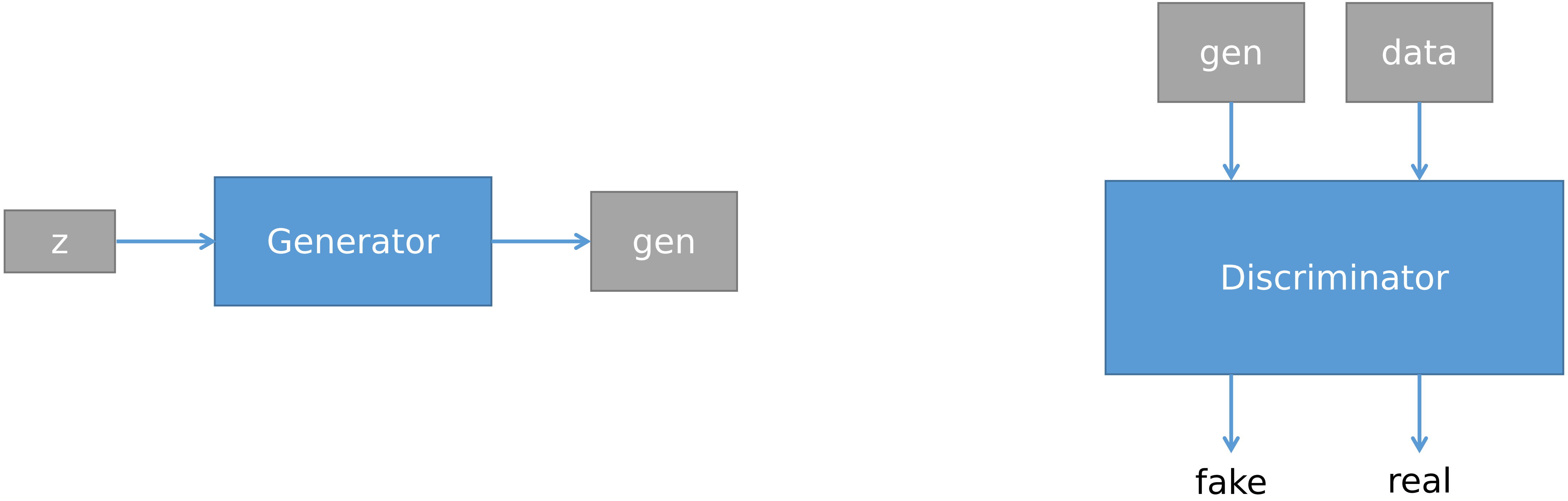
GAN

Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a **type** of deep learning **model composed** of **two** neural **networks**: a **generator** and a **discriminator**.

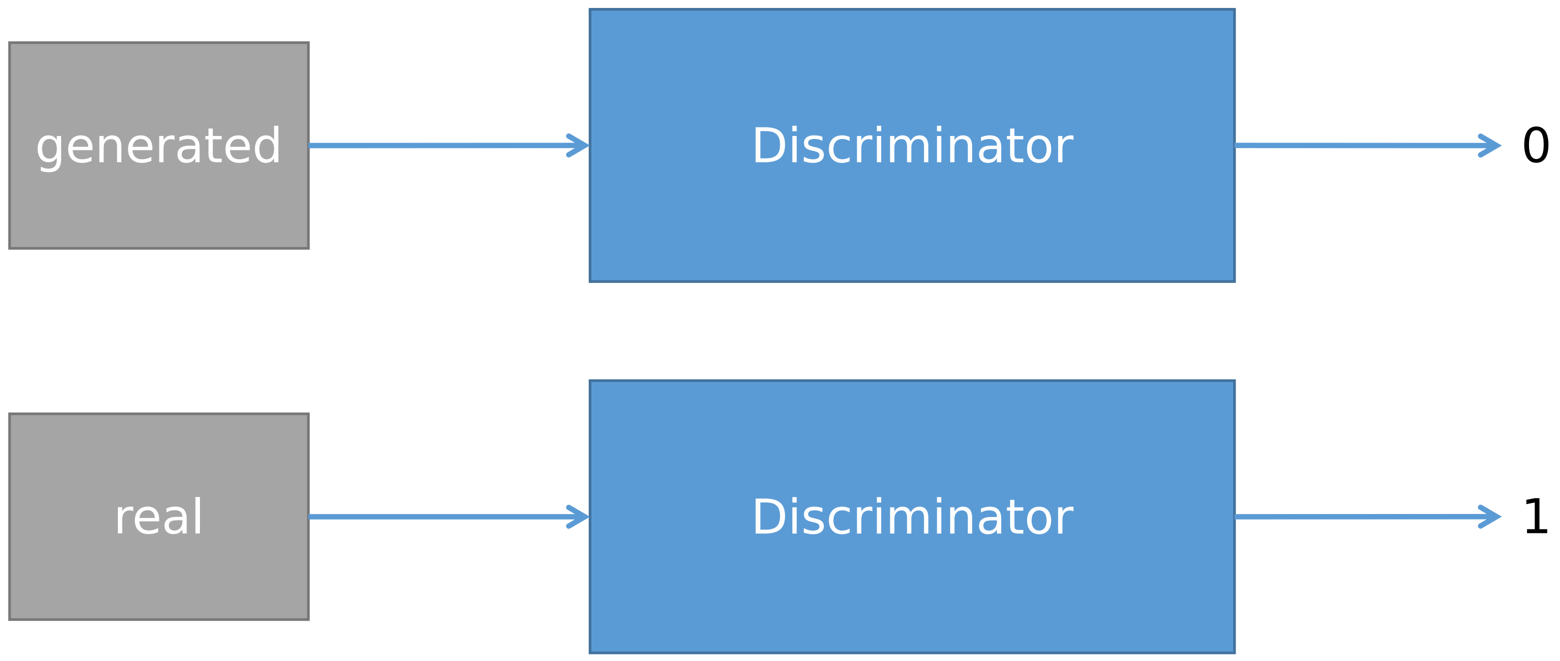
GANs are **designed** for **generating** data that closely **resembles** the **one** in the **training set**.



