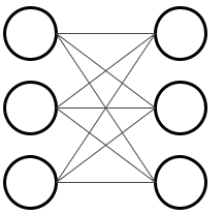
# Lecture 8: Advanced Convolutional Models

## Contents





**DSCI 572**

Supervised Learning II

## Lecture Learning Objectives

- Explain the difference between generative vs. discriminative models
- Describe what an autoencoder is at a high level and what they can be useful for

- Describe what a generative adversarial network is at a high level and what they can be useful for

# Imports

```
import numpy as np
import pandas as pd
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset
from torchvision import transforms, datasets, utils, models
from torchsummary import summary
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
import sys, os
sys.path.append(os.path.join(os.path.abspath(".."), "code"))
from plotting import *
from set_seed import *
DATA_DIR = DATA_DIR = os.path.join(os.path.abspath(".."), "data/")

from PIL import Image
plt.rcParams.update({'font.size': 16, 'axes.labelweight': 'bold', 'axes.grid':
import plotly.io as pio
pio.renderers.default = "png"
```

# ❓ ❓ Questions for you

## iClicker Exercise

**Select all of the following statements which are TRUE.**

1. I feel confident in implementing CNNs from scratch, including defining layers and training the model, as long as I have access to documentation and lecture notes.
2. I understand the key components of CNNs conceptually, but I struggle when translating this understanding into code.
3. I find it challenging to keep track of the input and output dimensions across different layers in a CNN.
4. I find CNNs overwhelming and struggle with understanding both the concepts and implementation.
5. I am unsure why we are learning CNNs and how they fit into data science applications.

You are working on a multiclass classification problem using `nn.CrossEntropyLoss`. Below are the training and validation blocks from a typical training loop.

Discuss with your neighbor:

1. Why is `torch.softmax` applied in the validation block but not in the training block?
2. What does `nn.CrossEntropyLoss` expect as input? What would happen if we applied `torch.softmax` before passing the outputs to it?
3. What if we omitted `torch.softmax` in the validation block? How would that affect the model's predictions and evaluation?

```
criterion = nn.CrossEntropyLoss()


# Training
model.train()
for X, y in trainloader:
    X, y = X.to(device), y.to(device)
    optimizer.zero_grad()
    y_hat = model(X)
    loss = criterion(y_hat, y)
    loss.backward()
    optimizer.step()
    train_batch_loss += loss.item()


# Validation
model.eval()

with torch.no_grad():  # this stops pytorch doing computational graph stuff un
    for X, y in validloader:
        X, y = X.to(device), y.to(device)
        y_hat = model(X)
        _, y_hat_labels = torch.softmax(y_hat, dim=1).topk(1, dim=1)
        loss = criterion(y_hat, y)
        valid_batch_loss += loss.item()
        valid_batch_acc += (y_hat_labels.squeeze() == y).type(torch.float32).m
```

Note that `nn.CrossEntropyLoss` already incorporates the softmax function internally. From the documentation

- The input is expected to contain the unnormalized logits for each class (which do not need to be positive or sum to 1, in general).

```
criterion = nn.CrossEntropyLoss()
y = torch.tensor([0, 3]) # targets for two examples assuming a 4-class classif

y_preds_good = torch.tensor([[2.0, 0.1, 0.2, 0.4], [0.22, 0.7, 1.0, 4.0]])
y_preds_bad = torch.tensor([[0.1, 0.3, 3.0, 1.2], [3.1, 1.3, 0.9, 0.2]])
```

```
criterion(y_preds_good, y) # low loss as expected
```

```
tensor(0.2602)
```

```
criterion(y_preds_bad, y) # high loss as expected
```

```
tensor(3.1694)
```

# Using torchsummary

Manually calculating the output map shape of complex CNNs with multiple layers before the final flattening layer can be tedious.

In these instances, the `summary` function from `torchsummary` can be useful.

```
class test_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.main = nn.Sequential(
            nn.Conv2d(3, 8, (5, 5)),
            nn.ReLU(),
            nn.MaxPool2d((2, 2)),
            nn.Conv2d(8, 4, (3, 3)),
            nn.ReLU(),
            nn.MaxPool2d((3, 3)),
            nn.Dropout(0.2),
            # nn.Flatten(),
            # nn.Linear(324, 128),
            # nn.ReLU(),
            # nn.Linear(128, 1)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```
from torchsummary import summary

model = test_CNN()
# The second argument is the size of your input.
summary(model, (3, 64, 64));
```

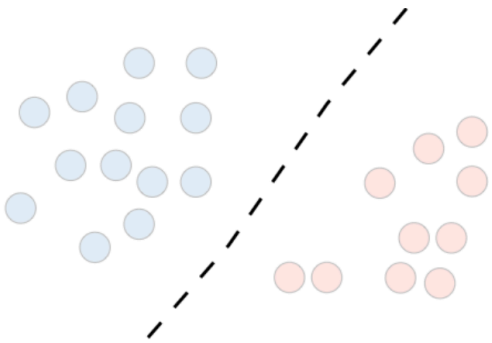
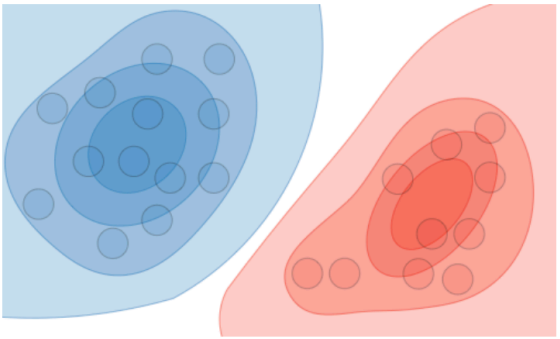
```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
├─Sequential: 1-1                        [-1, 4, 9, 9]             --
|    └─Conv2d: 2-1                       [-1, 8, 60, 60]           608
|    └─ReLU: 2-2                         [-1, 8, 60, 60]           --
|    └─MaxPool2d: 2-3                    [-1, 8, 30, 30]           --
|    └─Conv2d: 2-4                       [-1, 4, 28, 28]           292
|    └─ReLU: 2-5                         [-1, 4, 28, 28]           --
|    └─MaxPool2d: 2-6                    [-1, 4, 9, 9]             --
|    └─Dropout: 2-7                      [-1, 4, 9, 9]             --
==========================================================================
Total params: 900
Trainable params: 900
Non-trainable params: 0
Total mult-adds (M): 2.39
==========================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 0.24
Params size (MB): 0.00
Estimated Total Size (MB): 0.29
==========================================================================
```

# 1. Generative vs. Descriminative approaches

Both approaches can be used for supervised classification. But the way they think about separating classes is very different.

- **Generative** models can generate new data instances
  - Example: Generate or draw a cat
  - They build a “model” for each class; they try to capture some characterization of the distribution
  - They capture the joint probability $p(X, y)$ or just $p(X)$ (if no labels are available)
  - You predict the class based on which model becomes more likely for the given example
- **Discriminative** models descriminate between different kinds of data instances
  - Example: distinguish dog images from cat images

- They put all their effort in identifying the boundary between different classes
- They capture the conditional probability $p(y|X)$

| | Discriminative model | Generative model |
|---|---|---|
| **Goal** | Directly estimate $P(y\|x)$ | Estimate $P(x\|y)$ to then deduce $P(y\|x)$ |
| **What's learned** | Decision boundary | Probability distributions of the data |
| **Illustration** |  |  |

Source

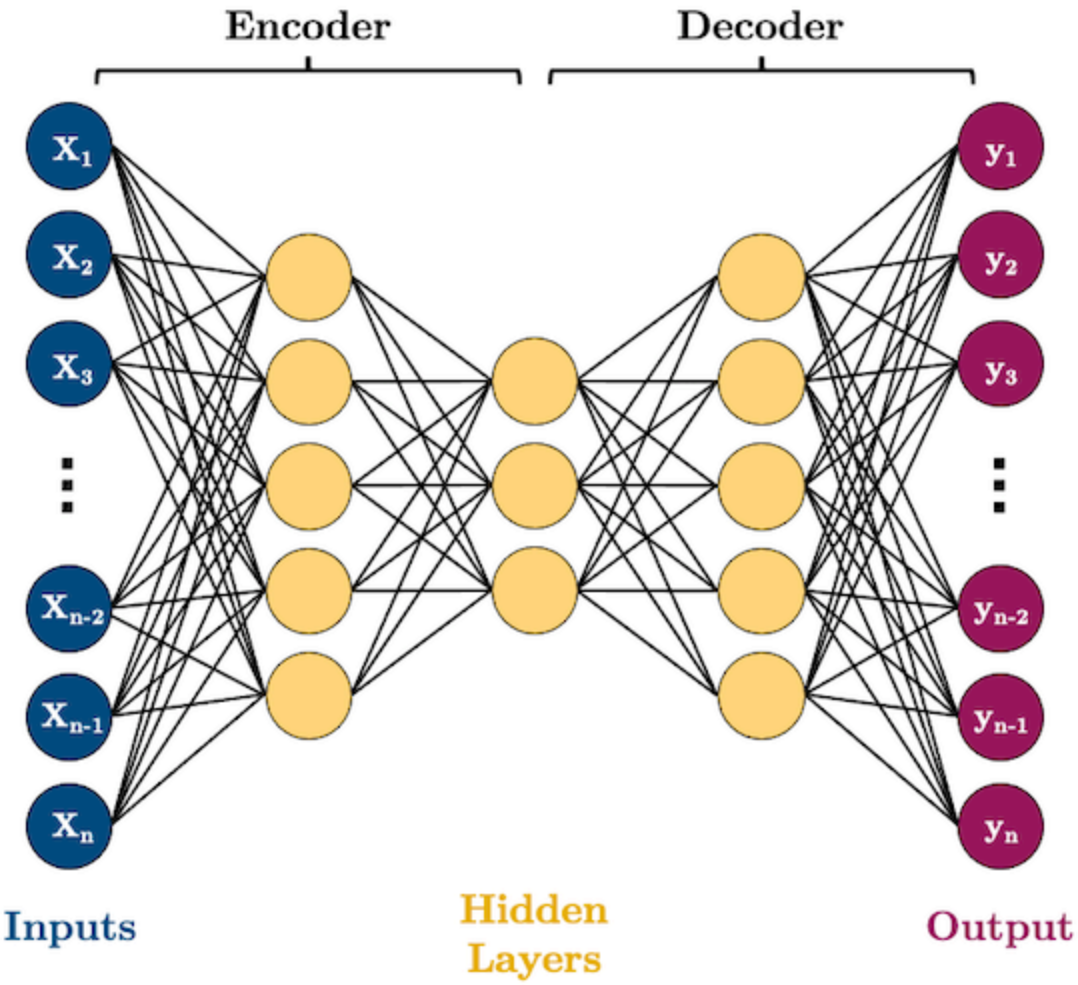
## Activity

Identify which of the following models are generative vs. discriminative

- Logistic regression
- Tree-based models
- Naive Bayes
- CNNs
- Large language models (e.g., ChatGPT)

