The following model is the standard GAN which is part of **Exercise 1**. It is a very simple example and you can improve it by adding convolutions and many other ideas that we talked about if you want. Fill in the missing pieces and train it.

In [1]:
```
%matplotlib inline

import os
import numpy as np
import math
import multiprocessing
import matplotlib.pyplot as plt

import torchvision.transforms as transforms
from torchvision.utils import save_image, make_grid
from torch.optim.optimizer import Optimizer, required
from torch.utils.data import DataLoader
from torchvision import datasets
from torch.autograd import Variable

import torch.nn as nn
import torch.nn.functional as F
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

os.makedirs("images_gan", exist_ok=True)
os.makedirs("images_cgan", exist_ok=True)

batch_size = 128                                          #size of the batches
lr = 0.0005                                               #adam: learning rate
b1 = 0.5                                                  #adam: decay of first or
b2 = 0.999                                                #adam: decay of second o
n_cpu = multiprocessing.cpu_count()                       #number of cpu threads to
latent_dim = 100                                          #dimensionality of the la
img_size = 28                                             #size of each image dimen
channels = 1                                              #number of image channels
sample_interval = 400                                     #interval between image s


img_shape = (channels, img_size, img_size)
torch.manual_seed(42)
```

Out[1]:
```
<torch._C.Generator at 0x7fcff0ad5490>
```

Excercise 1. Generative Adversarial Networks (GANs) (34%)

Implement a GAN and train it on the MNIST dataset (a). Plot 10 samples generated using the GAN (b). Finally, train a classifier C that classifies MNIST images. Use that classifier to approximate the marginal distribution of the generator p(y), (i.e. the probability that the GAN samples a particular class). Visualize the distribution using a bar-plot (c). You can use one-to-one updates applying one gradient step for the discriminator and generator successively. If you want to keep this simple you can also just use linear layers.

In [2]:
```
def to_onehot(digits, num_classes):
    """ [[3]] => [[0, 0, 1]]
    """
    labels_onehot = torch.zeros(digits.shape[0], num_classes).to(device)
    labels_onehot.scatter_(1, digits.view(-1, 1), 1)
    return labels_onehot

def plot_class_distributions(y_pred, num_classes):
    class_distributions = [np.sum(y_pred == i) for i in range(num_classe
    plt.bar(list(range(num_classes)), class_distributions, tick_label=li
    plt.ylabel("Number of predictions")
    plt.xlabel("Class")
    plt.plot()


class GeneratorBlock(nn.Module):
    def __init__(self, in_feat, out_feat, activation, use_norm=True):
        super().__init__()
        self.dense = nn.Linear(in_feat, out_feat)
        self.activation = activation
        if use_norm:
            self.bn = nn.BatchNorm1d(out_feat, 0.8)
        else:
            self.bn = None

    def forward(self, x):
        x = self.dense(x)
        x = self.activation(x)
        if self.bn != None:
            x = self.bn(x)
        return x


class Generator(nn.Module):
    def __init__(self, num_classes=0):
        super().__init__()
        self.b1 = GeneratorBlock(latent_dim + num_classes, 128, activati
        self.b2 = GeneratorBlock(128, 256, activation=nn.LeakyReLU(0.2))
        self.b3 = GeneratorBlock(256, 512, activation=nn.LeakyReLU(0.2))
        self.b4 = GeneratorBlock(512, 1024, activation=nn.LeakyReLU(0.2)
        self.b5 = GeneratorBlock(1024, 1 * 28 * 28, activation=nn.Tanh()

    def forward(self, x, y=None):
        if y != None:
            x =  torch.cat((x, y), dim=1)
        x = self.b1(x)
        x = self.b2(x)
        x = self.b3(x)
        x = self.b4(x)
        x = self.b5(x)
        x = x.view(-1, 1, 28, 28)
        return x


class Discriminator(nn.Module):
    def __init__(self, num_classes=0, use_sigmoid=True):
```