GANs: Discriminator

The **discriminator** is a **binary classifier** with **one** output.

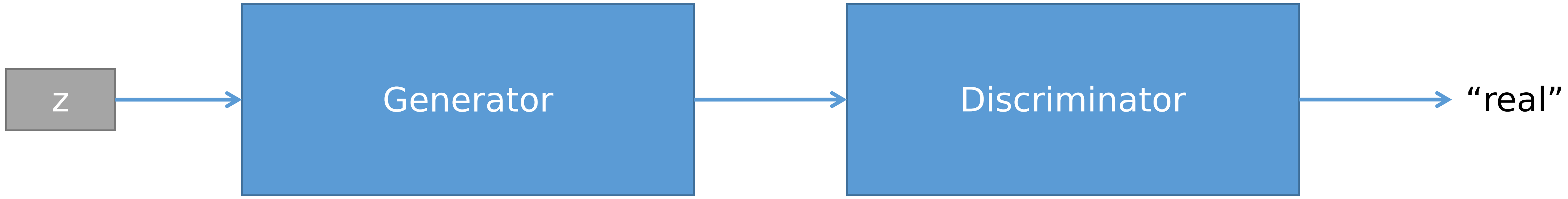
The **discriminator** is **trained** to **distinguish** between **real** images (output 1) and **generated** images (output 0)



