ITMO.Hack

ITMO ML LAB

BOTAN INVESTMENTS

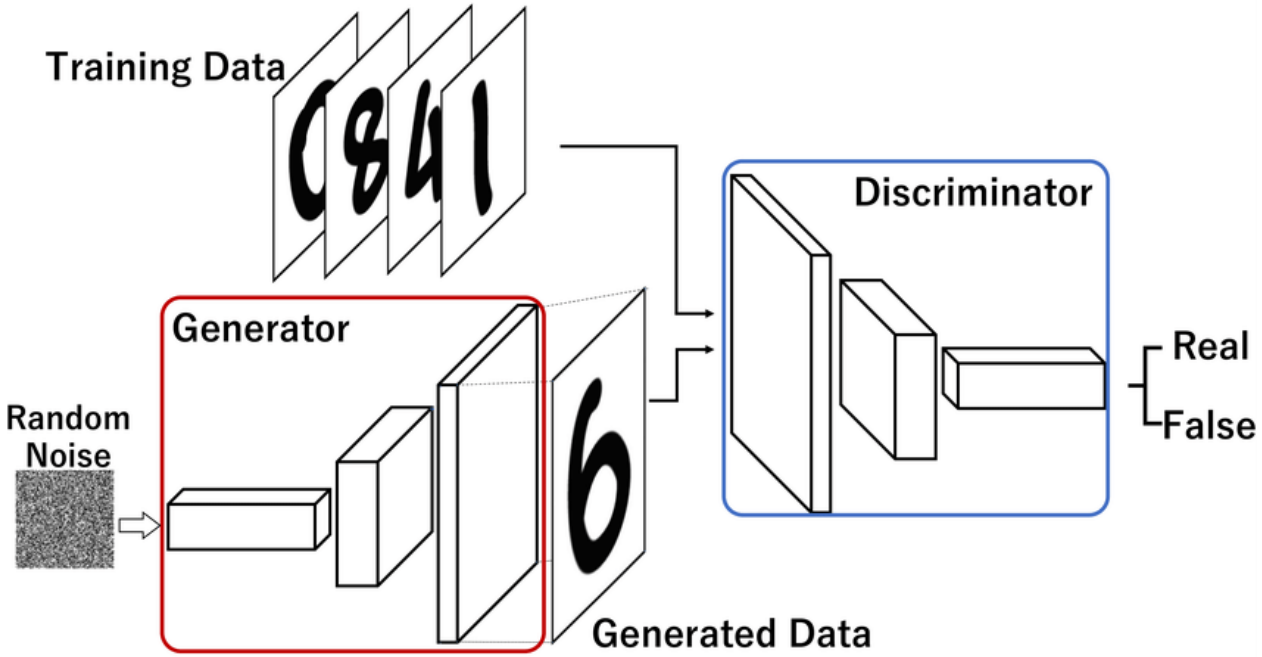
SOTA ML School

GANs for Image Generation

Tutorial

Igor Kuznetsov

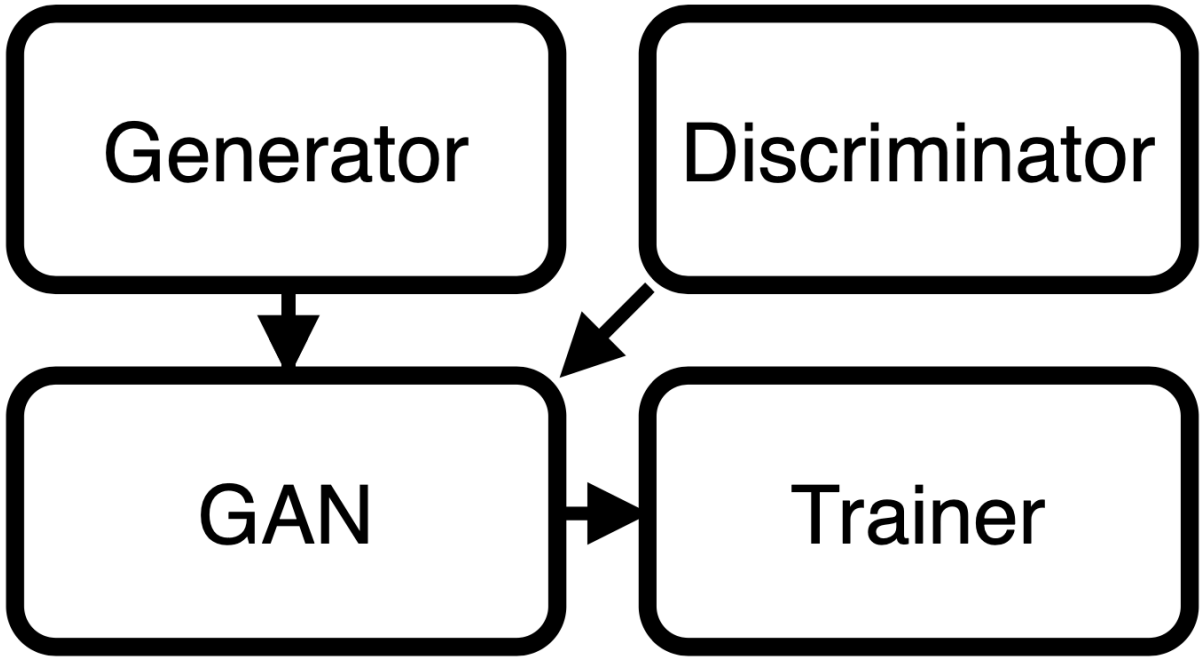
GANs Recap



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

Generative Adversarial Networks, 2014, Ian J. Goodfellow et. al

GAN Implementation



GAN Implementation: Generator

```python
class Generator(nn.Module):

    def __init__(self, z_dim=10, im_dim=784, hidden_dim=128):
        super(Generator, self).__init__()
        self._g = nn.Sequential(
            self._g_block(z_dim, hidden_dim),
            self._g_block(hidden_dim, hidden_dim * 2),
            self._g_block(hidden_dim * 2, hidden_dim * 4),
            self._g_block(hidden_dim * 4, hidden_dim * 8),
            nn.Linear(hidden_dim * 8, im_dim),
            nn.Sigmoid()
        )

    def _g_block(self, input_dim, output_dim):
        return nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.BatchNorm1d(output_dim),
            nn.ReLU(inplace=True))

    def forward(self, noise):
        return self._g(noise)
```

GAN Implementation: Discriminator

```python
class Discriminator(nn.Module):

    def __init__(self, im_dim=784, hidden_dim=128):
        super(Discriminator, self).__init__()
        self._d = nn.Sequential(
            self._d_block(im_dim, hidden_dim * 4),
            self._d_block(hidden_dim * 4, hidden_dim * 2),
            self._d_block(hidden_dim * 2, hidden_dim),
            nn.Linear(hidden_dim, 1))

    def _d_block(self, input_dim, output_dim):
        return nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.LeakyReLU(0.2))

