image(image)
  style_outputs = vgg(preprocessed_style_image)
  gram_style_features =
     [gram_matrix(style_layer) for style_layer in style_outputs[:num_style_layers]]

  return gram_style_features
```

Get Style Feature Representation at Layers

```
def get_style_loss(features, targets):
  return tf.reduce_mean(tf.square(features - targets))
```

Update the generated Image

```
def get_style_content_loss(style_targets, style_outputs,
                           content_targets, content_outputs,
                           style_weight, content_weight):

  style_loss = tf.add_n([get_style_loss(style_output, style_target)
                           for style_output, style_target in zip(style_outputs, style_targets)])

  style_loss *= style_weight / num_style_layers

  content_loss = tf.add_n([get_content_loss(content_output, content_target)
               for content_output, content_target in zip(content_outputs, content_targets)])
  content_loss *= content_weight / num_content_layers

  loss = style_loss + content_loss
  return loss
```

```
def calculate_gradients(image, content_targets,
                        style_targets, style_weight,
                        content_weight,with_regularization=False):

    with tf.GradientTape() as tape:
        style_features = get_style_image_features(image)
        content_features = get_content_image_features(image)
        loss = get_style_content_loss(style_targets, style_features,
                                      content_targets, content_features,
                                      style_weight, content_weight)

    gradients = tape.gradient(loss, image)

    return gradients
```

```python
def update_image_with_style(image, content_targets, style_targets,
                            optimizer, style_weight, content_weight,
                            with_regularization=False):

  gradients = calculate_gradients(image, content_targets,
                                  style_targets, style_weight,
                                  content_weight,with_regularization)

  optimizer.apply_gradients([(gradients, image)]) #Apply gradients on image
```

```python
def fit_style_transfer(input_image, style_image, optimizer,
                       epochs=1, steps_per_epoch=1,
                       with_regularization=False, style_weight = 0.01):


  for n in range(epochs):
    for m in range(steps_per_epoch):
        update_image_with_style(input_image, content_targets,
                                style_targets, optimizer,
                                style_weight, content_weight,
                               with_regularization=with_regularization)

  return input_image, images
```

Visualizing Outputs - Final Results

https://upload.wikimedia.org/wikipedia/commons/b/b4/Vassily_Kandinsky%2C_1913_-_Composition_7.j

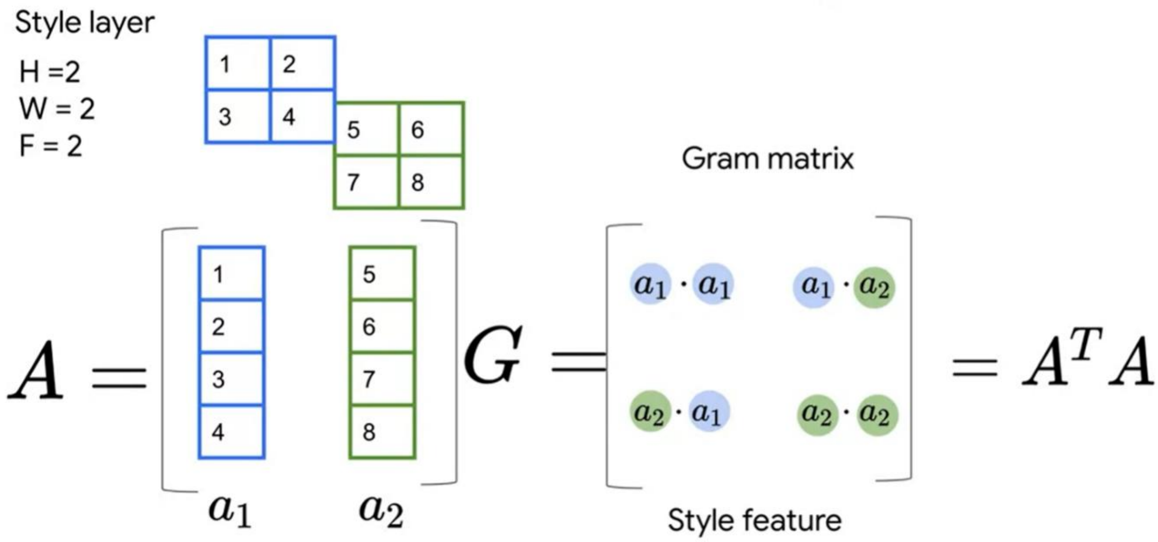
https://cdn.pixabay.com/photo/2017/02/28/23/00/swan-2107052_1280.jpg

Gram Matrix

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}$$

G_{ij}^{l}: inner product between vectorized feature maps i and j in layer l.

Style Loss (one layer)



Style Loss (code)

```
style_layer = tf.constant([1,2,3,4,5,6,7,8],
                          shape=(2,2,2))
```

```
[[[1, 2],
  [3, 4]],

 [[5, 6],
  [7, 8]]]
```

```
A = tf.transpose(
        tf.reshape(style_layer,
                   shape=(2,4)))
```

```
[[1, 5],
 [2, 6],
 [3, 7],
 [4, 8]]
```

```
AT = tf.transpose(A)
```

```
[[1, 2, 3, 4],
 [5, 6, 7, 8]]
```

```
G = tf.matmul(AT,A)
```

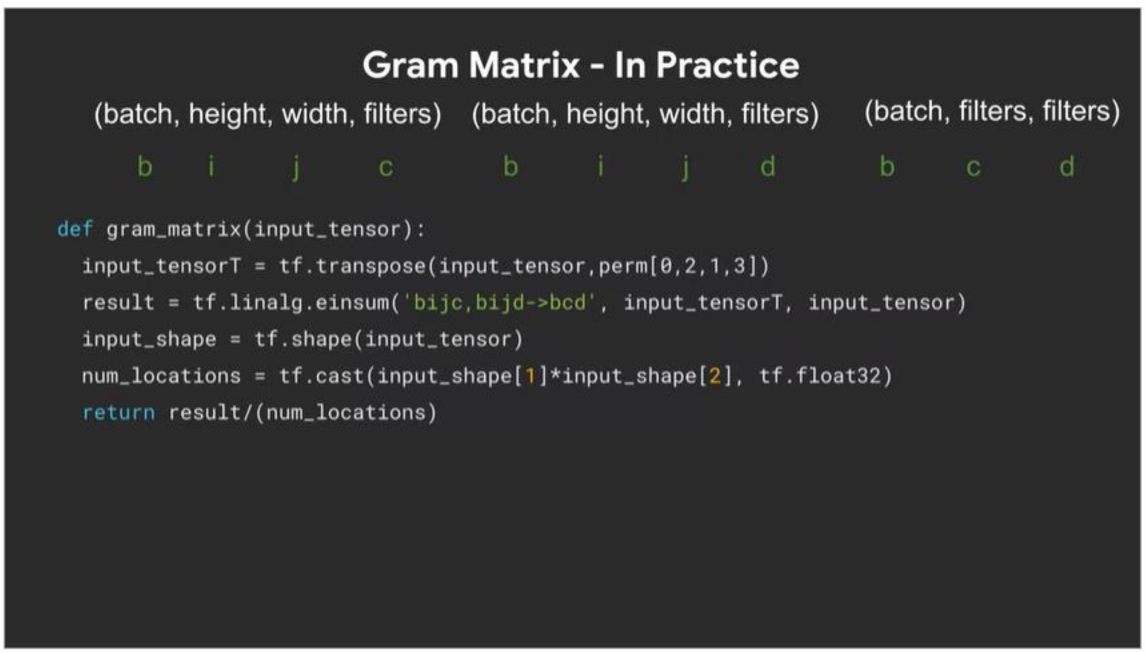
```
G = tf.linalg.einsum('cij,dij->cd',
                     style_layer,
                     style_layer)
```

```
[[ 30,  70],
 [ 70, 174]]
```

Recall that the core lab for this lesson calculates the gram matrix with something called Einstein notation. In code, you can just use tf dot L-I-N-A-L-G, linalg dot einsum, where einsum stands for Einstein sum. If you run this line of code passing in the following string then the original style layer for both of the input tensors, you'll see that it does the same thing as flattening each filter's matrix into a column vector, putting those vectors into the matrix A and performing the matrix multiplication of A transpose with A.

The string with C-I-J comma D-I-J arrow C-D may look rather magical at the moment, and in the next optional sectionI'll discuss this Einstein function a bit so that you'll know what's going on when you practice it in the core lab.It's optional, but it's kind of fun.
So check it out if you have some time.

Optional - Einsum in Code



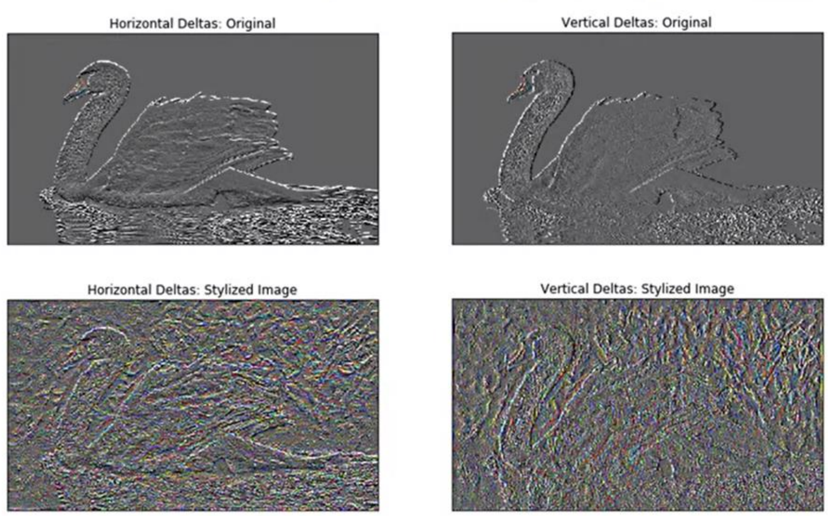
Now that you've seen how to do neural style transfer, you might have noticed that there were a lot of artifacts in the final image. Whereas the initial image had a lot of smooth colors, there were edges and there were ridges in the stylized one, which you can see if you look closely here.

To work on removing these, we can use edge detector filters known as high pass filters, and you can see some here. A high pass filter smooths out the image by only allowing values above a certain threshold through. There's lots of ways that you could do this. And this algorithm determines the differences between adjacent pixels on both the x and y axes to help us regularize them.

Extract High Frequency Components

```python
def high_pass_x_y(image):
  x_var = image[:,:,1:,:] - image[:,:,:-1,:]
  y_var = image[:,1:,:,:] - image[:,:-1,:,:]

  return x_var, y_var
```

Visualization-High Frequency Artifacts



Solution - Total Variation Loss

- Decrease high frequency artefacts.
- Explicit regularization term on high frequency components

Factoring in Total Variation using TensorFlow

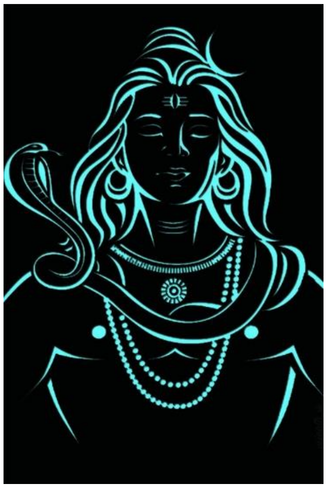
```
loss += total_variation_weight * tf.image.total_variation(image)
```

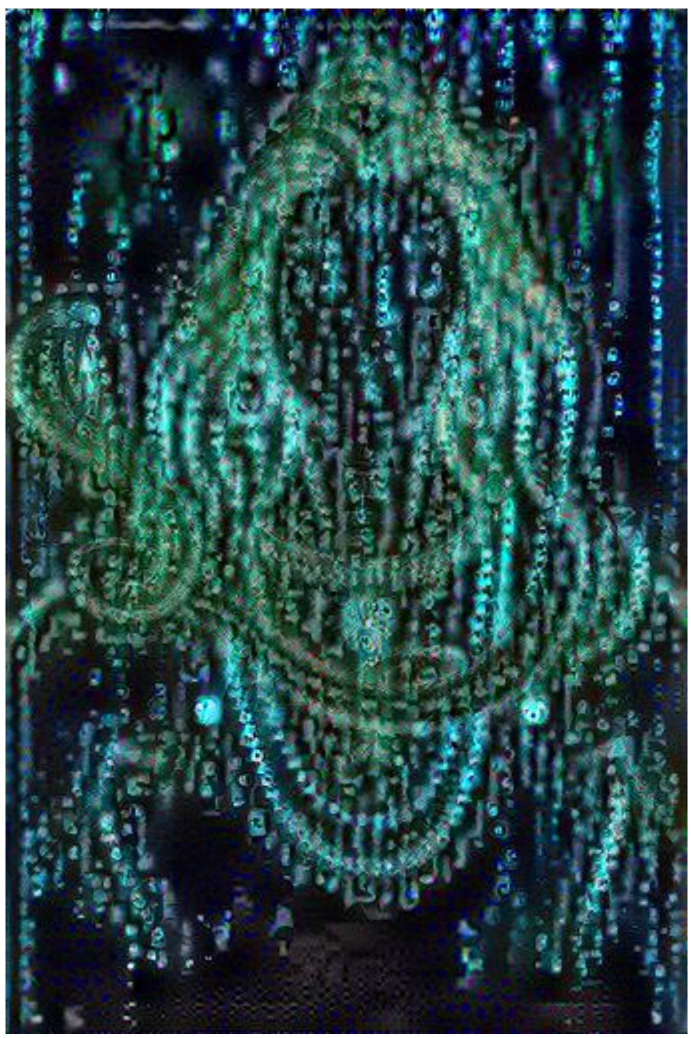
If we want to reduce this, we can do it on the loss function. With an update to make it something called a total variation loss. Its goal is to decrease the high frequency artifacts and apply explicit regularization on the components that have high frequency values. We can then calculate the total variation of an image using a function in tf.image that implements the algorithm for us. We multiply this by a variation weight that we assigned for our stylization.