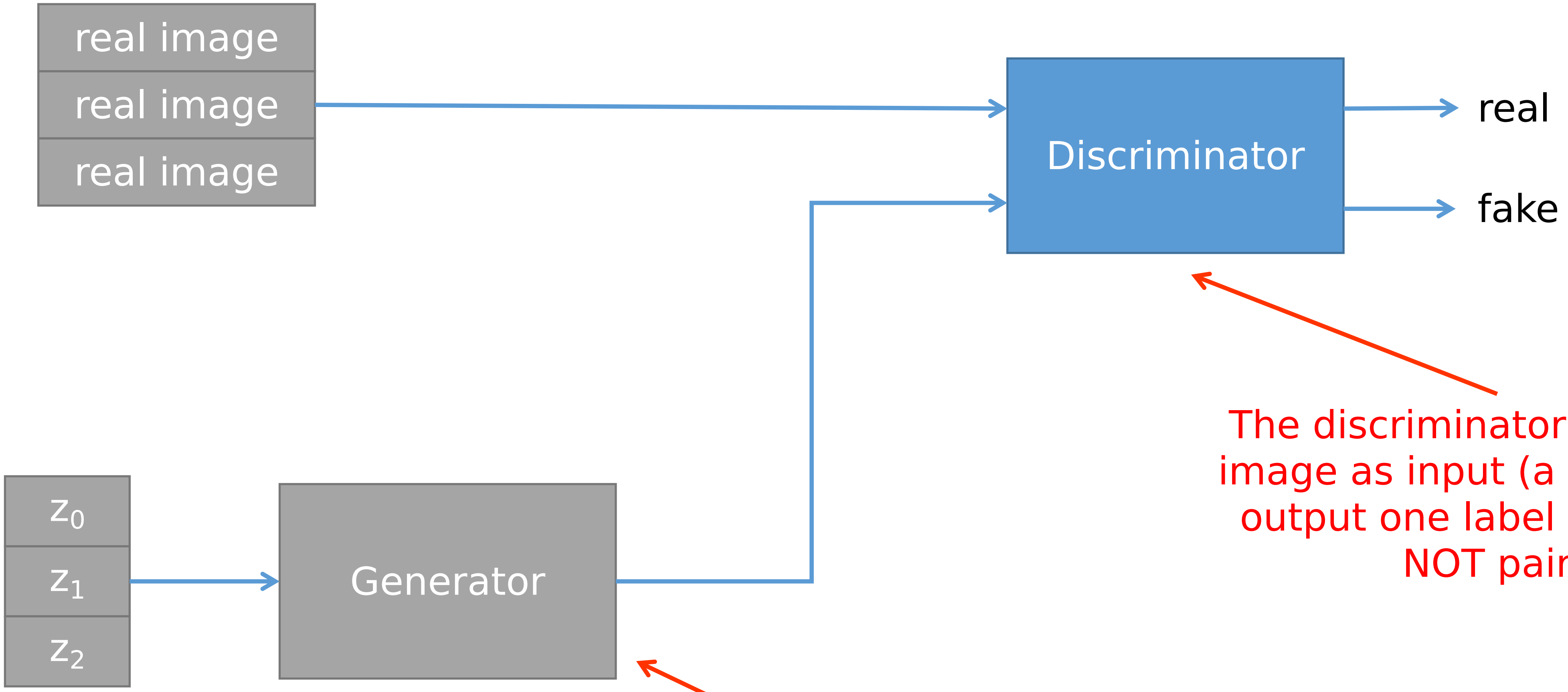
GANs: Generator

The **generator** is trained to **produce** images that can **deceive** the **discriminator**.

It takes **random** noise as **input** and aims to **generate** an **output** that, when **presented** to the **discriminator**, is **classified** as **'real'**.



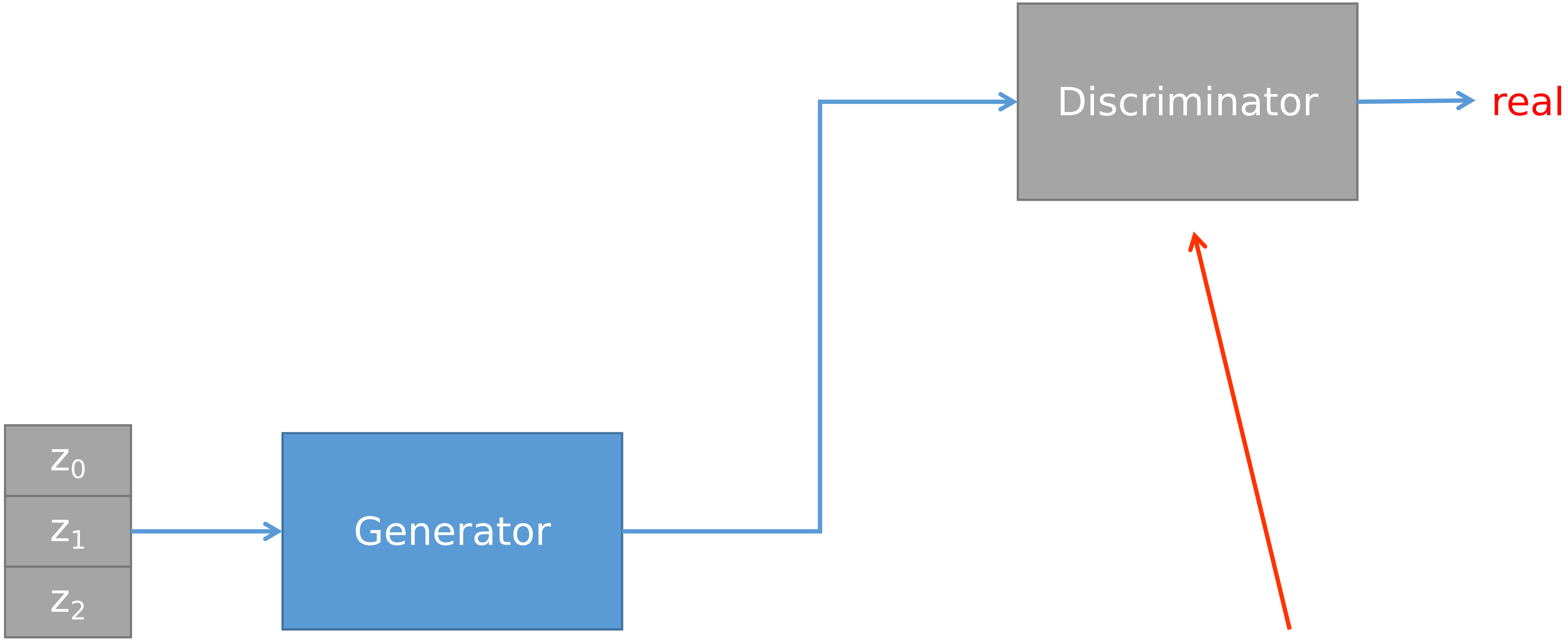
GANs: Discriminator Training



The discriminator takes one image as input (a batch) and output one label (a batch). NOT pairs