    def forward(self, image):
        return self._d(image)
```

GAN Implementation: Model

```
class GAN:

    def __init__(self, z_dim, criterion, lr, device):
        self._g = Generator(z_dim).to(device)
        self._g_opt = t.optim.Adam(self._g.parameters(), lr=lr)

        self._d = Discriminator().to(device)
        self._d_opt = t.optim.Adam(self._d.parameters(), lr=lr)

        self._z_dim = z_dim
        self._criterion = criterion
        self._device = device
```

GAN Implementation: Losses

```python
def _calc_d_loss(self, batch_real, num_images):
    # Generate noise image nad get sample from G
    z = gen_noise(num_images, self._z_dim, self._device)
    fake_samples = self._g(z)

    # Get D's prediction for fake images
    y_fake_ = self._d(fake_samples.detach())
    fake_loss = self._criterion(y_fake_, t.zeros_like(y_fake_))

    # Get D's predition of real images
    y_real_ = self._d(batch_real)
    real_loss = self._criterion(y_real_, t.ones_like(y_real_))

    _d_loss = (fake_loss + real_loss) / 2.

    return _d_loss
```

GAN Implementation: Losses

```python
def _calc_d_loss(self, batch_real, num_images):
    # Generate noise image nad get sample from G
    z = gen_noise(num_images, self._z_dim, self._device)
    fake_samples = self._g(z)

    # Get D's prediction for fake images
    y_fake_ = self._d(fake_samples.detach())
    fake_loss = self._criterion(y_fake_, t.zeros_like(y_fake_))

    # Get D's predition of real images
    y_real_ = self._d(batch_real)
    real_loss = self._criterion(y_real_, t.ones_like(y_real_))

    _d_loss = (fake_loss + real_loss) / 2.

    return _d_loss
```