```
def calculate_gradients(image, content_targets,
                        style_targets, style_weight,
                        content_weight,with_regularization=False ):

    total_variation_weight = 30
    with tf.GradientTape() as tape:
      if with_regularization:
        loss += total_variation_weight*tf.image.total_variation(image)

    gradients = tape.gradient(loss, image)
    return gradients
```

My own creations

Without total variation loss



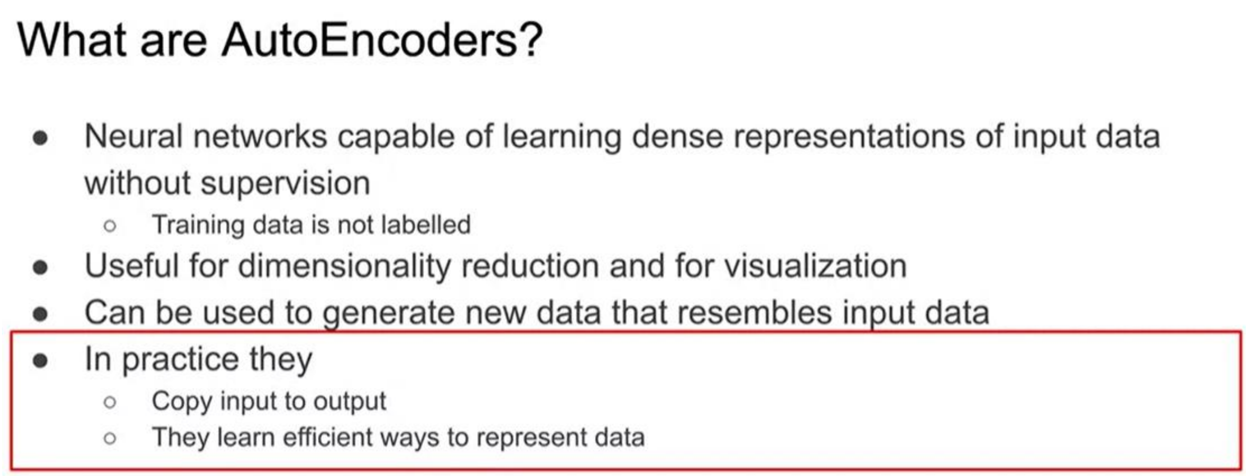


Fast Neural Style Transfer

```
import tensorflow_hub as hub

h_m = hub.load
        ('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')

content_image, style_image = load_images("content.jpg", "style.jpg")

stylized_image = h_m(tf.image.convert_image_dtype(content_image, tf.float32),
                     tf.image.convert_image_dtype(style_image, tf.float32))[0]

tensor_to_image(stylized_image)
```

- Exploring the structure of a real-time, arbitrary neural artistic stylization network (Ghiasi, Lee, Kudlur, Dumoulin & Shlens, 2017)

What Are AutoEncoders?



Welcome to this week where you look at auto encoders and how they can be used to generate new content. You'll start with a look at the architecture of an auto encoder before diving into coding, some for m nest and other data sets. So let's get started by exploring what auto encoders actually are. After this, you will explore the architecture of a basic auto encoder.

Ultimately, an auto-encoder is a neural network that's able to learn an internal representation of its input data without supervision. Your data is not labeled and I often like to think of an auto-encoder as similar in concept to compression methodology, that takes a big image and turns it into a smaller image without losing too much information. For example, JPEG is a common format for images, but it's not a direct compression of the pixels. It's a lossy representation of the content in the image, but one which is able to maintain the sense of the original image.

As such auto-encoders can be very useful for dimensionality reduction where, for example, in an NLP problem, you might have an embedding that has 10 or more dimensions, but you can reduce those while still maintaining some kind of context of the data and similarly, this makes it useful to visualize data. It's hard to visualize ten dimensional data but relatively easy to do so with three dimensional data. Such an AutoEncoder could be very useful here but this course is about generative deep learning.

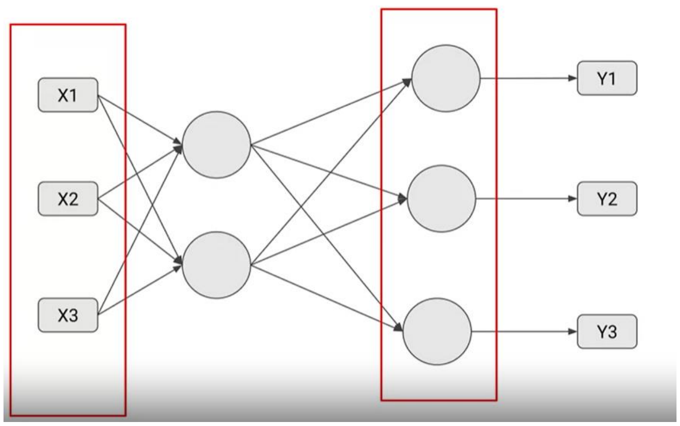
So we'll explore how AutoEncoders can be used to create new data by learning a representation of the input data, and then creating things in the method of the internal representation so that a decoding can create something new.

You do this by copying inputs to outputs via learned ways to represent the data. For example, attempting to learn a representation of the data that closely resembles the data itself, tested by comparing the output which is reconstructed from the representation with the original input.

83 12 21 42 99 18 51

4 9 16 25 36 49 64 81 100 121 144 169

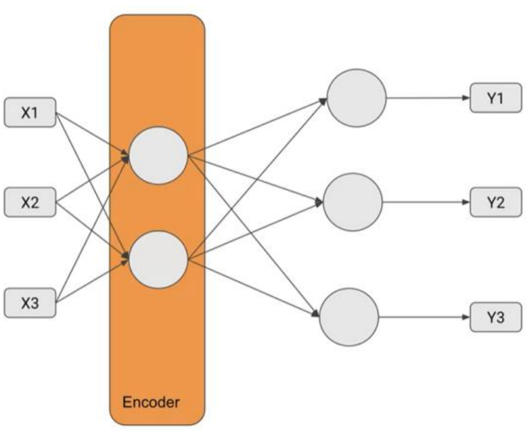
Consider, for example, these two sets of numbers. If I were to ask you to look at them for a moment and then look away, which one do you think you could remember more?
At first glance, you might think that this sequence is easier to remember because it's shorter.But then on second glance, I bet you'll do better with this set because after looking at it for a moment, you probably realize that they're all squares. Thus if you're asked to repeat them back, you wouldn't need to remember them, you could figure it out based on the fact that there's a pattern between them.
This pattern can be called a latent representation and the goal of an autoencoder is to create a latent representation. So that when it's asked to produce an output, it can do so in much the same way as you just did when you observe that the list of squares was the best way to remember this list so that you could output a new one.



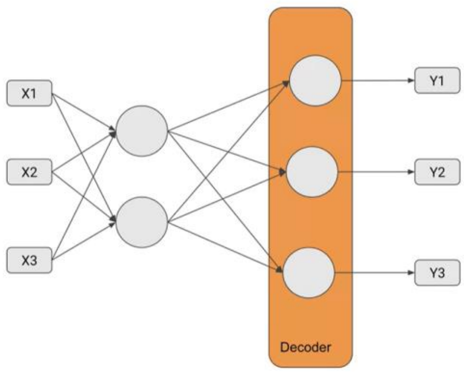
So when creating an autoencoder, your architecture can look a little bit like this, and while it might look like it's just a DNN, there are a couple of constraints.

First is that the number of output neurons in the output layer should be the same as the number of inputs.

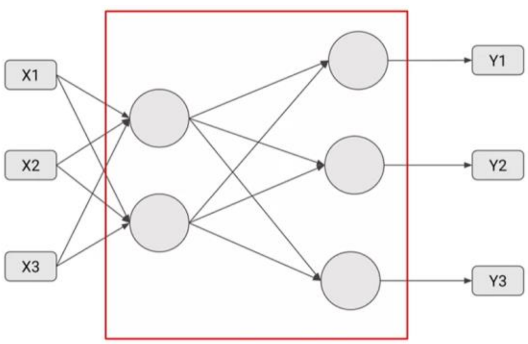
So here you can see I have three input values X1, X2 and X3, and I have three output neurons that give me Y1,Y2 and Y3.



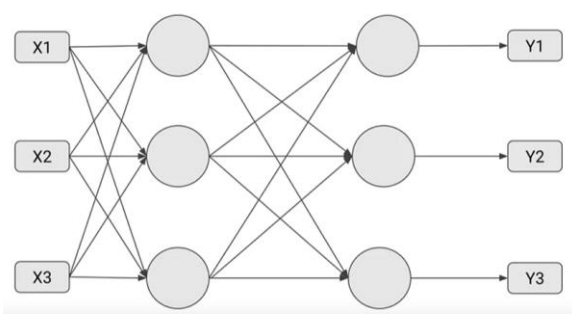
Then the layer that takes the inputs as shown here will encode the latent representation of the input data and as such, that's called the encoder layer.



And the layer that decodes the latent representation and turns it into an output is called the decoder.



So you might be wondering why are there different numbers of neurons in these layers? Why, for example, aren't there three neurons in the input layer like this?



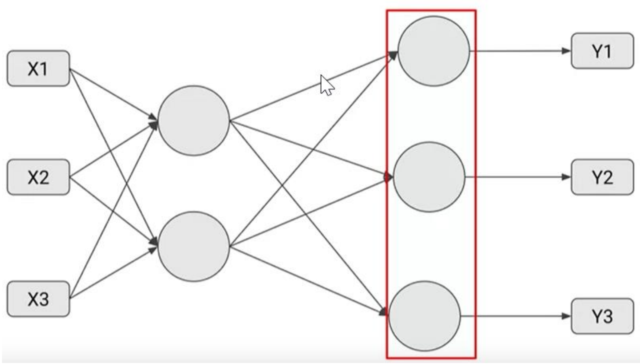
Well, the answer is that if the network ends up looking like this, we can have a straight pass through from the inputs to the outputs and no pattern to represent them would be learned.

So, we design a network like this in a matter which is called under complete.

The internal representation has two neurons, which could be seen as two dimensions to represent the three dimensions of original data, hence the term under complete.

We'll next look at creating an auto encoder that represents 3D data using two dimensions.

First Encoder



Now that you've had a look at what autoencoders are, let's take a look at building a very simple one.

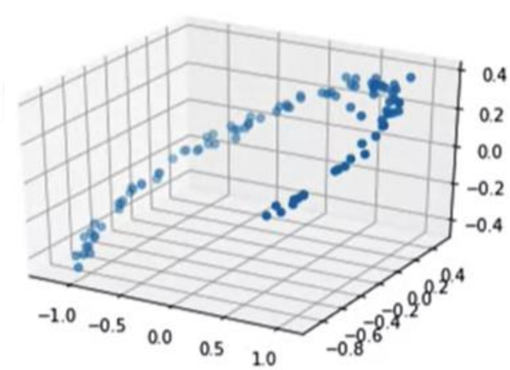
We'll have a three-dimensional set of data plotted on a chart, we'll then use an autoencoder to learn a 2D latent representation of that data, which we'll plot in two dimensions. Then, from this data we'll attempt to reconstruct something closer to three-dimensional regional using a decoder.

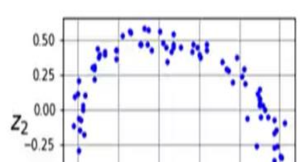
Let's consider the autoencoder that we saw earlier. It has 3D data as an input, X1, X2, and X3. When the data is encoded, it will lose a dimension and be encoded into a 2D representation.

When it's decoded, it will be returned to a 3D representation. But of course, given that the dimensions were
appr

Let's explore this. This image is a plot of our 3D data. If you look closely, it's roughly circular or horseshoe shaped. The values are randomly scattered on the z-axis. If we encode with two neurons as we saw on the previous architecture, and then, look at the encodings, we can see something like the image on the right.

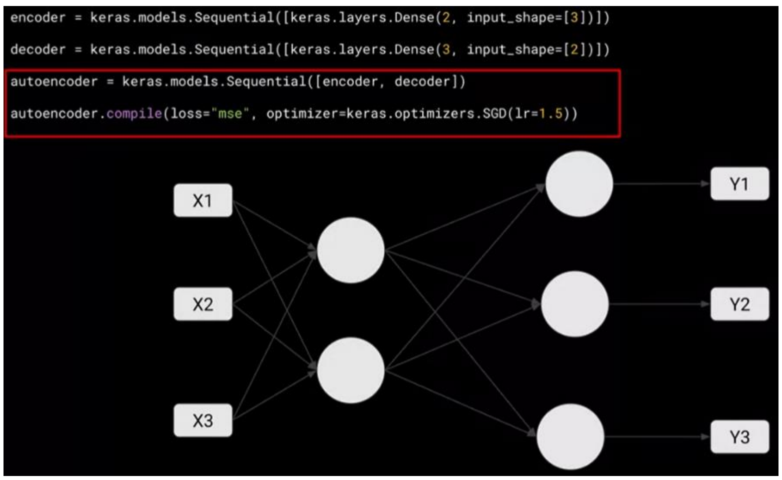
Despite the data being reduced to two





We'll get this, where you can see that the rough shape of the data was maintained, but the approximation lost a lot of the noise in the input data and regularized it somewhere.

Still, it's pretty good replica of the original producing a 3D output from the 2D latent representation that was learned by the network.

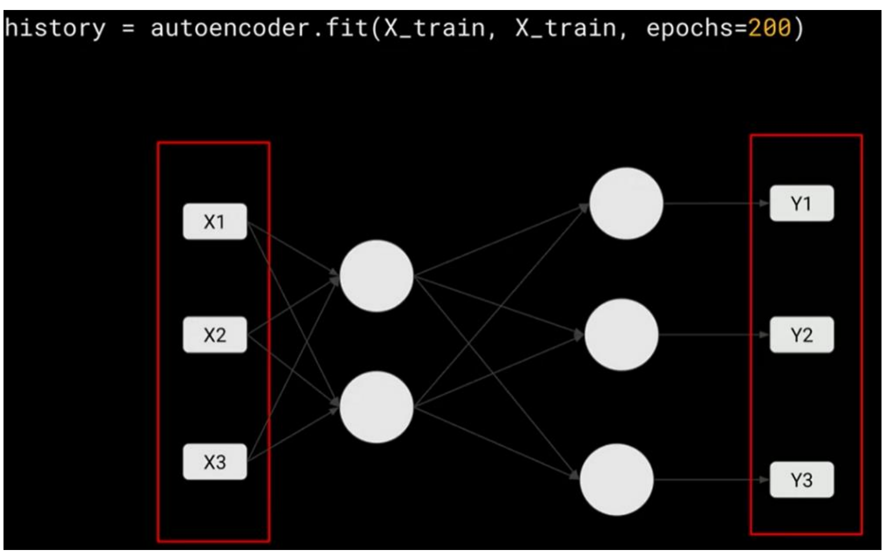


Creating this encode is really straightforward.

We've two dense layers, one for the encoder and one for the decoder. We can create the encoder as a single dense layer with two neurons and an input shape of three. As you can see in the diagram, the layer has two neurons and their three values input to it is called X1, X2, and X3.

Similarly, we'll create the decoder as a dense layer with three neurons, it's also our output layer. We specify it to have an input shape of two because it's designed to decode the 2D data.

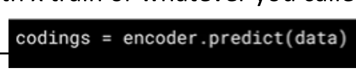
Later, we can call this with data to see if we can reconstruct what we had encoded. We then define our model as a sequential of these two layers and compile it.



Training with the autoencoder is a little unusual. You're typically used to fitting X to Y values. While that's what you're doing here, you're actually fitting the training data to itself. Indicated by the fact that you use x train twice in your model.fit.

Now, why would that be? Recall the model. Ultimately, if you give it input values of X1, X2, and X3, the best reconstruction would be if Y1, Y2, and Y3 were equal to X1, X2, and X3. By setting the training x and training y to be the same value, you'll get that impact.

Now, if we want to get the 2D encoding of our 3D data, we simply call predict on the encoder. If you want to see what it's like for our training set, you can simply replace data with x train or whatever you called your dataset.