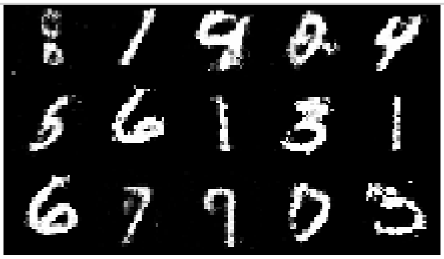
```
        super().__init__()
        self.dense_1 = nn.Linear(1 * 28 * 28 + num_classes, 512)
        self.dense_2 = nn.Linear(512, 256)
        self.dense_3 = nn.Linear(256, 1)
        self.leaky_relu = nn.LeakyReLU(0.2)
        self.use_sigmoid = use_sigmoid

    def forward(self, x, y=None):
        x = x.view(x.shape[0], -1)
        if y != None:
            x = torch.cat((x, y), dim=1)
        x = self.dense_1(x)
        x = self.leaky_relu(x)
        x = self.dense_2(x)
        x = self.leaky_relu(x)
        x = self.dense_3(x)
        if self.use_sigmoid:
            x = torch.sigmoid(x)
        return x
```

In [3]:
```
# Loss function
bce_loss = torch.nn.BCELoss()

# Initialize generator and discriminator
generator = Generator()
discriminator = Discriminator()

generator.to(device)
discriminator.to(device)
bce_loss.to(device)

# Configure data loader
os.makedirs("./mnist", exist_ok=True)
dataloader = torch.utils.data.DataLoader(
    torch.utils.data.ConcatDataset(
        [
            datasets.MNIST(
                "./mnist",
                train=True,
                download=True,
                transform=transforms.Compose(
                    [transforms.Resize(img_size), transforms.ToTensor(),
                ),
            ),
            datasets.MNIST(
                "./mnist",
                train=False,
                download=True,
                transform=transforms.Compose(
                    [transforms.Resize(img_size), transforms.ToTensor(),
                ),
            ),
        ]
    ),
    batch_size=batch_size,
    shuffle=True,
    num_workers=n_cpu
)

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=lr, betas=(b1,
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=
```

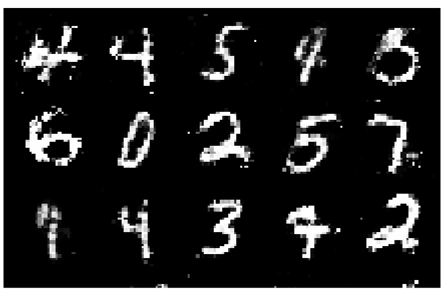
In [4]:
```
# ----------
#  Training
# ----------

def train_loop(generator, discriminator, dataloader, optimizer_g, optimi
    iterations = len(dataloader)
    gen_losses, disc_losses = [], []
    for epoch in range(num_epochs):
        running_gen_loss = 0.0
        running_disc_loss = 0.0
        for i, (real_imgs, y) in enumerate(dataloader):

            real_imgs =  real_imgs.to(device)
            if with_labels:
                y = to_onehot(y.to(device), 10)
            else:
                y = None

            # -----------------
            #  Train Generator
            # -----------------

            # WGANs train the discriminator more often than the generato
            if i % 5 == 0 or not use_wasserstein:
                optimizer_G.zero_grad()

                z = torch.randn((real_imgs.shape[0], latent_dim)).to(dev
                gen_imgs = generator(z, y)

                y_pred_fake = discriminator(gen_imgs, y)

                if use_wasserstein:
                    g_loss = loss_func(y_pred_fake)
                else:
                    g_loss = loss_func(y_pred_fake, torch.zeros_like(y_p
                g_loss.backward()
                optimizer_G.step()

            # ---------------------
            #  Train Discriminator
            # ---------------------

            optimizer_D.zero_grad()

            gen_imgs = generator(z, y)

            y_pred_real = discriminator(real_imgs, y)
            y_pred_fake = discriminator(gen_imgs, y)

            if use_wasserstein:
                d_loss = loss_func(y_pred_real) - loss_func(y_pred_fake)
            else:
                real_loss = loss_func(y_pred_real, torch.zeros_like(y_pr
                fake_loss = loss_func(y_pred_fake, torch.ones_like(y_pre
                d_loss = (real_loss + fake_loss) / 2
            d_loss.backward()
```

```
            optimizer_D.step()

            # clip weights of discriminator when using WGAN
            if use_wasserstein:
                for p in discriminator.parameters():
                    p.data.clamp_(-0.01, 0.01)

            running_gen_loss += g_loss.item()
            running_disc_loss += d_loss.item()

            batches_done = epoch * len(dataloader) + i
            if batches_done % sample_interval == 0:
                # You can also safe samples in your drive & maybe save y
                save_image(gen_imgs.data[:25], "images_gan/GAN-%d.png" %

        gen_loss = running_gen_loss / iterations
        disc_loss = running_disc_loss / iterations
        loss = gen_loss + disc_loss
        gen_losses.append(gen_loss)
        disc_losses.append(disc_loss)
        print(f"Epoch {epoch + 1}/{num_epochs} ==> loss: {loss}, gen_los
        grid = make_grid(gen_imgs.data[:25], nrow=5, normalize=True).cpu
        # Channels first (PyTorch) to channels last (matplotlib)
        grid = np.moveaxis(grid, 0, -1)
        plt.imshow(grid, cmap='gray')
        plt.axis('off')
        plt.show()
    return gen_losses, disc_losses
```