GAN Implementation: Losses

```python
def _calc_g_loss(self, num_images):
    # Generate fake images from noise vector
    z = gen_noise(num_images, self._z_dim, self._device)
    fake_samples = self._g(z)

    # Get D's prediction for the fake images
    y_ = self._d(fake_samples)
    _g_loss = self._criterion(y_, t.ones_like(y_))

    return _g_loss
```

GAN Implementation: Gradient Step

```python
def _update_step(self, batch_real, flatten_batch=True, *args):
    num_images = len(batch_real)
    if flatten_batch:
        batch_real = batch_real.view(num_images, -1)
    batch_real = batch_real.to(self._device)

    # D update
    self._d_opt.zero_grad()
    d_loss = self._calc_d_loss(batch_real, num_images)
    d_loss.backward(retain_graph=True)
    self._d_opt.step()

    # G update
    self._g_opt.zero_grad()
    g_loss = self._calc_g_loss(num_images)
    g_loss.backward()
    self._g_opt.step()

    return d_loss.item(), g_loss.item()
```

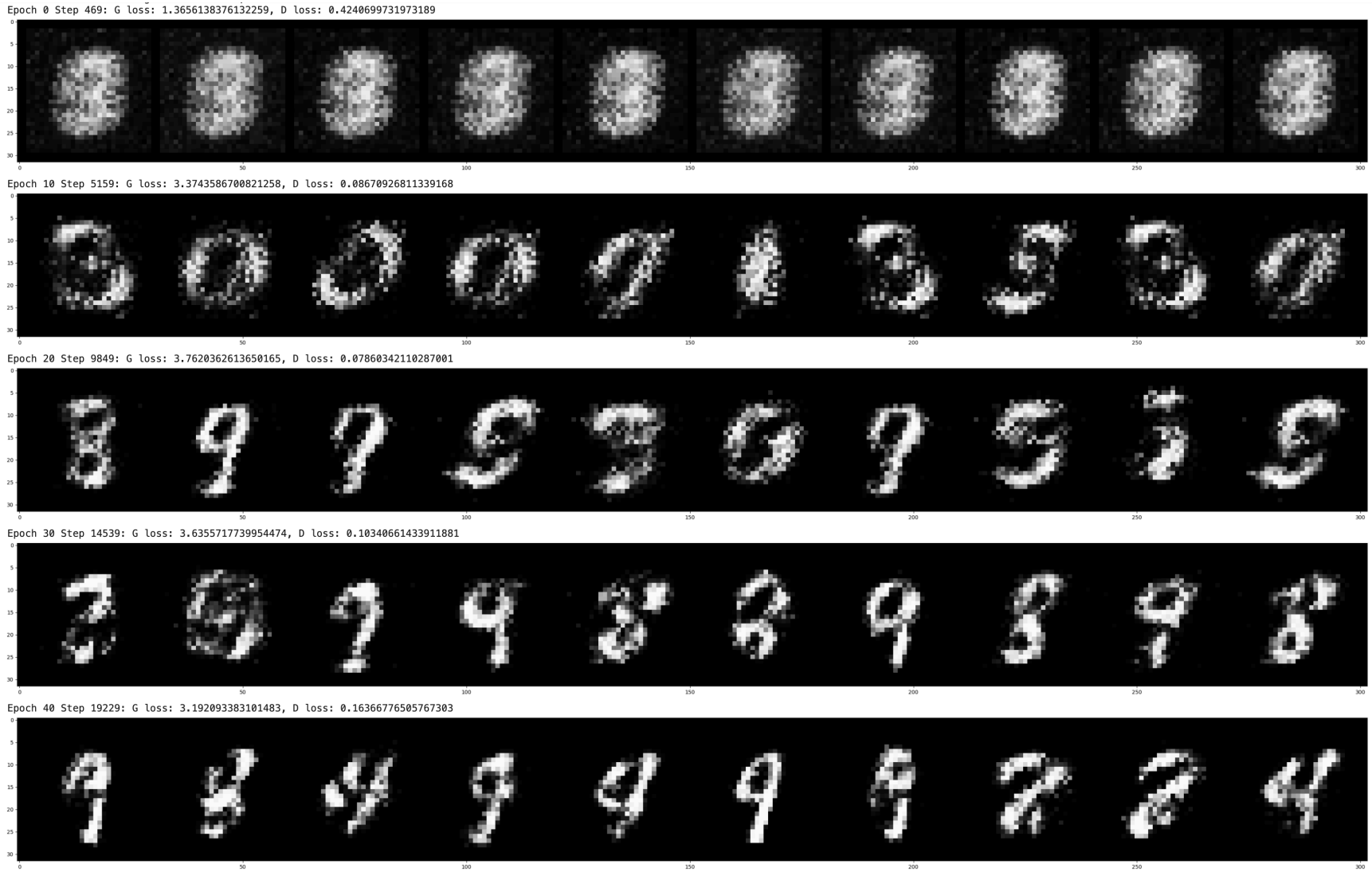
GAN Implementation: Trainer

```python
class Trainer:

    def __init__(self, data_loader, model, batch_size=128, device='cuda:0'):
        self.batch_size = batch_size
        self.dataloader = data_loader
        self.model = model

    def run(self, n_epochs, display_epoch=10, **kwargs):
        update_step = 0          # Total udpate step
        epoch_update_step = 0    # Update step within epoch
        mean_g_loss = 0          # Meach G loss for epoch
        mean_d_loss = 0          # Meach D loss for epoch
        display_epoch = 10       # Visualize once in
        device = config['device']

        for epoch in tqdm(range(n_epochs)):
            for batch_real, labels in self.dataloader:
                cur_batch_size = len(batch_real)

                d_loss, g_loss = self.model._update_step(batch_real,
                                                         labels=labels, **kwargs)
```