# 2. Autoencoders

- Autoencoders (AE) are networks that are designed to reproduce their input at the output layer
- They are composed of an “encoder” and “decoder”
- The hidden layers of the AE are typically smaller than the input layers, such that the dimensionality of the data is reduced as it is passed through the encoder, and then expanded again in the decoder:

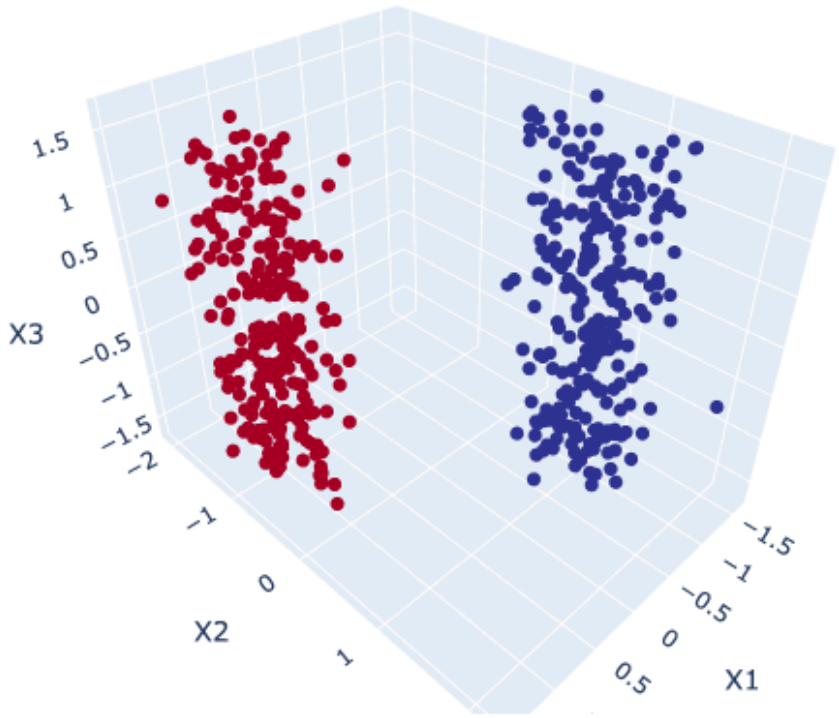



- Why would you want to use such a model? As you can see, AEs perform dimensionality reduction by learning to represent your input features using fewer dimensions
- That can be useful for a range of tasks but we'll look at some specific examples below

## 2.1. Example 1: Dimensionality Reduction

- Here's some synthetic data of 3 features and two classes
- Can we reduce the dimensionality of this data to two features while preserving the class separation?

```
n_samples = 500
X, y = make_blobs(n_samples, n_features=2, centers=2, cluster_std=1, random_st
X = np.concatenate((X, np.random.random((n_samples, 1))), axis=1)
X = StandardScaler().fit_transform(X)
plot_scatter3D(X, y)
```




- We can see that `X1` and `X2` split the data nicely, and the `X3` is just noise
- The question is, can an AE learn that this data can be nicely separated in just two of the three dimensions?
- Let's build a simple AE with the following neurons in each layer: 3 -> 2 -> 3:

```
class autoencoder(torch.nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 2),
            nn.Sigmoid()
        )
        self.decoder = nn.Sequential(
            nn.Linear(2, input_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

```
BATCH_SIZE = 100
torch.manual_seed(1)
X_tensor = torch.tensor(X, dtype=torch.float32)
dataloader = DataLoader(X_tensor,
                        batch_size=BATCH_SIZE)
model = autoencoder(3, 2)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters())
```

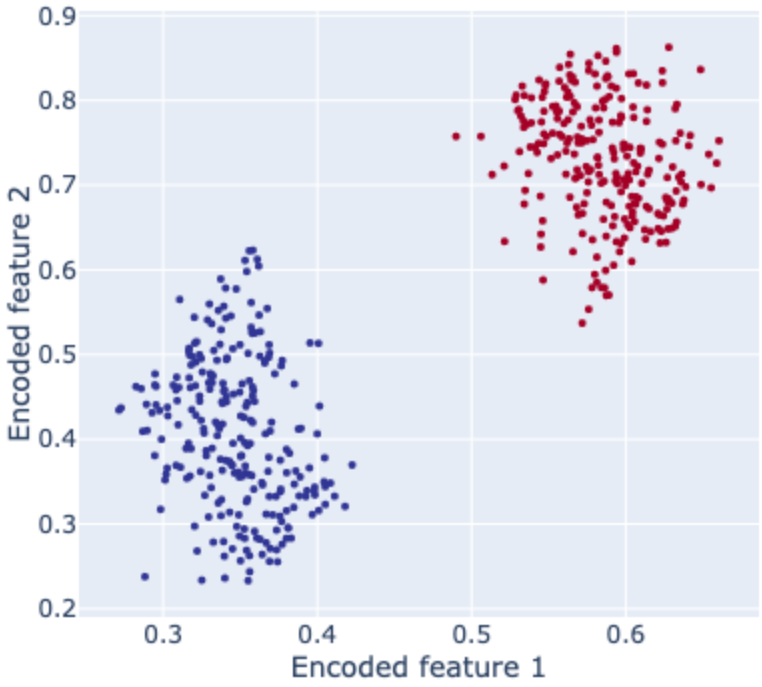
```
EPOCHS = 5

for epoch in range(EPOCHS):
    for batch in dataloader:
        optimizer.zero_grad()           # Clear gradients w.r.t. parameters
        y_hat = model(batch)            # Forward pass to get output
        loss = criterion(y_hat, batch)  # Calculate loss
        loss.backward()                 # Getting gradients w.r.t. parameters
        optimizer.step()                # Update parameters
```

- We only care about the encoder now, does it represent our data nicely in reduced dimensions?

```
model.eval()
print(f"Original X shape = {X_tensor.shape}")
X_encoded = model.encoder(X_tensor)
print(f" Encoded X shape = {X_encoded.shape}")
```

```
Original X shape = torch.Size([500, 3])
 Encoded X shape = torch.Size([500, 2])
```

```
plot_scatter2D(X_encoded, y)
```

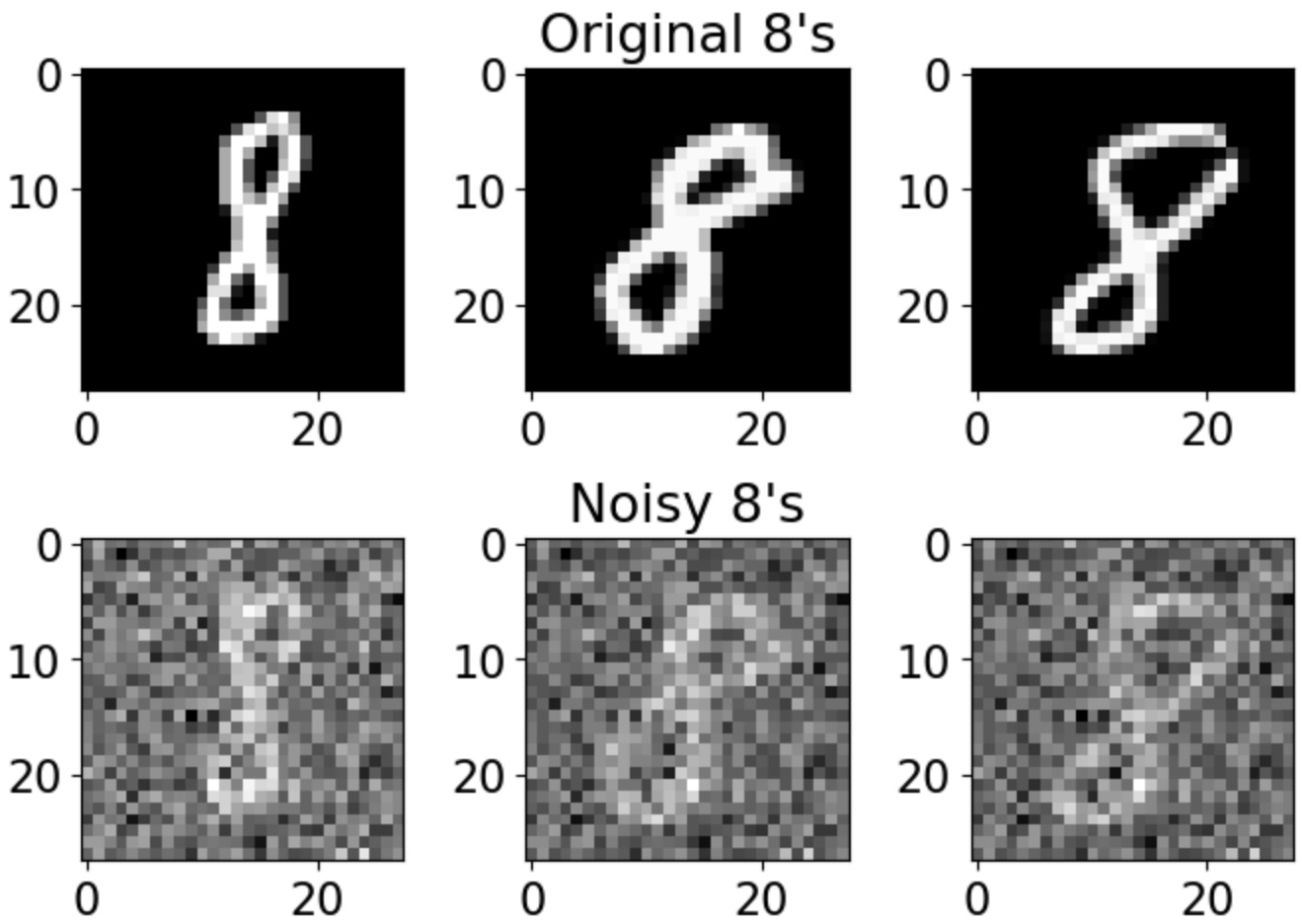



- What did we just do? We used an AE to effectively reduce the number of features in our data
- This is very similar to concepts of unsupervised learning and clustering that we'll discuss in DSCI 563.

## 2.2. Example 2: Image Denoising

- Okay, let's do something more interesting
- We saw above that AEs can be useful feature reducers (i.e., they can remove unimportant features from our data)
- This also applies to images and it's a fun application to de-noise images!
- Take a look at these images of 8's from the MNIST dataset, I'm going to mess them up by adding some noise to them:

```
BATCH_SIZE = 32

# Download data
transform = transforms.Compose([transforms.ToTensor()])
trainset = datasets.MNIST('data/', download=True, train=True, transform=transf
idx = trainset.targets == 8  # let's only work with the number 8
trainset.targets = trainset.targets[idx]
trainset.data = trainset.data[idx]
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shu