In [5]:
```
gen_losses, disc_losses = train_loop(
    generator,
    discriminator,
    dataloader,
    optimizer_G,
    optimizer_D,
    bce_loss,
    20,
    False,
    use_wasserstein=False
)
```



```
Epoch 20/20 ==> loss: 1.5051823876239698, gen_loss: 0.82724329115485
8, disc_loss: 0.677939096469112
```

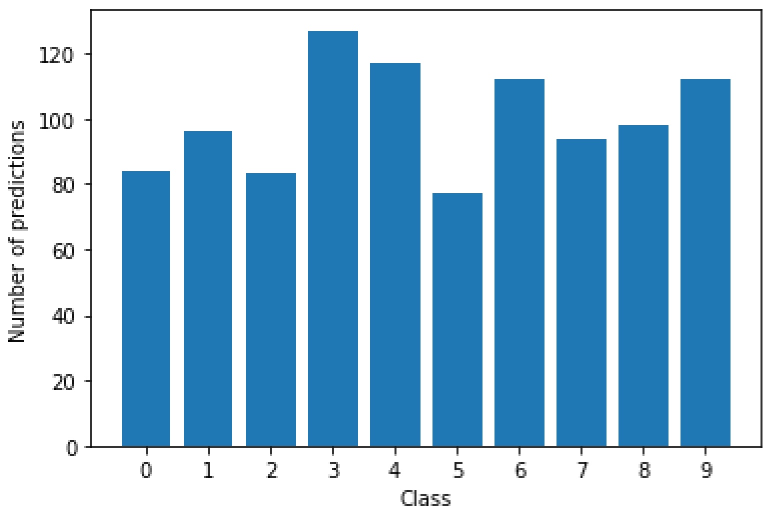


In [7]:
```
class Classifer(nn.Module):
    def __init__(self, in_dims):
        super().__init__()
        self.conv1 = nn.Conv2d(in_dims, 16, kernel_size=(3, 3), padding=
        self.conv2 = nn.Conv2d(16, 32, kernel_size=(3, 3), padding=1, st
        self.conv3 = nn.Conv2d(32, 64, kernel_size=(3, 3), padding=1, st
        self.dense1 = nn.Linear(4 * 4 * 64, 10)


    def forward(self, x):
        x = self.conv1(x)
        x = torch.nn.functional.relu(x)
        x = self.conv2(x)
        x = torch.nn.functional.relu(x)
        x = self.conv3(x)
        x = torch.nn.functional.relu(x)
        x = x.view(-1, 4 * 4 * 64)
        x = self.dense1(x)
        x = torch.softmax(x, dim=1)
        return x

clf = Classifer(1).to(device)

clf_optim = torch.optim.Adam(clf.parameters(), lr=0.001)
criterion = torch.nn.CrossEntropyLoss().to(device)


for epoch in range(1, 11):
    running_loss = 0
    running_accuracy = 0
    iterations = 0
    for i, (x, y) in enumerate(dataloader):
        x, y = x.to(device), y.to(device)
        clf_optim.zero_grad()
        y_pred = clf(x)
        loss = criterion(y_pred, y)
        loss.backward()
        clf_optim.step()
        running_loss += loss.item()
        iterations += 1
        with torch.no_grad():
            accuracy = torch.mean(((torch.argmax(y_pred, 1) == y) * 1).f
            running_accuracy += accuracy.item()
    loss = running_loss / iterations
