Generative Adversarial Networks

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Synthetic Data

- Synthetic data is data that is generated by a machine.
- For example:
  - A “real” face image is a photograph of someone’s face.
  - A “synthetic” face is an image made up by a computer program, that may have been designed to resemble someone, or may have been designed to not resemble anyone.
- Synthetic data can have many forms:
  - Synthetic text, for example a story or script or joke (or attempted joke) produced by a computer program.
  - Synthetic music.
  - Synthetic images and video.

Uses of Synthetic Data

- What are possible uses of synthetic data?
- Synthetic data is often used as training data, especially when "real" data is not as abundant as we would like.
- One example is hand pose estimation.

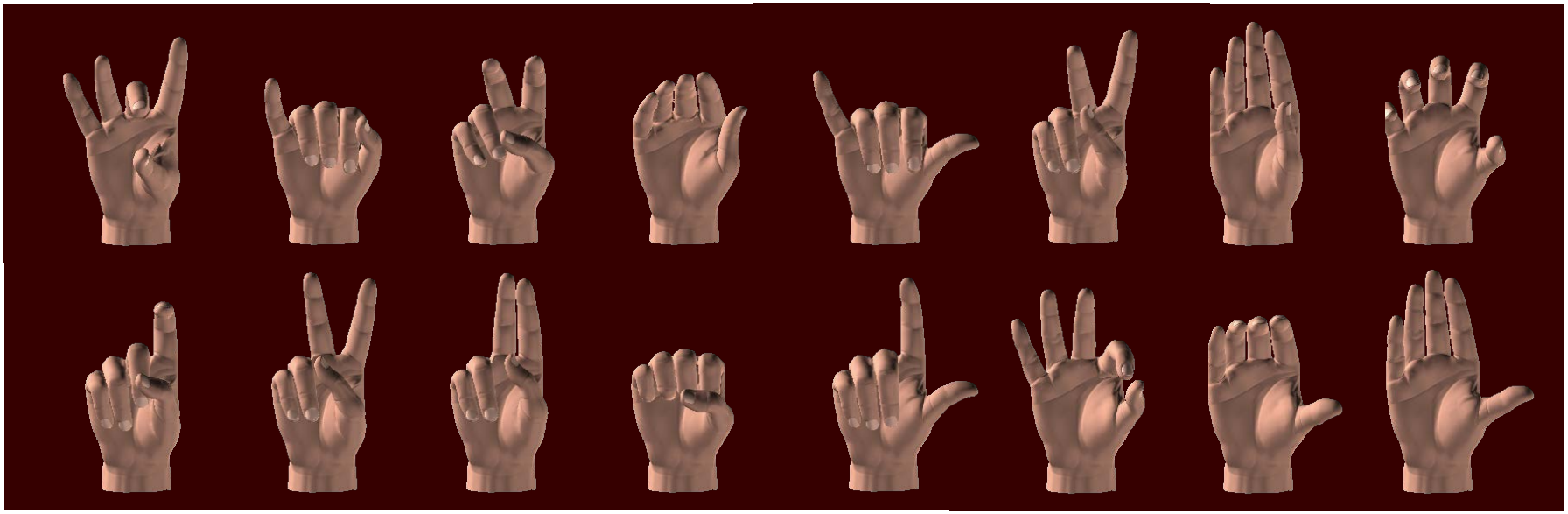
input image

Hand pose:

- Hand shape
- 3D hand orientation.

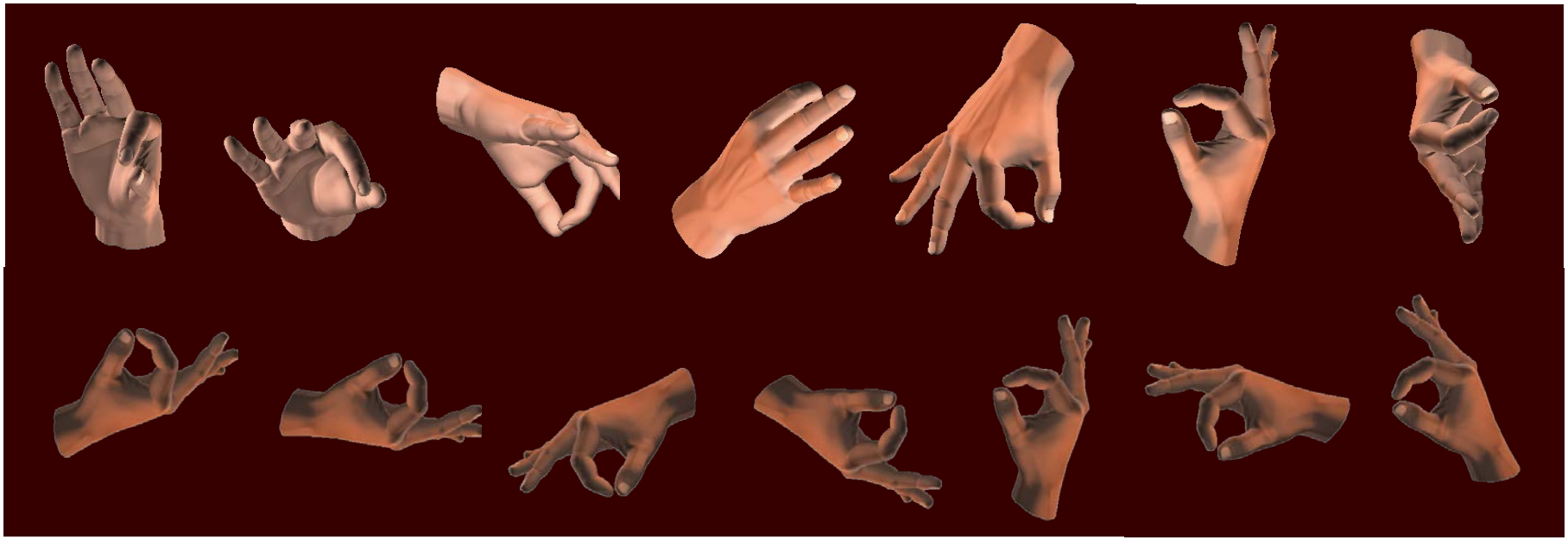
Hand Shapes

- Handshapes are specified by the joint angles of the fingers.
- Hands are very flexible.
  - Each finger has three joints, whose angles can vary.



3D Hand Orientation

- Images of the same handshape can look VERY different.
- Appearance depends on the 3D orientation of the hand with respect to the camera.
- Here are some examples of the same shape seen under different orientations.



Hand Pose Estimation: Applications

- There are several applications of hand pose estimation (if the estimates are sufficiently accurate, which is a big if):
  - Sign language recognition.
  - Human-computer interfaces (controlling applications and games via gestures).
  - Clinical applications (studying the motion skills of children, patients…).

input image

Hand pose:

- Hand shape
- 3D hand orientation.

Labeling Data for Hand Pose

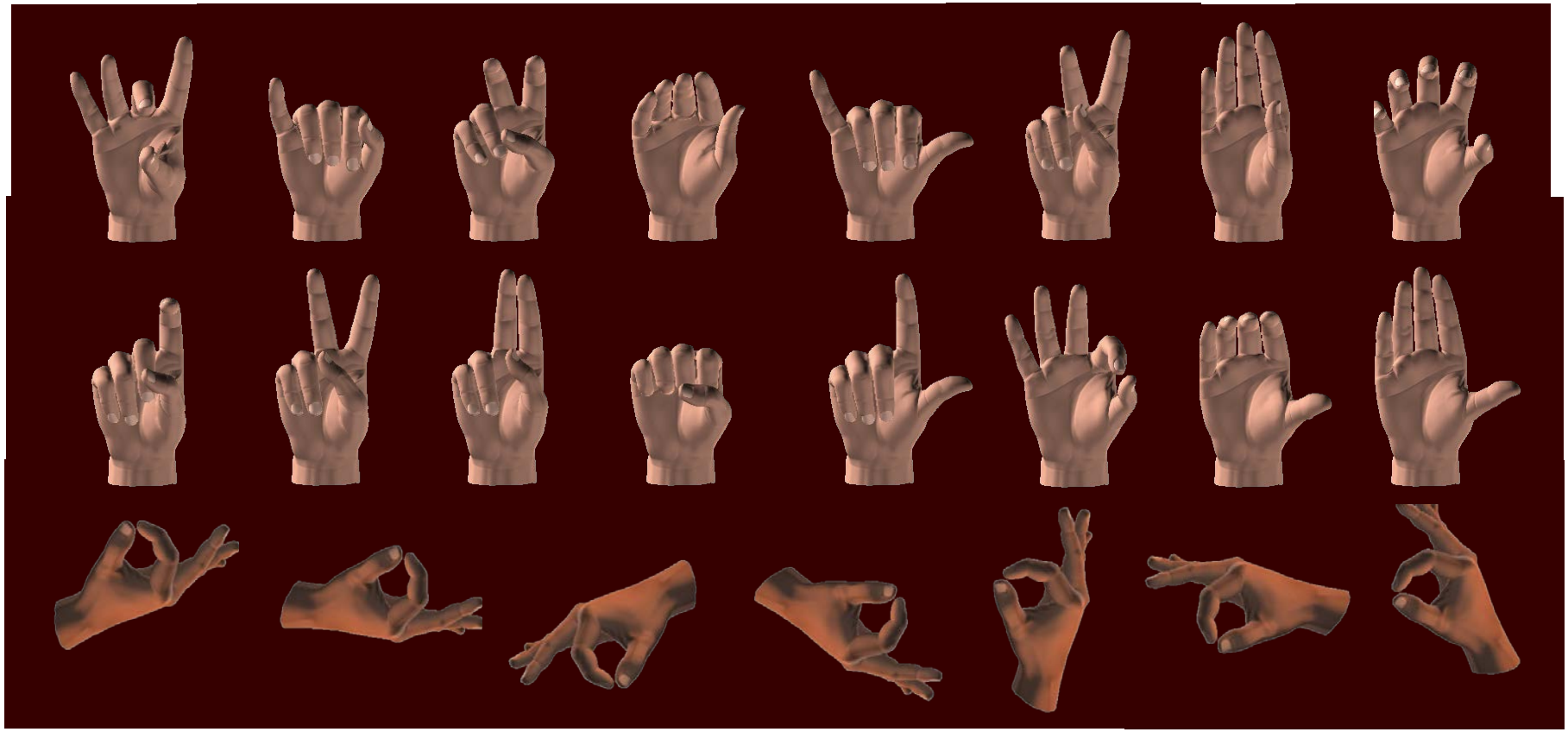
- In order to apply the methods we have learned this semester for hand pose estimation, we need a **labeled** training set.
- The term **labeled** simply means that for every training input we know the target output.
  - Every single dataset we have used this semester was labeled.
  - Usually the labels (target outputs) were given as part of the dataset.
  - Sometimes you had to write code that generated the labels automatically (for example, by reversing the word order in a sentence and then labeling that sentence as reverse).
- Instead of the term “labels” you will often see terms like “ground truth” or “annotations”.