GAN Implementation: Training

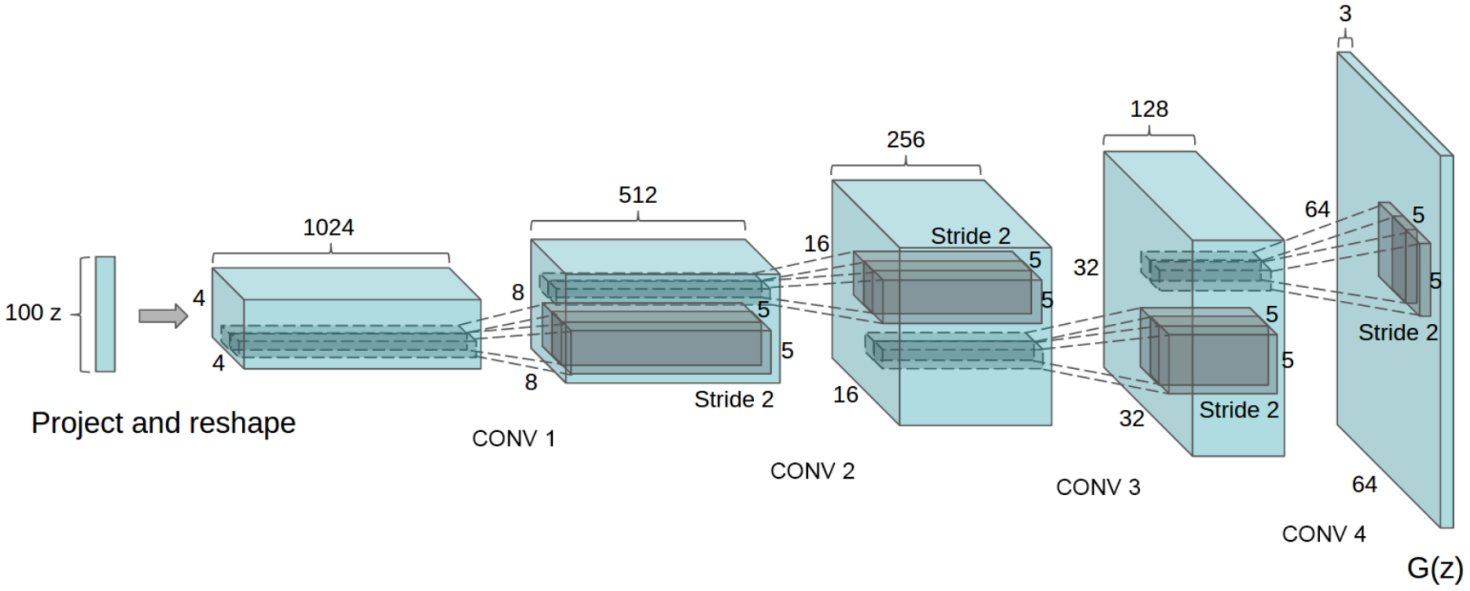
```
# Load MNIST dataset
batch_size = 128
dataloader = DataLoader(
    MNIST('.', download=True, transform=transforms.ToTensor()),
    batch_size=batch_size,
    shuffle=True)

config = {
    'criterion': nn.BCEWithLogitsLoss(),
    'z_dim': 64,
    'lr': 0.00001,
    'device': 'cuda:0'
}
model = GAN(**config)

trainer = Trainer(dataloader, model)
trainer.run(n_epochs=50)
```