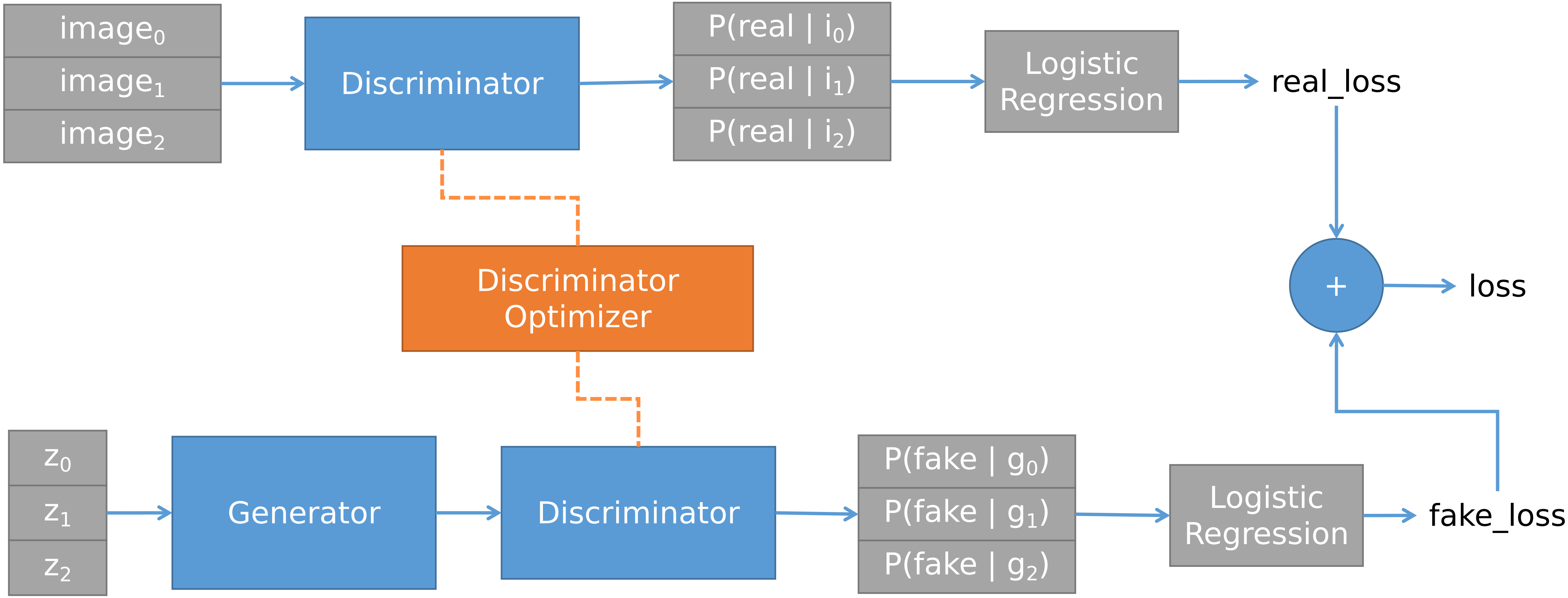
We do NOT train the generator. Compute gradients only for the discrmminator

