

Generative Adversarial Nets(GANs)

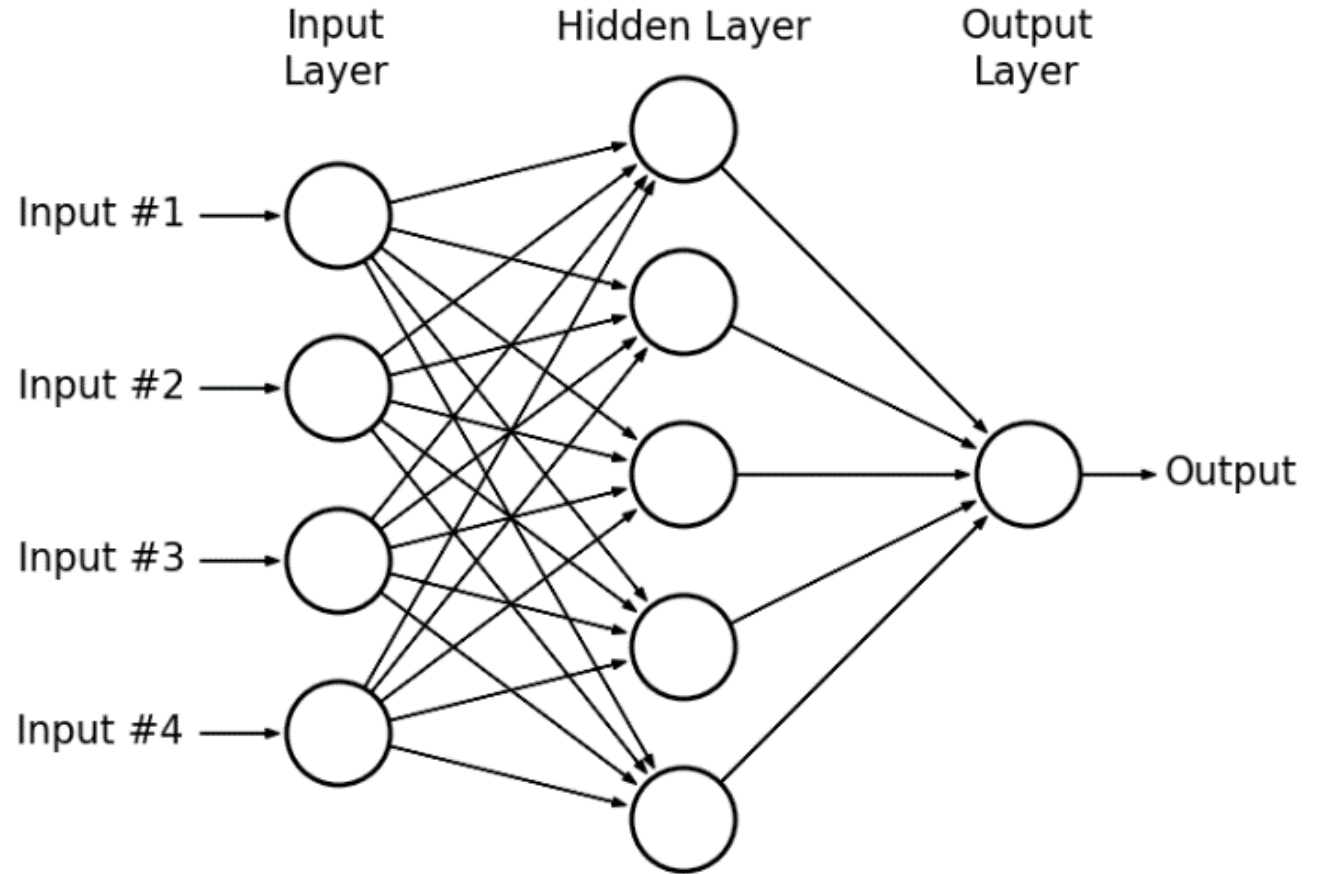
Troy Cary and Chenzhi Zhao

What is a GAN?