GAN Implementation: Results



DCGAN

- no FC layers were used
- no pooling were used



DCGAN Generator

[Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks, 2016]

DCGAN Impelementation: Generator

```python
class DCGenerator(nn.Module):
    def __init__(self, z_dim=10, n_channels=1, hidden_dim=64):
        super(DCGenerator, self).__init__()
        self.z_dim = z_dim

        self._g = nn.Sequential(
            self._g_block(z_dim, hidden_dim * 4),
            self._g_block(hidden_dim * 4, hidden_dim * 2, kernel_size=4, stride=1),
            self._g_block(hidden_dim * 2, hidden_dim),
            nn.Sequential(
                nn.ConvTranspose2d(hidden_dim, n_channels, kernel_size=4, stride=2),
                nn.Tanh())
        )

    def _g_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
        ''' TransConv -> BN -> Activation. '''
        return nn.Sequential(
            nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
            nn.BatchNorm2d(output_channels),
            nn.ReLU())

    def unsqueeze_noise(self, noise):
        return noise.view(len(noise), self.z_dim, 1, 1)

    def forward(self, noise):
        x = self.unsqueeze_noise(noise)
        return self._g(x)
```

DCGAN Impelementation: Discriminator

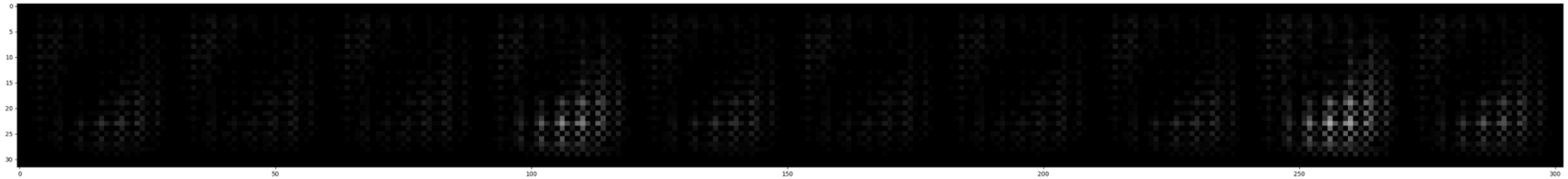
```
class DCDiscriminator(nn.Module):
    def __init__(self, n_channels=1, hidden_dim=16):
        super(DCDiscriminator, self).__init__()
        self._d = nn.Sequential(
            self._d_block(n_channels, hidden_dim),
            self._d_block(hidden_dim, hidden_dim * 2),
            nn.Sequential(
                nn.Conv2d(hidden_dim * 2, 1, kernel_size=4, stride=2)
            ))

    def _d_block(self, input_channels, output_channels, kernel_size=4, stride=2):
        ''' Conv -> BN -> Activation. '''
        return nn.Sequential(
            nn.Conv2d(input_channels, output_channels, kernel_size, stride),
            nn.BatchNorm2d(output_channels),
            nn.LeakyReLU(0.2))

    def forward(self, image):
        disc_pred = self._d(image)
        return disc_pred.view(len(disc_pred), -1)
```

DCGAN Impelementation: Model

```
class DCGAN(GAN):

    def __init__(self, z_dim, criterion, lr, betas, device):
        self._g = DCGenerator(z_dim).to(device)
        self._g_opt = t.optim.Adam(self._g.parameters(),
                                   lr=lr, betas=betas)

        self._d = DCDiscriminator().to(device)
        self._d_opt = t.optim.Adam(self._d.parameters(),
                                   lr=lr, betas=betas)

        # Initialize weights
        self._g = self._g.apply(_init_weights)
        self._d = self._d.apply(_init_weights)

        self._z_dim = z_dim
        self._criterion = criterion
        self._device = device
```