GANs: Generator Training

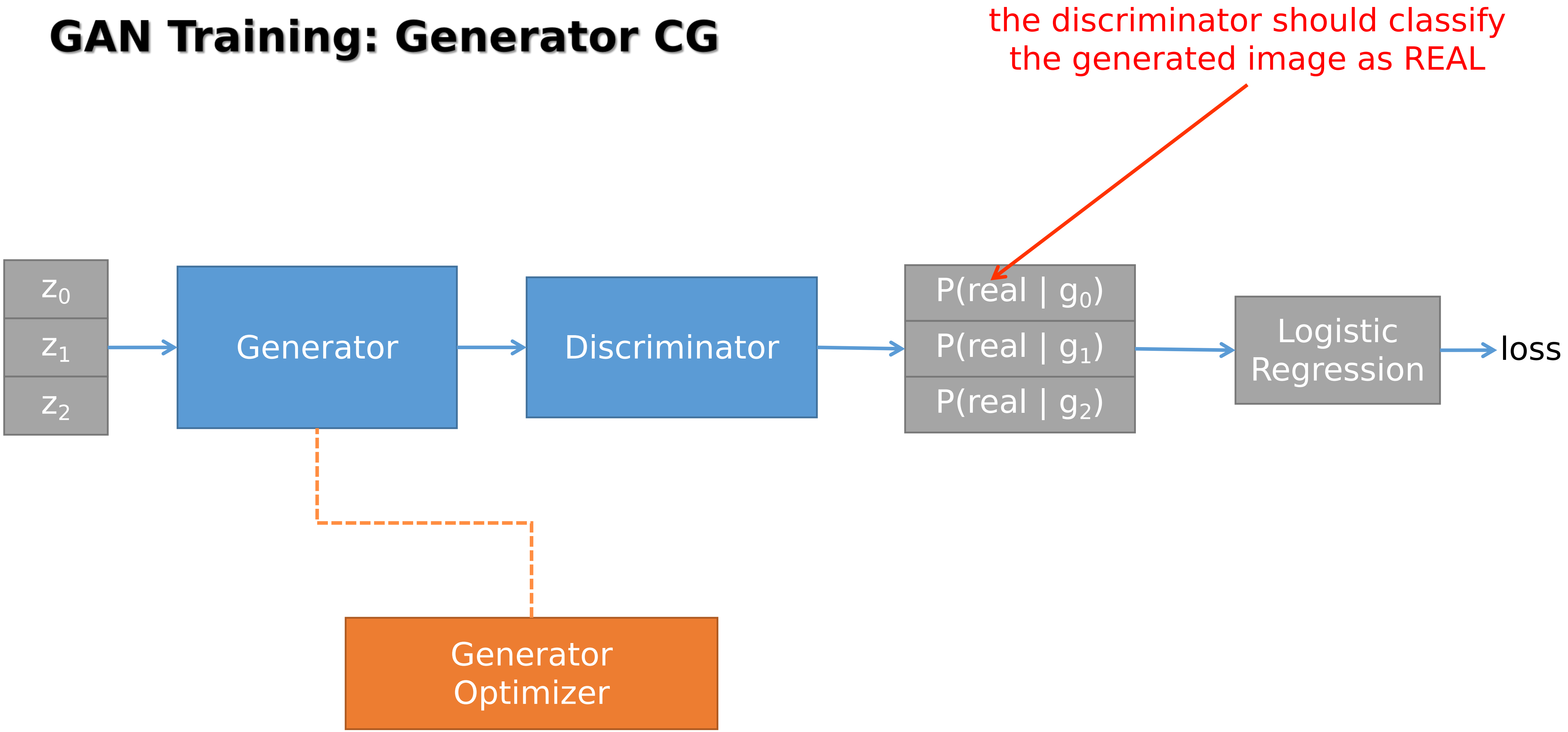


We do NOT train the discriminator.
Compute gradients only for the generator

GAN Training: Discriminator CG



GAN Training: Generator CG



GANs: Discriminator training in practice

To **train** a **GAN** network we use **two optimizer**: **one** for **each network**.

To train the **discriminator**:

- randomly **sample** some **real data** from the **dataset**
- **generate** some **fake data** using the **generator**
- **feed** the **discriminator** with the **real** data and **compute** the **error** of the **predicted** label w.r.t. **1** (1 is the real label)
- **feed** the **discriminator** with the **fake** data and **compute** the **error** of the **predicted** labels w.r.t. **0** (0 is the fake label)
- **compute** the **total error** as the **mean** of the two **errors**
- **compute** the **gradients** of the **error** w.r.t. the **discriminator weights**
- **update** the **discriminator weights**

GANs: Generator training in practice

To train the **generator**:

- **generate** some **fake data** using the **generator**
- **feed** the **discriminator** with the **fake** data and **compute** the **error** of the **predicted** labels w.r.t. **1** (1 is the real label)
- **compute** the **gradients** of the **error** w.r.t. the **generator weights**
- **update** the **generator weights**

GANs: PyTorch (initialization)

```
generator = Generator(latent_dim=64, output_dim=28)
discriminator = Discriminator(input_dim=28)
```

```
generator_optimizer = torch.optim.Adam(generator.parameters(), lr=0.0002)
discriminator_optimizer = torch.optim.Adam(discriminator.parameters(), lr=0.0002)
```

GANs: PyTorch (training loop)

```
for epoch in range(0, 20):
    for real_images, _ in tqdm(train_dl):
        real_images = real_images.to("cuda")

        discriminator_optimizer.zero_grad()
        generator_optimizer.zero_grad()
        train_discriminator(
            generator=generator,
            discriminator=discriminator,
            optimizer=discriminator_optimizer,
            real_images=real_images
        )

        discriminator_optimizer.zero_grad()
        generator_optimizer.zero_grad()
        train_generator(
            generator=generator,
            discriminator=discriminator,
            optimizer=generator_optimizer,
            real_images=real_images
        )
```

GANs: PyTorch (discriminator training)

```python
def train_discriminator(*, optimizer, generator, discriminator, real_images):
    batch_size = real_images.shape[0]
    device = real_images.device

    z = torch.randn(batch_size, generator.latent_dim).to(device)
    fake_images = generator(z)

    fake_label = torch.zeros(batch_size).to(device)
    real_label = torch.ones(batch_size).to(device)

    fake_logits = discriminator(fake_images)[:, 0]
    real_logits = discriminator(real_images)[:, 0]

    fake_loss = torch.nn.functional.binary_cross_entropy(fake_logits, fake_label)
    real_loss = torch.nn.functional.binary_cross_entropy(real_logits, real_label)

    loss = (fake_loss + real_loss) / 2

    loss.backward()
    optimizer.step()

    fake_accuracy = (fake_logits < 0.5).float().mean()
    real_accuracy = (real_logits >= 0.5).float().mean()

    return fake_loss.item(), real_loss.item(), fake_accuracy.item(), real_accuracy.item()
```

GANs: PyTorch (generator training)