Labeling Data for Hand Pose

- Labeling the hand pose in an image is relatively time-consuming and error-prone.
- To better understand the difficulty, consider labeling an MNIST image, which is relatively fast and reliable.
  - If a human looks at an MNIST image, most of the times the human knows immediately what the correct label is, and can provide that label by pressing a key.
- On the other hand, if we look at a hand image, we may understand intuitively what the pose is, but our brain cannot convert this intuitive understanding to actual joint angles.
- Alternatively, instead of labeling joint angles, we can label the pixel positions of the 15 joints.
  - That is an easier task, but still rather time-consuming.

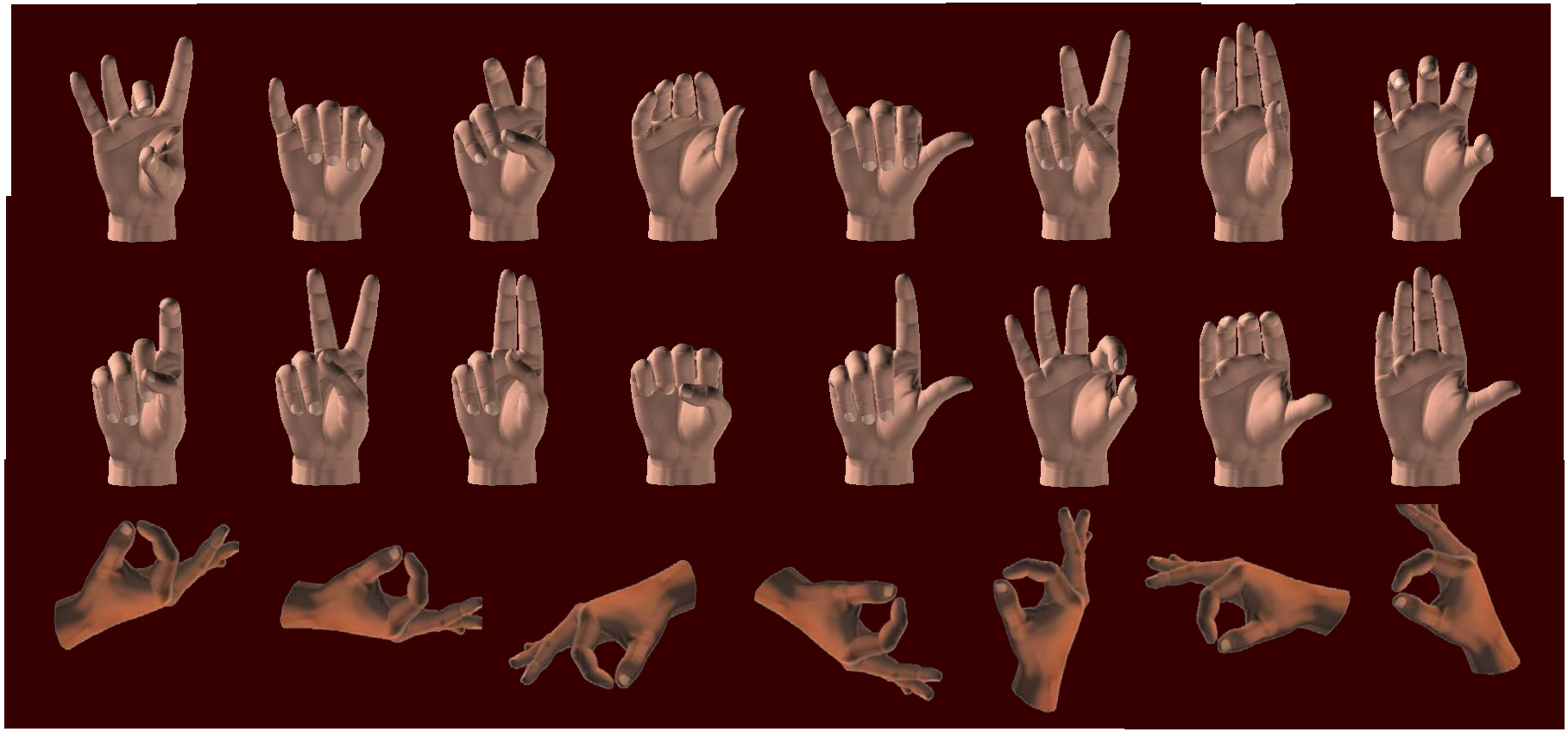
Synthetic Hand Images

- The images shown here were computer-generated.
  - Given joint angles, the program produces an image.
  - We can write a script that generates millions of joint angle combinations and the corresponding images.



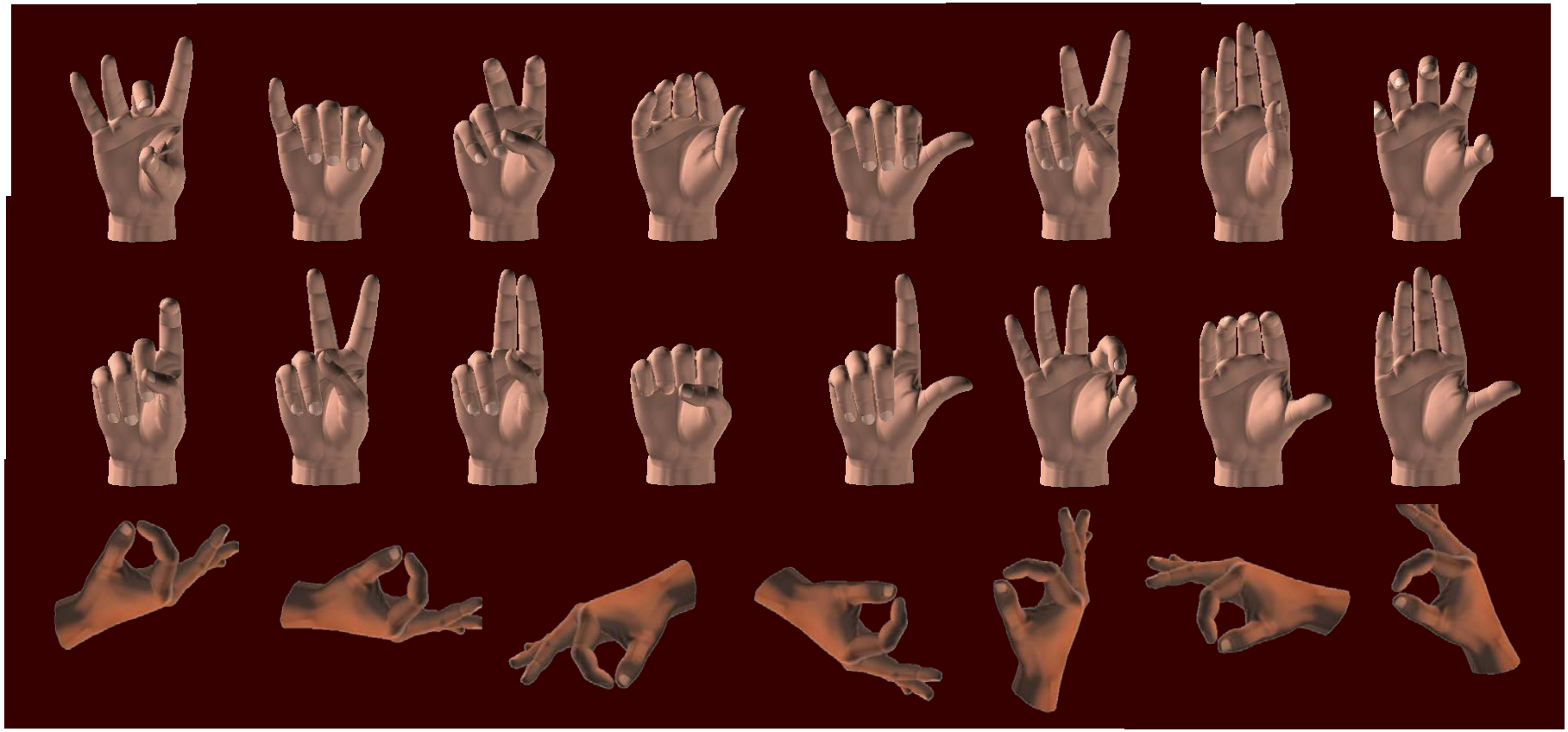
Synthetic Hand Images

- For synthetic hand images, we get the labels for free.
  - The joint angles shown in the images are produced by our own code.
- This means that we can generate a large training dataset easily.