# Sample plot
X, y = next(iter(trainloader))
noise = 0.5
plot_eights(X, noise)
```




```
X.shape
```

```
torch.Size([32, 1, 28, 28])
```

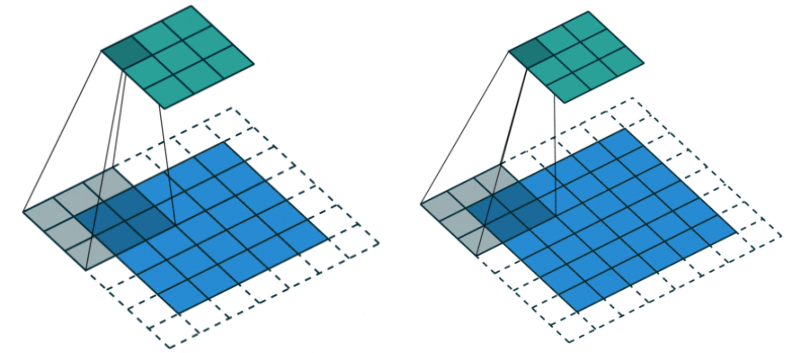
- Can we train an AE to get rid of that noise and reconstruct the original 8's? Let's give it a try!

- I'm going to use convolutional layers in my AE now as we are dealing with images
- We'll use `Conv2D()` layers to *compress* our images into a reduced dimensonality, and then we need to "upsample" it back to the original size
- One ingredient you'll need to know to do this is "transposed convolutional layers". These are just like "convolutional layers" but for the purpose of "upsampling" (increasing the size of) our data. Rather than simply expanding the size of our data and interpolating, we use `nn.ConvTranspose2d()` layers to help us learn how to best upsample our data:

## Convolution Layers

Convolution layers **downsample** input features. In other words, the goal of convolution layers is to go from **larger features (images)** to **smaller features (images)**.

Here is an animation of how kernels are applied to input features:

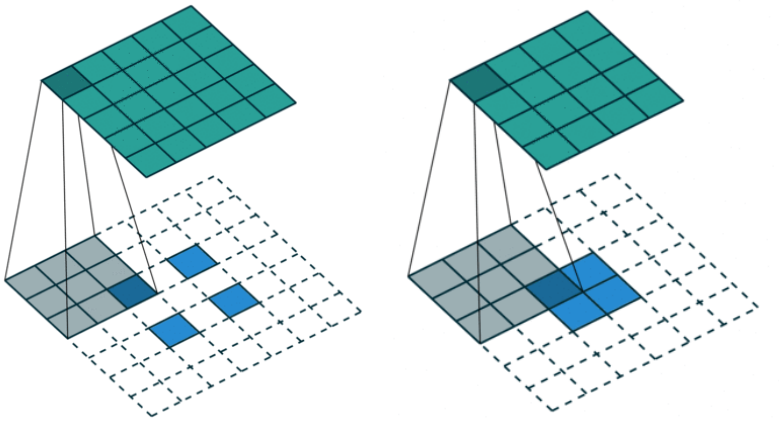


(image source)

## Transposed Convolution Layers

This is the first time we see transposed convolution layers. These layers do the opposite of what convolution layers do; that is, instead of downsampling images, they upsample. Transposed convolutions are used in the generator of a GAN to generate a image from some random vector.

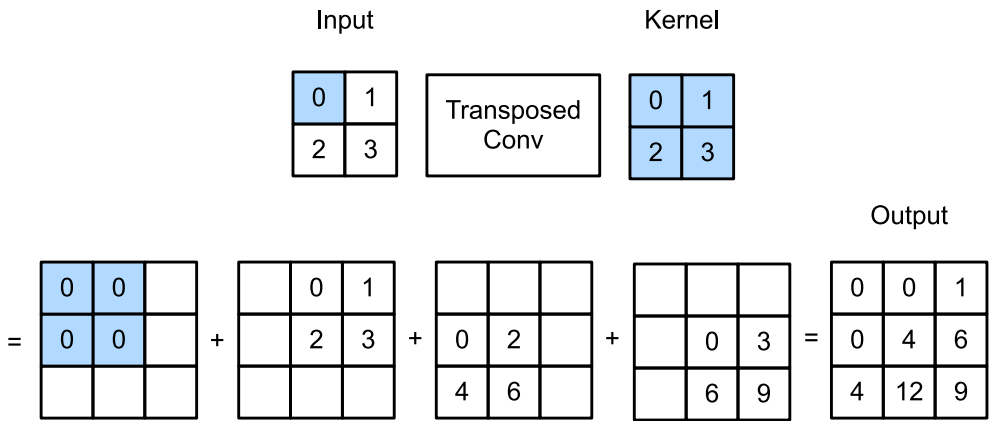
The goal of transposed convolution layers is to go from **smaller features (images)** to **larger features (images)**.



(image source)

While downsampling seems very intuitive, upsampling might look like doing magic: we try to generate information (pixel values) that did not exist before. But it's practically not hard to do.

We do something similar to what we did with convolution layers: we convolve (pass) the kernel over the inputs, and multiply each input element by all kernel elements. The resulting array will be part of the larger image:




(image source)

If we apply this operation across multiple layers, we can gradually increase the size of the input images. Later in this lecture, we'll see that this is exactly how the generator in a generative adversarial network (GAN) works; it starts with random noise and progressively transforms it into larger and more detailed images.

Let's try it out in the context of autoencoders.

```
def conv_block(input_channels, output_channels):
    return nn.Sequential(
        nn.Conv2d(input_channels, output_channels, 3, padding=1),
        nn.ReLU(),
        nn.MaxPool2d(2)  # reduce x-y dims by two; window and stride of 2
    )

def deconv_block(input_channels, output_channels, kernel_size):
    return nn.Sequential(
        nn.ConvTranspose2d(input_channels, output_channels, kernel_size, strid
        nn.ReLU()
    )

class autoencoder(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            conv_block(1, 32),
            conv_block(32, 16),
            conv_block(16, 8)
        )
        self.decoder = nn.Sequential(
            deconv_block(8, 8, 3),
            deconv_block(8, 16, 2),
            deconv_block(16, 32, 2),
            nn.Conv2d(32, 1, 3, padding=1)  # final conv layer to decrease cha
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        x = torch.sigmoid(x)  # get pixels between 0 and 1
        return x
```

```
model = autoencoder()
summary(model, (1, 28, 28));
```

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
├─Sequential: 1-1                        [-1, 8, 3, 3]             --
|    └─Sequential: 2-1                   [-1, 32, 14, 14]          --
|    |    └─Conv2d: 3-1                  [-1, 32, 28, 28]          320
|    |    └─ReLU: 3-2                    [-1, 32, 28, 28]          --
|    |    └─MaxPool2d: 3-3               [-1, 32, 14, 14]          --
|    └─Sequential: 2-2                   [-1, 16, 7, 7]            --
|    |    └─Conv2d: 3-4                  [-1, 16, 14, 14]          4,624
|    |    └─ReLU: 3-5                    [-1, 16, 14, 14]          --
|    |    └─MaxPool2d: 3-6               [-1, 16, 7, 7]            --
|    └─Sequential: 2-3                   [-1, 8, 3, 3]             --
|    |    └─Conv2d: 3-7                  [-1, 8, 7, 7]             1,160
|    |    └─ReLU: 3-8                    [-1, 8, 7, 7]             --
|    |    └─MaxPool2d: 3-9               [-1, 8, 3, 3]             --
├─Sequential: 1-2                        [-1, 1, 28, 28]           --
|    └─Sequential: 2-4                   [-1, 8, 7, 7]             --
|    |    └─ConvTranspose2d: 3-10        [-1, 8, 7, 7]             584
|    |    └─ReLU: 3-11                   [-1, 8, 7, 7]             --
|    └─Sequential: 2-5                   [-1, 16, 14, 14]          --
|    |    └─ConvTranspose2d: 3-12        [-1, 16, 14, 14]          528
|    |    └─ReLU: 3-13                   [-1, 16, 14, 14]          --
|    └─Sequential: 2-6                   [-1, 32, 28, 28]          --
|    |    └─ConvTranspose2d: 3-14        [-1, 32, 28, 28]          2,080
|    |    └─ReLU: 3-15                   [-1, 32, 28, 28]          --
|    └─Conv2d: 2-7                       [-1, 1, 28, 28]           289
==========================================================================
Total params: 9,585
Trainable params: 9,585
Non-trainable params: 0
Total mult-adds (M): 3.16
==========================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.44
Params size (MB): 0.04
Estimated Total Size (MB): 0.48
==========================================================================
```

The formula to calculate the shape of the output map is:

$$\text{Output size} = (I - 1) \times S - 2P + K + \text{Output Padding}$$

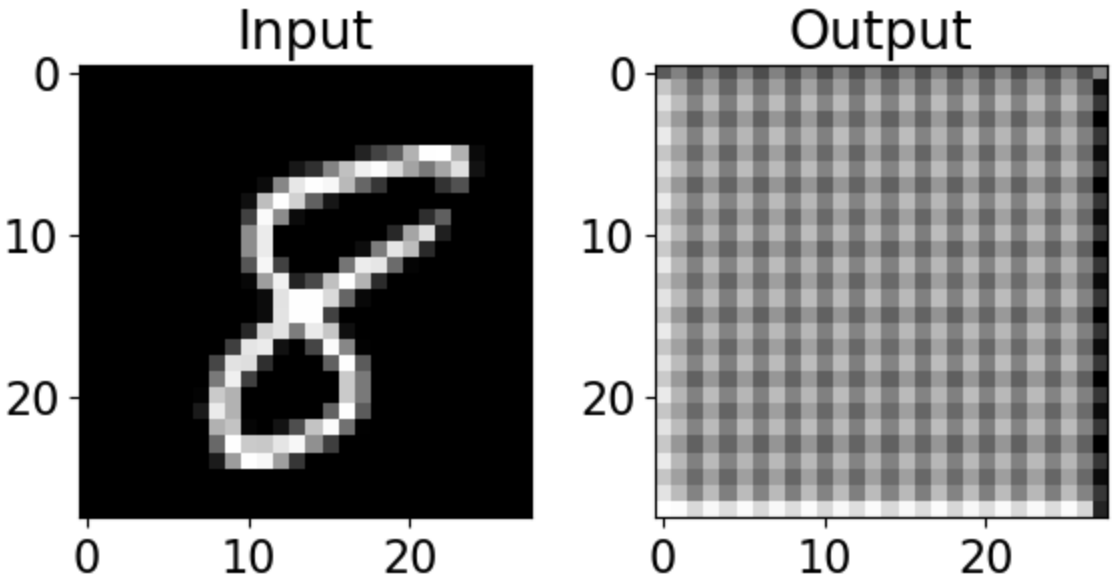
Where

- $I \rightarrow$ Input size (height or width)
- $S \rightarrow$ Stride
- $P \rightarrow$ Padding (used on the input)
- $K \rightarrow$ Kernel size