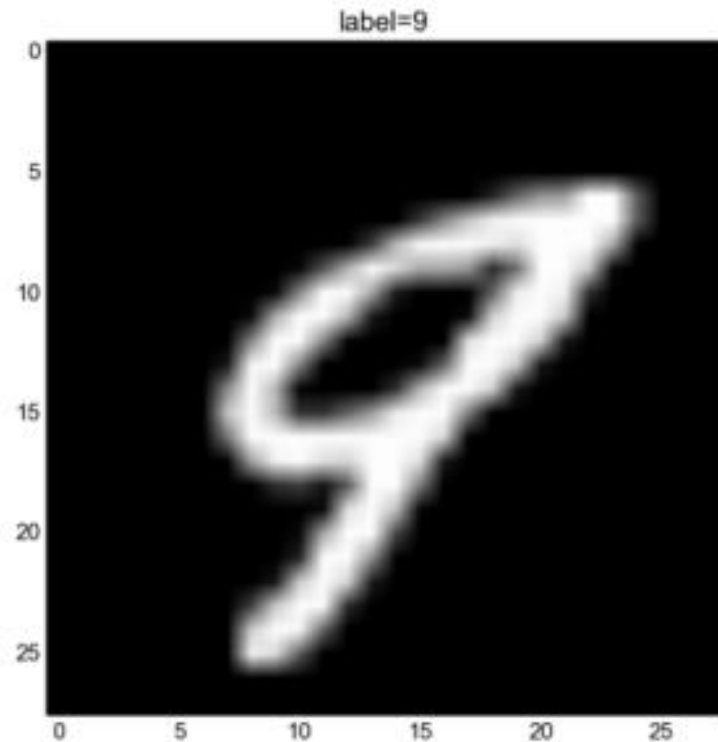
- A generative adversarial net is a type of neural net, used in deep learning/machine learning problems
- The goal of a GAN is to train two simultaneous models: a generative model G and a discriminative model D
- The generative model uses training data to create fake data points, and the discriminative model estimates the probability that the fake data points came from training data rather than G .

Neural Nets

- A neural net is a framework used to have machines “learn”, and is inspired by the brain.
- The inputs are neurons, and hold a value between 0 and 1
- Loosely analogous to biological networks of neurons “firing” causing others to “fire”
- Output layer is determined by specific problem