Training on Synthetic Hand Images

- Problem: synthetic images are not quite the same as real images.
  - A model can learn to predict hand pose very accurately in synthetic images, and still be very inaccurate in real images.
  - Therefore, we want synthetic images that are as realistic as possible.



Anonymizing Images and Video

- Another application of synthetic data is in anonymizing images and video.
- For people using English (or any other language that people know how to read and write), it is straightforward to write anonymous text expressing their thoughts and opinions.
- For American Sign Language, there is no commonly used way to write it as text.
  - The typical way for a sign language user to state their thoughts or opinions is for the user to record a video.
  - However, the video shows the user, so the user desires to be anonymous, video is a far worse option than text.
- Potential solution: convert the video so that it shows a made-up (but realistic-looking) person doing the signing.

Realistic Scenes in Games and Movies

- Realistic synthetic data is highly valued in the gaming and entertainment industry.
- For example:
  - Scenes in sci-fi and phantasy movies may integrate real actors and landscapes with imaginary creatures and landscapes.
  - Scenes in action movies showing explosions and massive destruction can be much safer and cheaper to produce if they are not real.
  - In computer games, it may be important for people, objects and/or scenery to look realistic.
  - Realistic motion is also important, and can be very challenging to synthesize (for example, realistic motion of smoke, fire, water, humans and animals).

Generative Adversarial Networks

- Generative Adversarial Networks (GANs) were introduced in 2014 by this paper:

Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua (2014). Generative Adversarial Nets (PDF). Proceedings of the International Conference on Neural Information Processing Systems (NIPS 2014). pp. 2672–2680.

https://arxiv.org/abs/1406.2661

- GANs have become very popular and are commonly used to generate realistic synthetic data.

Generator and Discriminator

- What we really want is a “generator”: a module that produces realistic synthetic data.
- However, in a GAN model, we essentially train two separate modules that **compete with each other**:
  - The generator module, that produces synthetic data that is hopefully very realistic.
  - A **discriminator** module, that is trained to recognize if a piece of data is real or synthetic.

Generator and Discriminator

- The word “adversarial” in Generative Adversarial Networks refers to the fact that the generator and the discriminator actually compete with each other.
- The goal of the generator is to be so good that it can fool the discriminator as often as possible.
  - A good generator produces synthetic data that cannot be distinguished from real data, so the discriminator fails at that task.
- The goal of the discriminator is to be so good that it cannot be fooled by the generator.
  - The discriminator should tell with high accuracy if a piece of data is real or synthetic.

How It (Hopefully) Works

- The first version of the generator is initialized with random weights. Consequently, it produces random images that are not realistic at all.
- The discriminator is trained on a training set that combines:
  - A hopefully large number of real images.
  - An equally large number of images produced by the generator.
- Since the generated images are not realistic, the discriminator should achieve very high accuracy on this initial training set.
- Now we can train a second version of the generator.
  - Each input is just a random vector, that is used to make sure that the output images are not identical to each other.
  - The loss function is computed by giving the output of the generator to the discriminator. The more confident the discriminator is that the image is synthetic, the higher the loss.

How It (Hopefully) Works

- The second version of the generator should be better than the initial version with random weights.
  - The output images should now be more realistic.
- We now train a second version of the discriminator, incorporating into the training set the output images of the second version of the generator.
- Then, we train a third version of the generator, using the second version of the discriminator.
- And so on, we keep training alternatively:
  - a new version of the discriminator, using the latest version of the discriminator.
  - a new version of the generator, using the latest version of the discriminator.

Problems With Convergence

- In all models we have studied before, we had a single loss function.
  - In training, the model weights converged to a local optimum.
- Here, we have two competing loss functions:
  - The generator loss function, that is optimized as the generator gets better at fooling the discriminator.
  - The discriminator loss function, that is optimized as the discriminator gets better at NOT being fooled by the generator.
- We optimizing these losses iteratively, one after the other.
- It would be nice to be able to guarantee that after each iteration, both the generator and the discriminator are better (or at least not worse) than they were before that iteration.
  - Unfortunately, the opposite can also happen.

Problems With Convergence

- For example, suppose that we get to a point where the generator is really great, and it fools the discriminator to the maximum extent.
- What is the “maximum extent”?
  - The discriminator has to solve a binary classification problem: “real” vs. “synthetic”.
  - A random classifier would attain 50% accuracy.
  - With a perfect generator, the discriminator will be no better and no worse than a random classifier.
- If the generator is perfect, training the discriminator will produce a useless model, equivalent to a random classifier.
- The previous version of the discriminator, trained with data from an imperfect generator, would probably be better than the current version.

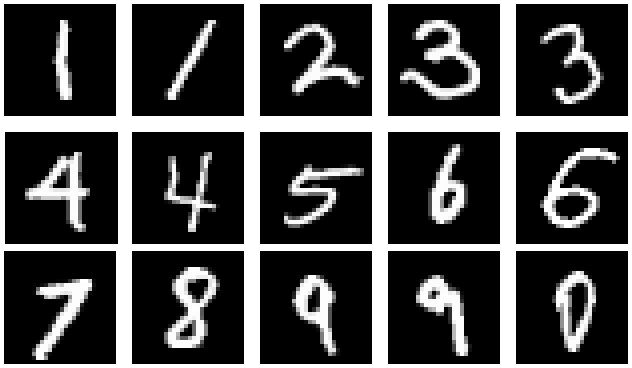
Problems With Convergence

- Conversely, suppose that we get to a point where the discriminator is 100% accurate, so that it is never fooled.
  - In that case, training the generator will produce a useless model, equivalent to a random image generator, since there will be no effect in the loss function by producing more realistic images.
  - The previous version of the generator, trained with data from an imperfect discriminator, would probably be better than the current version.
- So, overall, if one of the two components gets too good, then that makes it harder to improve the other component.
- In practice, GANs are used and often produce great results, but the system designer may need to manually intervene to guide the training to the right direction.
  - Overall, training GANs is somewhat complicated and heuristic.

Case Study: Training on MNIST

- See code in mnist_gan.py, posted under today's lecture.
- Input: training set of MNIST images.
- The GAN generator is trained to produce synthetic images, that "look like" the training images.

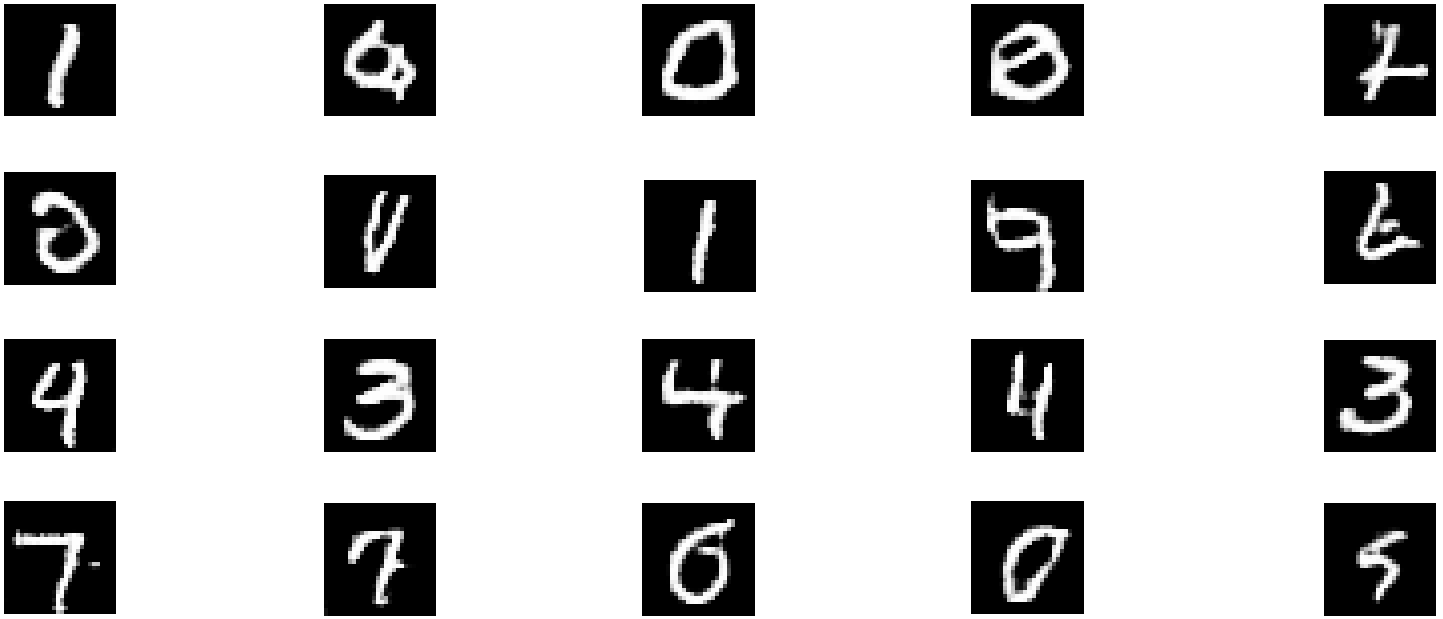
example real images,
from MNIST training set



example synthetic images,
produced by GAN model after
74 epochs of training.