    acc = running_accuracy / iterations
    print(f"Epoch {epoch}/10 ==> train loss: {loss}, train acc: {acc}")
```

```
Epoch 1/10 ==> train loss: 1.6623974343322532, train acc: 0.806054534
598286
Epoch 2/10 ==> train loss: 1.5195921829457257, train acc: 0.943880174
4520338
Epoch 3/10 ==> train loss: 1.4898731527642315, train acc: 0.972793973
6690556
Epoch 4/10 ==> train loss: 1.4841277424988406, train acc: 0.977984624
0005284
Epoch 5/10 ==> train loss: 1.4805172697063773, train acc: 0.981432815
```

```
35649
Epoch 6/10 ==> train loss: 1.4788157107407258, train acc: 0.983001844
5386312
Epoch 7/10 ==> train loss: 1.4771130475091323, train acc: 0.984458654
4106604
Epoch 8/10 ==> train loss: 1.4746645930916125, train acc: 0.987098867
3222348
Epoch 9/10 ==> train loss: 1.474323284473454, train acc: 0.9871315127
97075
Epoch 10/10 ==> train loss: 1.4732294851944694, train acc: 0.98815984
91773309
```

In [8]:
```
z = torch.randn((1000, latent_dim)).to(device)

# Generate a batch of images
with torch.no_grad():
    gen_imgs = generator(z)
    y_pred = clf(gen_imgs)

y_pred_probs_ngan = y_pred
y_pred = np.argmax(y_pred.cpu().numpy(), axis=1)
plot_class_distributions(y_pred, 10)
```



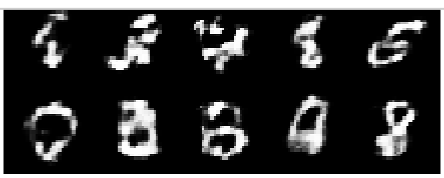
Excercise 2. Wasserstein GANs (46%)

Unless you have been lucky, you should have seen a mode collapse (one class being sampled much more often that another). Now Wasserstein GANs are not only proven to generate better images, but they also do not have such a bad mode collapse. However, we need to enforce 1-lipschitz continuity of the discriminator D in order for WGANs to work. You can either implement Spectral-Norm-Layers1 or do Gradient Penalty2 to ensure this. Train your model on the data set. (33%)