```
decodings = decoder.predict(codings)
```

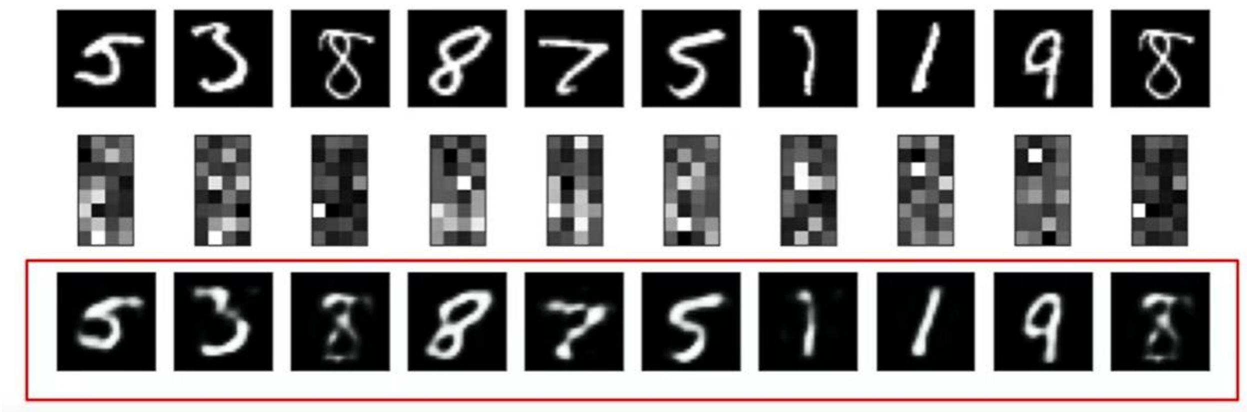
Similarly, if we want to get our 3D prediction, we can call predict on our decoder passing it 2D data, just like our codings.

Now, that was a very simple autoencoder, but hopefully it demonstrates the overall concept where the model can learn a latent representation of your data that could be used to reconstruct a replica of that data.

It worked with a very simple 3D data. But what happens if your data is more complex like, for example, images? Well, in a similar manner to what you did when you first started learning DNNs for image classification, the next step is to have multi-layered networks to help encode and decode more complex data.

In the autoencoder world, these are referred to as stacked autoencoders and you'll explore them soon.
But first, check out the Colab for this simple example and then play with tweaking the parameters such as the function that generates the 3D data or hyperparameters on the network and see if you can discover any interesting and fun effects.

MNIST AutoEncoder



Now that you've played with your first auto encoder and you saw how it could reduce dimensions in a 3D set, you then created an approximation of that set. Let's now move to something a little bit more challenging.

For the rest of this week, you're going to explore auto encoders for computer vision, starting with a simple one that works on MNIST, which gives you output like this. We'll start with the MNIST dataset, a few samples of which are here.

We'll use an encoder to encode them into a representation that is lower resolution like these, from which the decoder can recreate them quite well, with the images here being the decoded versions of the values that the encoder created

Now let's take a look at the code for building this. Remember that MNIST is 28 by 28. So when we flatten them out into a single dimension vector, it will be 784 by 1. So our input shape will be 794 times something, where something is the number of images that we're loading in.

As before, we'll have an encoder that downsizes the inputs to create an internal representation. In this case the encoder will take the 784 inputs and try to represent them as 32 neurons. The decoder will have the same number of neurons as the number of input values, which in this case is 784. So let's create a function to implement them.

```
inputs =  tf.keras.layers.Input(shape=(784,))

def simple_autoencoder():
    encoder = tf.keras.layers.Dense(units=32, activation='relu')(inputs)
    decoder = tf.keras.layers.Dense(units=784, activation='sigmoid')(encoder)
    return encoder, decoder

encoder_output, decoder_output = simple_autoencoder()

encoder_model = tf.keras.Model(inputs=inputs, outputs=encoder_output)

autoencoder_model = tf.keras.Model(inputs=inputs, outputs=decoder_output)
```

And as you can see here, the function will return the encoder and the decoder layers. We'll then call this function and load the return values into variables called the encoder output and decoder output respectively.

You can instantiate a model by using the tf.keras.model class passing it inputs and outputs so we can create an encoder model that takes the inputs, but gives us its outputs as the encoder outputs. Calling this model will return the encoded representation of our input values.

And then we can also see the path of an image through the auto encoder by creating another model which takes the inputs, but then outputs the decoder output. Calling this model will then encode the image into an internal representation and then decode the image.

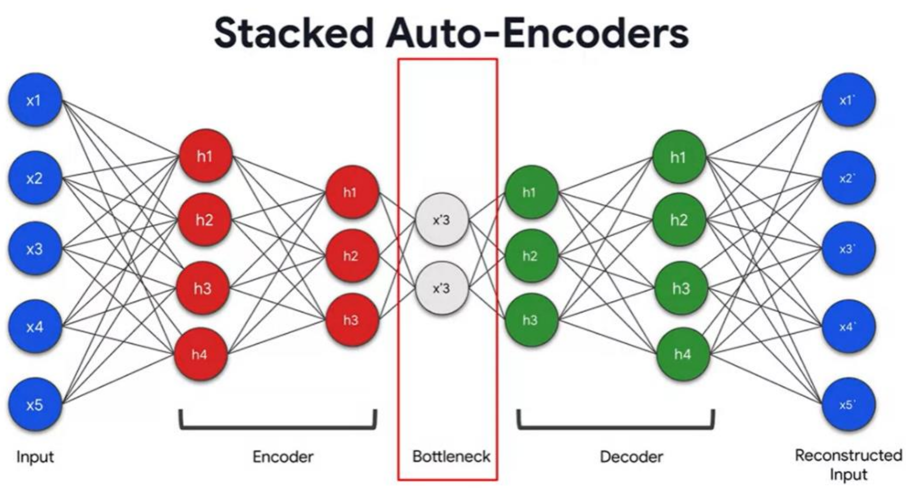
This is what we saw previously when we looked at the image that internal representation and then the reconstruction of the image based on that representation

```
autoencoder_model.compile(
        optimizer=tf.keras.optimizers.Adam(),
        loss='binary_crossentropy')
```

We can then compile our model with an optimizer and a loss function, but pay attention to our loss function in particular. Remember that it's an unsupervised learning so we aren't matching values to labels like with a classification task.

Instead, our concept of labels are the pixel outputs from the decoder. We want them to match the input labels so we could see each pixel in the output with our value representing the probability that the value is white. A 0 value in the pixel is black, and a 1 value is white, and
everything in between is grayscale. This maps really nicely to 0 probability is black and 1 is white, and everything in between can be scaled to a grayscale.

MNIST Deep AutoEncoder



Earlier you built a simple auto encoder that learned encodings based on the MS data set, and it was then able to decode and MS digits easily. This used a shallow encoding decoding network where there was only a single layer for each of the encoder and the decoder.

And this is possible because MNIST is a pretty straightforward and easy data set. As you get into more complex data sets, you may need to use multiple layers for each of the encoder and the decoder.

Think about it in the same way as you did with classification. For more complex images, a single dense layer usually wasn't enough, and it's very, very similar here. So in order to achieve encoding and decoding for more complex images, a stacked auto encoder architecture can be used and it can look a little like this.

As before, your input is fed into the encoder. But the initial layers in the encoder have a lower dimensionality. But the encoder architecture might be more sophisticated than a single layer. So we

Where our encoder now has three layers with our initial input of 784 values being filtered through a deep neural network with three layers of 128, 64 and 32 neurons respectively.

So our latent representation is handled by 32 neurons. And the output of these 32 neurons will be fed when encoding to the decoder which also has three layers. These are 64, 128 and 784 neurons. Remember that the final layer needs to have the same number as the initial input, which in this case is 784 because each image had 784 pixels in it.

The rest is the same as before, where we get the encoder model and the auto-encoder model and train them on the data. And the results can be seen here. The top of the original images in the middle of the latent encodings, and at the bottom are the reconstructions made from those encodings. They're a little bit more accurate than our shallow auto encoder from earlier, and they're probably hitting the law of diminishing returns at this point. But as you experiment with more complex images, you'll see that this technique will become essential.

Next up, you're going to work on the deep auto-encoder co-lab after which the logical next step is to explore convolutions and how you can build a convolutional auto-encoder.

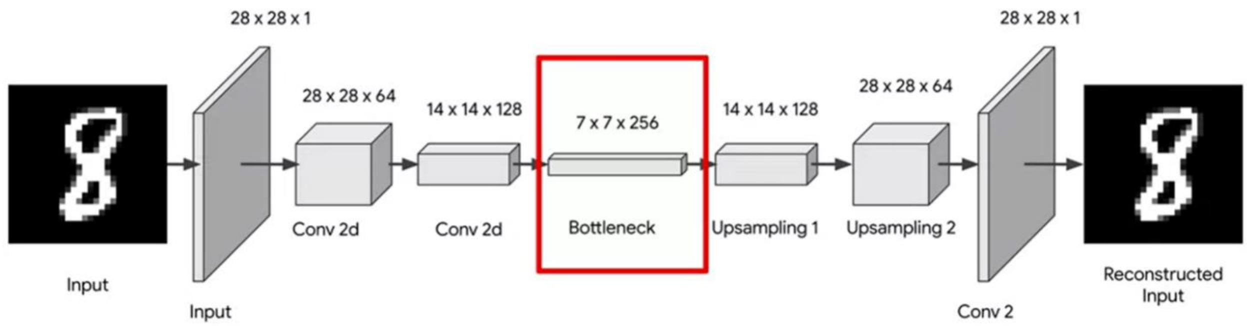
```
def deep_autoencoder():
  encoder = tf.keras.layers.Dense(units=128, activation='relu')(inputs)
  encoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
  encoder = tf.keras.layers.Dense(units=32, activation='relu')(encoder)

  decoder = tf.keras.layers.Dense(units=64, activation='relu')(encoder)
  decoder = tf.keras.layers.Dense(units=128, activation='relu')(decoder)
  decoder = tf.keras.layers.Dense(units=784, activation='sigmoid')(decoder)

  return encoder, decoder

deep_encoder_output, deep_autoencoder_output = deep_autoencoder()

deep_encoder_model = tf.keras.Model(inputs=inputs, outputs=deep_encoder_output)
deep_autoencoder_model = tf.keras.Model(inputs=inputs, outputs=deep_autoencoder_output)
```

Convolutional AutoEncoder

So far this week, you've seen how autoencoders work and we used M-NEST as an example for first creating a simple auto encoder. Then looked into how it could be extended to be a deeper autoencoder. As you might suspect, autoencoders can use multiple layer types.

When it comes to computer vision, convolutional layers are really powerful for feature extraction and thus for creating a latent representation of an image.
In this video, you'll explore what a convolutional autoencoder could look like.
For example, let's compare the outputs of an autoencoder for fashion amnesty trained with the DNN and trained with a CNN. Here you can see the 10 input items and they're output from an autoencoder that's based on a DNN architecture. It looks pretty good.
Here are the same items, but with our output from a CNN based autoencoder. Now, I think they look a lot better, so let's explore. For example, take these three shoes and compare their DNN output at the top with their CNN output at the bottom.

Convolutional Auto-Encoders



Let's now explore a convolutional architecture, it is pretty straightforward. The encoder will use convolutional layers combined with pooling to extract features from the image and then perform the reduction of the image size.

This will give you a lower resolution, seven by seven latent representation of the image and this will be our bottleneck. Your decoder will do the opposite using upsampling with convolutions, it will increase the resolution of the image back to the original 28 by 28.

```
def encoder(inputs):
  conv_1 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),
                                  activation='relu', padding='same')(inputs)

  max_pool_1 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_1)

  conv_2 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3),
                                  activation='relu', padding='same')(max_pool_1)

  max_pool_2 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_2)

  return max_pool_2
```

```
def bottle_neck(inputs):
  bottle_neck = tf.keras.layers.Conv2D(filters=256, kernel_size=(3,3),
                                       activation='relu', padding='same')(inputs)


  encoder_visualization = tf.keras.layers.Conv2D(filters=1, kernel_size=(3,3),
                                                 activation='sigmoid',
                                                 padding='same')(bottle_neck)

  return bottle_neck, encoder_visualization
```

```python
def decoder(inputs):
  conv_1 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3),
                                  activation='relu', padding='same')(inputs)
  up_sample_1 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_1)

  conv_2 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3),
                                  activation='relu', padding='same')(up_sample_1)
  up_sample_2 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_2)

  conv_3 = tf.keras.layers.Conv2D(filters=1, kernel_size=(3,3),
                                  activation='sigmoid',
                                  padding='same')(up_sample_2)

  return conv_3
```

First, the encoder, the first layers in the encoder are a Conv2D followed by a MaxPool. This will reduce the resolution from 28 by 28 to 14 by 14 while extracting features.
The next layer will do the same. Create filters for the next layer of abstraction with the pool, reducing the resolution to seven by seven and then will return max pool 2 so the bottleneck function can use this as its input.

Will see that next. Here's the bottleneck. It will take the output of the encoder, which was the second max pool as its input. Will give you the bottleneck of 256 filters. There's no max pool here, so you don't reduce the dimensionality any further. The images will stay seven by seven and you can add another layer here
that doesn't impact the autoencoder.

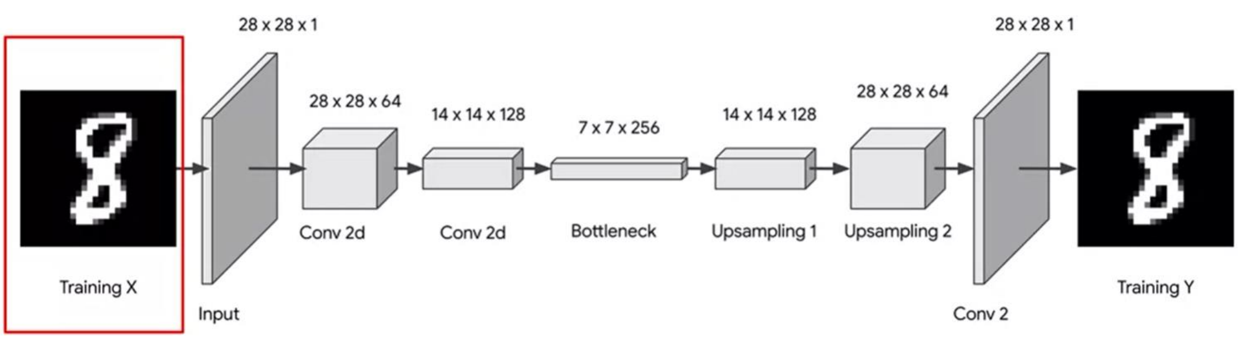
This is utility that we can use to visualize the internal representations. You'll return the bottleneck and the visualization. The former will be the input to the decoder, the ladder for drawing the internal representations
if you want to examine them. Now, comes the decoder and this should look very similar to the encoder as you want to return the images back to that 28 by 28 resolution.

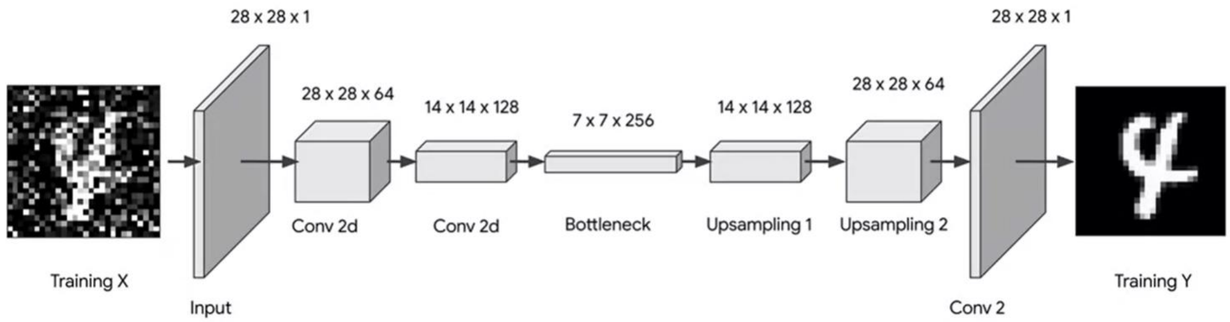
You'll have a convolutional block with a two by two up sampler and this will give you 14 by 14 images. Then another one with a two by two upsampler that will give you 28 by 28 images before a final output layer with just a single filter in it and this will reconstruct the image from all of the previous filter's, giving us a 28 by 28 by 1 output. This layer will be the return from the function. Now, to build your convolutional autoencoder architecture. You get started by defining the input shape is 28 by 28 by 1, because this is a CNN which needs all three dimensions.
You call the three functions that we
just discussed to create the encoder,
the bottleneck, and the decoder.
Then you can create the master model
by specifying the input as
your input and the outputs
as the outputs from the decoder.
Another model that you'll call the encoder model that
will output the latent representation
from the bottleneck.
You can use that to visualize the filters. That's it.
Next up, you'll play with the code to
create a convolutional autoencoder.
You can try it out for yourself.