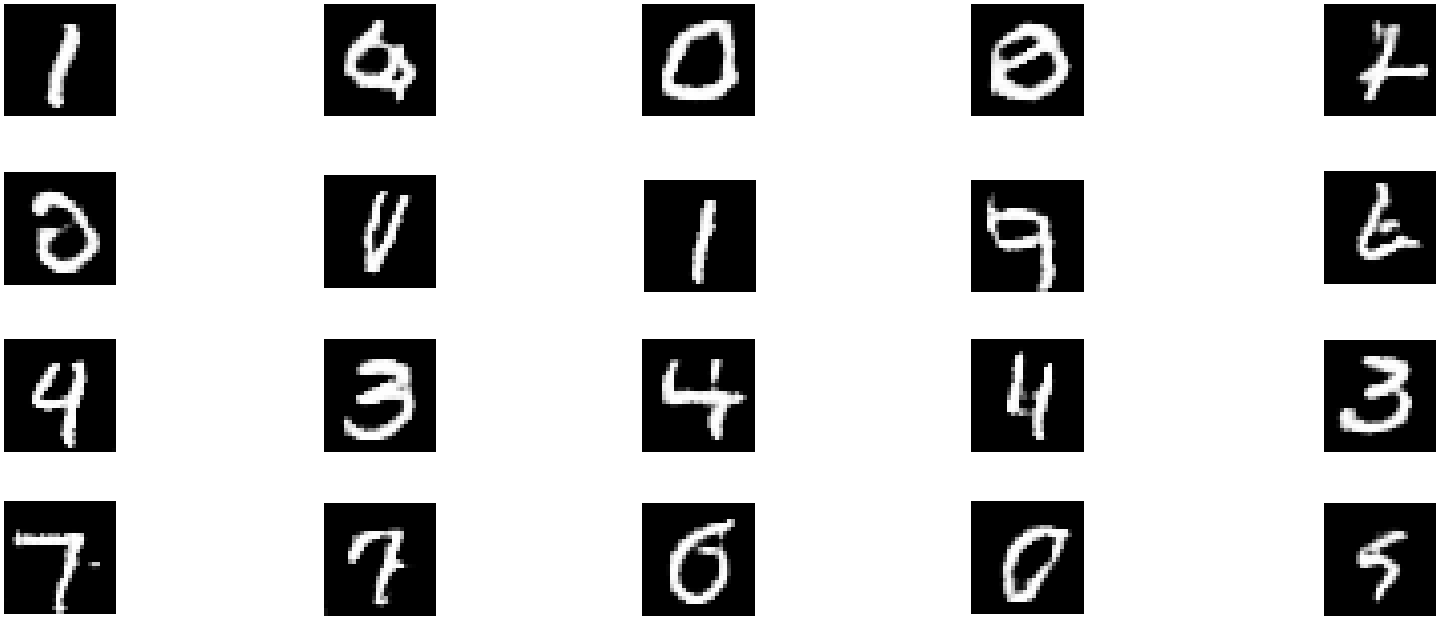
Evaluating Results

- Here are 20 example images.
- Can you tell which ones are real and which ones are synthetic?



Evaluating Results

- Here are 20 example images.
- Can you tell which ones are real and which ones are synthetic?
- Answer: they are all synthetic, produced by the GAN generator.
  - Some are easier to tell, but some look very convincing.



Code

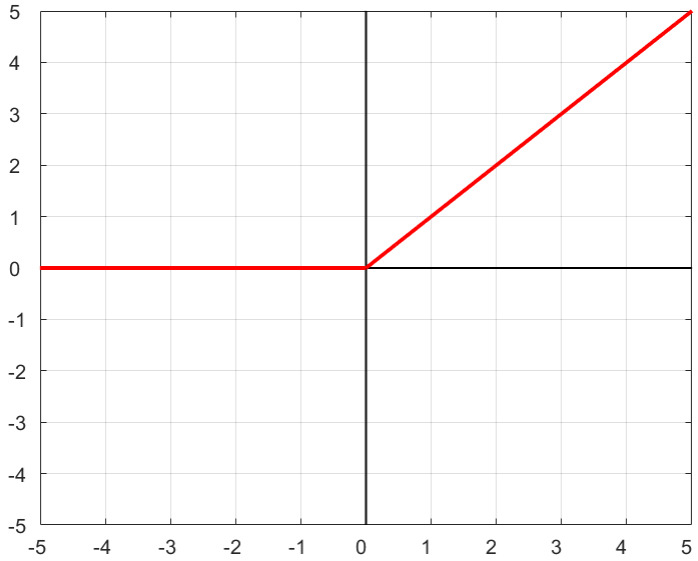
- The textbook includes code that trains a GAN on the CelebA dataset of over 200,000 images of faces of celebrities.
  - See file celeba_gan.py, posted under this lecture.
- I have not (yet) run this code for more than a fraction of an epoch, so I did not get any results.
  - It was taking more than 13 hours per epoch on my desktop computer
- I adapted that code to train a GAN on the MNIST dataset, to get something that runs as quickly as possible.
  - See file mnist_gan.py, posted under this lecture.

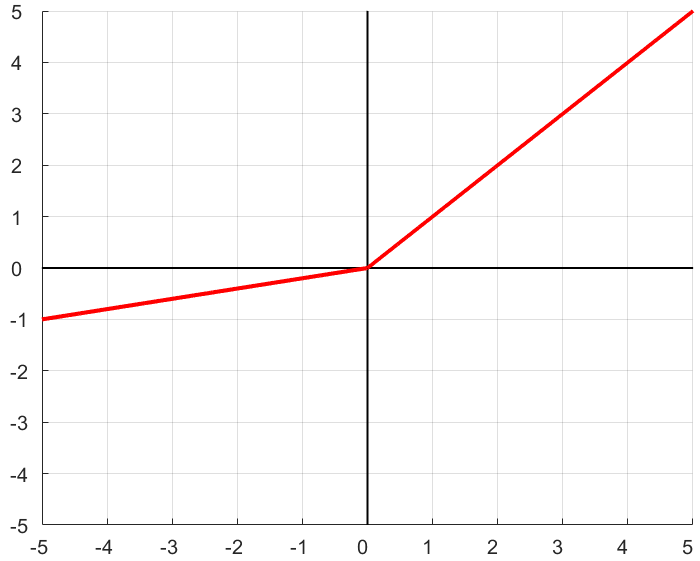
The Discriminator

```
discriminator_v2 = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(4,4),
                  strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(64, kernel_size=(4,4),
                  strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(64, kernel_size=4,
                   strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Flatten(),
    layers.Dropout(0.2),
    layers.Dense(1, activation="sigmoid"),
])
```

- The discriminator looks (mostly) like CNN models that we have already used.
  - It does binary classification of input images as “real” or “synthetic”.
- Key differences:
  - We do not use max pooling. Instead, we use strides=2, so that the output will have half the rows and half the columns with respect to the input.
  - Instead of ReLU activation, we use “Leaky ReLU” (see next slide).

Leaky ReLU





- Left: ReLU activation function.

$$\mathrm{ReLU}(x) = \max(x, 0)$$

- With ReLU, the derivative is 0 when $x < 0$.

- Right: LeakyReLU activation function.

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

- In our code and the figure, $\alpha = 0.2$.
- With LeakyReLU, the derivative is never 0.

The Discriminator

```
discriminator_v2 = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(4,4),
                  strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(64, kernel_size=(4,4),
                  strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(64, kernel_size=4,
                   strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Flatten(),
    layers.Dropout(0.2),
    layers.Dense(1, activation="sigmoid"),
])
```

- Key design choices:
  - No max pooling. Instead, we use strides=2.
  - Leaky ReLU instead of ReLU.
- Why these choices?
  - As mentioned before, GANs can be difficult to train. People using these models have found that some choices tend to lead to better results.
  - Regarding strides vs. max_pooling: the textbook says that strides work better in preserving information about where features are located.

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- Input: a random vector (so that the output is different each time).
- The dimensionality of the vector is a hyperparameter.
  - latent_dim = 64 in the code.

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- We then do a matrix multiplication layer.
  - Dense layer, no activation.
- We reshape the result to a 7x7x8 array.
  - 7 rows, 7 columns, 8 channels.

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- We then apply a Deconvolution operation, also called a **transposed convolution** operation.
  - This is implemented by a Conv2DTranspose layer in Keras.
- Deconvolution is the inverse operation of a convolution.
  - Let F be a convolution function that maps an $R \times C \times B$ array to an $R' \times C' \times B'$ array.
  - Then, F^{-1} is a deconvolution function. It maps an $R' \times C' \times B'$ array to an $R \times C \times B$ array.

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- A good reference in deconvolutions is this paper:

  "A guide to convolution arithmetic for deep learning", by Vincent Dumoulin, and Francesco Visin.

  https://arxiv.org/pdf/1603.07285v1.pdf

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- In this case, the deconvolution layer maps the $7 \times 7 \times 8$ input to a $14 \times 14 \times 64$ array.
- padding= "same" means that, when strides=1, the output has the same rows and columns as the input.
  - If strides = s, the output of deconvolution has s times the rows and s times the columns of the input.
- Note that we again use LeakyReLU as activation function.

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- The second deconvolution maps its $14 \times 14 \times 64$ input to a $28 \times 28 \times 128$ output.
- The third deconvolution maps its $28 \times 28 \times 128$ input to a $28 \times 28 \times 128$ output.
  - The output has the same number of rows and columns as the input, because strides=1 AND padding="same".

The Generator

```
generator_v2 = keras.Sequential([
    keras.Input(shape=(latent_dim,)),
    layers.Dense(7 * 7 * 8),
    layers.Reshape((7, 7, 8)),
    layers.Conv2DTranspose(64, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=2, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2DTranspose(128, kernel_size=4,
                           strides=1, padding="same"),
    layers.LeakyReLU(alpha=0.2),
    layers.Conv2D(1, kernel_size=5,
                  padding="same",
                  activation="sigmoid")])
```

- The last layer does normal convolution, that maps its $28 \times 28 \times 128$ input to a $28 \times 28 \times 1$ output.
  - Note that this is exactly the size of MNIST images.
  - It is important to produce synthetic images that have the same size as the real training images.
  - The discriminator takes both types of images (real and synthetic) as inputs, so they need to be the same size.