- Output Padding → Additional padding added to control output size (default is 0 in torch.nn.ConvTranspose2d)

- So we want to train our model to remove that noise I added
- Generally speaking, the idea is that the model learns what pixel values are important, we are reducing the dimensionality of the imaages, so our model must learn only the crucial information (i.e., not the noise) needed to reproduce the image
- Right now, our model probably produces gibberish because it isn't trained:

```
model = autoencoder()
input_8 = X[:1, :1, :, :]
output_8 = model(input_8)
plot_eight_pair(input_8, output_8)
```




- How do we train it?
- Well we feed in a noisy image, compare it to the non-noisy version, and let the network learn how to make that happen
- We want the value of the predicted pixels to be as close as possible to the real pixel values, so we'll use `MSELoss()` as our loss function:

```
```

```
EPOCHS = 20
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
img_list = []

def train_denoising_autoencoder(model, trainloader, noise, criterion, optimize
    for epoch in range(EPOCHS):
        losses = 0
        for batch, _ in trainloader:
            noisy_batch = batch + noise * torch.randn(*batch.shape)
            noisy_batch = torch.clip(noisy_batch, 0.0, 1.0)
            optimizer.zero_grad()
            y_hat = model(noisy_batch)
            loss = criterion(y_hat, batch)
            loss.backward()
            optimizer.step()
            losses += loss.item()
        print(f"epoch: {epoch + 1}, loss: {losses / len(trainloader):.4f}")
        # Save example results each epoch so we can see what's going on
        with torch.no_grad():
            noisy_8 = noisy_batch[:1, :1, :, :]
            model_8 = model(input_8)
            real_8 = batch[:1, :1, :, :]
        img_list.append(utils.make_grid([noisy_8[0], model_8[0], real_8[0]], p

# train_denoising_autoencoder(model, trainloader, noise, criterion, optimizer,
```

```
epoch: 1, loss: 0.0305
epoch: 2, loss: 0.0302
epoch: 3, loss: 0.0301
epoch: 4, loss: 0.0298
epoch: 5, loss: 0.0296
epoch: 6, loss: 0.0294
epoch: 7, loss: 0.0293
epoch: 8, loss: 0.0290
epoch: 9, loss: 0.0289
epoch: 10, loss: 0.0287
epoch: 11, loss: 0.0286
epoch: 12, loss: 0.0285
epoch: 13, loss: 0.0283
epoch: 14, loss: 0.0282
epoch: 15, loss: 0.0282
epoch: 16, loss: 0.0281
epoch: 17, loss: 0.0280
epoch: 18, loss: 0.0279
epoch: 19, loss: 0.0279
epoch: 20, loss: 0.0276
```

```
save_path = '../models/denoising_autoencoder.pth'
# torch.save(model.state_dict(), save_path)
```

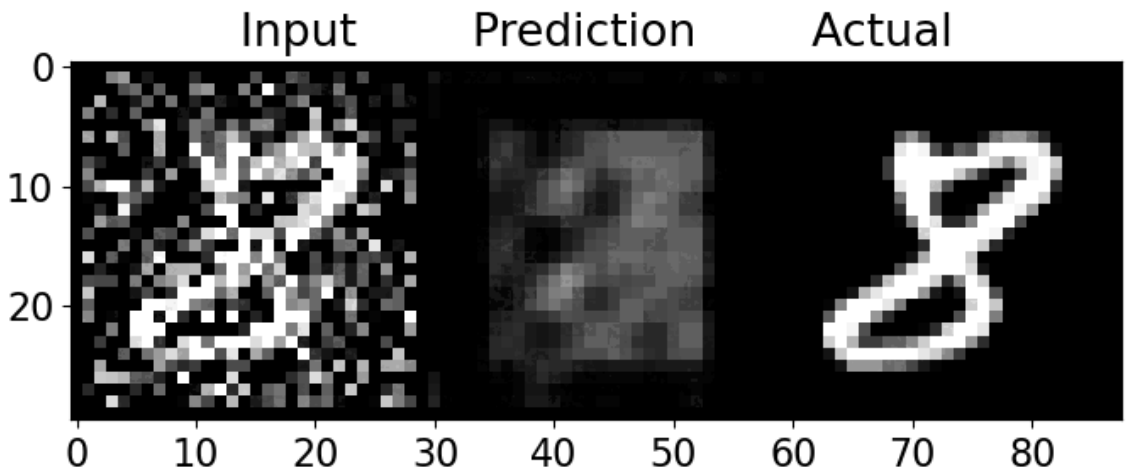
```
model.load_state_dict(torch.load(save_path))
```

```
/var/folders/b3/g26r0dcx4b35vf3nk31216hc0000gr/T/ipykernel_74872/8766937.py:1:

You are using `torch.load` with `weights_only=False` (the current default valu
```

```
<All keys matched successfully>
```

```
%%capture
fig, ax = plt.subplots(1, 1, figsize=(8, 5))
ax.set_title("Input        Prediction        Actual")
ims = [[plt.imshow(np.transpose(i, (1,2,0)), animated=True)] for i in img_list
```

```
# ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,
# ani.save('eights.gif', writer='imagemagick', fps=2)
# HTML(ani.to_jshtml()) # run this in a new cell to produce the below animatio
```




- Pretty cool!

# Course evaluations (~10 minutes)

The online course evaluations are up, it'll be great if you can fill them in when you get a chance: https://canvas.ubc.ca/courses/153823/external_tools/53187

- We know you have busy scheduled and many evaluations to complete, but your feedback is valuable in improving our courses.
- It'll be great to know what worked for you and what could be improved in this course.
- Other than helping us in improving our courses and teaching style, they also play an important role in our careers, promotions etc.

# 3. Generative Adversarial Networks (GANs)

## 3.1 What are GANs?

GANs are a type of neural network models that are used to generate new data, that is indistinguishable from the data that exists in a dataset.

For example, suppose that we have a dataset of 10,000 images. The question is: can we somehow generate images that are so real-looking that we can't tell if they are **real** (could potentially come from the dataset) or **fake** (generated by some algorithm)?

Here, there aren't any real labels. We just want to be able to produce images that are as real-looking as possible; we don't classify them. This is why **GAN modeling** is regarded as an **unsupervised learning** task. In other words, we just need a bunch of images (or input data),

no labels (or target data or outputs) would be required. We will learn more on unsupervised learning in DSCI 563.

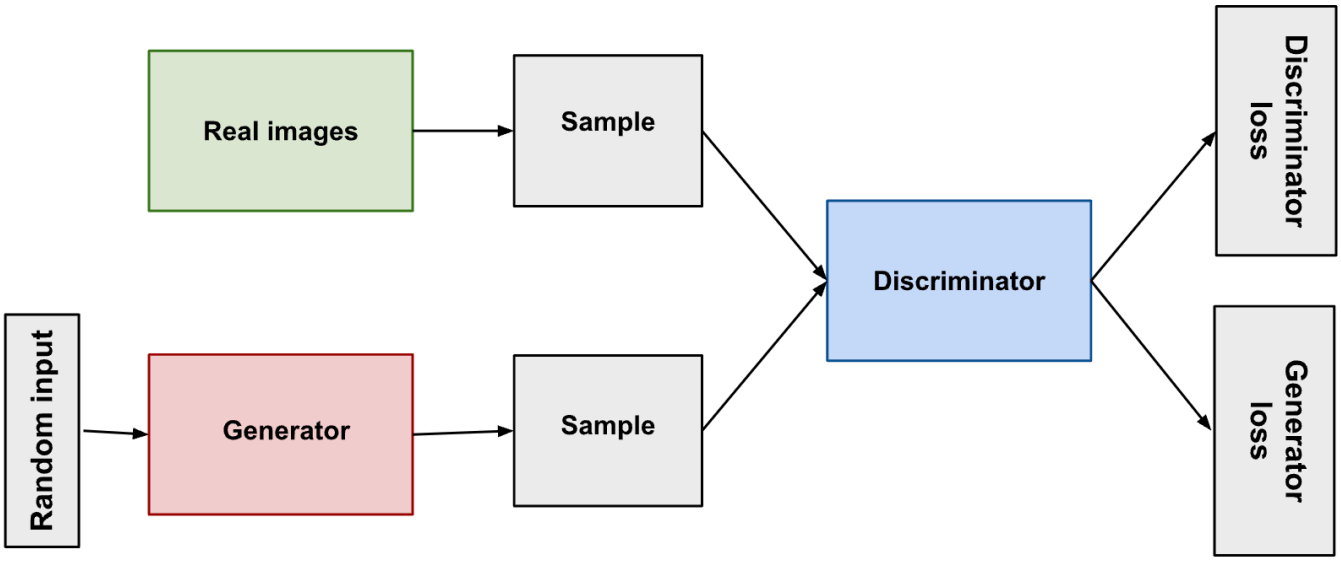
GANs were invented in 2014 by Ian Goodfellow and colleagues (see the original paper here); and have been called "*the most interesting idea in the last 10 years in ML*" by Yann LeCun, Facebook's AI research director.

Now take a look at the following image:

Believe it or not, **this is not a real person!**

The image above is produced by a GAN that is trained on human faces. If you want to see more, visit www.thispersondoesnotexist.com. The website is connected to a GAN model living on the cloud, and each time the page refreshes, it generates a new image of a person who **does not exist!**.

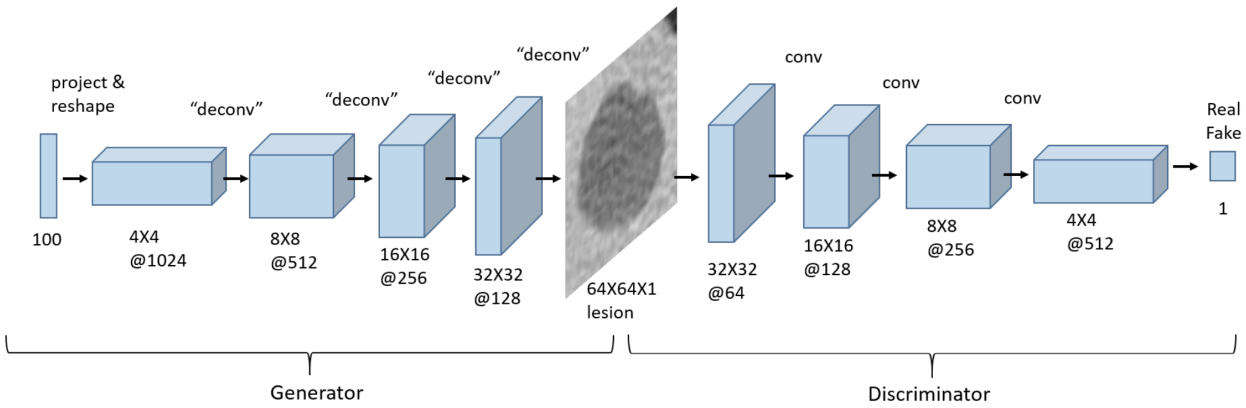
# 3.2 Structure of a GAN




Source

In this section, I describe how GANs work for image data, but remember that the idea of GANs is generalizable to any kind of data, not just images.