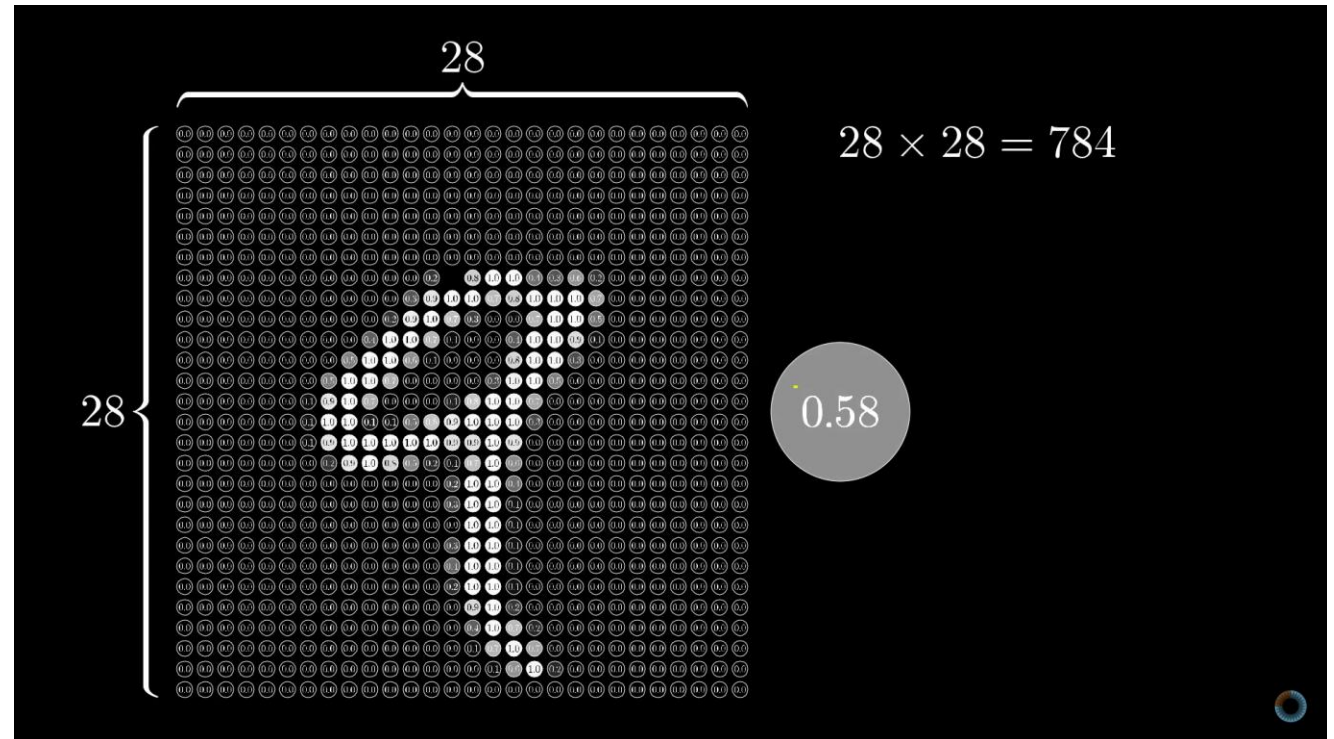


Example: MNIST Dataset



Input Layer

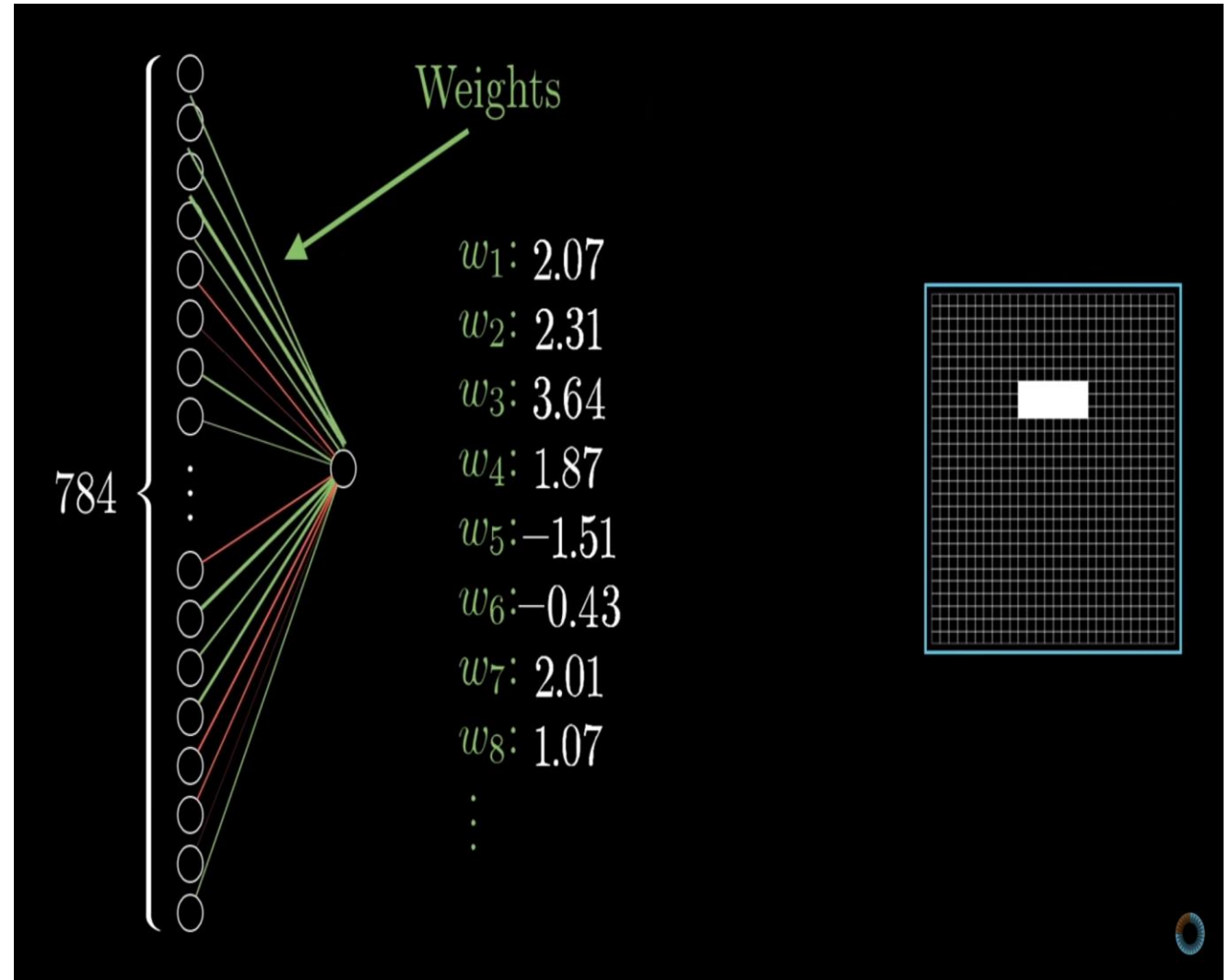
- In the MNIST Handwritten Digits dataset, each digit is represented by a pixel of grayscale values
- Each neuron in the input layer would correspond to a pixel with a value between 0 and 1.