The GAN model

```
class GAN(keras.Model):
    def __init__(self, discriminator, generator,
                 latent_dim):
        # see next slides for the code

    def compile(self, d_optimizer, g_optimizer,
                loss_fn):
        # see next slides for the code

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]

    def train_step(self, real_images):
        # see next slides for the code
```

- The GAN model includes both the discriminator and the generator.
- It is implemented as a subclass of keras.Model.
  - We have already seen how to define custom layers, this is the first time we see how to define a custom model.
- The next slides go over the code for each method.

The GAN Model Constructor

```
def __init__(self, discriminator, generator,
                    latent_dim):
    super().__init__()
    self.discriminator = discriminator
    self.generator = generator
    self.latent_dim = latent_dim
    self.d_loss_metric =
            keras.metrics.Mean(name="d_loss")
    self.g_loss_metric =
            keras.metrics.Mean(name="g_loss")
```

- Inputs:
  - The component models, i.e. the discriminator and the generator.
  - Also, the latent dimensions, i.e., the dimensions for the random vector that is given as input to the generator.

The GAN Model **compile()** Method

```
def compile(self, d_optimizer, g_optimizer,
                    loss_fn):
    super(GAN, self).compile()
    self.d_optimizer = d_optimizer
    self.g_optimizer = g_optimizer
    self.loss_fn = loss_fn
```

- Inputs:
  - The optimizers for the discriminator and the generator.
    - In our code, when we call this method, we pass Adam for both optimizers
  - The loss function.
    - When we call this method, we pass BinaryCrossentropy as the loss function.

The **train_step()** Method

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
    generated_images =
        self.generator(random_latent_vectors)
    combined_images =
        tf.concat([generated_images,
                   real_images], axis=0)
    labels = tf.concat([tf.ones((batch_size, 1)),
                        tf.zeros((batch_size, 1))],
                        axis=0)
    labels += 0.05 *
              tf.random.uniform(tf.shape(labels))

    # code continues, see next slides.
```

- The **train_step()** method specifies how to do training.
  - In particular, it specifies how to process a single batch of training data. In our case, the batch of training data is a batch of real images from the MNIST dataset.
- Inputs:
  - **real_images**, it is the batch of training data.
- First step: get the batch size.
  - **real_images** is an array of shape [batch_size, 28, 28, 1].

The **train_step()** Method

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
    generated_images =
        self.generator(random_latent_vectors)
    combined_images =
        tf.concat([generated_images,
                   real_images], axis=0)
    labels = tf.concat([tf.ones((batch_size, 1)),
                        tf.zeros((batch_size, 1))],
                        axis=0)
    labels += 0.05 *
              tf.random.uniform(tf.shape(labels))

    # code continues, see next slides.
```

- Second step: generate a batch of random vectors, that are used as input to the generator.
- Third step: apply the current version of the generator to generate a batch of synthetic images given the batch of random vectors.

The **train_step()** Method

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
    generated_images =
        self.generator(random_latent_vectors)
    combined_images =
        tf.concat([generated_images,
                   real_images], axis=0)
    labels = tf.concat([tf.ones((batch_size, 1)),
                        tf.zeros((batch_size, 1))],
                        axis=0)
    labels += 0.05 *
              tf.random.uniform(tf.shape(labels))

    # code continues, see next slides.
```

- Fourth step: create a training batch for the discriminator.
- **combined_images** will be the training inputs, half part generated images, half part real images.
- **labels** will be the target outputs.
  - Class label = 1 for synthetic images, 0 for real images.
  - We add random noise to the labels, as a random value between 0 and 0.05. This is yet another empirical hack.

GradientTape

```
# second part of code for train_step().

with tf.GradientTape() as tape:
    predictions = self.discriminator(
                combined_images)
    d_loss = self.loss_fn(labels, predictions)

grads = tape.gradient(d_loss,
            self.discriminator.trainable_weights)
self.d_optimizer.apply_gradients(
        zip(grads,
            self.discriminator.trainable_weights))
```

- **GradientTape** is a topic that merits significant coverage on its own.
- Remember, the whole point of Tensorflow is to do automatic calculation of gradients in a computational graph.
  - This semester, this automatic calculation has been happening when we train the Keras model using the model.fit() method.
  - Here, the **train_step()** method customizes what model.fit() does, and we need to be explicit about gradient calculations.

GradientTape

```
# second part of code for train_step().

with tf.GradientTape() as tape:
    predictions = self.discriminator(
                combined_images)
    d_loss = self.loss_fn(labels, predictions)

grads = tape.gradient(d_loss,
            self.discriminator.trainable_weights)
self.d_optimizer.apply_gradients(
        zip(grads,
            self.discriminator.trainable_weights))
```

- In general, Keras is a high-level wrapper around Tensorflow.
  - When Keras does what we want, the Keras code is much more simple than the non-Keras Tensorflow equivalent.
  - When Keras cannot do something we want, usually Tensorflow lets us implement it.
  - When using Tensorflow directly, we usually need to be explicit about gradients and optimization, and GradientTape is used a lot.
  - Look up GradientTape at the textbook's index, to see where it is used and discussed.

GradientTape

```
# second part of code for train_step().

with tf.GradientTape() as tape:
    predictions = self.discriminator(
                combined_images)
    d_loss = self.loss_fn(labels, predictions)

grads = tape.gradient(d_loss,
            self.discriminator.trainable_weights)
self.d_optimizer.apply_gradients(
        zip(grads,
            self.discriminator.trainable_weights))
```

- The line highlighted in red tells Tensorflow to start a GradientTape scope.
- Within that scope, when any calculations are performed, Tensorflow keeps track of partial derivatives with respect to some specified variables.
- A model's trainable_weights are included, by default, in those specified variables.

GradientTape

```
# second part of code for train_step().

with tf.GradientTape() as tape:
    predictions = self.discriminator(
                combined_images)
    d_loss = self.loss_fn(labels, predictions)

grads = tape.gradient(d_loss,
            self.discriminator.trainable_weights)
self.d_optimizer.apply_gradients(
        zip(grads,
            self.discriminator.trainable_weights))
```

- Essentially, this chunk of code:
  - Applies the discriminator to the training batch of images.
  - Computes the discriminator's loss. Remember, the loss function in the code is BinaryCrossentropy, which makes sense, since the discriminator is a binary classifier.
  - Retrieves the gradient of the loss with respect to the discriminator's trainable weights.
  - Calls the optimizer (we use Adam in the code) to update the discriminator's trainable weights based on the gradients.

The **train_step()** Method

```
# third part of code for train_step().

random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
misleading_labels = tf.zeros((batch_size, 1))

with tf.GradientTape() as tape:
   predictions = self.discriminator(
        self.generator(random_latent_vectors))
   g_loss = self.loss_fn(misleading_labels,
                         predictions)
grads = tape.gradient(g_loss,
     self.generator.trainable_weights)
self.g_optimizer.apply_gradients(
        zip(grads,
            self.generator.trainable_weights))
```

- Now, we need to train the generator.
- We first generate another batch of images using the generator.
- As before, first we generate the random vectors that are given as inputs to the generator.

The **train_step()** Method

```
# third part of code for train_step().

random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
misleading_labels = tf.zeros((batch_size, 1))

with tf.GradientTape() as tape:
   predictions = self.discriminator(
        self.generator(random_latent_vectors))
   g_loss = self.loss_fn(misleading_labels,
                         predictions)
grads = tape.gradient(g_loss,
     self.generator.trainable_weights)
self.g_optimizer.apply_gradients(
        zip(grads,
            self.generator.trainable_weights))
```

- **misleading_labels** are class labels for these generated images.
- All the misleading labels are set to 0.
  - Remember, 0 is supposed to mean “real images”.
  - This is why these labels are called “misleading”.

The **train_step()** Method

```
# third part of code for train_step().

random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
misleading_labels = tf.zeros((batch_size, 1))

with tf.GradientTape() as tape:
   predictions = self.discriminator(
        self.generator(random_latent_vectors))
   g_loss = self.loss_fn(misleading_labels,
                         predictions)
grads = tape.gradient(g_loss,
     self.generator.trainable_weights)
self.g_optimizer.apply_gradients(
        zip(grads,
            self.generator.trainable_weights))
```

- The highlighted lines:
  - Apply the generator on the random vectors to generate images.
  - Apply the discriminator to get its predictions on the generated images.
  - Now, we compute the generator's loss.
  - Ideally (for the generator), the discriminator will be fooled for every single generated image, and output 0. That is why the target labels (**misleading_labels**) are all 0.
  - The loss function is again BinaryCrossEntropy.

The **train_step()** Method

```
# third part of code for train_step().

random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
misleading_labels = tf.zeros((batch_size, 1))

with tf.GradientTape() as tape:
   predictions = self.discriminator(
        self.generator(random_latent_vectors))
   g_loss = self.loss_fn(misleading_labels,
                         predictions)
grads = tape.gradient(g_loss,
     self.generator.trainable_weights)
self.g_optimizer.apply_gradients(
        zip(grads,
            self.generator.trainable_weights))
```

- The highlighted lines:
  - Retrieve the gradient of the loss with respect to the generator's trainable weights.
  - Call the optimizer (again, we use Adam in the code) to update the generator's trainable weights based on the gradients.

The **train_step()** Method

```
# fourth and last part of code for train_step().