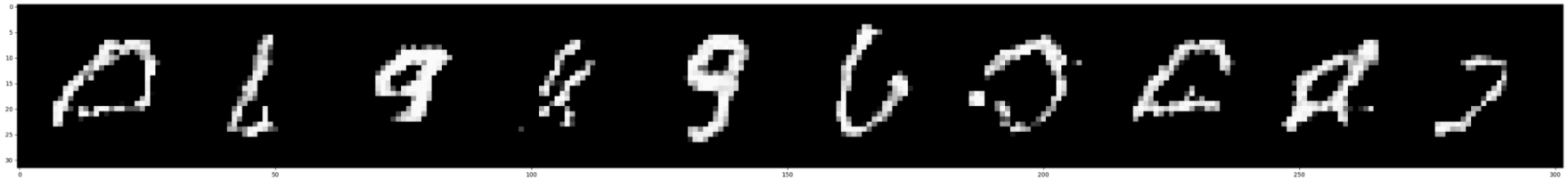
DCGAN Impelementation: Results

```
HBox(children=(FloatProgress(value=0.0, max=31.0), HTML(value='')))
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch 0 Step 469: G loss: 0.9324319826514482, D loss: 0.5064703847553684
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch 10 Step 5159: G loss: 0.7814101284857419, D loss: 0.6616245626827162
```

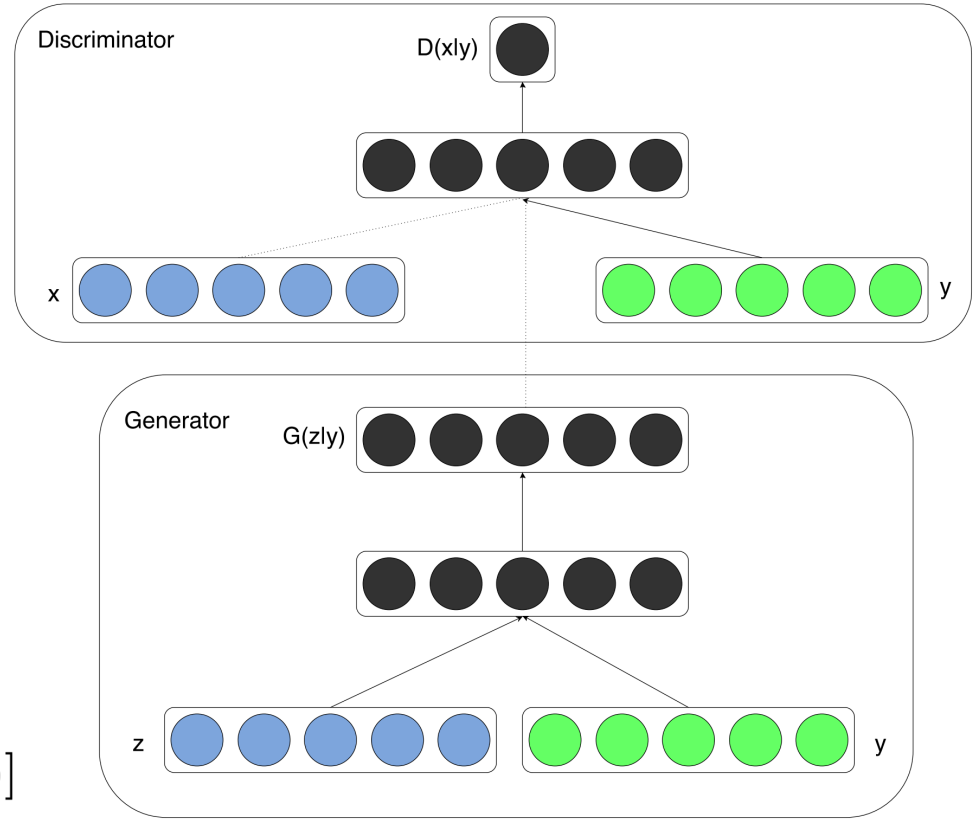
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch 20 Step 9849: G loss: 0.736259622136582, D loss: 0.6879140250196005
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Epoch 30 Step 14539: G loss: 0.706077394787859, D loss: 0.7003879404397938
```

Conditional GAN

- Examples can generated for particular class
- Training datsaet needs to be labeled



GAN minimax equation

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

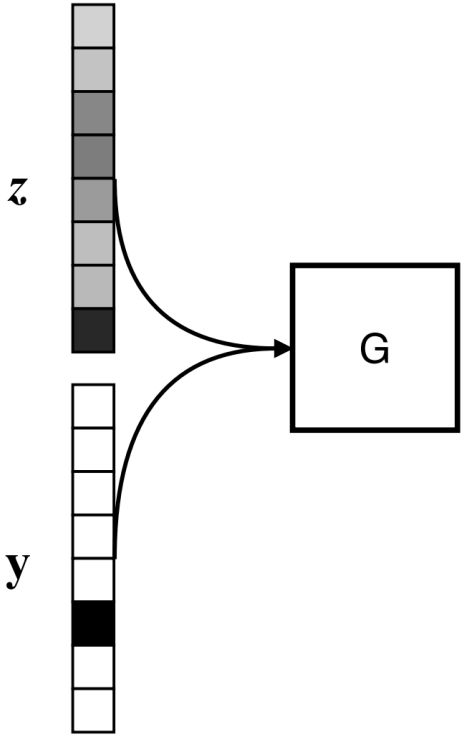
Conditional GAN minimax equation

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

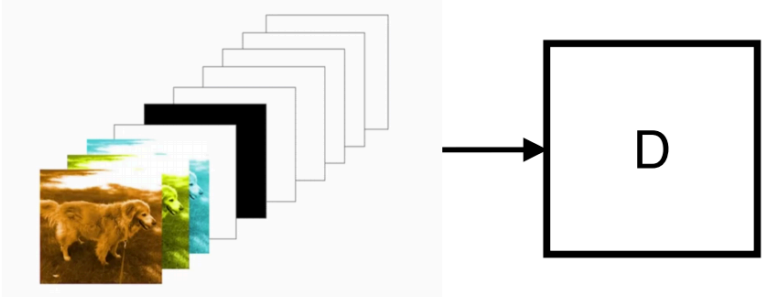
Conditional Generative Adversarial Nets, 2014

Conditional GAN: Input modification

Generator Input



Discriminator Input



Conditional GAN Implementation