Screenshots taken from 3Brown1Blue YouTube channel

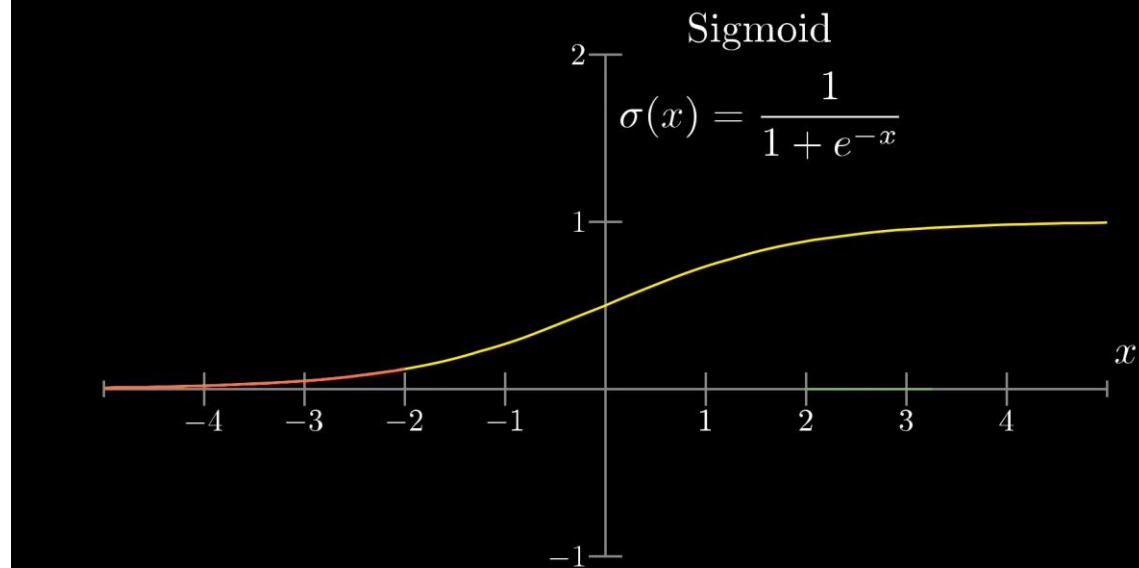
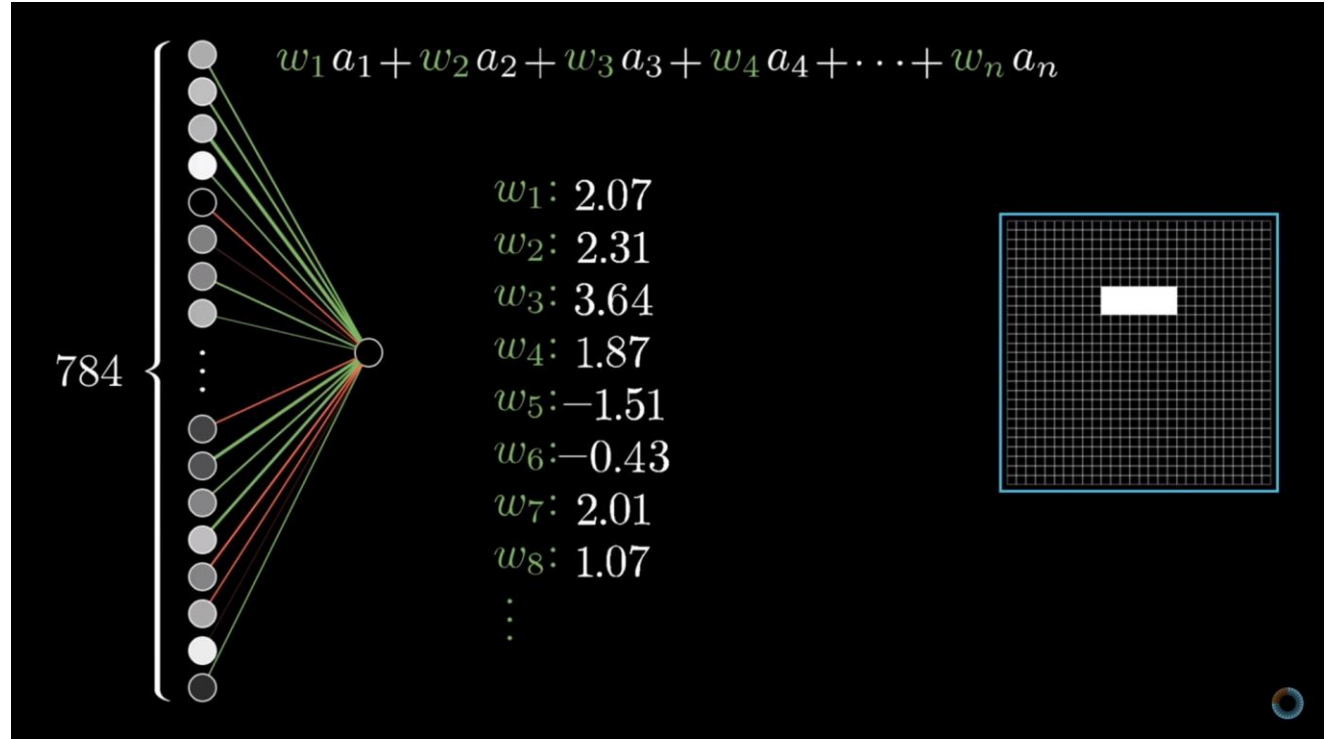
Hidden Layers