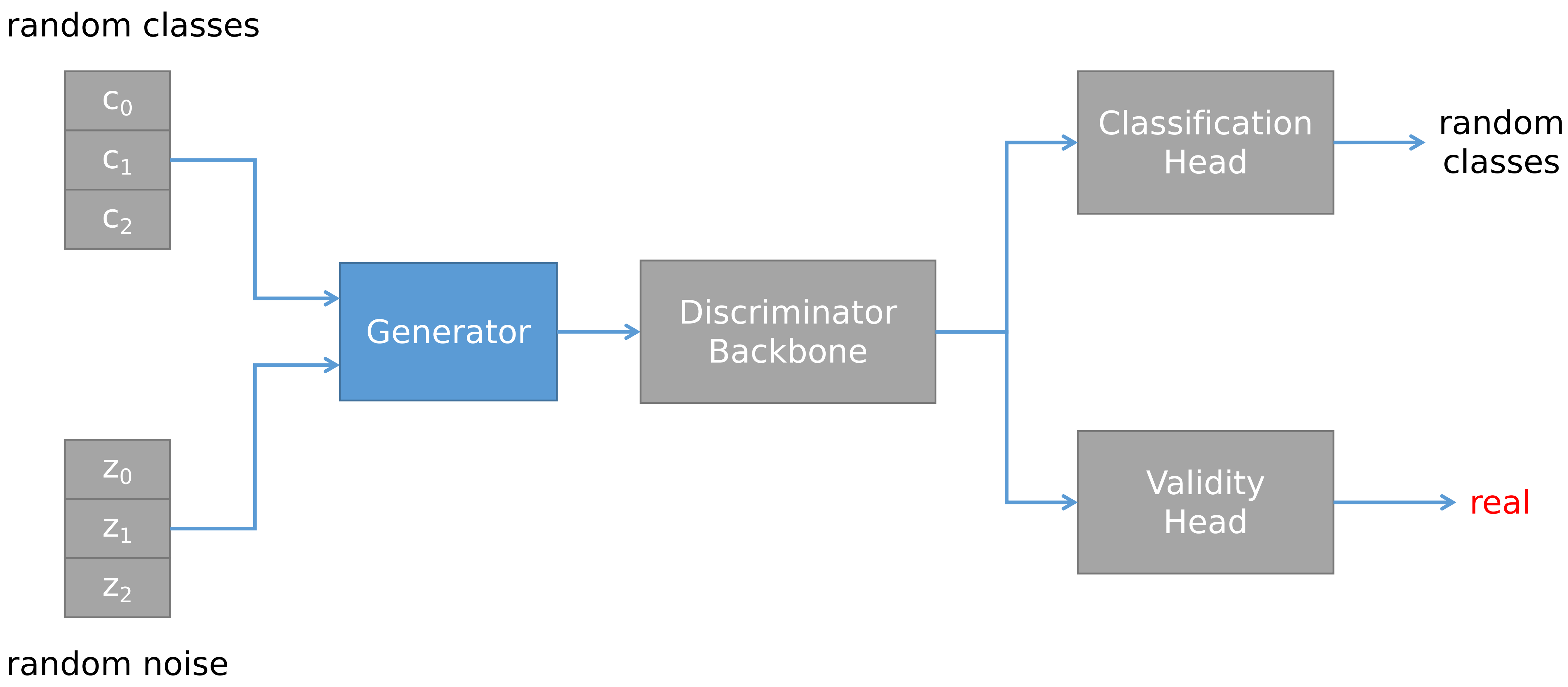
```python
def train_generator(*, optimizer, generator, discriminator, real_images):
    batch_size = real_images.shape[0]
    device = real_images.device

    z = torch.randn(batch_size, generator.latent_dim).to(device)
    fake_images = generator(z)

    real_label = torch.ones(batch_size).to(device)

    fake_logits = discriminator(fake_images)[:, 0]

    loss = torch.nn.functional.binary_cross_entropy(fake_logits, real_label)

    loss.backward()
    optimizer.step()

    accuracy = (fake_logits >= 0.5).float().mean()

    return loss.item(), accuracy
```