```python
def convolutional_auto_encoder():
  inputs = tf.keras.layers.Input(shape=(28, 28, 1,))

  encoder_output = encoder(inputs)

  bottleneck_output, encoder_visualization = bottle_neck(encoder_output)

  decoder_output = decoder(bottleneck_output)

  model = tf.keras.Model(inputs =inputs, outputs=decoder_output)
  encoder_model = tf.keras.Model(inputs=inputs, outputs=encoder_visualization)

  return model, encoder_model
```

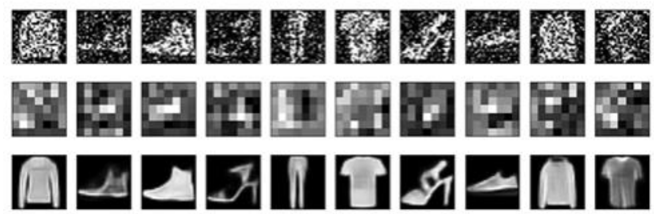
Denoising with an AutoEncoder

Previously, you saw how to build a CNN-based auto-decoder that after reducing a
28 by 28 Fashion-MNIST image to its latent representations was then able to reconstruct those images remarkably well. For a little fun, let's see what will happen if we add some noise to the training data.
Will it be able to reconstruct the images just as well as it had before and effectively remove the noise?
Recall when training the neural network, you provide the same image as the training x and the training y. You don't label the image but you could also see it that you provide the image as a label to itself. You're effectively telling the network to train itself to create the same output as the input. The difference, of course, as you already know with autoencoders, is that there's a latent representation in the middle here,
which effectively gives you the rules for encoding an image accurately if it's able to learn with little loss.
Then what happens if you try to introduce noise at the training stage?





Instead of the training image being a nice, clean one like this, you make it noisy like this. The question will then become, can the autoencoder learn the desired output of a clean image from input images like noisy ones?



Here's an example of some images with noise added and then trained with exactly the same architecture as you saw in the previous video.

Here's the results when done with Fashion-MNIST. It's not bad, right?

```
def map_image_with_noise(image, label):
  noise_factor = 0.5
  image = tf.cast(image, dtype=tf.float32)
  image = image / 255.0

  factor = noise_factor * tf.random.normal(shape=image.shape)
  image_noisy = image + factor
  image_noisy = tf.clip_by_value(image_noisy, 0.0, 1.0)

  return image_noisy, image
```

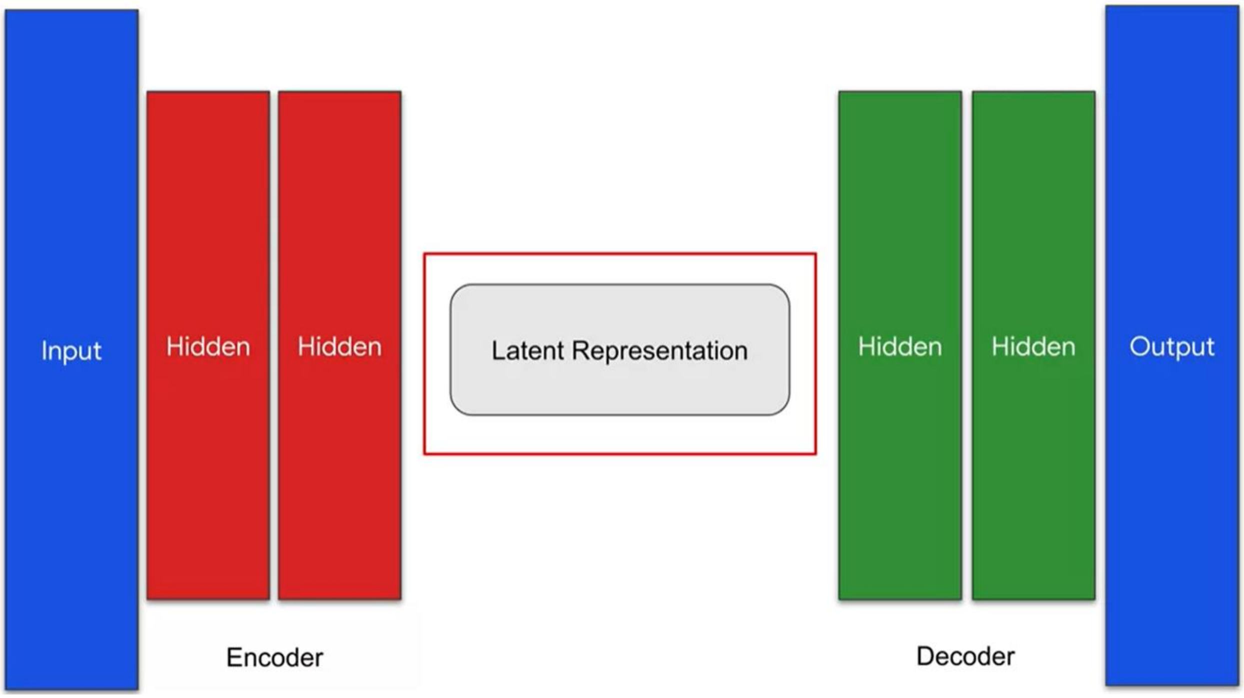
Adding noise is super easy because you're loading Fashion-MNIST from TFTS and this allows you to use a mapping function. You've typically used that to do things like normalizing your data, but of course it could be used also to add noise. This is the code that you'd typically see to normalize your image.

As your image is just an array of values, you can create a new array of the same size and shape called factor here. You'll add the values in this to your image values and because the image is normalized, you want to ensure that adding values doesn't make any of the image values to be more than one, so you'll use tf.clip by value to ensure that they're in the range from 0-1.

Mapping functions usually return the image and the labels in a scenario like this one and in all the previous examples, you've just returned image, image, making the image effectively its own label for unsupervised learning. But in this case, you want the training input or x to be the noisy image and the label, or y to be the original.

You'll return the noisy image followed by the image as this returns xy, or image label or data label, that type of thing. Our data is the image noisy and our label will be the image.

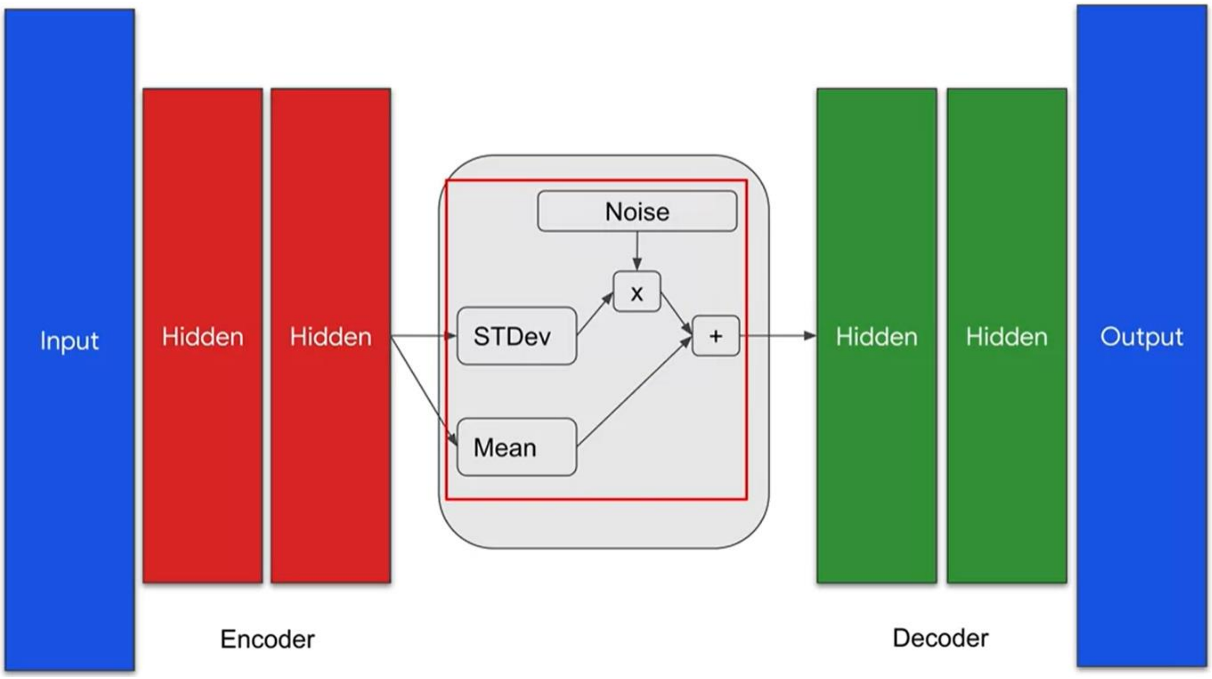
VAE Model



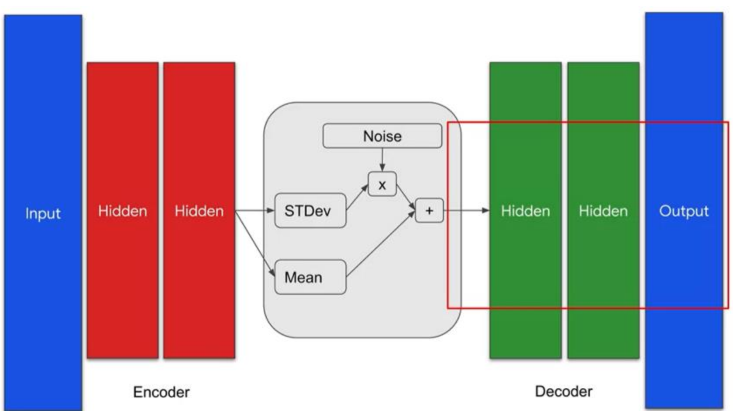
Last week you looked at auto encoders and how they could be used to create a latent space of the inputs, which was typically a reduced amount of dimensions and thus data, which could then be reconstructed into data that looked like the original. In some ways, I liken it to a compression methodology where you can reconstruct something that looks like the original, despite losing data from the compression.

You also saw how to use this to, for example, remove noise from an image. And while we could use auto encoders to generate new data, it wasn't fully generative. It was more a case of reconstruction from original data. However, the introduction of variational auto encoders by Diederich kingma and Max welling, makes using them to create entirely new data possible. We're going to explore that this week.

So to recap, auto encoders would look a little like this, where there's an input vector, which goes through an encoder to the bottleneck. Were a learned representation of the input data is encoded this could then pass through a decoder to give a reconstruction of the input. We explored various architectures for the encoder and decoder that will give us a latent representation that was somewhat analogous to the input data does giving us a rough neural content generator. And we could rewrite that diagram to simplify it a little like this. Our hidden layers for encoding and decoding don't necessarily have to be dense. But what I want to focus on for variational auto encoders is this latent representation of your data with variational auto encoders.

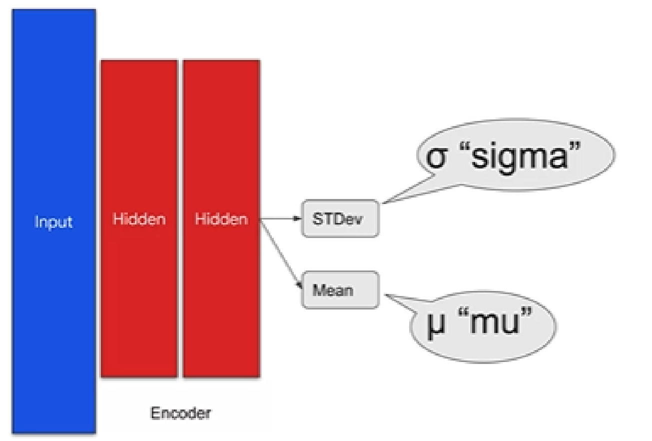


We'll make this more probabilistic so that the later output is partially determined by chance. So that they can create new instances of the data that look and feel, like they came from the training set.
And we'll do this with a more complex latent representation of the data. Let's explore this. First of all, instead of just the output from the encoder being in the latent space, will take two outputs from the encoder.
We'll call these the mean encoding and the standard deviation of the encoding. We'll also generate noise using a Gaussian distribution so that our actual encoding will be sampled from the Gaussian distribution using our standard deviation and our mean by multiplying out the noise by the standard deviation and then adding the results to the mean.

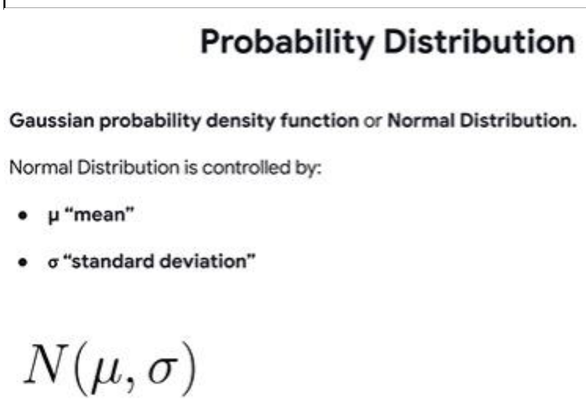


Once you've done this, then a sample from a Gaussian distribution is enough for you to create something that looks like it's in the latent space. And it can then be combined with the learned values from the encoder to emulate what the training data would look like, so that the output layers can reconstruct.

In this case effectively generate something completely new. So key difference in this approach is that our encoder has to produce the multiple outputs of standard deviation and mean. We'll explore that next.

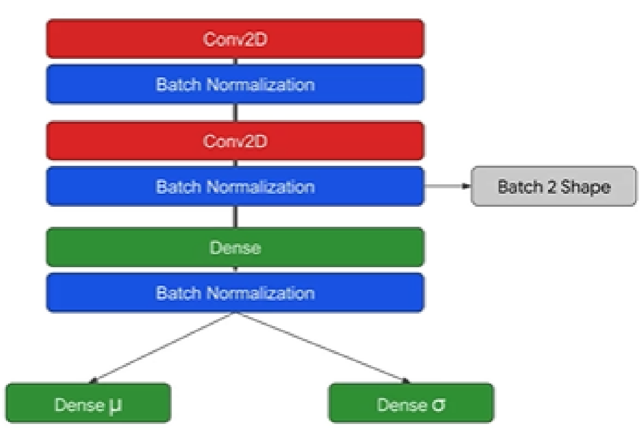


Previously, we saw how the variational autoencoder builds on the encoder architecture by changing the latent space to be a combination of the learned values of the encoder with a Gaussian distribution, so that future values could be predicted from the same distribution. This would require some architectural changes to how we would implement an encoder. Let's explore that next. We left off by looking at how our encoder has to output a standard deviation and a mean that would be used to build a latent representation of the data. These two outputs from the encoder are what's going to be important to us. Typically, the Greek letter Sigma and Mu are use for this, and that's what you'll see in the code.



This function is controlled by two values, a mean and a standard deviation. We'll need to have these values output from our neural network. When looking at the code, remember that you won't see the outputs calculate these values. They will be learned over time as the neural network learns what matches an input value to the output value when the output is generated using the normal distribution.

Let's go back here and take a look at the neural network code that will produce these two outputs that we'll call Mu and Sigma.

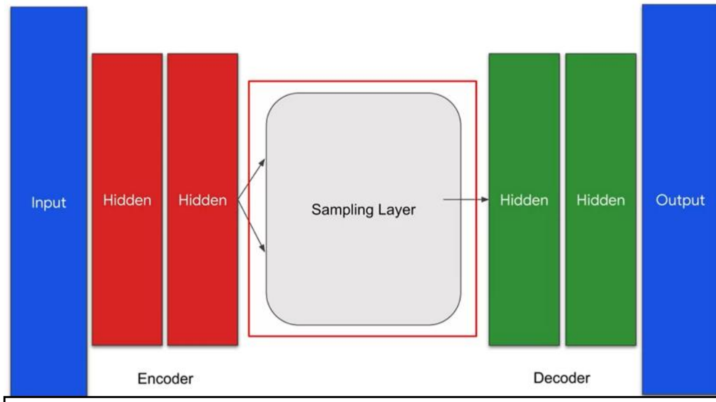


Well, code 1 that looks like this, with a number of hidden layers. We'll start with a Conv2D layer that's followed by a batch normalization. This is followed by similar pairing, a Conv2D followed by a batch normalization. We'll flatten out the results and then feed these into a dense layer, which also gets batch normalized. The output of this will be two dense layers that will represent our Mu and Sigma values. We'll also later want to understand the shape of the output of this layer, so we'll also return the shape of the output at this layer.

Given that this network has multiple outputs, we can use the functional API to construct it.