- Weights are assigned to each neuron, which correspond to the connection between the neurons and the next layer
- For this example, the weights are initialized randomly(explained later)



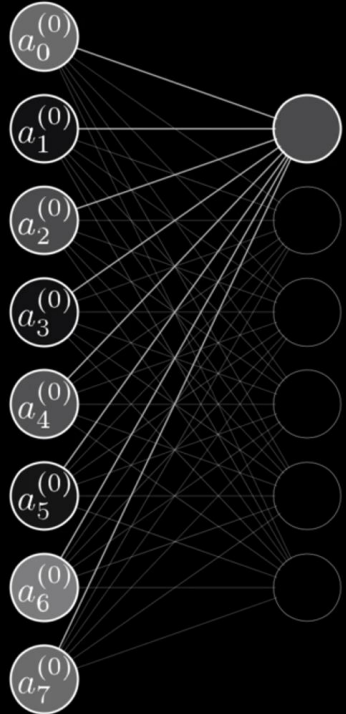
Hidden Layers cont.

- Then, the activation value from the input layer is used to compute a weighted sum of all the inputs
- This weighted sum is then put into a sigmoid function in order to get all outputs between 0 and 1
- In addition, a bias term may be added based on specific problem



Hidden Layers cont.

- This can be broken down into a matrix multiplication, and components of linear algebra can be used to solve this problem.


$$\sigma \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \right) = \begin{bmatrix} \sigma(x) \\ \sigma(y) \\ \sigma(z) \end{bmatrix}$$
$$\sigma \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \dots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right)$$

Hidden Layers cont.(last one I promise)

- Utilizing a loss function and stochastic gradient descent is essential to choosing the weights in these problems
- Since the weights were initially chosen randomly, we can define a cost function(average of the loss function) to measure how off our guesses were
- The cost function is large when the guesses are far off, so minimizing cost is the goal
- Stochastic gradient descent is then used to optimize this cost

Gradient descent, how neural networks learn | Deep learning, chapter 2

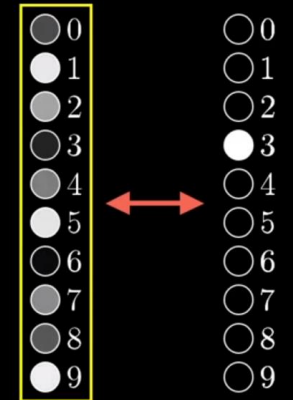
Average cost of
all training data...