self.d_loss_metric.update_state(d_loss)
self.g_loss_metric.update_state(g_loss)
return {"d_loss": self.d_loss_metric.result(),
        "g_loss": self.g_loss_metric.result()}
```

- The last lines of **train_part()**:
  - Record the loss that was calculated for the discriminator and the generator.
  - Return those two losses.

The **GANMonitor** Class

```
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=3,
                       latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim



    def on_epoch_end(self, epoch,
                                logs=None):
        # see next slides
```

- The GANMonitor class is a subclass of keras.callbacks.Callback.
- It is used to define a function, called **on_epoch_end()**, that should be called at the end of every epoch.

The **GANMonitor** Class

```
def on_epoch_end(self, epoch, logs=None):
    random_latent_vectors =
        tf.random.normal(shape=(self.num_img,
                                self.latent_dim))
    generated_images =   self.model.generator(
            random_latent_vectors)
    generated_images *= 255
    generated_images.numpy()
    for i in range(self.num_img):
        img = keras.utils.array_to_img(
              generated_images[i])

img.save(f"mnist_results/generated_img_{epoc
h:03d}_{i}.png")
```

- Method **on_epoch_end()**:
  - Generates a certain number of images (10 in our code).
  - Saves those images to disk.
- This way, as the training goes on, we can monitor the quality of the generated results.

Running the Code

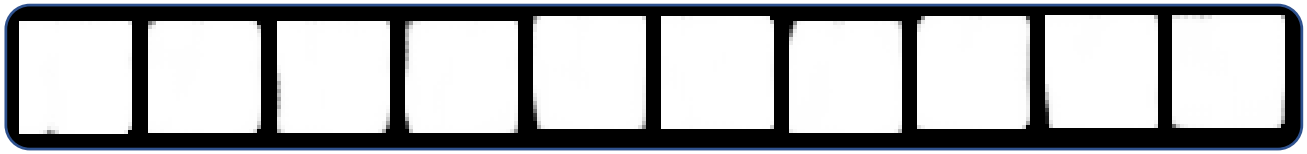
```
gan = GAN(discriminator=discriminator,  generator=generator, latent_dim=latent_dim)
gan.compile(d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
                g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
                loss_fn=keras.losses.BinaryCrossentropy(),)