```
# This function defines the encoder's layers
def encoder_layers(inputs, latent_dim):
  x = tf.keras.layers.Conv2D(filters=32, kernel_size=3, strides=2,
                             padding="same", activation='relu',
                             name="encode_conv1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=2,
                             padding='same', activation='relu',
                             name="encode_conv2")(x)
  batch_2 = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Flatten(name="encode_flatten")(batch_2)
  x = tf.keras.layers.Dense(20, activation='relu', name="encode_dense")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  mu = tf.keras.layers.Dense(latent_dim, name='latent_mu')(x)
  sigma = tf.keras.layers.Dense(latent_dim, name ='latent_sigma')(x)

  return mu, sigma, batch_2.shape
```

We have a deep encoder, so let's go through it little by little to understand what's going on. This is the first set of layers with a Conv2D followed by a batch normalization layer. There's a subsequent set of Conv2D batch normalization, so we code that here. Note that I named this layer batch_2 instead of x because I want to have its output shape as one of the return values from this function. After the second Conv2D batch normalization block, I've flattened the values into a dense layer. The Mu and Sigma layers will then follow this so my function can return these as the outputs of these encoder layers. We'll return Mu, Sigma, and the output shape of batch_2 to the caller.



As we've been looking at how to build a variational auto encoder, we saw that we needed to change our input and encoding layer to provide multiple outputs that we called sigma and mew. These will then be used with a Gaussian distribution to sample their latent encoding pseudo randomly. The math for this is a bit beyond the scope of this course. But if we dig into the code we can see how it all works. So let's check it out recall the architecture that we've been looking at for a variational autoencoder.

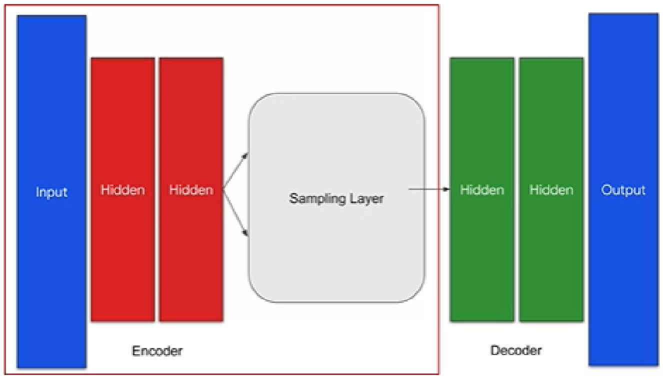
So next we want to get the output of the latent space which includes these values along with a gaussian distribution giving us a level of random noise and probability. Instead of just a learned feature from the encoder. As we want this to behave as part of the neuro network, we can actually create a custom layer to handle it.

And now we change our architecture to look a little bit like this. Where we create a new layer that we call a sampling layer to handle a construction of the encoding in the linked in space.

```
class Sampling(tf.keras.layers.Layer):
  def call(self, inputs):
    mu, sigma = inputs
    batch = tf.shape(mu)[0]
    dim = tf.shape(mu)[1]
    epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
    return mu + tf.exp(0.5 * sigma) * epsilon
```

So let's look at the code. And here's the code for the custom layer that can handle that for us. We'll pass in mu and sigma two as its inputs. These both had the shape of the batch and the dimensions of the batch in them at index zero and one respectively. So we can get the shape using either a mew or sigma. But at those index, these are needed to get the normal distribution across that batch size for that dimension.

And you can see that here as the random normal function in Keras generates the random normal Gaussian distribution for us in the desired shape, which we'll call epsilon. And one of those many ways that we could combine them when that is proven effective is to multiply our epsilon value by the exponent of half sigma and then add mew.



So that code gives us this. We saw the code earlier for the layers in the encoder. So let's put them together to get this part of the architecture, giving us our full encoder.

```
def encoder_model(LATENT_DIM, input_shape):
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu, sigma, conv_shape = encoder_layers(inputs, latent_dim=LATENT_DIM)
  z = Sampling()((mu, sigma))
  model = tf.keras.Model(inputs, outputs=[mu, sigma, z])
  return model, conv_shape
```

So here's the code to create the encoder model from what we've already seen. We'll call it with a latent dimension and an input shape. The dimension in this case will be two as we're working with the monochrome characters. The input shape will be 28 by 28. Our inputs will simply be an input layer of the desired shape in this case 28 by 28, we'll call the function we created earlier to create our encoder layers. And returned mu sigma and our convolution shape from the second layer. You'll see how that's used a little later. But the mu and sigma are passed to our custom layer called sampling. And this will give us a z. With the functional API to get a model, you define the inputs and the outputs. So we'll create a new model giving it the inputs. There are outputs are a set of new sigma and Z. You might expect it to be just z at this point, because the decoder doesn't need mu and sigma.

But one thing we'll look at shortly is a reconstruction loss function, which does need these values.
So we'll output them to make them easy for us to get later. The decoder will also need our convolutional shape for reconstruction. So we can return it from this function. So now we've created this part of the architecture. The next step we need to create the decoder

```python
def decoder_layers(inputs, conv_shape):
  units = conv_shape[1] * conv_shape[2] * conv_shape[3]
  x = tf.keras.layers.Dense(units, activation = 'relu',
                            name="decode_dense1")(inputs)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Reshape((conv_shape[1], conv_shape[2], conv_shape[3]),
                               name="decode_reshape")(x)
  x = tf.keras.layers.Conv2DTranspose(filters=64, kernel_size=3, strides=2,
                                      padding='same', activation='relu',
                                      name="decode_conv2d_2")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2DTranspose(filters=32, kernel_size=3, strides=2,
                                      padding='same', activation='relu',
                                      name="decode_conv2d3")(x)
  x = tf.keras.layers.BatchNormalization()(x)

  x = tf.keras.layers.Conv2DTranspose(filters=1, kernel_size=3, strides=1, padding='same',
                                      activation='sigmoid', name="decode_final")(x)

  return x
```

Okay, so for the variational autoencoder, you've now explored the architecture and coded up the encoder and the latent representation of the data that included sampling from a Gaussian distribution. Next, you'll need to code up the decoder before finally training the network to be able to generate new data based on the training set. Let's get started. You've already seen how to code all of this. Let's now look at how to reconstruct data from the latent space and effectively generate new data.

That's the job of the decoder and you can see the decoder here. The conv_shape is what we save from the encoding layers. For that architecture and for m-nest, this will be 7 by 7 by 64. Namely that we'll have 64, 7 by 7 images output from that layer. We'll start the decoding process by loading a dense layer with that number of neurons, 7 by 7 by 64, and we'll then batch normalize it.

The output of that will be reshaped into the 7 by 7 by 64 filters, reversing the flatten from the encoder. Then it will be fed into two Conv2DTranspose layers, effectively inverting the convolutional filters. These will have the same number of filters as their corresponding encoding ones. In this case 64 and 32 respectively. Then as with any other autoencoder,a single filter Conv2DTranspose layer can give us our output in the same shape as our original inputs, namely a 28 by 28 by 1 image.

```python
def decoder_model(latent_dim, conv_shape):
  inputs = tf.keras.layers.Input(shape=(latent_dim,))
  outputs = decoder_layers(inputs, conv_shape)
  model = tf.keras.Model(inputs, outputs)
  return model
```

Putting together a decoder model is actually quite straightforward. We'll create a function that takes the latent space dimensions and the convolutional shapes from the encoder as its inputs.

We'll use these to create an input layer called the inputs, which we can then use along with the convolution shape to call the decoder layers function that creates the decoder layers that we saw just a moment ago. Now we have inputs and outputs that we can use to construct a tf.keras Model.

We've now created both of these. Our final task is to put it all together. For a variational autoencoder, there's an additional step that you'll need, which is creating a reconstruction loss to ensure that our random normal data is good for reconstructing images and we'll see that next.

Loss Function and Model Definition

```python
# Define a kl reconstruction loss function
def kl_reconstruction_loss(inputs, outputs, mu, sigma):
  kl_loss = 1 + sigma - tf.square(mu) - tf.math.exp(sigma)
  return tf.reduce_mean(kl_loss) * -0.5
```

So you've now put together a model for an encoder that includes variational latent space encoding, as well as a decoder that can put data from that space together into an output that should match the type of things learn from the training set. Next up, you'll put these together into the final variational auto encoder model. But first you're going to have to consider your loss function.

So recall our architecture looks a little bit like this. We have a latent space in the middle, that's more complex than the dimensional reduction that we've seen in previous lessons. When it comes to loss, our loss functions are really good at having the network. Learn how to reconstruct data that comes directly from the encoder like this.

But the encodings in our latent space are much more complex, taking into account a random normal distribution, and having this act on what the encoder learns. For this, we're going to need an additional loss function.

And we'll use a callback kleibler cost function here. And this is one that's commonly used with variational auto encoders. Thankfully, it's also really simple to write and the code is here. I won't go into the details here, but note that this is a much more complex loss function than you might be used to, as it takes in different parameters mu and sigma, instead of the usual predicted value and target value.

```python
def vae_model(encoder, decoder, input_shape):
  inputs = tf.keras.layers.Input(shape=input_shape)
  mu = encoder(inputs)[0]
  sigma = encoder(inputs)[1]
  z = encoder(inputs)[2]
  reconstructed = decoder(z)
  model = tf.keras.Model(inputs=inputs, outputs=reconstructed)
  loss = kl_reconstruction_loss(inputs, z, mu, sigma)
  model.add_loss(loss)
  return model
```

Using these we'll calculate our loss value. You can learn more about this loss function at it's Wikipedia page, and it's impact on variational order encoders, it's discussed in the archive paper that we share here also. Finally, we can put it all together to create a vae_model. We'll start with defining our inputs, they're simply an input layer with our input shape.

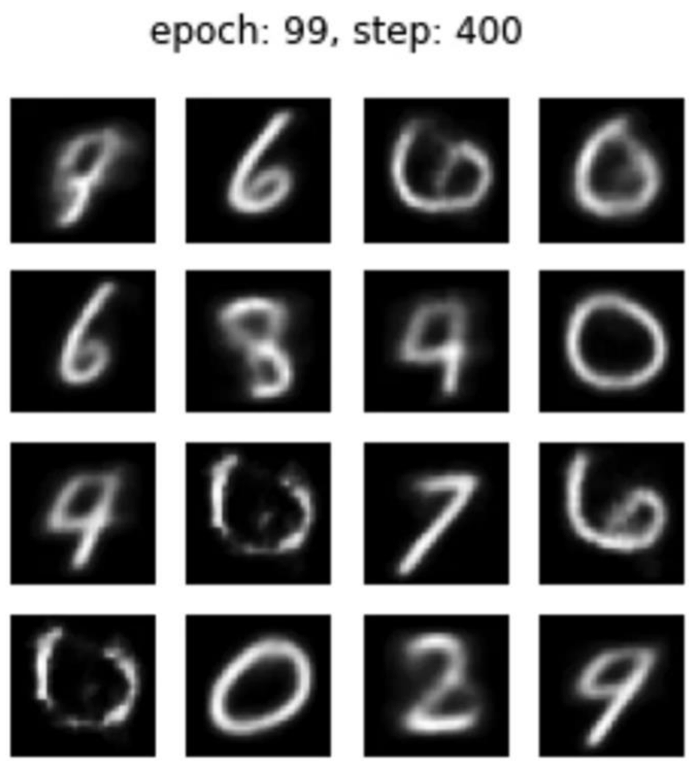
We can then call our encoder model with the inputs, and we'll get back mu sigma and z. Now remember z is the output of the latent space by combining the random Gaussian,the mew and the sigma values. To get our reconstruction will decode z. And this will give us our output from the final model. And now that we have our inputs and our outputs, we can create a new model.

Our last function is a bit more exotic than the standard ones. We're not taking the typical white bread and white label values. And instead we'll work off the z, mu and sigma. So we'll instantiate it and then use the models objects added last method, to add it as a last that the model will recognize. So you've now looked at how to code the encoder, decoder and latent space for a variation auto encoder. All that's left to do is to train the model. You'll see that next.

```python
for epoch in range(epochs):
  for step, x_batch_train in enumerate(train_dataset):
    with tf.GradientTape() as tape:
      reconstructed = vae(x_batch_train)
      flattened_inputs = tf.reshape(x_batch_train, shape=[-1])
      flattened_outputs = tf.reshape(reconstructed, shape=[-1])
      loss = bce_loss(flattened_inputs, flattened_outputs) * 784
      loss += sum(vae.losses)  # Add KLD regularization loss