Auxiliary Classifier GAN

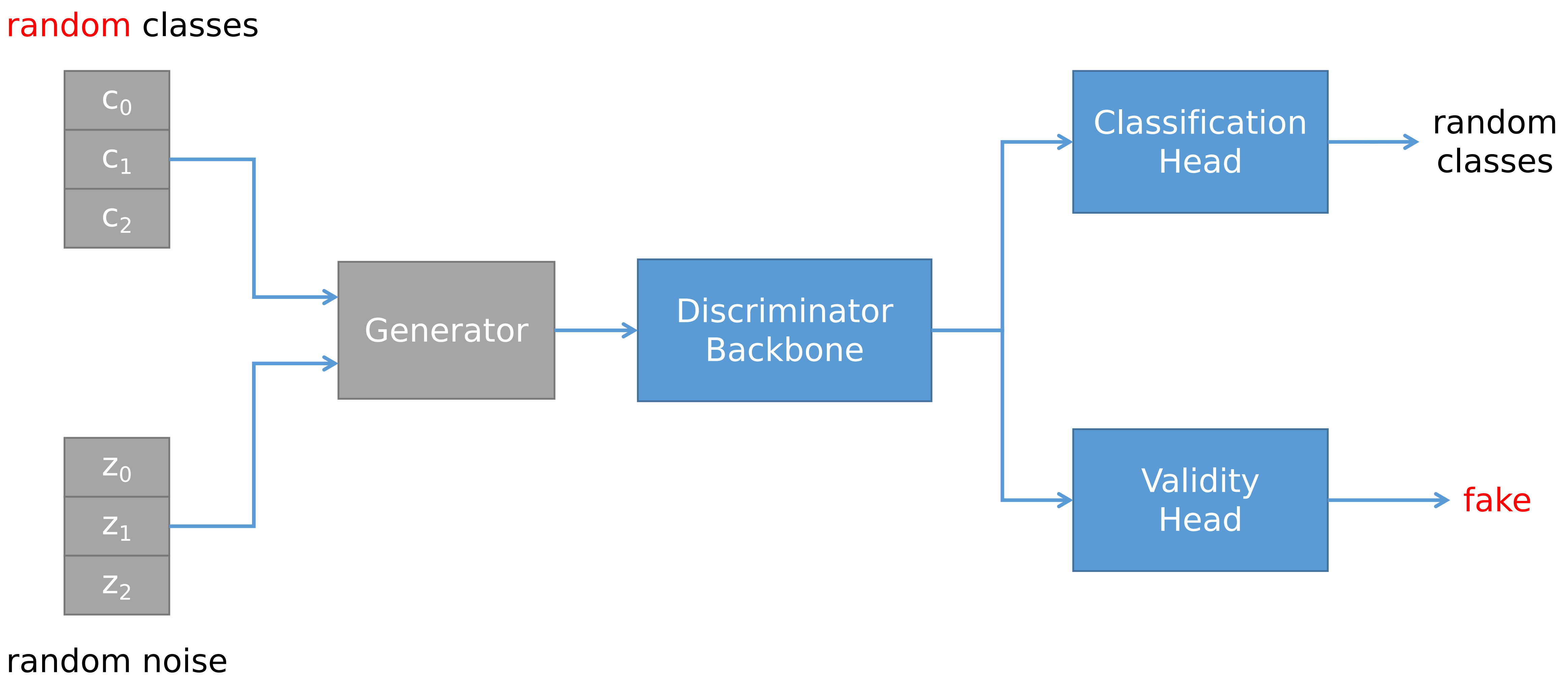
Auxiliary Classifier GAN (ACGAN) is a **variant** of the standard **GAN architecture** that **enhances** generative capabilities by **integrating** an **auxiliary classifier** into the **discriminator**.

It is **ideal** for **conditional image generation** and other tasks requiring **class-specifi**c data synthesis.