Here is the visualization of the structure of a GAN:




(image source)

The structure of a GAN consists of a **discriminator** and a **generator**:

- A discriminator is just a **typical CNN** that receives an image as the input, and generates a vector of probabilities of the input belonging to some class
- A generator is an **inverted CNN** that receives a vector of random numbers and generates an image in the output

The word “adversarial” comes from the fact that we actually have two networks battling each other:

- The generator: tries to generate fake images that look as realistic as possible such that it can **fool** the discriminator
- The discriminator: takes in real data and fake data and tries to correctly determine whether an input was real or fake

**An analogy:**

Think of the “Generator” as a new counterfeit artist trying to produce realistic-looking famous artworks to sell.

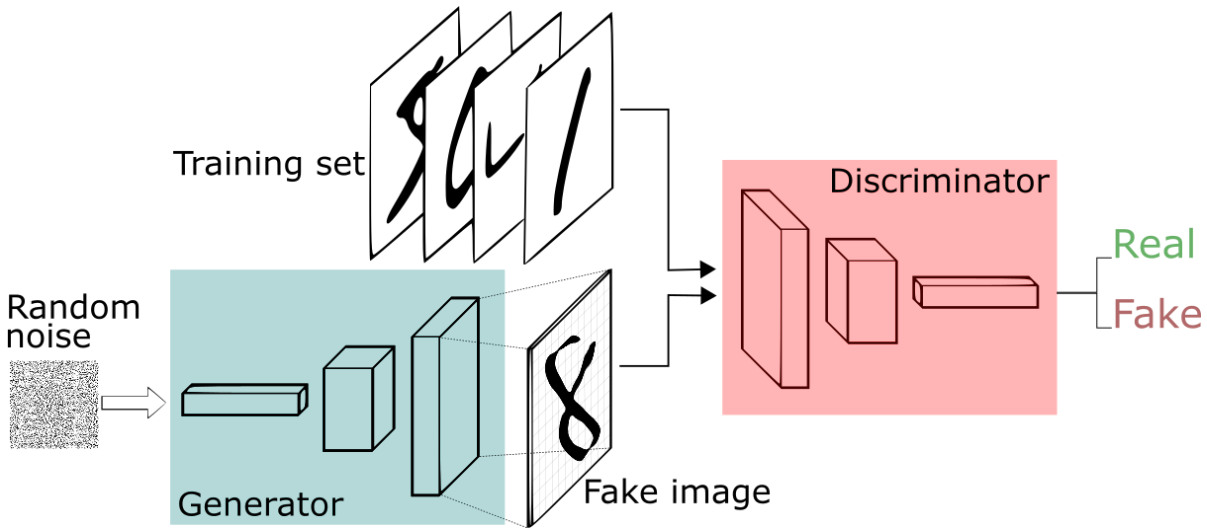
The “Discriminator” is an art critic, trying to determine if a piece of art is “real” or “fake”.

At first, the “Generator” produces poor art-replicas which the “Discriminator” can easily tell are fake. But over time, the “Generator” learns ways to produce art that fools the “Discriminator”. Eventually, the “Generator” becomes so good that the “Discriminator” can’t tell if a piece of art is real or fake.

# 3.3 Training GANs

Training a GAN happens in two iterative phases:

1. **Train the Discriminator:**
    - Generate some fake images with the generator
    - Show the discriminator real images and fake images and get it to classify them correctly (a simple binary classification problem)
2. **Train the Generator:**
    - Generate fake images with the generator but label them as “real”
    - Pass these fake images through the discriminator, and ask it for its judgment, i.e. the probability of this image being real
    - Pass this judgment to a loss function, and see how far it is from the ideal output. The ideal output is that the generator was so good that it has fooled the discriminator to give it the label of “real”.
    - Do backpropagation based on the gradients of this loss value to adjust the parameters of the generator, such that it can better and better fool the discriminator.
3. **Repeat**.




(image source)

# 3.4 PyTorch Implementation

Alright, now's the time to implement a GAN in PyTorch. Since training GANs is a very resource-intensive job, we need to do our computations on a GPU. Here, I'll write the code **so that you can take this notebook and run it directly on Kaggle**. If you want to run it on your own computer, you need to change the folder paths of the dataset that we're going to use.

```
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
print(f"Using device: {device.type}")
```

```
Using device: mps
```

For the purpose of demonstrating how training a GAN works in PyTorch, we'll use the Face Recognition Dataset from Kaggle, which contains face images of celebrities.

This dataset contains two folders: `Face Dataset` and `Extracted Faces`. We'll use the images in `Extracted Faces`, which are already `128x128` pixels. Therefore, there is no need to resize them, which is why I've commented out `transforms.Resize(IMAGE_SIZE)` in the code below. This speeds up the computations significantly, as resizing is done on CPU and it would have been the bottle-neck of our computations. Fortunately, we don't need to do this here:

```
DATA_DIR = "/kaggle/input/face-recognition-dataset/Extracted Faces/Extracted F

BATCH_SIZE = 64
IMAGE_SIZE = (128, 128)

data_transforms = transforms.Compose([
#     transforms.Resize(IMAGE_SIZE),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = datasets.ImageFolder(root=DATA_DIR, transform=data_transforms)
data_loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE, shuf

# Plot samples
sample_batch = next(iter(data_loader))
plt.figure(figsize=(8, 8)); plt.axis("off"); plt.title("Sample Training Images
plt.imshow(np.transpose(utils.make_grid(sample_batch[0], padding=1, normalize=

print(f'Size of dataset: {len(data_loader) * BATCH_SIZE}')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[24], line 12
      4 IMAGE_SIZE = (128, 128)
      6 data_transforms = transforms.Compose([
      7 #     transforms.Resize(IMAGE_SIZE),
      8     transforms.ToTensor(),
      9     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
     10 ])
---> 12 dataset = datasets.ImageFolder(root=DATA_DIR, transform=data_transforms
     13 data_loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SI
     15 # Plot samples

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torchvision/datasets/
    319 def __init__(
    320     self,
    321     root: str,
   (...)
    326     allow_empty: bool = False,
    327 ):
--> 328     super().__init__(
    329         root,
    330         loader,
    331         IMG_EXTENSIONS if is_valid_file is None else None,
    332         transform=transform,
    333         target_transform=target_transform,
    334         is_valid_file=is_valid_file,
    335         allow_empty=allow_empty,
    336     )
    337     self.imgs = self.samples

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torchvision/datasets/
    138 def __init__(
    139     self,
    140     root: Union[str, Path],
   (...)
    146     allow_empty: bool = False,
    147 ) -> None:
    148     super().__init__(root, transform=transform, target_transform=target
--> 149     classes, class_to_idx = self.find_classes(self.root)
    150     samples = self.make_dataset(
    151         self.root,
    152         class_to_idx=class_to_idx,
   (...)
    155         allow_empty=allow_empty,
    156     )
    158     self.loader = loader

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torchvision/datasets/
    207 def find_classes(self, directory: Union[str, Path]) -> Tuple[List[str],
    208     """Find the class folders in a dataset structured as follows::
    209
    210         directory/
   (...)
    232         (Tuple[List[str], Dict[str, int]]): List of all classes and di
```

```
    233     """
--> 234     return find_classes(directory)

File ~/miniforge3/envs/jbook/lib/python3.12/site-packages/torchvision/datasets/
     36 def find_classes(directory: Union[str, Path]) -> Tuple[List[str], Dict
     37     """Finds the class folders in a dataset.
     38 
     39     See :class:`DatasetFolder` for details.
     40     """
---> 41     classes = sorted(entry.name for entry in os.scandir(directory) if 
     42     if not classes:
     43         raise FileNotFoundError(f"Couldn't find any class folder in {d

FileNotFoundError: [Errno 2] No such file or directory: '/kaggle/input/face-re
```

Example output:

### 3.4.1 Creating the Generator

The generator takes in a random vector called **latent vector**, which can be thought of as a `1x1` pixel image having an arbitrary number of **channels** (specified in the code by `LATENT_SIZE`).

Through the generator, we pass this latent vector through the **deconvolution** layers (known as **transposed convolution** layers in PyTorch), and progressively expand its size, such that in the output we'll have an image similar in dimensions to the images in our dataset. Here for example, our images in the dataset are `128x128`, so the goal of the generator is to start from an image of `1x1` pixel and generate an image of `128x128` pixels.

**Details:**

- In the following code, I've used `nn.BatchNorm2d()` for all layers, `nn.LeakyReLU()` as activation for intermediate layers, and `nn.Tanh()` as activation for the output of the generator. These are suggested to be used based on empirical evidence in training GANs.
- We usually do in-place modification of tensors in `nn.LeakyReLU()` by setting `inplace=True` to save some memory.
- We set `bias=False` because the batch normalization layer contains a bias term, so we don't want to do it twice.

Note that we mainly play with the strides to progressively expand the size of the input latent vector.