    grads = tape.gradient(loss, vae.trainable_weights)
    optimizer.apply_gradients(zip(grads, vae.trainable_weights))
```

Here's the code for the training loop. Let's take a look at it in a bit more detail. Our master model was called vae, so we'll pass our batch of training data to that and get reconstructed values back. We want to measure the loss on these, and we'll use that to optimize the next training epoch. To measure the loss, we need to reshape the inputs and outputs into the format that are expected.

Then once we have them, we'll pass them to bce_loss, which is simply a binary cross-entropy loss function. Remember, we're not categorizing them with 10 labels as if we were building a classifier.
It's binary with values closer to one are good and expected, and values closer to zero are not what's expected. The star 784 is because MNIST is 28 by 28, or 784 overall. Earlier we created the KL regularization loss and added it to the model. We can use that here by adding it to our loss. We now have the total loss for the predictions. Once we have the loss, we can get the gradient for them against the trainable weights, and the optimizer will use its apply gradients function to update the trainable weights within the vae.

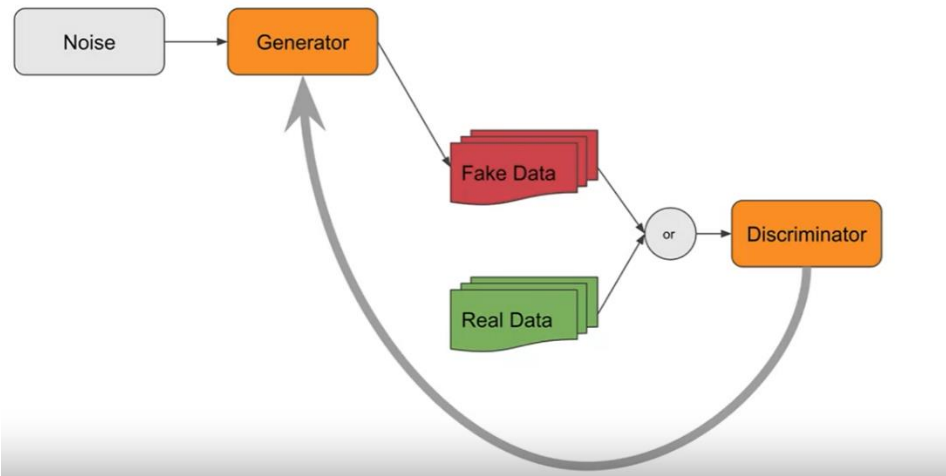


Here is the output from the code that you've just seen when trained on MNIST for 100 epochs in about half an hour. All of these visuals have been generated from scratch. It's not perfect, but it's definitely headed in the right direction. Now that you've explored vaes, it's time to give it a try yourself. Next up, you'll see a Colab where you can run the code that you saw here to generate an MNIST VAE.

References: Kullback–Leibler divergence, Balancing reconstruction error and Kullback-Leibler divergence in Variational Autoencoders

- Kullback–Leibler divergence
- Balancing reconstruction error and Kullback-Leibler divergence in Variational Autoencoders (Asperti & Trentin, 2020)

GANs



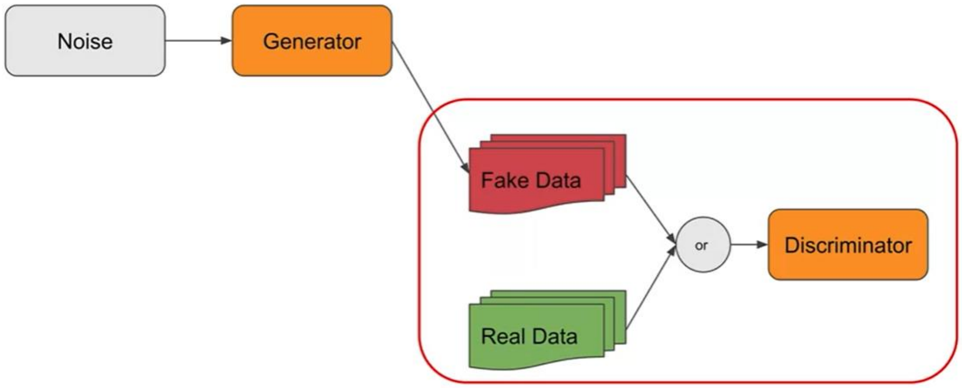
Welcome to week four of this course on generative machine learning. You've come a long way. And this week we're going to look at the methodology for generating content that probably gets the most attention, that's called Generative Adversarial Networks or GANs for short.

GaNS is a huge an interesting field of ml research and I'm really only going to touch the surface.
But I find them particularly exciting and I hope you will say. There's also a great specialization from deep learning AI that goes much deeper into generative adversarial networks than I will and it's available on Coursera here Generative adversarial networks are first introduced by Ian Goodfellow, john pouget.
Abadie McNamara's things you David Ward Farley, shares yellows there Aaron Carville and yoshua bengio in their paper from 2014 at this URL, you can see some of the images from it here.

So what is a generative adversarial network? Well, the idea behind it is quite simple. It's similar to what you saw with variational auto encoders last week, where you have random noise that's being fed into a neural network that's called a generator. This works a little bit like the VAEs where it has learned from a random distribution.

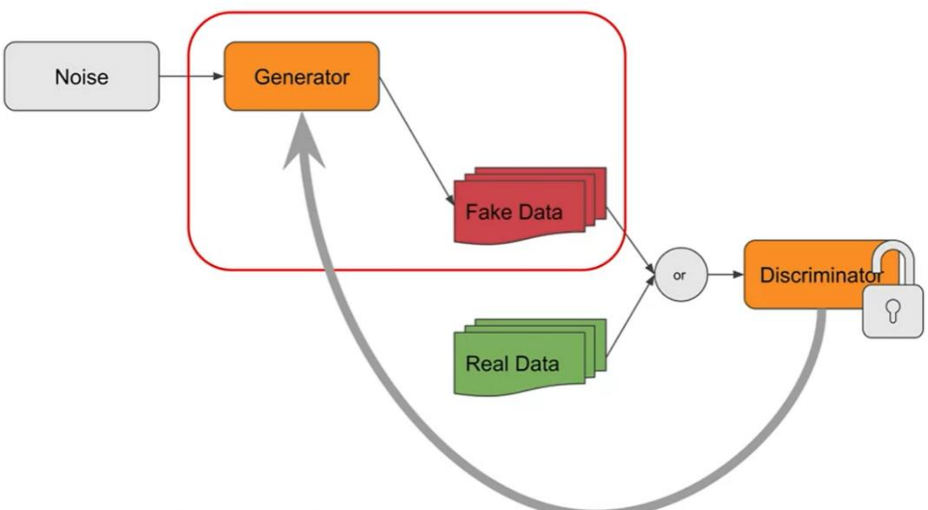
How to create g/facsimiles of the data that you'll call fake data. This data is then combined with real data, so that one or the other is fed into another neural network. And this is called the discriminator. Its job, as its name suggests, is to determine if the input image that it's given is fake or real. So in training, you have these two networks with opposite goals. The generators job is to create fake data that's real enough to fool the discriminator.

While the discriminators job is to learn how to be effective at telling real from fake data, and that's the adversarial part of the architecture. What's particularly interesting about this network It's that the generator never sees real images, it only ever sees the noise. It will learn instead from gradients that flow back from the discriminator. And of course the better the discriminator gets, the batter the knowledge flowing back to the generator, so the better it gets in creating fake data which in turn will make it better at generating fake data So over time, each network can make the other stronger and better. A key difference with GANS is in how the training loop takes place.



Training: Phase 1

Because there are two neural networks in the architecture training will also take two phases. In the first phase, you'll train the discriminator. It's given batches of labelled images which are labeled as fake or real. As this is a binary classification using an appropriate loss such as binary cross entropy is good. During this phase, only the parameters within the discriminator are updated.



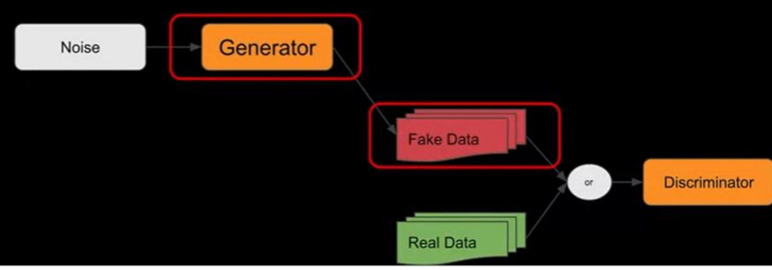
Training: Phase 2

In the second phase, you'll train the generator by getting it to produce a batch of fake images. But you'll also try to trick the discriminator into labeling those as real images. Despite the fact that you already know that the fake, the parameters of the discriminator will be frozen during this step, but the results of its judgments about whether the data is fake or real will be passed back to the generator, so it can optimize to produce more convincing fake data. Okay, so now that you've seen how again works in principle, let's get to coding, and I'll show you how to create a simple one that works using ms digits.

First GANs Architecture

```
generator = keras.models.Sequential([
    keras.layers.Dense(64, activation="selu",
                       input_shape=[random_normal_dimensions]),
    keras.layers.Dense(128, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
```

https://arxiv.org/abs/1706.02515



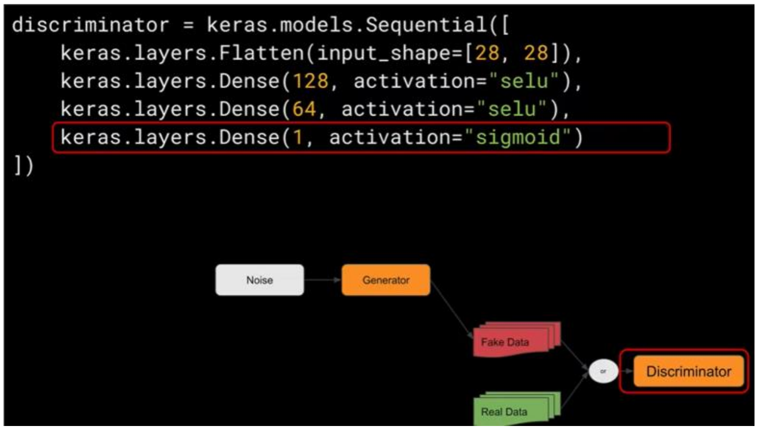
Previously, you got an introduction to how GANs work from a high level with two neural networks. Once you a create fake data called a generator and the other to determine how good the fakes are by comparing them with real data called a discriminator.

By pitting these networks against each other, the theory is that the generator will get better at its job. You can have a neural network that effectively creates new data from scratch. The architecture is here. The two neural networks, the generator, and discriminator are highlighted.
In this video, we'll create your very first, very simple GAN using MNIST data and this architecture.
Let's start by looking at the discriminator. The job of this network is to take input noise and use it to generate 28 by 28 MNIST digits.

It's fed by noise random_normal, so our input will be in the dimensionality of that noise. You'll see later how this is created. But for now, the important thing is its shape and this provides our input layer. This will be fed through a dense layer for learning weights before a final dense layer,
that is the shape of the image that we want to generate, and that's namely 28 by 28.

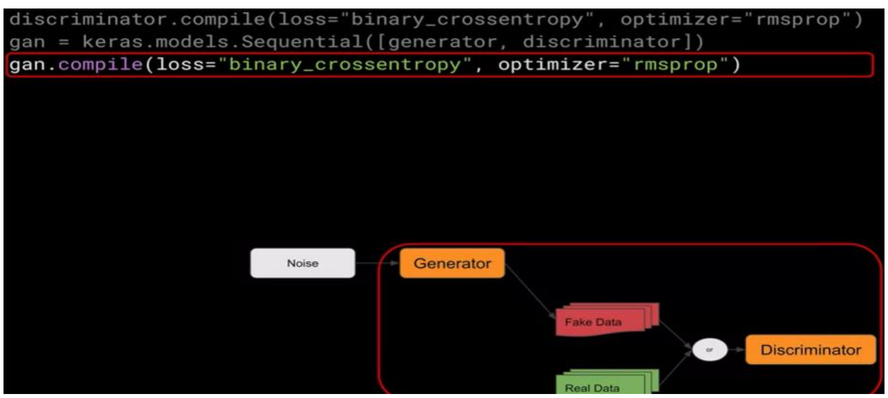
Note also that the activation on the 28 by 28 layer is a sigmoid. We'll have neurons that contain a value between zero and one. This is exactly the pixel representation in MNIST with zero being black and one being white, and the values in between being various shades of gray. You'll then reshape this. It's 28 by 28, which is the size of the MNIST images. We can create our fake data in the correct shape.

You may have noticed that the activation function on the dense layers was SELU and not ReLU. That's a key difference in these networks. ReLU is useful in removing noise when classifying data by preventing negative values from canceling out positive ones.
But here, you don't want to remove data, and the SELU function is very useful for that. Explaining how it works is a little bit beyond the scope of this course.But if you want to learn more, this paper on self-normalizing neural networks is really good.



Let's now look at the discriminator network. Here's the architecture. Because it's input is MNIST 28 by 28 or fake MNIST 28 by 28, the input shape will be set to that and it will be flattened before it goes into a dense network. The labels will not be the classes of the images, but the label zero for fake and one for real.

You'll then have two dense layers, each similar to the ones in the generator with 128 and 64 neurons respectively, and these will be activated by SELU. You'll have the classification layer activated by sigmoid for the labels. Recalling, we're not classifying 10 MNIST digits, but whether the incoming data is fake or real. Zero, one will be fine for this, and we only need one neuron activated by sigmoid to be able to handle that.



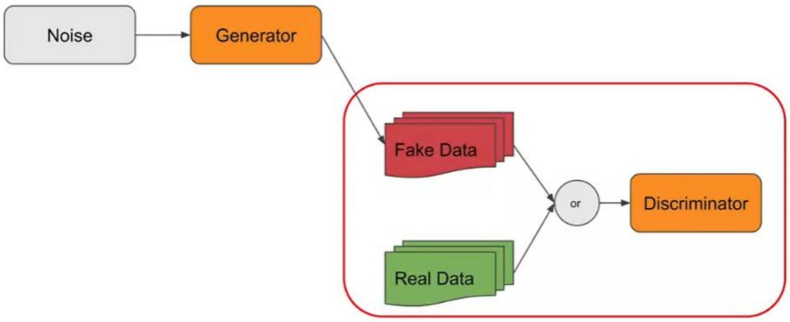
- [Self-Normalizing Neural Networks](#) (Klambauer, Unterthiner, Mayr & Hochreiter, 2017)

Now to create your overall model, you'll need a couple of steps, and here's the code. The first thing you need to do is to compile the discriminator. As it has a binary input, fake or real data, you'll use a binary loss function. It's binary cross-entropy. RMSprop is a reasonable optimizer, but you could experiment with this if you want.

When using the models API, you define a model using its inputs and its outputs. You can create your GAN model with the generator asits inputs and the discriminator as its outputs. This model can then be combined with the same loss function and optimizer. That was pretty straightforward.
What you will look at next is the training loop for this GAN. It's a little different than anything you might have seen before because it has that two-phase training approach that we mentioned earlier.

You'll also alternate between making the discriminator trainable or not. You'll see all of that code in the next video.

First GAN Training Loop

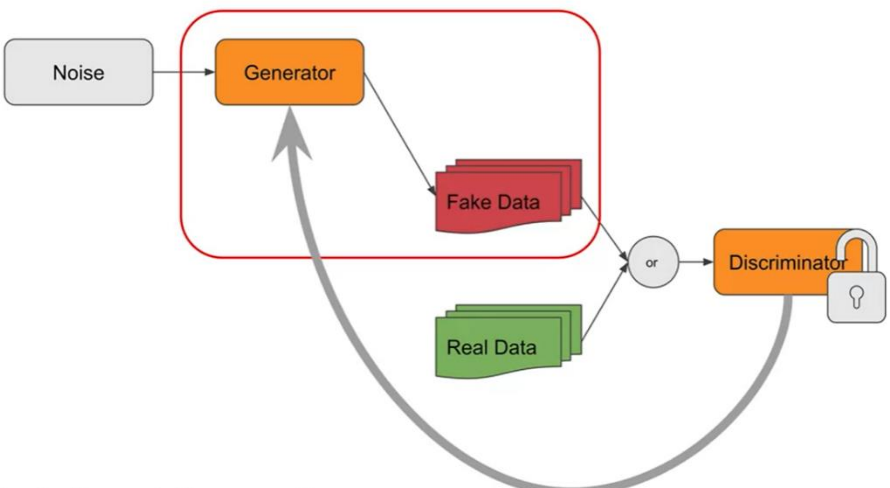


Training: Phase 1

So far you've looked at GANs by creating a simple GAN that generates MNIST images. You saw the architecture and then you looked at the model architecture code for the generator and the discriminator.

Training again is a little different than a normal network as it operates in two distinct phases. This requires creating a custom training loop, and we'll explore that in this video. Recall the architecture of the GAN that looked a little like this. It has two trainable models: the generator and the discriminator.

When it comes to training, you'll do it in two phases. The first is where you train the discriminator by giving it labeled fake and real data, and it attempts to learn the difference between them.



Training: Phase 2

In the second phase, you'll train the generator by getting it to produce a batch of fake images. But you'll also try to trick the discriminator by labeling them as real images, despite the fact that you already know they're fake. The parameters of the discriminator will be frozen in this step. But the results of its judgments about whether the data is fake or real will be passed back to the generator, so they can optimize to produce more convincing fake data.