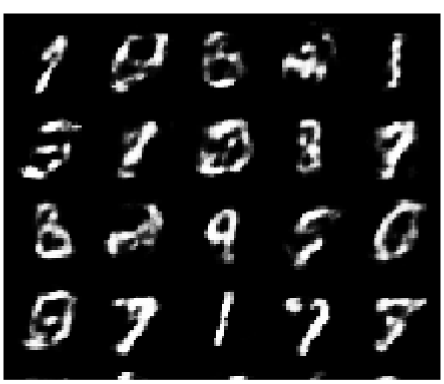
In [9]:
```
def wasserstein_loss(y_pred):
    return torch.mean(y_pred)


# It's recommended to use RMSProp for WGAN
generator = Generator()
discriminator = Discriminator(0, use_sigmoid=False)

optimizer_G = torch.optim.RMSprop(generator.parameters(), lr=0.00005)
optimizer_D = torch.optim.RMSprop(discriminator.parameters(), lr=0.00005

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

Out[9]:
```
Discriminator(
  (dense_1): Linear(in_features=784, out_features=512, bias=True)
  (dense_2): Linear(in_features=512, out_features=256, bias=True)
  (dense_3): Linear(in_features=256, out_features=1, bias=True)
  (leaky_relu): LeakyReLU(negative_slope=0.2)
)
```

In [10]:
```
gen_losses, disc_losses = train_loop(
    generator,
    discriminator,
    dataloader,
    optimizer_G,
    optimizer_D,
    wasserstein_loss,
    200,
    False,
    use_wasserstein=True
)
```

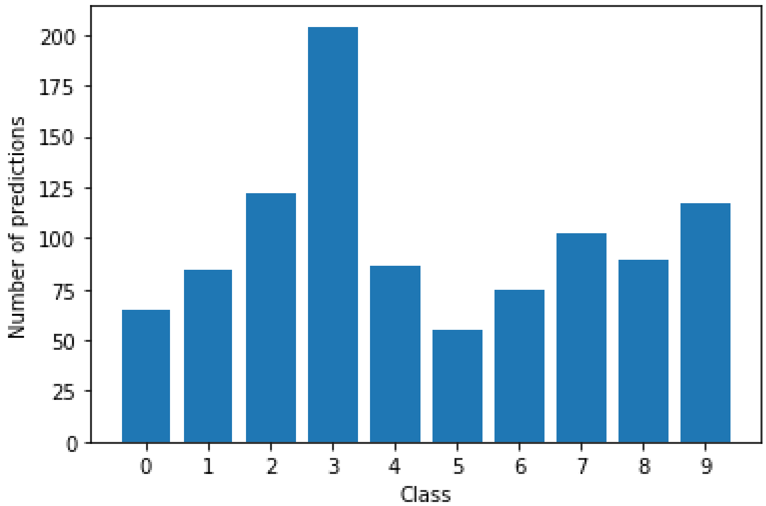


```
Epoch 181/200 ==> loss: -0.830482601901513, gen_loss: -0.698210825649
8682, disc_loss: -0.1322717762516447
```



(a) Calculate a marginal distribution, again, and create a bar-plot. (13%)

In [11]:
```
z = torch.randn((1000, latent_dim)).to(device)

# Generate a batch of images
with torch.no_grad():
    gen_imgs = generator(z)
    y_pred = clf(gen_imgs)

y_pred_probs_wgan = y_pred
y_pred = np.argmax(y_pred.cpu().numpy(), axis=1)
plot_class_distributions(y_pred, 10)
```



(b) Calculate the Inception Score (IS) for both models. (10%)

In [17]:
```
def inception_score(y_pred, chunk_size=10):
    inc_scores = []
    for chunk_start in range(0, y_pred.shape[0], chunk_size):
        y_pred_cur = y_pred[chunk_start: chunk_start + chunk_size]
        # calculate marginal probability, which is just the average prob
        marginal_dist = torch.mean(y_pred_cur, dim=0)

        # calculate kl divergence for each sample between it's probabil
        # Add a small eps in case a class has probability 0.00, because
        eps = 1e-14
        kl_div = y_pred_cur * (torch.log(y_pred_cur + eps) - torch.log(m

        # sum over all classes
        kl_div_samples = torch.sum(kl_div, dim=0)

        # average over all samples
        avg_kl = torch.mean(kl_div_samples)
        inc_score = torch.exp(avg_kl)
        inc_scores.append(inc_score.cpu().numpy())
    return np.mean(inc_scores)