gan.fit(train_dataset, epochs=epochs,
        callbacks=[GANMonitor(num_img=10, latent_dim=latent_dim)])
```

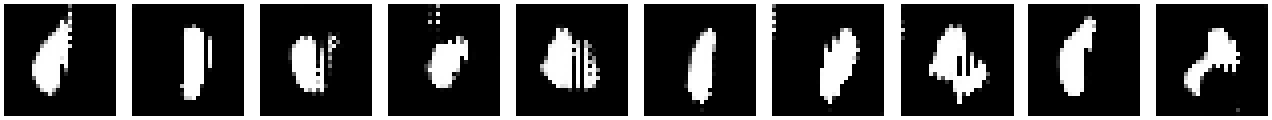
- These lines run the code.
  - We create the GAN model by calling the GAN constructor.
  - We compile the model.
  - We call **fit()** to train.

Results: Generated Images

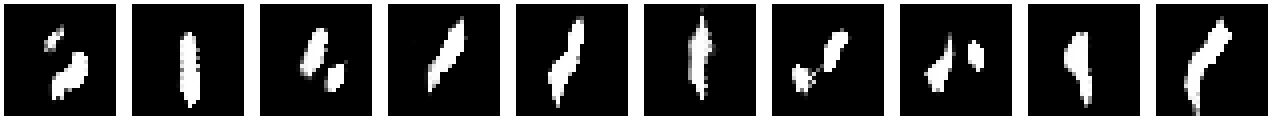
After 1 epoch: (all images mostly white)



After 2 epochs:

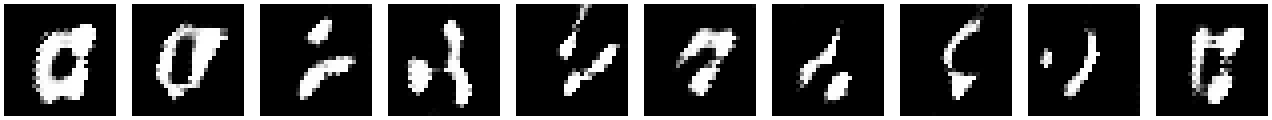


After 3 epochs:

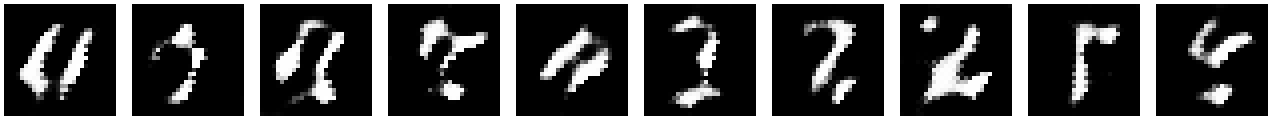


After 4 epochs:

After 5 epochs:



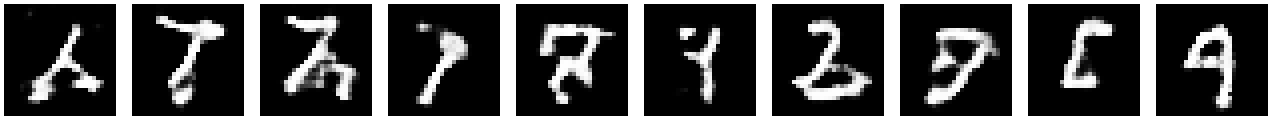
After 6 epochs:



After 7 epochs:

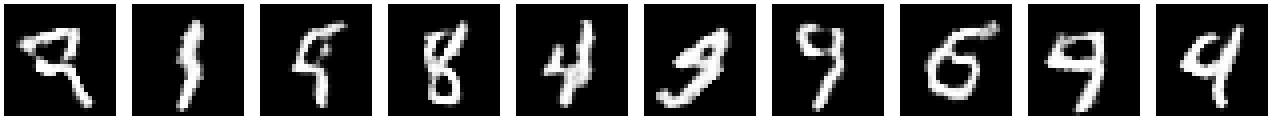
Results: Generated Images

After 10 epochs:

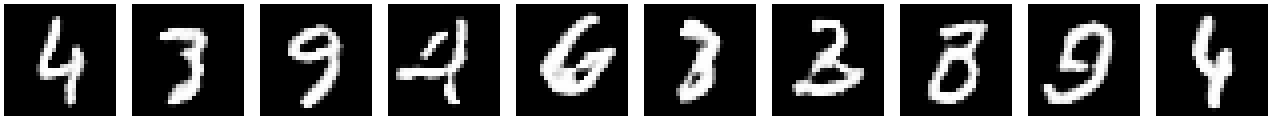


After 20 epochs:

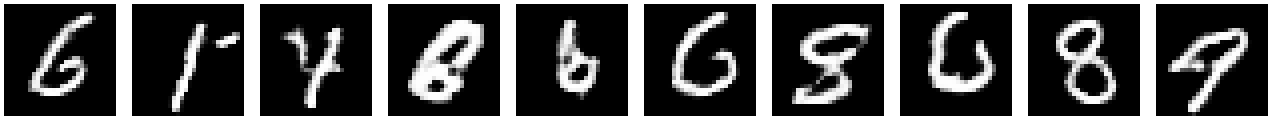
After 30 epochs:



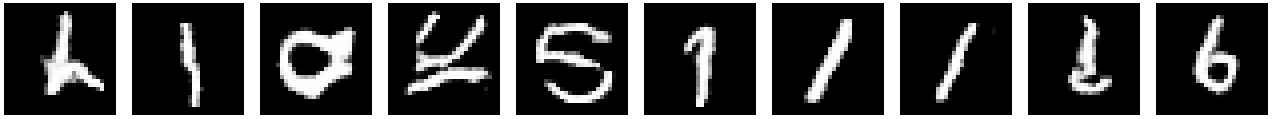
After 40 epochs:



After 50 epochs:

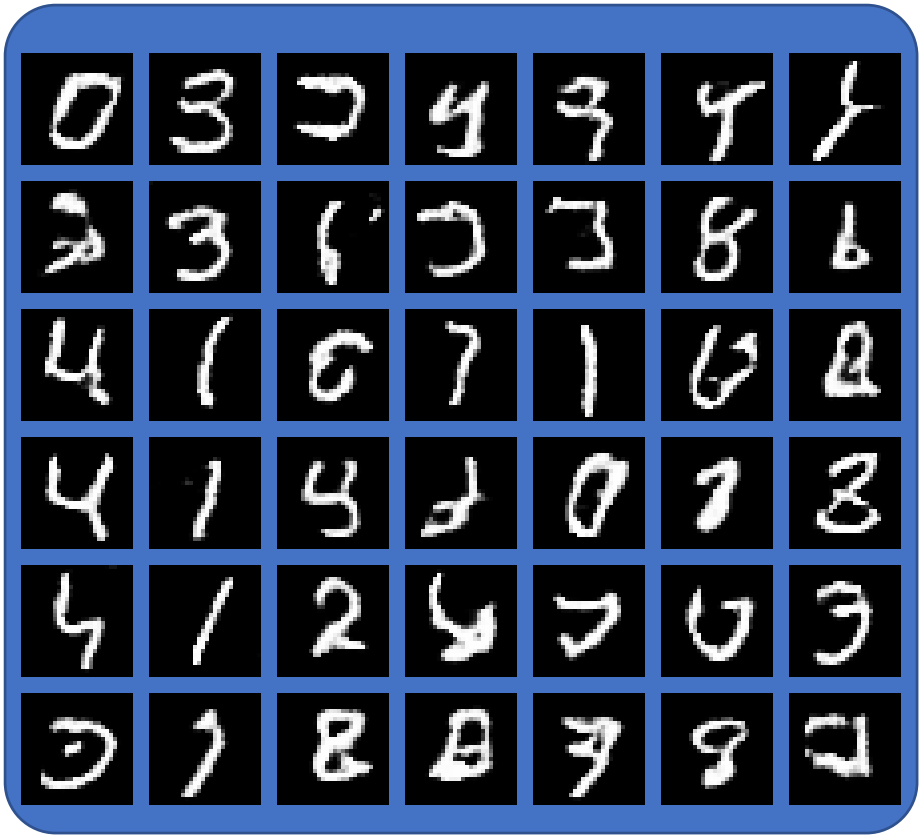


After 75 epochs:

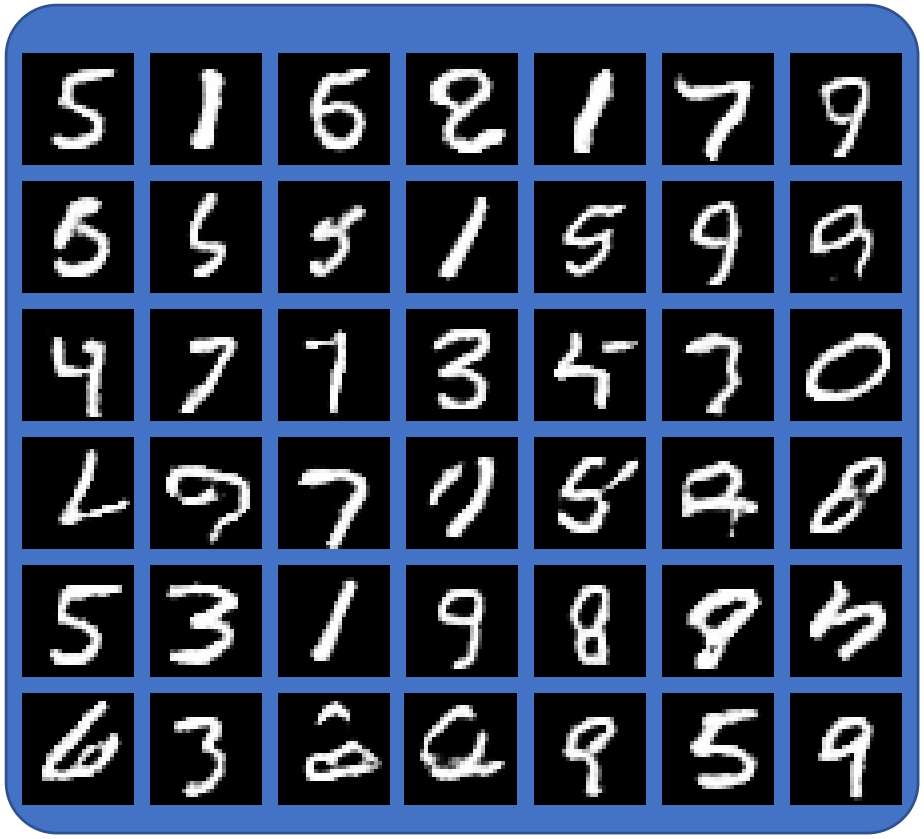


After 100 epochs:

Results: GAN Version 1 vs. Version 2



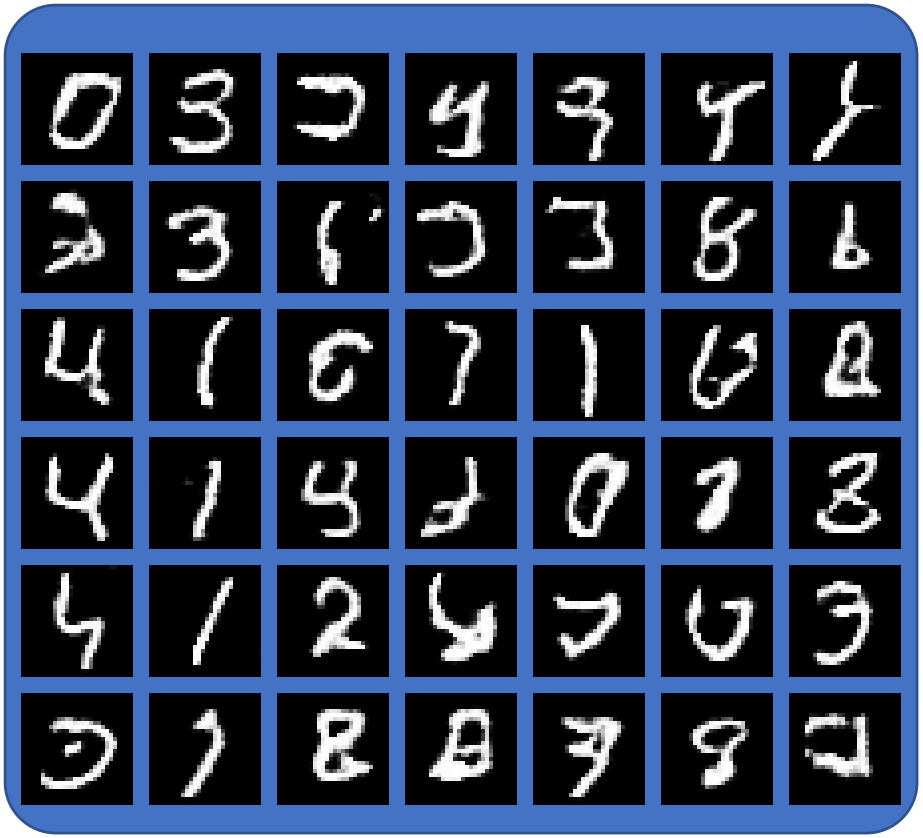
Results from version ???, epochs 95-99.



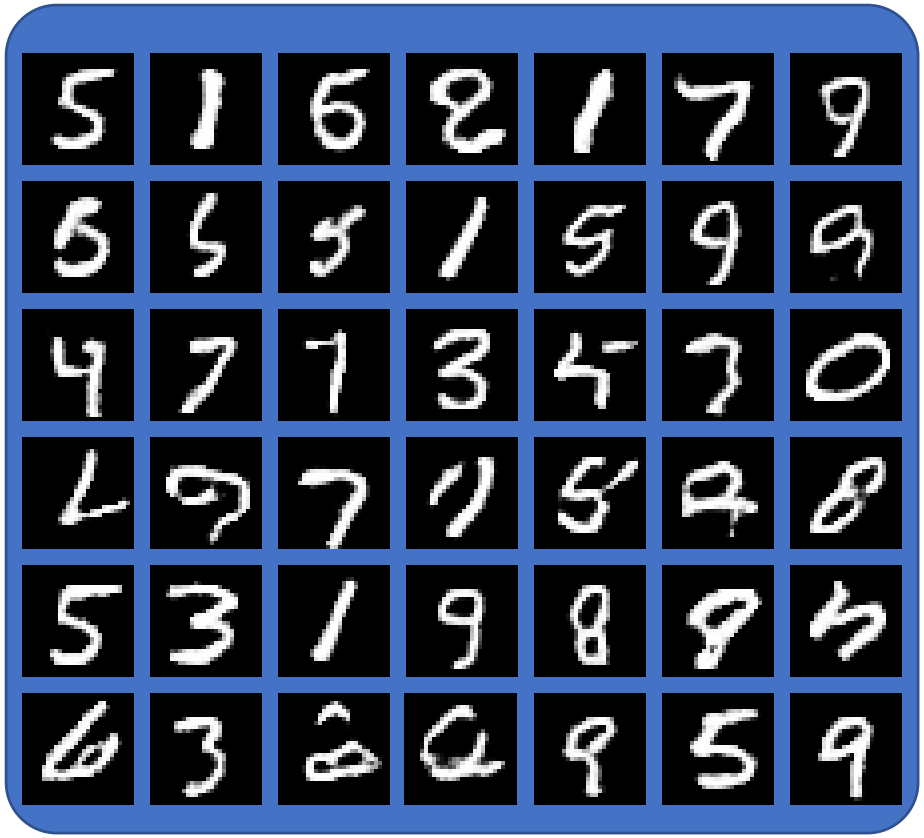
Results from version ???, epochs 95-99.

Which results look better to you?

Results: GAN Version 1 vs. Version 2

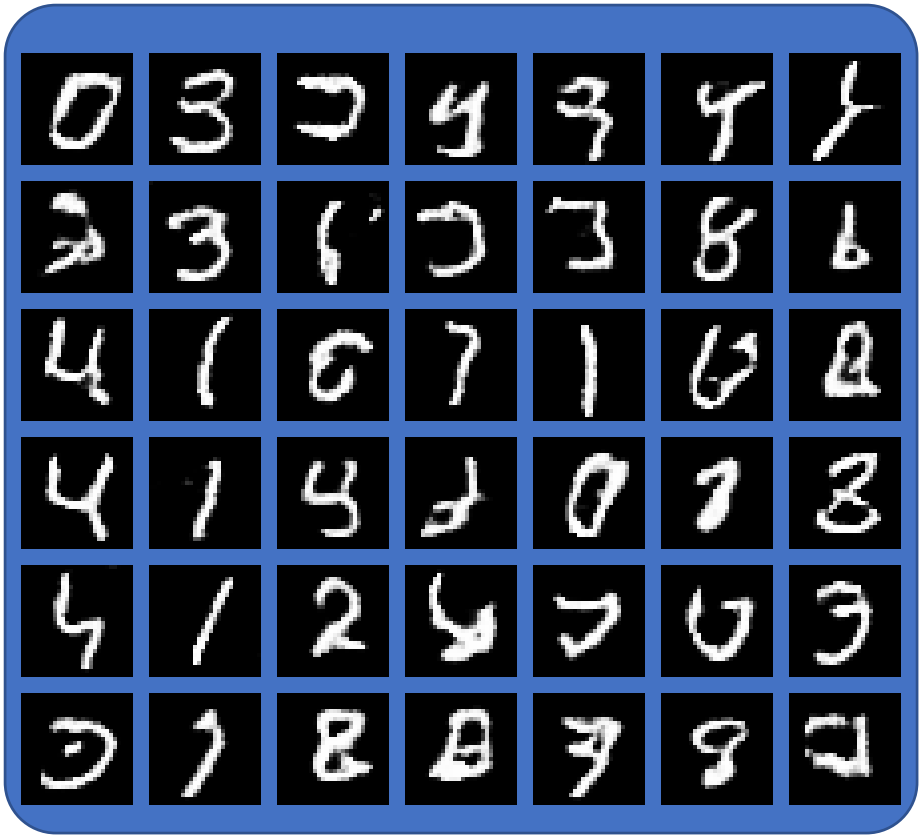


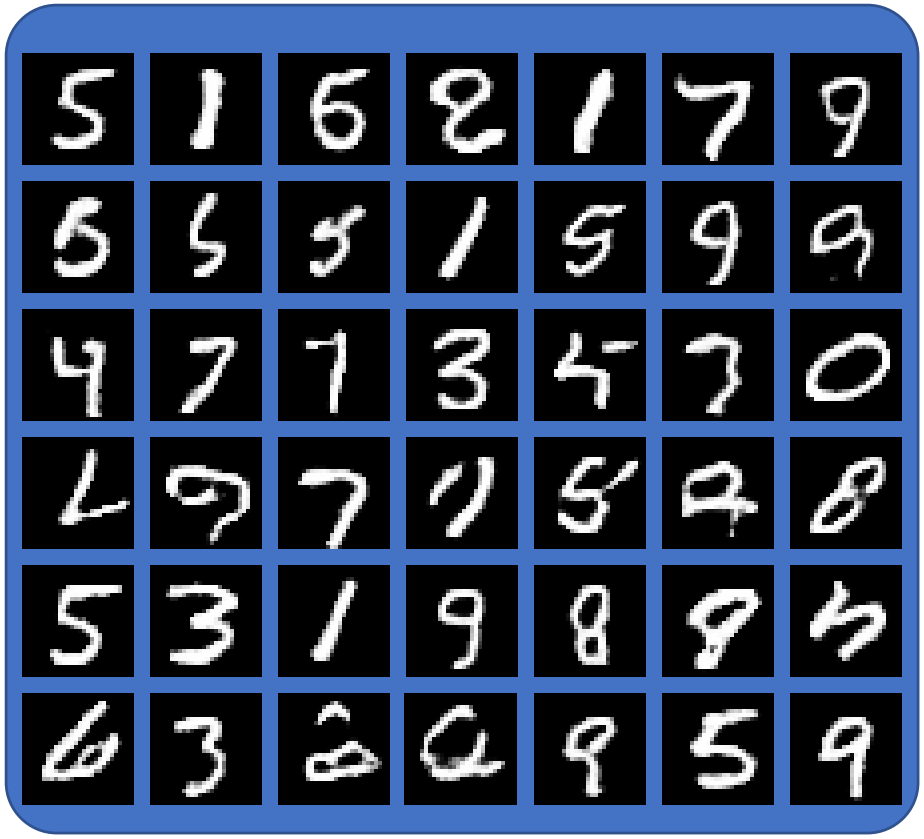