```
# Train the generator - PHASE 2
noise = tf.random.normal(shape=[batch_size, random_normal_dimensions])
generator_labels = tf.constant([[1.]] * batch_size)
discriminator.trainable = False
gan.train_on_batch(noise, generator_labels)
```

We'll now take a look at this training and starting with phase 1. Here's the code as part of a custom training loop. Your first task is to create a random normal that's the same size as our batch of real images and is in the custom dimension space. Random normal dimensions is a hyperparameter that you set up, and I defaulted it to 32. You'll then pass this to the generator to get the first set of fake images. They're not going to be very good because the generator hasn't been trained yet. Based on the random initialization of weights, you'll have a bad set of fakes.

You will then create a new set of data called mixed images. This will concat the fake images that we just generated with real images that we'll pull from the dataset. Each of these lists are the same size and they're determined by the batch size hyperparameter. The labels will be zeros for the fake data and ones for the real data. As such, you can create a new array of batch size number of zeros followed by batch size number of ones.

You'll be training the discriminator in phase one, but freezing it in phase two. Let's right now ensure that it's trainable. Then you can call its train on batch method passing in the mixed images list and the discriminator labels.

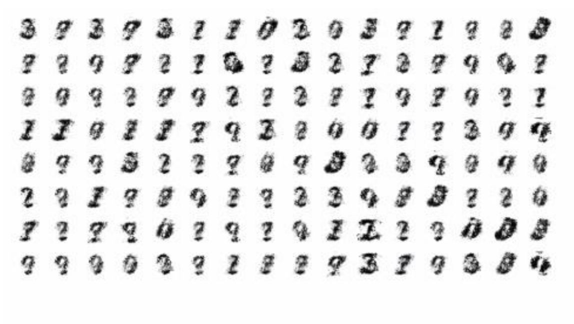
```
# Train the generator - PHASE 2
noise = tf.random.normal(shape=[batch_size, random_normal_dimensions])
generator_labels = tf.constant([[1.]] * batch_size)
discriminator.trainable = False
gan.train_on_batch(noise, generator_labels)
```

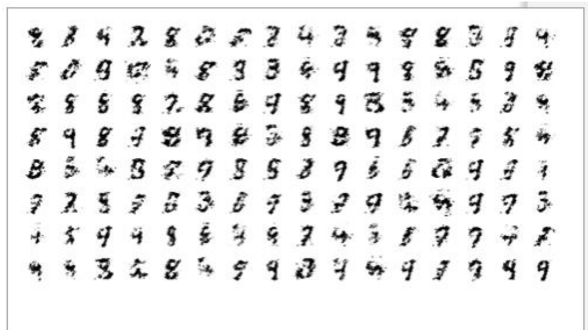
Phase 2 then means that you'll train the generator, so here's the code for that. First, you'll generate noise as before it using tf.random.normal. You'll then want to try to fool the discriminator, so despite all of those images being fake, you'll create the labels as all ones indicating that you're claiming that there real images and not fakes.

You can then freeze the discriminator. You just want it to flow feedback back to the generator with its predictions for these labels, and not update its weights based on this data.

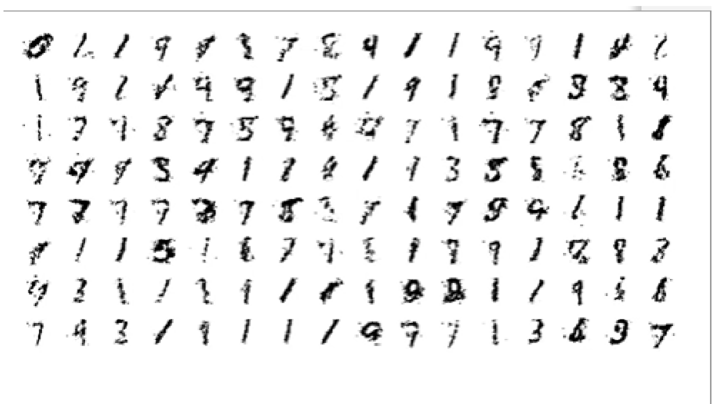
Otherwise, the weights could get corrupted by the fact that you're lying about the labels. Then you'll train the GAN using the noise and the generator labels to get it to update the generator. Remember that the discriminator is frozen at this point.

Here's some of the results. After just one epoch, the generator created digits that looked like this. You're beginning to see some numbers emerge, but they could be twos, sevens or nines in particular. After five epoch, some other shapes like three or four start to emerge.

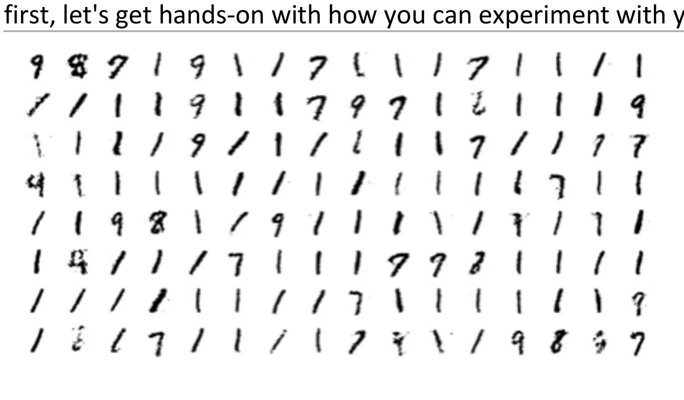




After 20 epochs, you're beginning to see some very clear values, and particularly the ones.
Now there's a lot of ones here. There's a distinctive zero, there's some nines, there's some threes and there's some fives.



After reaching a 100 epochs, you'll see that the GAN is now optimizing primarily for generating ones, sevens and nines. It is generating data, but there's a clear bias towards some numbers, and this is called **mode collapse**. The generator is getting better at creating shapes that fool the discriminator, and as a result, over time, it becomes biased towards creating those particular ones. As such, simple shapes like one and seven in particular, can be generated in a way that they fool the discriminator. Fixing this problem is a huge area of research in GANs. For the rest of this week we'll look at some architectures that have evolved to fix it. But first, let's get hands-on with how you can experiment with your first GANs, creating MNIST digits.



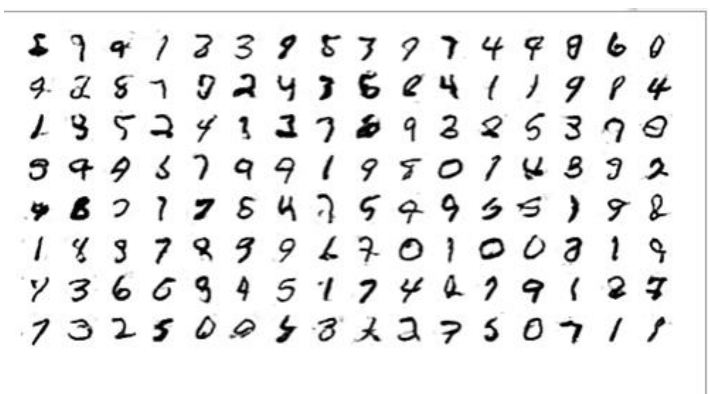
DCGANs (Deep GANs)

- [Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks](#) (Radford, Metz & Chintala, 2016)

- [tf.keras.layers.LeakyReLU](#)

You've just seen GANs and how they can be used to create new content using a generator and a discriminator. You saw how they worked, but they did have their limitations, which you saw here when the GANs started over specializing to create ones and sevens and maybe a few nines. This happened because of the whole adversarial relationship where both the generator and
the discriminator wanted to win.

If the generator gets better at producing convincing digits, it will fool the discriminator with those;
and the discriminator in turn, seeing more and more of the ones that was fooled by, will optimize towards them.



Different architectural approach might be necessary, and the logical next step is to look at Convolutional Neural Networks for the generator and the discriminator. Let's explore those.
These are the results when using a deep convolutional network architecture within a GAN. As you can see, they're greatly improved and they don't end up skewing towards any particular digit.

Deep convolutional GANs wereproposed in the paper by Alec Radford,Luke Metz, and Soumith Chintala in the paper at this link. They demonstrate architectures and best practices that can be used to improve on GAN performance. I'll show you an example of those next.

```
generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),

    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2,
                                 padding="SAME", activation="selu"),
    keras.layers.BatchNormalization(),


    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="SAME",
                                 activation="tanh"),
])
```

For example, here's how the generator will work. Note that the research that led to DCGANs brought about a number of best practices. The first is that in the generator, instead of using pooling layers and conv2Ds, the conv2D transposed layers are used. Next, is that batch normalization should be used in the generator except for the output layer. All activation layer should be selu, except for the output layer, which should be tanh.

```
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="SAME",
                        activation=keras.layers.LeakyReLU(0.2),
                        input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),

    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="SAME",
                        activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),

    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
```

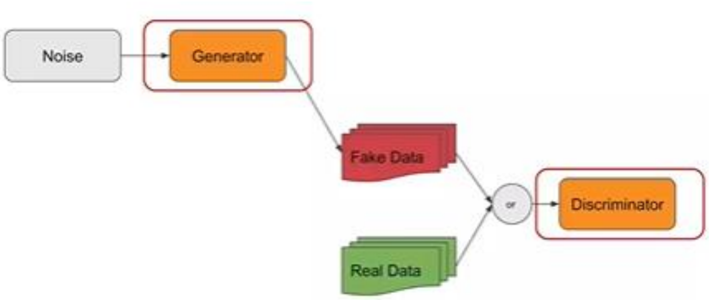
In the discriminator, we can classify the image using CNNs as usual, but again, there are some best practices that were presented in that paper. Instead of pooling after the convolutional layer, the use of strides can be used to reduce the dimensions. **LeakyReLu** should be used in the activation layers in the discriminator. If you're not familiar with LeakyReLu, you can learn more about it at the URL shown. It's similar to ReLu, except in that if the value is less than zero, it's not always set to zero. It's actually multiplied by a small value instead, so that some of the gradients are maintained.
One of the practice not shown here is that in addition to everything that we've discussed, you'll also avoid the use of dense layers in both the discriminator and the generator.

If you explore a data set like fashion MNIST, you can see that it becomes very good at creating facsimiles of the data in there, though not quite perfect. For example, you'll often see issues like this where the items like shirts, but have split up the middle, that looks like the legs of a pair of trousers will give us a very strange hybrid between the two; or handbags and shirts like here have somewhat been morphed together. Here you used GANs with very simple data sets, the MNIST and the fashion MNIST ones. Once you start getting into more advanced data sets, you'll see that the processing requirements will increase dramatically. In the next video, you're going to look at a notebook using GANs with real faces, and you'll see if you can build one that can create suit or real faces.

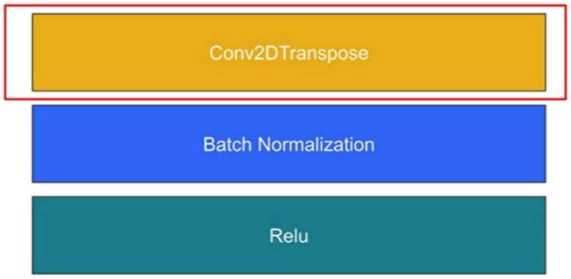
FACE Generator

In this video, you will explore a GAN that can be used to generate human faces, trained using the celebrity faces data set. The faces are quite low resolution so your generated ones will be too. Importantly though, you'll see that when dealing with GANs and more complex data like this, you'll require greatly increased processing power. The sample that you'll work through in this video can take a couple of hours to train using a TPA. These are the types of results that you should expect to see when running the code that I'm providing for face-generation. None of these faces are real. They were all generated by using a GAN, that was trained on the celebrity faces data-set and as you can see, some of the faces came out pretty well, but others are horribly distorted, and some, they may even look like Impressionist paintings.

So let's start with understanding the architecture for a generator that can be used in a GAN for images like these. One, the celebrity data-set has a variety of image sizes. We're going to pre-process the data to center the faces in a 64-by-64 picture. The results will look something like this. Data is being lost and sometimes it might be hair or ears. As a result, you could expect the generated images to be somewhat skewed. So bear this in mind when you create your own models. But for now, we'll work with these kind of 64-by-64 images.

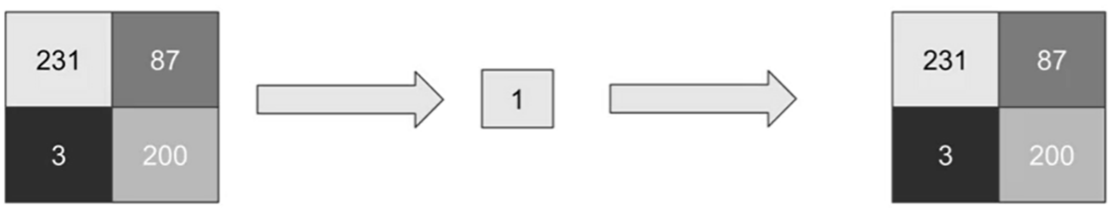


Recall that with a GAN, your generator takes any noisy data and uses this to create fake data. The fake data is then merged with real data for the discriminator to learn from and pass back intel that can be used to create better fake data in the future.



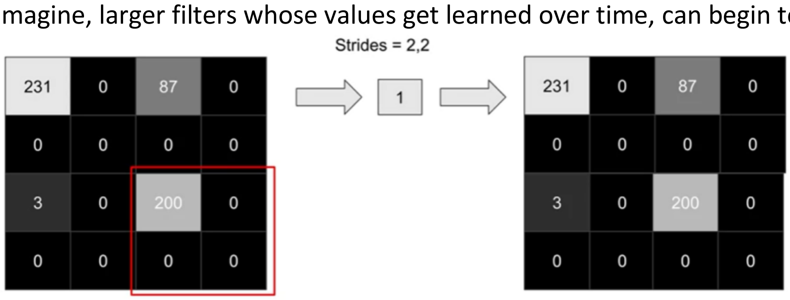
So to create a generator that can manage 64-by-64 images, you will start with our noise. It's a normal distribution which has the dimensions one-by-one by something and then have an architecture that scales that up to 64-by-64 images with three channels of depth because they're in color. So it'll be 64-by-64-by-3. A common architecture block for scaling this up might look like this. A Conv2DTtranspose followed by a batch normalization, followed by an activation function like Relu. Conv2DTranspose layers often called deconvolution layers are the opposite of convolutionlayers.

Instead of taking an image and applying filters to it to get a filtered image which can be smaller than the original. The transpose is designed to go in the opposite direction to effectively reconstruct from filters and upscale the image.

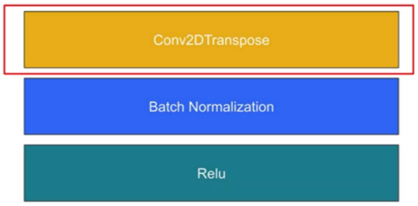


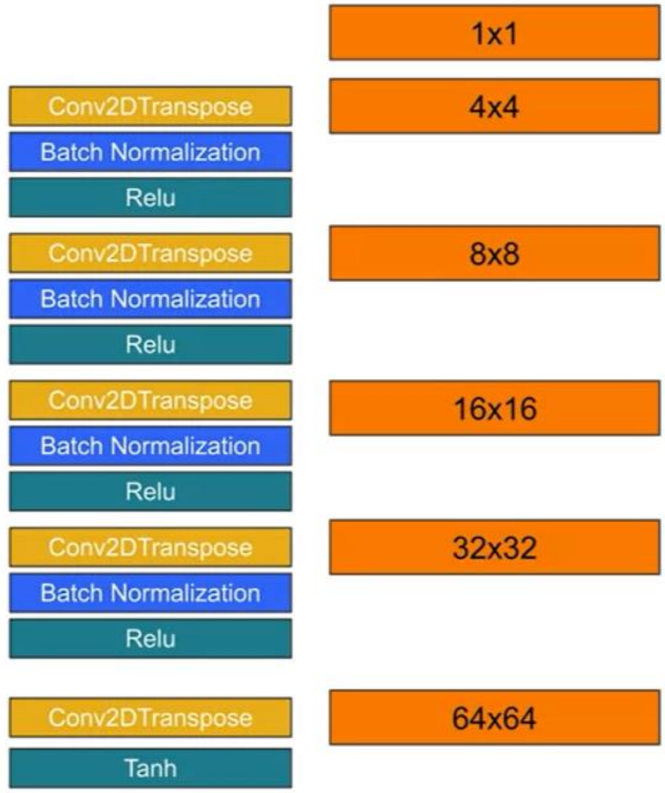
Consider this image. It's a simple two-by-two, one. If you apply a one-by-one filter to it and that filter was the value one and its stride was one. I.e, it step through the pixels one at a time. The output for the first pixel will be 231-by-one, which is 231 and similarly the other pixels will be the same and nothing would have changed in the image.

But what will happen if it does so in a two-by-two manner, it's not going to work because you have no image that you can jump across this way, the input image is too small. So a two by two stride will look like this and when this is multiplied out by the filter, you'll get the same value. The second stride will do the same as will the third and the fourth. As you can see when using D-convolutions with a stride length above one, you can up-sample the image into a higher resolution and here you have a very simple one-by-one filter that doesn't change the image. But as you can imagine, larger filters whose values get learned over time, can begin to construct new images.



Which brings me back to this block. You'll use those D-convolutions or Conv2DTranspose to perform the upsampling. Batch normalization, as its name suggests, is a methodology that let you normalize an input across its batches. The goal is to reduce peaks and troughs in the output data and then smooth that out. Of course Relu is an activation function which will remove negative values to prevent them from canceling out positive ones.



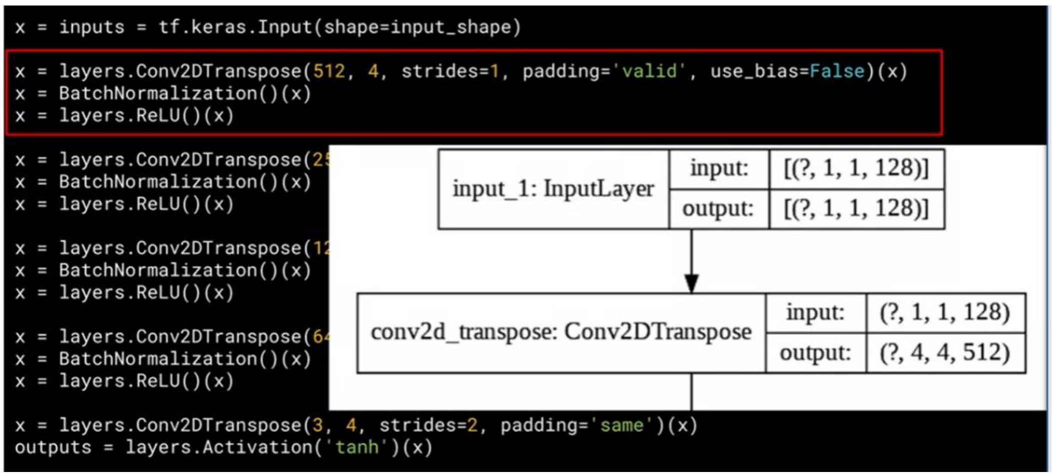


So now using a block of four of these and setting the Conv2DTranspose properly, you can get your normal distribution of one by one and then upscale to four-by-four using a four-by-four filter with a stride size of one and subsequently continue
to double the axes to get to eight-by-eight, 16-by-16, and 32-by-32, using four-by-four filters with a stride size of two.

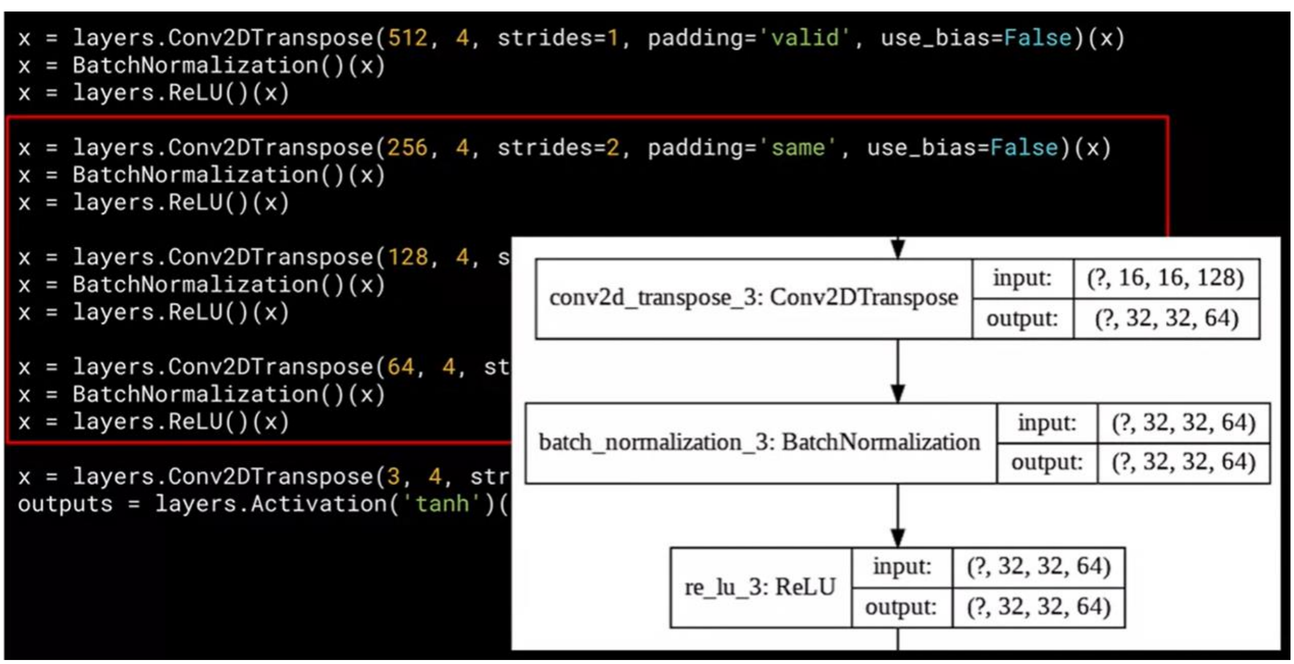
Before you'll come to one final Conv2DTranspose that you don't need to batch normalize because it's your output. According to the best practices we discussed earlier, you will activate that were **Tanh** and this gets us up to 64-by-64, which is the dimension of the training images in the data-set.