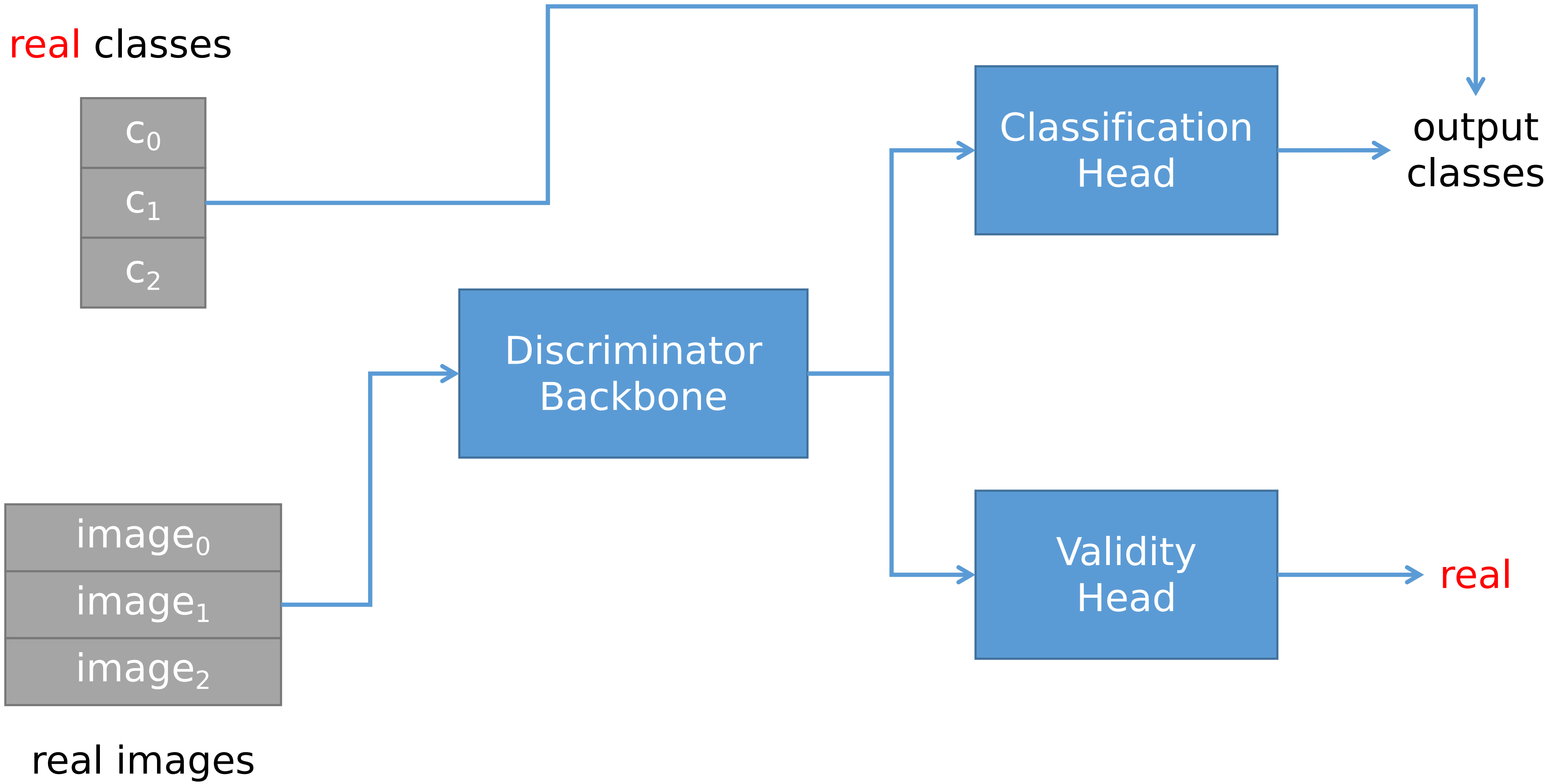
ACGANs: Generator Training



ACGANs: Discriminator Training (generated images)



ACGANs: Discriminator Training (real images)



GAN training ticks

Training a GAN is a very **unstable** process, and **finding** the right **balance** between the **generator** and **discriminator** can be **quite tricky**.

There are some known **tricks** to help **speed** up and **stabilize** the **training process**:

- **Normalize** images between **-1** and **1**
  - Use torchvision.transforms.Normalize((0.5,), (0.5,)) for MNIST
- **Avoid MLP**, **use CNN** (called **DCGAN** Deep Convolutional GAN)
- Use torch.nn.**BatchNorm2d** every layer
- **Avoid MaxPool2d**, **use strided** convolutions
- **Initialize convolution weights** to almost **0** (normal mean 0 and std 0.02)
- **Initialize batchnorm weights** to almost **1** (normal mean 1 and std 0.02)

GAN training ticks: in PyTorch (Initialization)

```python
def init_weights(module):
    for m in module.modules():
        if isinstance(m, torch.nn.Conv2d):
            torch.nn.init.normal_(m.weight, 0, 0.02)
            torch.nn.init.constant_(m.bias, 0)
        if isinstance(m, torch.nn.BatchNorm2d):
            torch.nn.init.normal_(m.weight, 1.0, 0.02)
            torch.nn.init.constant_(m.bias, 0)
```

GAN training ticks: in PyTorch (Conv block)

```
class ConvBlock(torch.nn.Module):
    def __init__(self, *, in_channels, out_channels, upsample=False, kernel=3, stride=1, padding=1):
        super(ConvBlock, self).__init__()

        self.block = torch.nn.Sequential(
            torch.nn.Upsample(scale_factor=2) if upsample else torch.nn.Identity(),
            torch.nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size=kernel,
                stride=stride,
                padding=padding
            ),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.LeakyReLU()
        )

        init_weights(self.block)

    def forward(self, x):
        return self.block(x)
```

GAN training ticks: in PyTorch (Generator example)

```
class Generator(torch.nn.Module):
    def __init__(self, latent_dim, output_dim):
        super(Generator, self).__init__()
        self.latent_dim = latent_dim
        self.init_size = output_dim // 4
        self.latent_channels = 128

        self.proj_in_cnn = torch.nn.Linear(latent_dim, self.latent_channels*self.init_size**2)
        self.generator = torch.nn.Sequential(
            torch.nn.BatchNorm2d(self.latent_channels),
            ConvBlock(
                in_channels=self.latent_channels,
                out_channels=self.latent_channels//2,
                upsample=True
            ),
            ConvBlock(
                in_channels=self.latent_channels//2,
                out_channels=self.latent_channels//4,
                upsample=True
            ),
            torch.nn.Conv2d(self.latent_channels//4, 1, 3, stride=1, padding=1),
            torch.nn.Tanh()
        )

        init_weights(self.generator)
```

Exercise 1 (doable)

Train a **DCGAN** on the **MNIST** dataset

- Train the **DCGAN** for **20 epochs**
- After **each epoch** plot **10** random **images generated** by the **generator**
- Tip: be gentle, use a small learning rate (like 0.0002)

Exercise 2 **(very hard)**

Train a **ACGAN** on the **MNIST** dataset

- Train the **ACGAN** for **20 epochs**
- After **each epoch** plot **one** random **image** for **each class generated** by the **generator**
- **Be creative**, come up with a **teqnique** to **mix** the **class** and **noise** in a way that **preserve** both **randomness** and **class information**
- Tip: If your **ACGAN** does **not** generate **conditional** images (**ignore** the **class** information) you can **weight** the **validity** and **classification losses** (0.01 for the validity loss is a good number)
- Tip: You can **use** the **torch.nn.Embedding** to **project integers** into **vectors** (like the VQ-VAE codebook)