inc_score_ngan = inception_score(y_pred_probs_ngan)
inc_score_wgan = inception_score(y_pred_probs_wgan)
print(f"GAN has an inception score of   {inc_score_ngan}")
print(f"WGAN has an inception score of  {inc_score_wgan}")
```

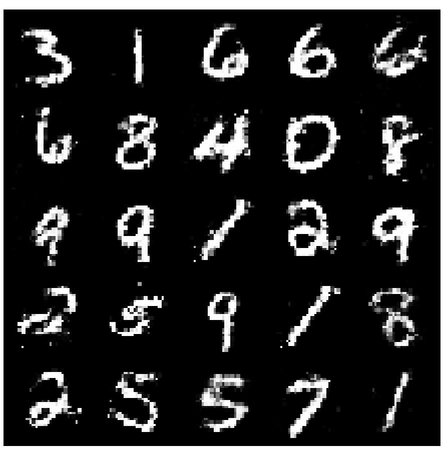
```
GAN has an inception score of   5.711552143096924
WGAN has an inception score of  5.353795051574707
```

Excercise 3. Conditional GANs (CGANs) (20%)

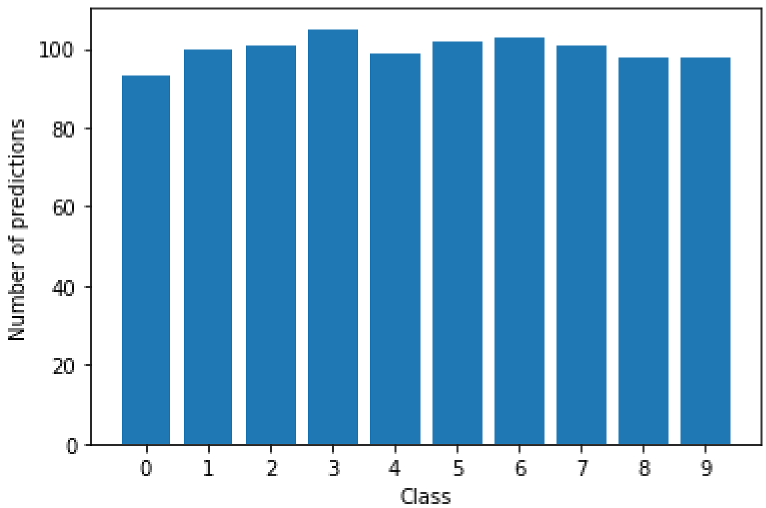
Now if we want to ultimately prevent mode collapse, we need to provide the information of what to sample additionally. This is what CGANs do. Implement a CGAN and train it on the data set. (15%)

In [13]:
```
# Initialize generator and discriminator
generator = Generator(10)
discriminator = Discriminator(10)

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

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=lr, betas=(b1,
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=

gen_losses, disc_losses = train_loop(
    generator,
    discriminator,
    dataloader,
    optimizer_G,
    optimizer_D,
    bce_loss,
    20,
    True,
    use_wasserstein=False
)
```

```
Epoch 20/20 ==> loss: 1.4606071692062252, gen_loss: 0.773821665120517
1, disc_loss: 0.6867855040857082
```



In [19]:
```
z = torch.randn((1000, latent_dim)).to(device)

y = [i % 100 for i in range(1000)]

# Generate a batch of images
with torch.no_grad():
    y = torch.LongTensor([i % 10 for i in range(1000)]).to(device)
    y = to_onehot(y, 10)
    gen_imgs = generator(z, y)
    y_pred = clf(gen_imgs)

y_pred_probs = y_pred
y_pred = np.argmax(y_pred.cpu().numpy(), axis=1)
plot_class_distributions(y_pred, 10)
```



In [21]:
```
inc_score_cgan = inception_score(y_pred_probs)
print(f"GAN has an inception score of   {inc_score_ngan}")
print(f"WGAN has an inception score of  {inc_score_wgan}")
print(f"CGAN has an inception score of  {inc_score_cgan}")
```

```
GAN has an inception score of   5.711552143096924
WGAN has an inception score of  5.353795051574707
CGAN has an inception score of  9.517807006835938
```

As we see the CGAN is clearly our best model.

(a) Sample a few pictures of the classes of your choice. (5%)

In [41]:
```
fig, ax = plt.subplots(nrows=2, ncols=5, figsize=(25, 10))

for label in range(10):
    z = torch.randn((25, latent_dim)).cuda()
    with torch.no_grad():
        y = torch.ones((25, )).long().cuda() * label
        y = to_onehot(y, 10)
        gen_imgs = generator(z, y)
    grid = make_grid(gen_imgs, nrow=5, normalize=True).cpu().numpy()
    # Channels first (PyTorch) to channels last (matplotlib)
    grid = np.moveaxis(grid, 0, -1)
    ax[label // 5][label % 5].imshow(grid, cmap='gray')
    ax[label // 5][label % 5].axis('off')
    ax[label // 5][label % 5].set_title(label)
plt.show()
```