Cost of

3

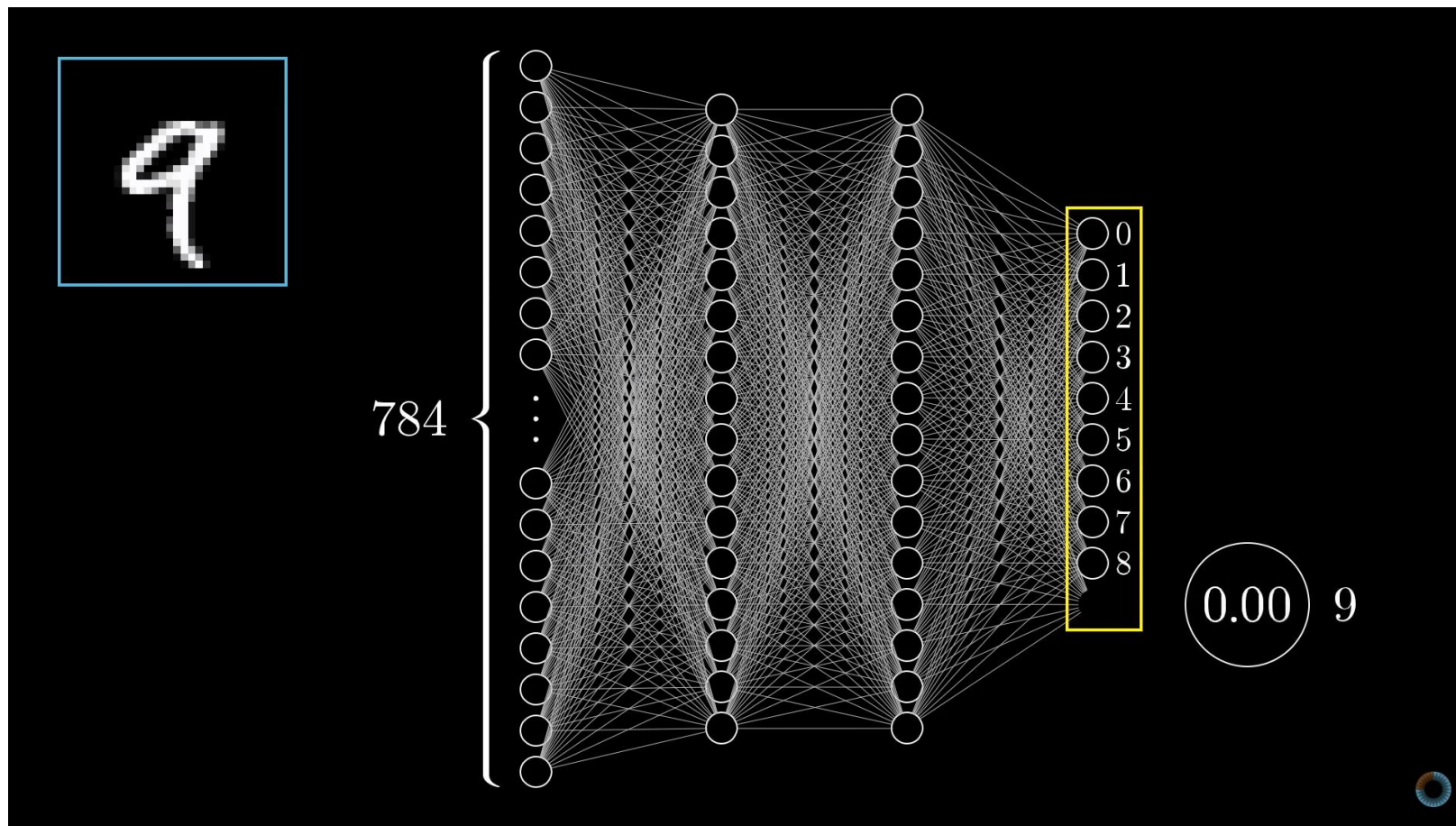
$$\left\{ \begin{array}{l} (0.29 - 0.00)^2 + \\ (0.91 - 0.00)^2 + \\ (0.65 - 0.00)^2 + \\ (0.14 - 1.00)^2 + \\ (0.50 - 0.00)^2 + \\ (0.91 - 0.00)^2 + \\ (0.05 - 0.00)^2 + \\ (0.55 - 0.00)^2 + \\ (0.35 - 0.00)^2 + \\ (0.94 - 0.00)^2 \end{array} \right.$$

What's the “cost”
of this difference?