```
x = inputs = tf.keras.Input(shape=input_shape)

x = layers.Conv2DTranspose(512, 4, strides=1, padding='valid', use_bias=False)(x)
x = BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.Conv2DTranspose(256, 4, strides=2, padding='same', use_bias=False)(x)
x = BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.Conv2DTranspose(128, 4, strides=2, padding='same', use_bias=False)(x)
x = BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.Conv2DTranspose(64, 4, strides=2, padding='same', use_bias=False)(x)
x = BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.Conv2DTranspose(3, 4, strides=2, padding='same')(x)
outputs = layers.Activation('tanh')(x)
```

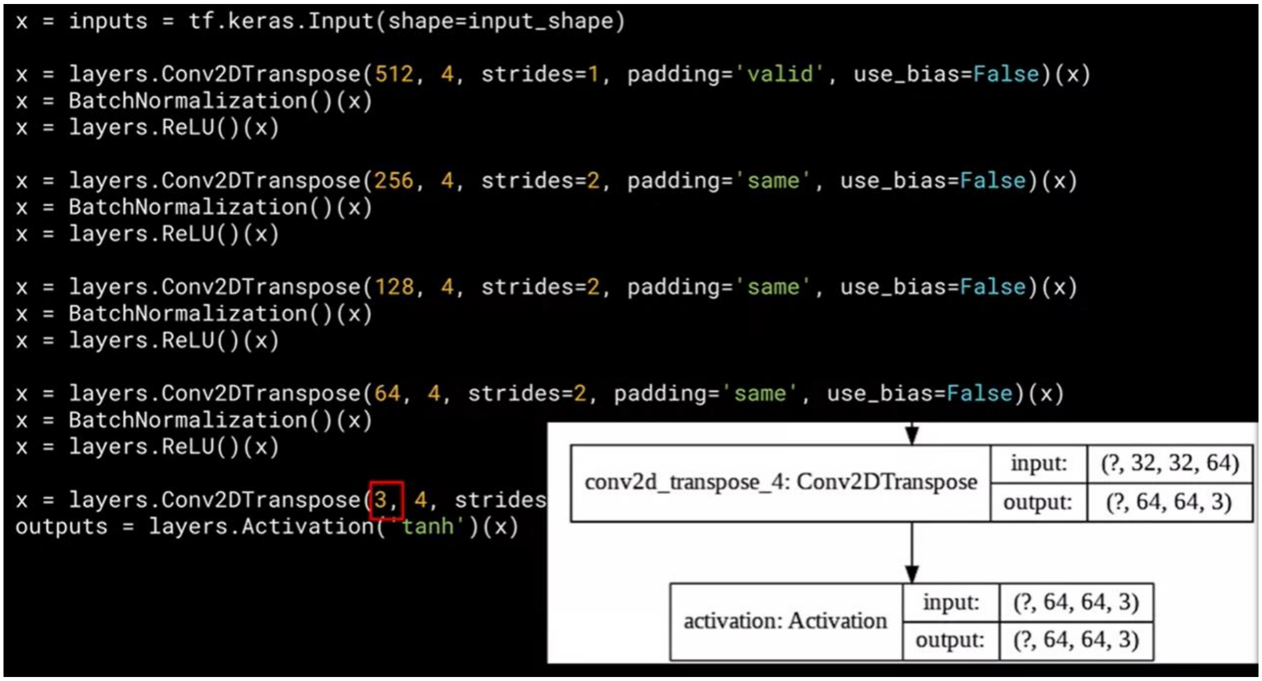
Let's now look at the code to achieve this. Your first block has 512 four-by-four filters with a stride size of one. This takes the input of one-by-one-by-128 from the normalizer and it gives you four-by-four-by-512 output, as you'll see in the Keras plot on the next slide. But before that note the use of use_bias. A convolution is a filter over the image, which can then be multiplied over the image with a bias added. When you're deconvolving, you don't necessarily have a bias though you could learn one if you wanted. But to avoid using one and just learn the transpositions that allow things to upscale, you can say use_bias equals false and here's the Keras plot overlaid and we can see how the dimensionality is changing through the use of the Conv2DTranspose. We're going from one-by-one by 128 to four-by-four by 512.



After that, you'll have three successive layers of four-by-four transposes, each with a **stride of two** and they will double the axes size three times. So you go from four-by-four to eight-by-eight to 16-by-16 to 32-by-32. I've overlaid the Keras plot for the output of the third block here, note that the image is now 32-by-32, and we have 64 filters.

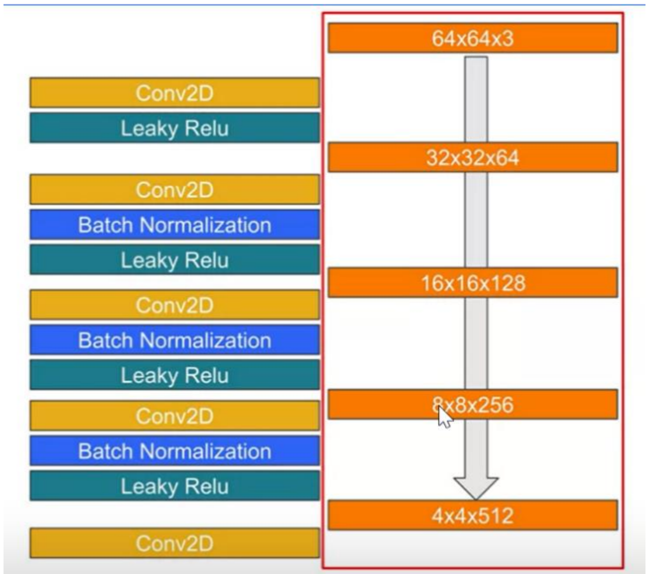


So by the time you get to the final layer, you'll have 32-by-32-by-64 dimensions, but you want your output to be 64-by-64-by-3, because your images 64-by-64 pixels and it's got three bytes of color depth.
Again, we can use a transpose with a four-by-four filter, and with two-by-two strides,
we can double the resolution to 64-by-64 and by specifying three filters we'll get into 64-by-64-by-3, which you can see in the final output here.
So you have now defined the generator for the face creator GAN. Next I'll show you that discriminator and you'll see that in the next video.



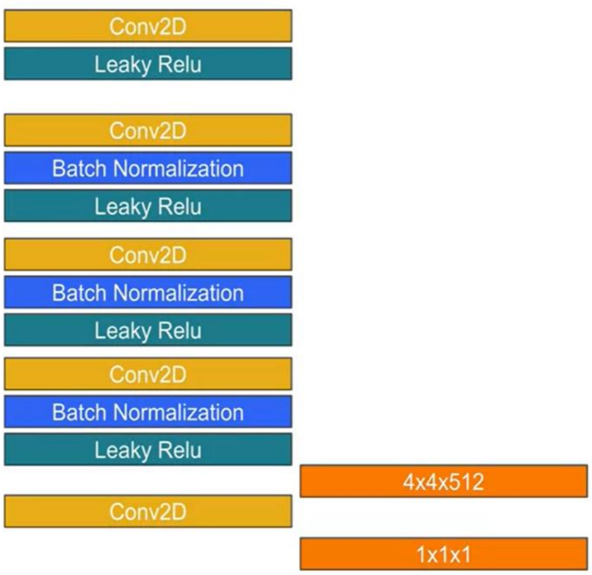
Face Generator Discriminator

Earlier, you learned about a more complex GAN, where you're generating realistic looking human faces off of a 64-by-64-by-3 color block. You looked into the architecture of the generator. Next up, let's explore the architecture of the discriminator.
Your task with the discriminator is to classify the images in much the same way as any other computer vision task would. You don't use pooling in this part of the guidelines, and instead you'll use strided convolutions, which you can normalize, and then activate with leaky ReLu. If you're not familiar with leaky ReLu, don't worry, it's very similar to ReLu, except that it isn't a flat 0 for values less than 0, and it does have a small gradient. This helps maintain some values when learning instead of zeroing them out, and is often credited for giving better generated images as a result. Given that the input is 64-by-64-by-3, and you want to do a classification of them, you'll need an architecture that contains blocks like these, that will get your size down to something that's easy to classify. Recall that based on the best practices, we want to do this without using dense layers.



Your discriminator architecture can look a little bit like this. Combined with an initial Conv2D layer whose job it is to reduce the dimensionality of the image from 64-by-64-by-3 to 32-by-32-by-something by applying filters and using strides.

In this case, we'll get it to 32-by-32-by-64. Subsequently, it will go through a dimension reduction with the strided convolutions. The number of filters will increase per layer so that it becomes 16 by-16-by-128, then 8-by-8-by-256, then 4-by-4-by-512, before going through a final Conv2D layer that has one filter and uses strides to reduce to a 1-by-1-by-1 that can be used to give you a classification.



Remember, the best practices for GANs don't allow us to use fully connected or dense layers, you'll be using convolutions in this case as a workaround.

Here's what you just saw in code, with the initial input being scaled down, by using a four-by-four filter with two strides here. Then three blocks of Conv2Ds each followed by batch normalization and activated by leaky ReLu, before a final output layer, which uses a single filter to collapse everything into a 1-by-1-by-1, which you can then use for classification. That's it. Next up, you can try the Colab for yourself and train a GAN that generates faces like those we saw earlier. There's a lot of code in it that we didn't cover here, including what you need to do distributed training across multiple TPUs for performance sake. As you learn more about GANs, there are some minor changes that can be done for different results, such as changing the normalization type from batch normalization to layer normalization too. There's a lot of code in there to try to generalize it as much as possible. Have a play with it and see if you can improve upon the face generation that I did.

```
x = inputs = tf.keras.Input(shape=input_shape)
x = layers.Conv2D(64, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU(alpha=0.2)(x)

x = layers.Conv2D(128, 4, strides=2, padding='same', use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.2)(x)

x = layers.Conv2D(256, 4, strides=2, padding='same', use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.2)(x)

x = layers.Conv2D(512, 4, strides=2, padding='same', use_bias=False)(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU(alpha=0.2)(x)

outputs = layers.Conv2D(1, 4, strides=1, padding='valid')(x)
```

- Layer Normalization (Ba, Kiros & Hinton, 2016)