Here is the code for the generator:

```
class Generator(nn.Module):

    def __init__(self, LATENT_SIZE):
        super(Generator, self).__init__()

        self.main = nn.Sequential(

            # input dim: [-1, LATENT_SIZE, 1, 1]
            # output size = (1 - 1)*1 - 2*0 + 4

            nn.ConvTranspose2d(LATENT_SIZE, 1024, kernel_size=4, stride=1, pad
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 1024, 4, 4]
            # output size = (input size - 1)*stride - 2*padding + kernel size
            # (4 - 1)*2 - 2 * 1 + 4 = 4 + 4 = 8

            nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, padding=1,
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 512, 8, 8]

            nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, b
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 256, 16, 16]

            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, b
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 128, 32, 32]

            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bi
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),

            # output dim: [-1, 64, 64, 64]

            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias

            # output dim: [-1, 3, 128, 128]

            nn.Tanh()
        )

    def forward(self, input):
        output = self.main(input)
        return output
```

```
LATENT_SIZE = 200

generator = Generator(LATENT_SIZE).to(device)
summary(generator, (LATENT_SIZE, 1, 1));
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 3, 128, 128]         --
|    └─ConvTranspose2d: 2-1              [-1, 1024, 4, 4]          3,276,800
|    └─BatchNorm2d: 2-2                  [-1, 1024, 4, 4]          2,048
|    └─LeakyReLU: 2-3                    [-1, 1024, 4, 4]          --
|    └─ConvTranspose2d: 2-4              [-1, 512, 8, 8]           8,388,608
|    └─BatchNorm2d: 2-5                  [-1, 512, 8, 8]           1,024
|    └─LeakyReLU: 2-6                    [-1, 512, 8, 8]           --
|    └─ConvTranspose2d: 2-7              [-1, 256, 16, 16]         2,097,152
|    └─BatchNorm2d: 2-8                  [-1, 256, 16, 16]         512
|    └─LeakyReLU: 2-9                    [-1, 256, 16, 16]         --
|    └─ConvTranspose2d: 2-10             [-1, 128, 32, 32]         524,288
|    └─BatchNorm2d: 2-11                 [-1, 128, 32, 32]         256
|    └─LeakyReLU: 2-12                   [-1, 128, 32, 32]         --
|    └─ConvTranspose2d: 2-13             [-1, 64, 64, 64]          131,072
|    └─BatchNorm2d: 2-14                 [-1, 64, 64, 64]          128
|    └─LeakyReLU: 2-15                   [-1, 64, 64, 64]          --
|    └─ConvTranspose2d: 2-16             [-1, 3, 128, 128]         3,072
|    └─Tanh: 2-17                        [-1, 3, 128, 128]         --
==========================================================================================
Total params: 14,424,960
Trainable params: 14,424,960
Non-trainable params: 0
Total mult-adds (G): 2.26
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 8.12
Params size (MB): 55.03
Estimated Total Size (MB): 63.15
==========================================================================================
```

The details of how exactly a transposed convolution layer works in PyTorch can be fairly confusing at first, but here's some further remarks to help you feel more comfortable with it:

The parameters `stride` and `padding` in `nn.ConvTranspose2d` are (unfortunately?) not what we're used to in using `nn.Conv2d`. For example, `stride=2` doesn't mean that the kernel in the transposed convolution moves in steps of 2 pixels each time. These parameters

are, instead, designed such that if you use the same stride and padding for a `ConvTranspose2d` as in a `Conv2d`, and apply it on the output of the that `Conv2d`, it will give you an image of the same shape (but not the same pixel values).

If you're wondering about the mechanics of computing a transposed convolution in various scenarios, **make sure to check out this article or this blog post.**

## 3.4.2 Creating the Discriminator

As discussed before, this is a conventional CNN that receives an image (`128x128` in our case here) and outputs the probability of this image belonging to some certain class:

- Using pooling layers is less common because they reduce spatial resolution of feature maps and can result in loss of fine-grained details
- It's more common to use strided convolutions instead
- We usually do in-place modification of tensors in `nn.LeakyReLU()` by setting `inplace=True` to save some memory.
- We set `bias=False` because the batch normalization layer contains a bias term, so we don't want to do it twice.

```
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator, self).__init__()

        self.main = nn.Sequential(

            # input dim: [-1, 3, 128, 128]

            nn.Conv2d(3, 32, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.2),

            # output dim: [-1, 32, 64, 64]

            nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.2),

            # output dim: [-1, 64, 32, 32]

            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.2),

            # output dim: [-1, 128, 16, 16]

            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.2),

            # output dim: [-1, 256, 8, 8]

            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.2),

            # output dim: [-1, 512, 4, 4]

            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0),

            # output dim: [-1, 1, 1, 1]

            nn.Flatten()
        )

    def forward(self, input):
        output = self.main(input)
        return output
```

```
discriminator = Discriminator().to(device)
summary(discriminator, (3, 128, 128));
```

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
├─Sequential: 1-1                        [-1, 1]                   --
|    └─Conv2d: 2-1                       [-1, 32, 64, 64]          1,536
|    └─BatchNorm2d: 2-2                  [-1, 32, 64, 64]          64
|    └─LeakyReLU: 2-3                    [-1, 32, 64, 64]          --
|    └─Dropout: 2-4                      [-1, 32, 64, 64]          --
|    └─Conv2d: 2-5                       [-1, 64, 32, 32]          32,768
|    └─BatchNorm2d: 2-6                  [-1, 64, 32, 32]          128
|    └─LeakyReLU: 2-7                    [-1, 64, 32, 32]          --
|    └─Dropout: 2-8                      [-1, 64, 32, 32]          --
|    └─Conv2d: 2-9                       [-1, 128, 16, 16]         131,072
|    └─BatchNorm2d: 2-10                 [-1, 128, 16, 16]         256
|    └─LeakyReLU: 2-11                   [-1, 128, 16, 16]         --
|    └─Dropout: 2-12                     [-1, 128, 16, 16]         --
|    └─Conv2d: 2-13                      [-1, 256, 8, 8]           524,288
|    └─BatchNorm2d: 2-14                 [-1, 256, 8, 8]           512
|    └─LeakyReLU: 2-15                   [-1, 256, 8, 8]           --
|    └─Dropout: 2-16                     [-1, 256, 8, 8]           --
|    └─Conv2d: 2-17                      [-1, 512, 4, 4]           2,097,152
|    └─BatchNorm2d: 2-18                 [-1, 512, 4, 4]           1,024
|    └─LeakyReLU: 2-19                   [-1, 512, 4, 4]           --
|    └─Dropout: 2-20                     [-1, 512, 4, 4]           --
|    └─Conv2d: 2-21                      [-1, 1, 1, 1]             8,193
|    └─Flatten: 2-22                     [-1, 1]                   --
==========================================================================
Total params: 2,796,993
Trainable params: 2,796,993
Non-trainable params: 0
Total mult-adds (M): 143.31
==========================================================================
Input size (MB): 0.19
Forward/backward pass size (MB): 3.88
Params size (MB): 10.67
Estimated Total Size (MB): 14.73
==========================================================================
```

## Initializing our GAN

Let's create the discriminator and generator objects, as well as the loss function and optimizers:

```
generator.to(device)
discriminator.to(device)

criterion = nn.BCEWithLogitsLoss()

optimizerG = optim.Adam(generator.parameters(), lr=0.001, betas=(0.5, 0.999))
optimizerD = optim.Adam(discriminator.parameters(), lr=0.001, betas=(0.5, 0.99
```

We explored how the starting point of the optimization can affect the final results. It is recommended that to initialize the weights of a GAN with values obtained randomly from a normal distribution. In PyTorch, we can define the initialization function as we like and apply it to the model parameters using the `.apply()` method:

```
def weights_init(m):
    if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif isinstance(m, nn.BatchNorm2d):
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

generator.apply(weights_init)
discriminator.apply(weights_init);
```

## 3.4.1 Training our GAN

We use the following cell to keep track of how a fixed noise (latent vector) is transformed to a generated image in each epoch. We will see that the generations become better and better throughout the epochs:

```
img_list = []
fixed_noise = torch.randn(BATCH_SIZE, LATENT_SIZE, 1, 1).to(device)
```

Finally, here is the training loop (you can also put everything inside a function):

```
NUM_EPOCHS = 40
from statistics import mean
print('Training started:\n')

D_real_epoch, D_fake_epoch, loss_dis_epoch, loss_gen_epoch = [], [], [], []

for epoch in range(NUM_EPOCHS):

    # D_real_iter: list that accumulates the average output of the discriminat
    # for real images for each batch. High values mean the discriminator is mo
    # confidently classifying real images as real.

    # D_fake_iter: Similar to D_real_iter but for fake images. Low values indi
    # the descriminator is confidently classifying them as fake.

    # loss_dis_iter: Stores the discriminator loss for each batch.
    # This is the sum of the loss on real images and the loss on fake images.
    # Monitoring this helps understand how well the discriminator is
    # differentiating between real and fake images.

    # loss_gen_iter: Stores the generator loss for each batch. This indicates
    # how well the generator is fooling the discriminator.

    D_real_iter, D_fake_iter, loss_dis_iter, loss_gen_iter = [], [], [], []

    for real_batch, _ in data_loader:

        # STEP 1: train discriminator
        # ================================
        # Train with real data
        optimizerD.zero_grad()

        real_batch = real_batch.to(device)
        real_labels = torch.ones((real_batch.shape[0],), dtype=torch.float).to

        output = discriminator(real_batch).view(-1)
        loss_real = criterion(output, real_labels)

        # Iteration book-keeping
        D_real_iter.append(torch.sigmoid(output).mean().item())

        # Train with fake data
        noise = torch.randn(real_batch.shape[0], LATENT_SIZE, 1, 1).to(device)

        fake_batch = generator(noise)
        fake_labels = torch.zeros_like(real_labels)

        output = discriminator(fake_batch.detach()).view(-1)
        loss_fake = criterion(output, fake_labels)

        # Update discriminator weights
        loss_dis = loss_real + loss_fake
        loss_dis.backward()
        optimizerD.step()
```

```
        # Iteration book-keeping
        loss_dis_iter.append(loss_dis.mean().item())
        D_fake_iter.append(torch.sigmoid(output).mean().item())

        # STEP 2: train generator
        # ================================
        optimizerG.zero_grad()

        # Calculate the output with the updated weights of the discriminator
        output = discriminator(fake_batch).view(-1)
        loss_gen = criterion(output, real_labels)
        loss_gen.backward()

        # Book-keeping
        loss_gen_iter.append(loss_gen.mean().item())

        # Update generator weights and store loss
        optimizerG.step()

    print(f"Epoch ({epoch + 1}/{NUM_EPOCHS})\t",
          f"Loss_G: {mean(loss_gen_iter):.4f}",
          f"Loss_D: {mean(loss_dis_iter):.4f}\t",
          f"D_real: {mean(D_real_iter):.4f}",
          f"D_fake: {mean(D_fake_iter):.4f}")

    # Epoch book-keeping
    loss_gen_epoch.append(mean(loss_gen_iter))
    loss_dis_epoch.append(mean(loss_dis_iter))
    D_real_epoch.append(mean(D_real_iter))
    D_fake_epoch.append(mean(D_fake_iter))

    # Keeping track of the evolution of a fixed noise latent vector
    with torch.no_grad():
        fake_images = generator(fixed_noise).detach().cpu()
        #img_list.append(utils.make_grid(fake_images, normalize=True, nrows=10

print("\nTraining ended.")
```

Example output:

```
Training started:

Epoch (1/40)     Loss_G: 8.8666 Loss_D: 2.2696   D_real: 0.7287 D_fake: 0.3082
Epoch (2/40)     Loss_G: 2.6550 Loss_D: 1.4906   D_real: 0.5930 D_fake: 0.4089
Epoch (3/40)     Loss_G: 2.3996 Loss_D: 1.4997   D_real: 0.5955 D_fake: 0.4059
Epoch (4/40)     Loss_G: 2.0674 Loss_D: 1.2792   D_real: 0.6034 D_fake: 0.3967
Epoch (5/40)     Loss_G: 2.3533 Loss_D: 1.3387   D_real: 0.6110 D_fake: 0.3960
Epoch (6/40)     Loss_G: 2.5160 Loss_D: 1.2768   D_real: 0.6145 D_fake: 0.3881
Epoch (7/40)     Loss_G: 2.5918 Loss_D: 1.3441   D_real: 0.6048 D_fake: 0.4016
Epoch (8/40)     Loss_G: 1.9902 Loss_D: 1.3769   D_real: 0.5607 D_fake: 0.4271
Epoch (9/40)     Loss_G: 2.0522 Loss_D: 1.2786   D_real: 0.5900 D_fake: 0.4136
Epoch (10/40)    Loss_G: 2.0852 Loss_D: 1.3246   D_real: 0.5846 D_fake: 0.4141
Epoch (11/40)    Loss_G: 2.0159 Loss_D: 1.3148   D_real: 0.5829 D_fake: 0.4147
Epoch (12/40)    Loss_G: 2.1032 Loss_D: 1.3134   D_real: 0.5828 D_fake: 0.4189
Epoch (13/40)    Loss_G: 2.1085 Loss_D: 1.3120   D_real: 0.5849 D_fake: 0.4106
Epoch (14/40)    Loss_G: 1.9567 Loss_D: 1.3210   D_real: 0.5778 D_fake: 0.4210
Epoch (15/40)    Loss_G: 2.1885 Loss_D: 1.2520   D_real: 0.6003 D_fake: 0.4022
Epoch (16/40)    Loss_G: 2.2106 Loss_D: 1.2717   D_real: 0.5913 D_fake: 0.4037
Epoch (17/40)    Loss_G: 2.2239 Loss_D: 1.2087   D_real: 0.6055 D_fake: 0.3884
Epoch (18/40)    Loss_G: 2.3172 Loss_D: 1.1886   D_real: 0.6161 D_fake: 0.3769
Epoch (19/40)    Loss_G: 2.4606 Loss_D: 1.2104   D_real: 0.6153 D_fake: 0.3805
Epoch (20/40)    Loss_G: 2.4683 Loss_D: 1.1270   D_real: 0.6342 D_fake: 0.3621
Epoch (21/40)    Loss_G: 2.3587 Loss_D: 1.1867   D_real: 0.6272 D_fake: 0.3721
Epoch (22/40)    Loss_G: 2.4577 Loss_D: 1.1671   D_real: 0.6211 D_fake: 0.3714
Epoch (23/40)    Loss_G: 2.4454 Loss_D: 1.1479   D_real: 0.6343 D_fake: 0.3625
Epoch (24/40)    Loss_G: 2.4671 Loss_D: 1.1206   D_real: 0.6401 D_fake: 0.3610
Epoch (25/40)    Loss_G: 2.3323 Loss_D: 1.1318   D_real: 0.6316 D_fake: 0.3648
Epoch (26/40)    Loss_G: 2.3177 Loss_D: 1.1357   D_real: 0.6349 D_fake: 0.3627
Epoch (27/40)    Loss_G: 2.1666 Loss_D: 1.1736   D_real: 0.6243 D_fake: 0.3681
Epoch (28/40)    Loss_G: 2.0290 Loss_D: 1.1241   D_real: 0.6333 D_fake: 0.3712
Epoch (29/40)    Loss_G: 2.0850 Loss_D: 1.1466   D_real: 0.6278 D_fake: 0.3675
Epoch (30/40)    Loss_G: 2.0212 Loss_D: 1.1367   D_real: 0.6310 D_fake: 0.3646
Epoch (31/40)    Loss_G: 2.1884 Loss_D: 1.1382   D_real: 0.6418 D_fake: 0.3658
Epoch (32/40)    Loss_G: 1.9387 Loss_D: 1.1644   D_real: 0.6212 D_fake: 0.3719
Epoch (33/40)    Loss_G: 2.0390 Loss_D: 1.1455   D_real: 0.6322 D_fake: 0.3697
Epoch (34/40)    Loss_G: 1.9449 Loss_D: 1.1432   D_real: 0.6249 D_fake: 0.3688
Epoch (35/40)    Loss_G: 2.0598 Loss_D: 1.1179   D_real: 0.6396 D_fake: 0.3603
Epoch (36/40)    Loss_G: 2.0184 Loss_D: 1.1234   D_real: 0.6350 D_fake: 0.3647
Epoch (37/40)    Loss_G: 2.1825 Loss_D: 1.1005   D_real: 0.6439 D_fake: 0.3536
Epoch (38/40)    Loss_G: 2.1285 Loss_D: 1.1032   D_real: 0.6414 D_fake: 0.3565
Epoch (39/40)    Loss_G: 2.0810 Loss_D: 1.0947   D_real: 0.6436 D_fake: 0.3535
Epoch (40/40)    Loss_G: 2.1922 Loss_D: 1.0856   D_real: 0.6492 D_fake: 0.3478

Training ended.
```

During the training of the discriminator in a GAN, it is important to prevent updates to the generator's parameters. This is achieved in the code by using the `detach()` method on the `fake_batch`. When `detach()` is applied to a tensor in PyTorch, it creates a new tensor that is disconnected from the current computation graph, meaning that it does not track gradients. Consequently, when `fake_batch.detach()` is used in our code, it instructs PyTorch not to calculate gradients for the generator during the discriminator's backward

pass. Without `detach()`, gradients from the discriminator's loss, when evaluating fake images, would propagate back through both the discriminator and the generator, which is not desirable during this phase of training.

Here's the relevant part of the code for training the discriminator with fake data:

```
output = discriminator(fake_batch.detach()).view(-1)
loss_fake = criterion(output, fake_labels)
```

On the other hand, in the generator training phase, the objective is to update the generator's parameters such that it becomes better at generating images. Here, the gradients need to flow through both the discriminator and the generator. This is why `detach()` is not used when passing `fake_batch` to the discriminator.

Here's the code segment for training the generator:

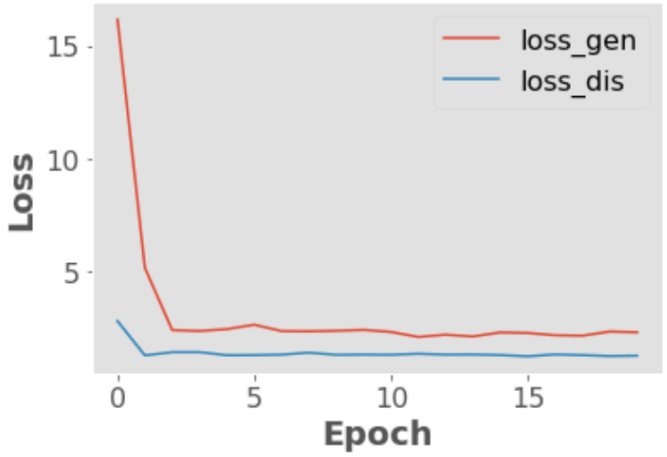
```
output = discriminator(fake_batch).view(-1)
loss_gen = criterion(output, real_labels)
```

## 3.4.2 Visualizing Training Progress

The following plots will help you see how the loss values of the generator and the discriminator, as well as the probabilities generated by the discriminator on real and fake images evolve during the training of our GAN:

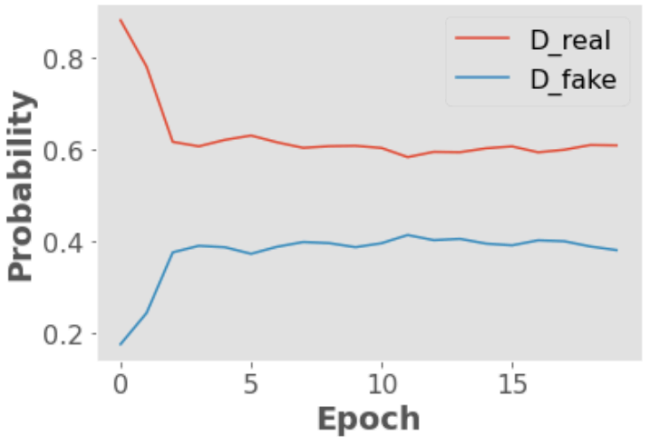
```
plt.plot(np.array(loss_gen_epoch), label='loss_gen')
plt.plot(np.array(loss_dis_epoch), label='loss_dis')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend();
```

Example output:




```
plt.plot(np.array(D_real_epoch), label='D_real')
plt.plot(np.array(D_fake_epoch), label='D_fake')
plt.xlabel("Epoch")
plt.ylabel("Probability")
plt.legend();
```

Example output:

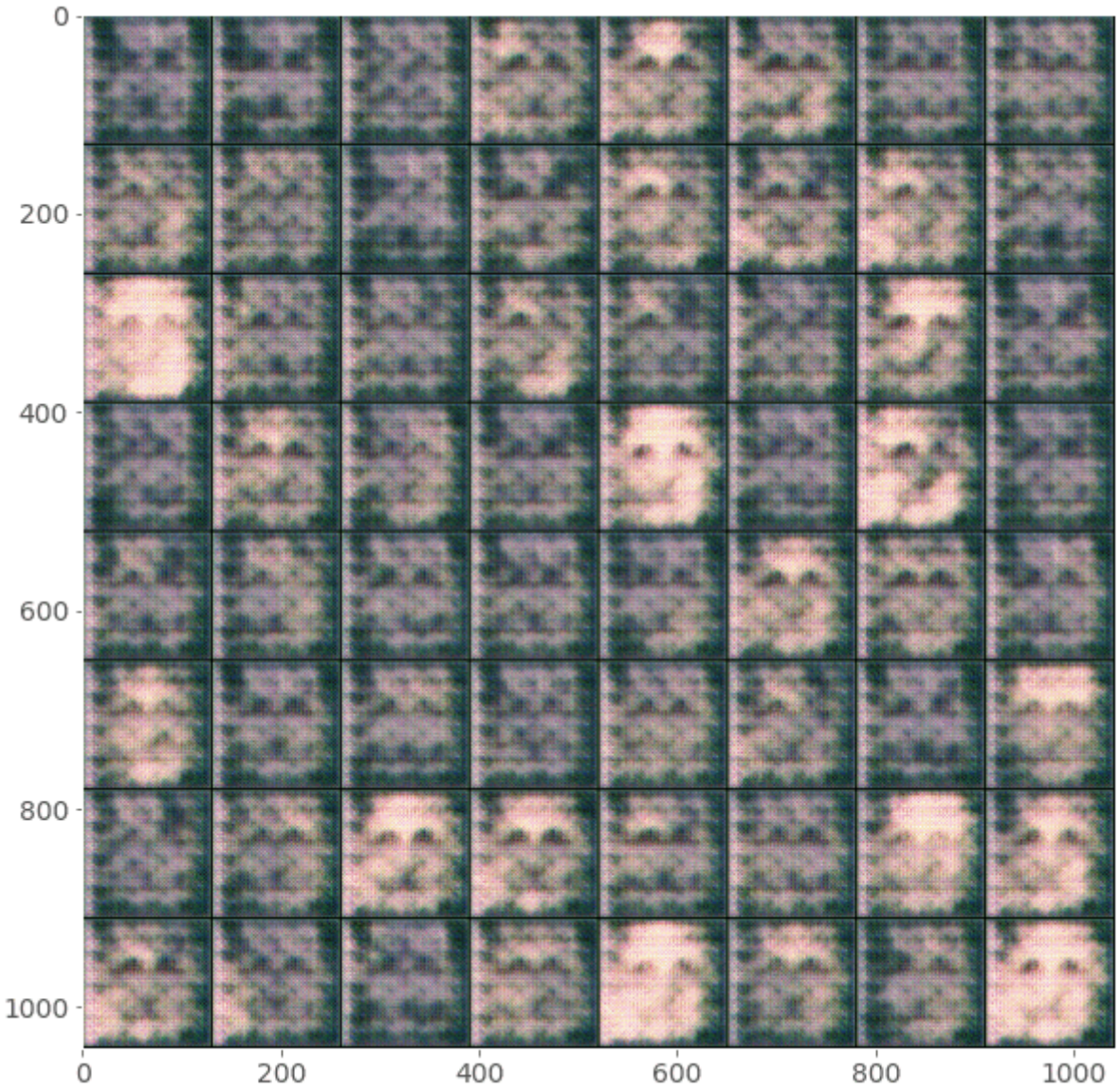



The following code cells help to see the evolution of one fixed noise vector throughout the epochs. The generator is applied on this fixed random noise in each epoch, and the results are saved as batches of generated images.