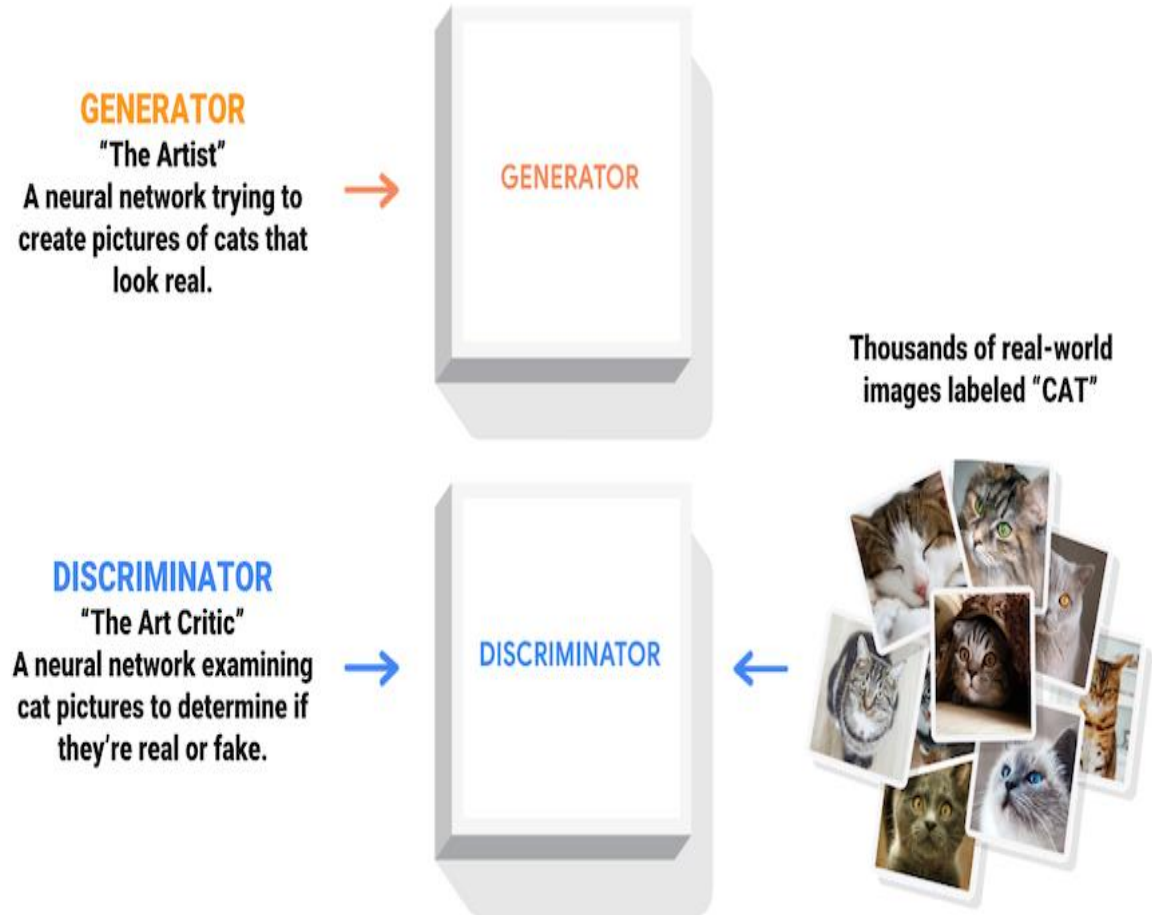
Utter trash

Output Layer



On to GANs

- The goal of a GAN is to train two simultaneous models: a generative model G and a discriminative model D
- The generative model uses training data to create fake data points, and the discriminative model estimates the probability that the fake data points came from training data rather than G.



GAN Paper

that x came from the data rather than p_g . We train D to maximize the probability of assigning the correct label to both training examples and samples from G . We simultaneously train G to minimize $\log(1 - D(G(z)))$. In other words, D and G play the following two-player minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]. \quad (1)$$

- For this project we read a paper called “Generative Adversarial Nets” by Ian J. Goodfellow and co. from the Universite de Montreal
- The paper contains proofs and theoretical results, however the purpose of this project was to focus more on the practical side
- The most important result of the paper is listed above, which determines the way the network trains

MNIST Fashion Dataset

- Similar to the MNIST Digit dataset, there is a fashion dataset which includes 60,000 examples of clothing, in a 28x28 grayscale image
- We utilized the Generative Adversarial Network to try and create our own fashion images



Code

Using python through google colab, here is the code for the generator and discriminator

▼ The Generator

The generator uses `tf.keras.layers.Conv2DTranspose` (upsampling) layers to produce an image from a seed (random noise). Start with a Dense layer that takes this seed as input, then upsample several times until you reach the desired image size of 28x28x1. Notice the `tf.keras.layers.LeakyReLU` activation for each layer, except the output layer which uses `tanh`.

```
[11] def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```


Code cont.

▼ The Generator

The generator uses `tf.keras.layers.Conv2DTranspose` (upsampling) layers to produce an image from a seed (random noise). Start with a Dense layer that takes this seed as input, then upsample several times until you reach the desired image size of 28x28x1. Notice the `tf.keras.layers.LeakyReLU` activation for each layer, except the output layer which uses `tanh`.