Results from version 1, epochs 95-99. Version 1 took about 3 minutes per epoch, for a total training time of about 300 minutes (5 hours).



Results from version 2, epochs 95-99. Version 2 took about 15 minutes per epoch, for a total training time of about 1500 minutes (25 hours).

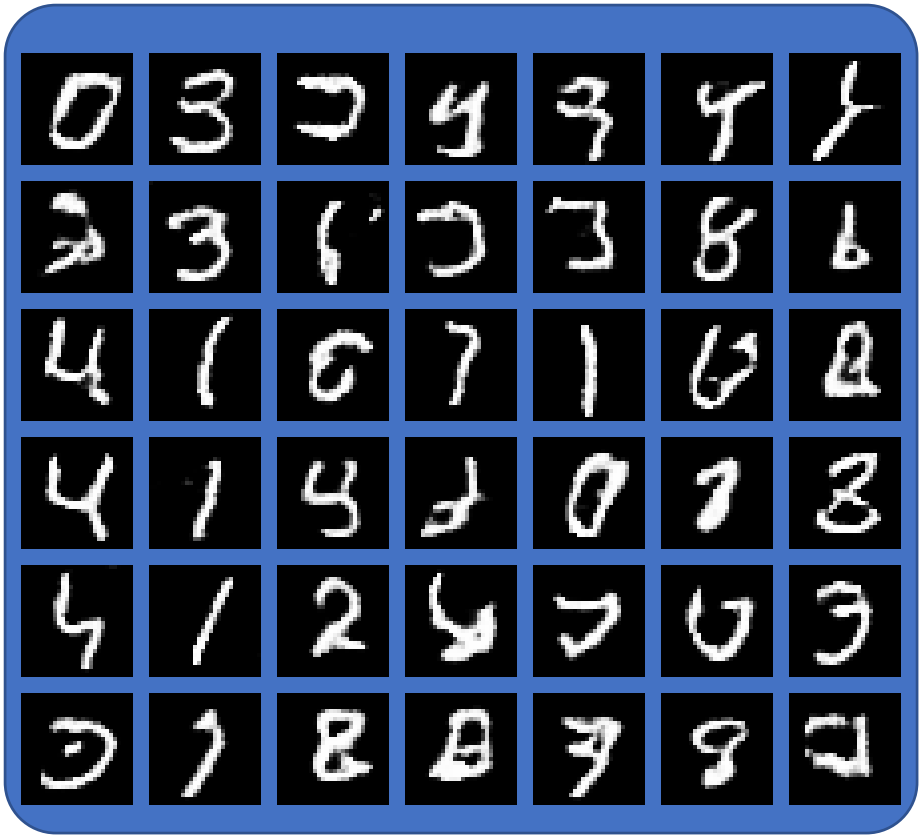
How Should We Evaluate Results?

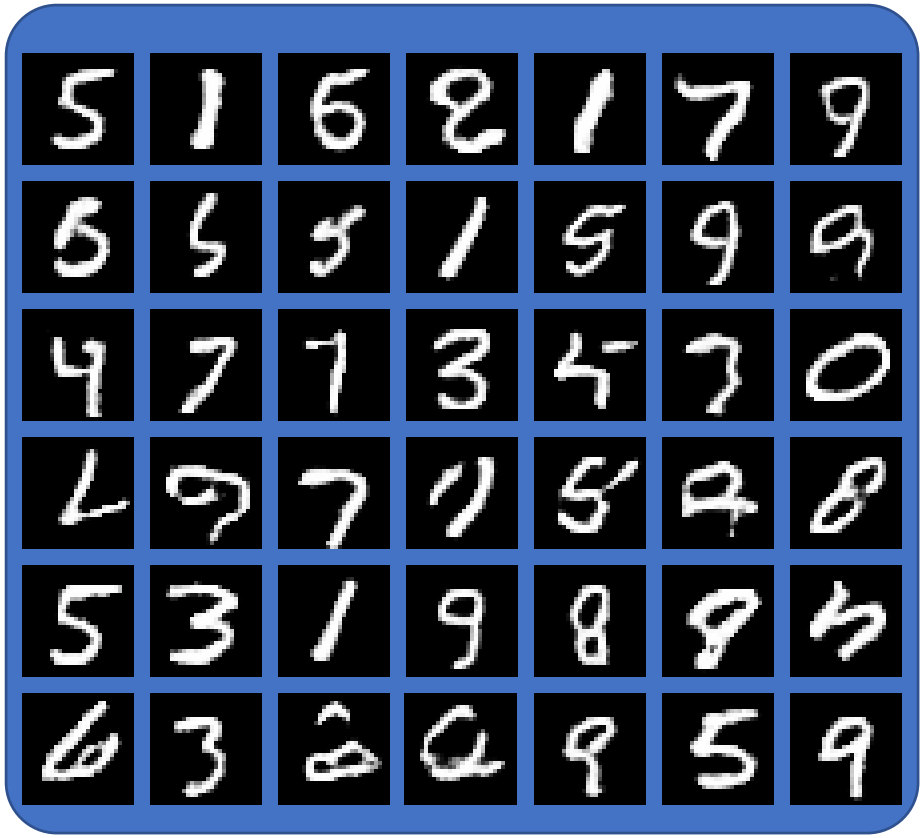




- In previous topics, we had test sets and evaluation criteria.
  - We mostly measured classification accuracy, for classification tasks.
  - We also used mean absolute error., for a regression task.

How Should We Evaluate Results?





- Here, we do not have a test set.
- Also, we do not have an obvious quantitative measure of quality.
  - We can ask people to subjectively rate the quality of the generated images.

GAN Models: Summary

- A GAN model contains two competing submodels.
  - The discriminator and the generator.
- The training goals for the two models are opposite.
  - The discriminator is trained to NOT be fooled by the generator.
  - The generator is trained to fool the discriminator.
- Training GAN models can be tricky, the models can actually start performing worse after a while.
  - Several specific heuristics are often used to make the training process more likely to produce useful results.
- Overall, GAN models have been very popular, so they do work quite a lot of times.
  - Various improvements and variations, that we have not discussed, can be used to improve the quality of results and the chances of success.