```
%%capture

fig = plt.figure(figsize=(10, 10))
ims = [[plt.imshow(np.transpose(i,(1, 2, 0)), animated=True)] for i in img_lis
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, bl
ani.save('GAN.gif', writer='imagemagick', fps=2)
```

```
HTML(ani.to_jshtml()) # run this in a new cell to produce the below animation
```

These are my results after running the GAN model for around 100 epochs:

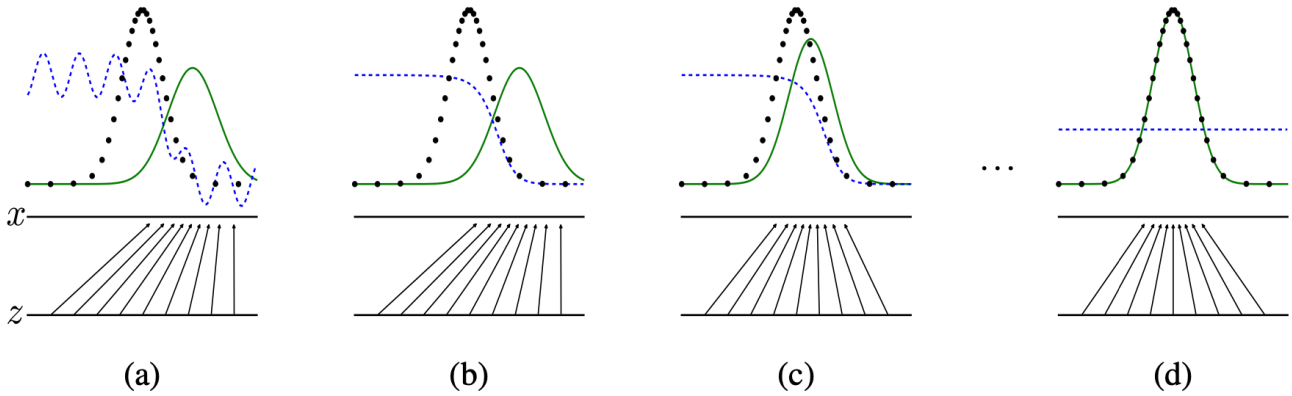



**Note:** You might have noticed the checker-board patterns that appear in the generated images, especially early in the training process. This is a known issue with transposed convolutions. This problem and potential solutions are discussed in great detail in this article.

Here is a picture of GAN training from the original paper.

- Data generating distribution $\longrightarrow$ Green solid line
- Real data distribution $\longrightarrow$ black dotted line
- Discriminator distribution $\longrightarrow$ Blue dashed line

- Lower horizontal line $\longrightarrow$ domain from which $z$ is sampled (uniform distribution in this case)




Source

- (a) Initially we have a partially accurate classifier
- (b) After training the discriminator for a few epochs it is able to classify real vs. fake examples.
- (c) The gradient of the discriminator guides the generator to move to the real data distribution
- (d) After several steps of training, if the generator distribution and the real data distribution align. The discriminator is unable to differentiate between the two distributions.

# ❓❓ Questions for you

## Exercise 8.2

**Which of the following is the desired outcome at the end of training a GAN?**

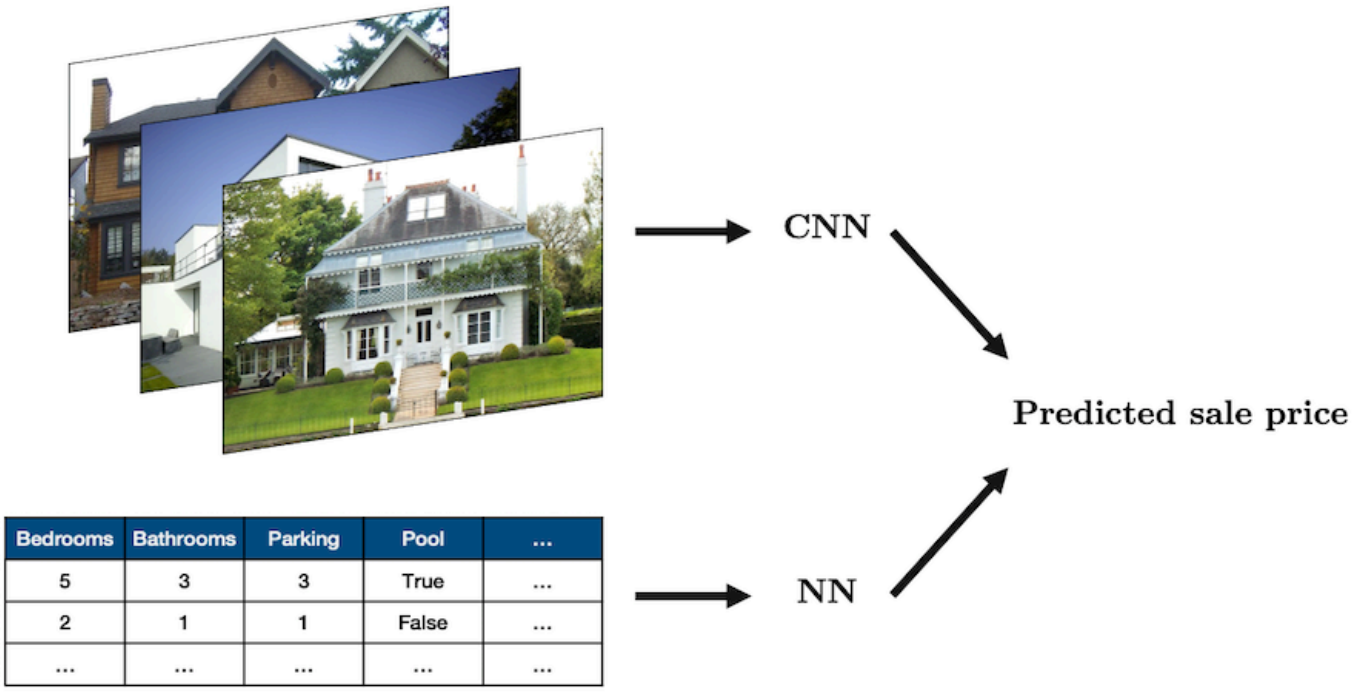
- (A) The discriminator always wins; it can identify real vs. fake images with perfect accuracy.

- (B) The discriminator never wins; it thinks all the real images are fake and all the fake images are real.
- (C) The discriminator output is no better than a random guess; it cannot tell real from fake.
- (D) The generator is able to recreate copies of images in the trainining data.

**V's Solutions!**

# 4. Multi-input Networks

- Sometimes you'll want to combine different types of data in a single network
- The most common case is combining tabular data with image data, for example, using both real estate data and images of a house to predict its sale price:




| Bedrooms | Bathrooms | Parking | Pool | ... |
|---|---|---|---|---|
| 5 | 3 | 3 | True | ... |
| 2 | 1 | 1 | False | ... |
| ... | ... | ... | ... | ... |

Source: "House" by oatsy40, "House in Vancouver" by pnwra, "House" by noona11 all licensed under CC BY 2.0.

- In such a problem you may want to combine:
    1. a NN for the tabular data
    2. a CNN for the image data
- The way we often do this is create these two models, and then combine them together into a model that produces a single output
- This sounds complicated but it's pretty easy! We only need one new ingredient which is a concatenation function: `torch.cat()`
- Below is a simple example:

```
class MultiModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(3, 16, 3, 2, 1),
            nn.ReLU(),
            nn.Conv2d(16, 8, 5, 2, 1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(1800, 5)
        )

        self.fc = nn.Sequential(
            nn.Linear(10, 50),
            nn.ReLU(),
            nn.Linear(50, 5)
        )

        self.multi = nn.Sequential(
            nn.Linear(10, 5),
            nn.ReLU(),
            nn.Linear(5, 1)
        )

    def forward(self, image, data):
        x_cnn = self.cnn(image)
        x_fc = self.fc(data)
        x_multi = torch.cat((x_cnn, x_fc), dim=1)
        return self.multi(x_multi)
```

```
model = MultiModel()
image = torch.randn(1, 3, 64, 64)
data = torch.randn(1, 10)
model(image, data).item()
```

```
0.48142674565315247
```

- The above network doesn't really do anything but show that we can easily combine a CNN and fully connected NN!

# 5. Conclusion

How did we do with the course learning objectives?

- Identify common computational issues caused by floating-point arithmatic, e.g., rounding, overflow, etc., and program defensively against these errors.
- Explain how the gradient descent algorithm and its variants work.
- Explain the fundamental concepts of neural networks including layers, nodes, and activation functions and gain proficiency in implementing basic neural networks using `PyTorch`.
- Illustrate the process of backpropagation in neural network training.
- Explain how convolutional neural networks work and implement them for image classification using PyTorch.
- Explain and apply transfer learning and the different flavours of it: "out-of-the-box", "feature extractor", "fine tuning".
- Describe at a high level the basic principles and architecture of Generative Adversarial Networks (GANs)

# Coming up ...

- Clustering, dimensionality reduction, word embeddings, recommender systems (DSCI 563)
- Training at scale/distributed computing (coming in DSCI 525)
- Working with other forms of data like time series (DSCI 574)
- Markov models, topic modeling, recurrent neural networks, and transformers (DSCI 575)

# Final remarks

That's all for this course! I hope you found it useful and learned something valuable. It was a lot of fun teaching you this fascinating material! Thank you for your engagement and thoughtful questions; they inspire us to keep improving.