```
[11] def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

Results

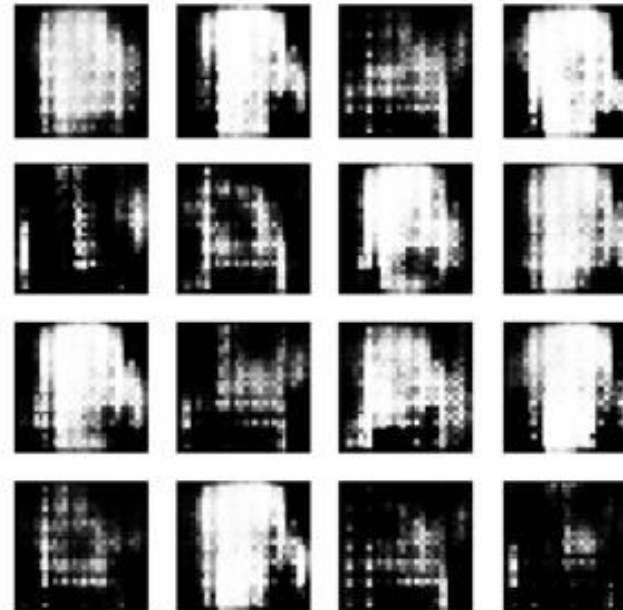
Over a period of 60 epochs, here
are some results:

Epoch 9



```
%%time  
train(train_dataset, EPOCHS)
```

...



Time for epoch 9 is 11.811452627182007 sec

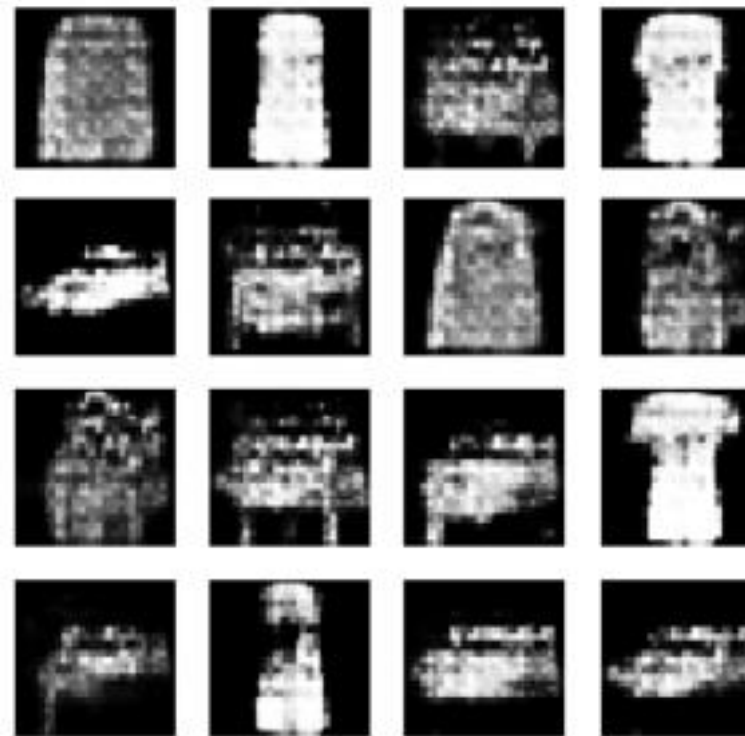
Results cont.

Epoch 33



```
%%time  
train(train_dataset, EPOCHS)
```

...



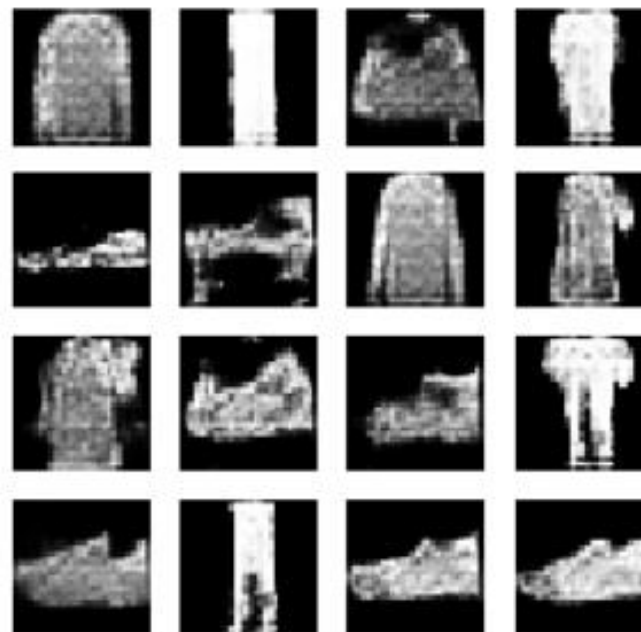
Time for epoch 33 is 12.254274606704712 sec

Results cont.

Epoch 60



```
%%time  
train(train_dataset, EPOCHS)
```



CPU times: user 3min 44s, sys: 59.3 s, total: 4min 43s
Wall time: 12min 14s