```
class ConditionalGAN(GAN):

    def __init__(self, z_dim, criterion, lr, betas, device,
                 n_classes=10, data_shape=(1, 28, 28)):


        gen_dim = z_dim + n_classes
        self._g = DCGenerator(gen_dim).to(device)
        self._g_opt = t.optim.Adam(self._g.parameters(),
                                   lr=lr, betas=betas)

        d_dim = data_shape[0] + n_classes
        self._d = DCDiscriminator(d_dim).to(device)
        self._d_opt = t.optim.Adam(self._d.parameters(),
                                   lr=lr, betas=betas)

        # Initialize weights
        self._g = self._g.apply(_init_weights)
        self._d = self._d.apply(_init_weights)

        self._z_dim = z_dim
        self._criterion = criterion
        self._device = device
        self._n_classes = n_classes
        self._data_shape = data_shape
```

Conditional GAN Implementation

```python
def _update_step(self, real, labels, **kwargs):
    # Move to cuda
    real = real.to(self._device)
    labels = labels.to(self._device)

    # Get noise corresponding to the current batch_size
    one_hot_labels = calc_one_hot(labels, self._n_classes).squeeze(0)
    b_size = len(labels)
    z = gen_noise(b_size, self._z_dim, device=self._device)
    z_and_labels = t.cat([z, one_hot_labels], dim=-1)
    fake = self._g(z_and_labels)

    image_one_hot_labels = one_hot_labels[:, :, None, None]
    image_one_hot_labels = image_one_hot_labels.repeat(1, 1, self._data_shape[1], self._data_shape[2])

    self._d_opt.zero_grad()
    d_loss = self._calc_d_loss(real, fake, image_one_hot_labels)
    d_loss.backward(retain_graph=True)
    self._d_opt.step()

    self._g_opt.zero_grad()
    g_loss = self._calc_g_loss(fake, image_one_hot_labels)
    g_loss.backward()
    self._g_opt.step()

    return d_loss.item(), g_loss.item(), fake.cpu()
```

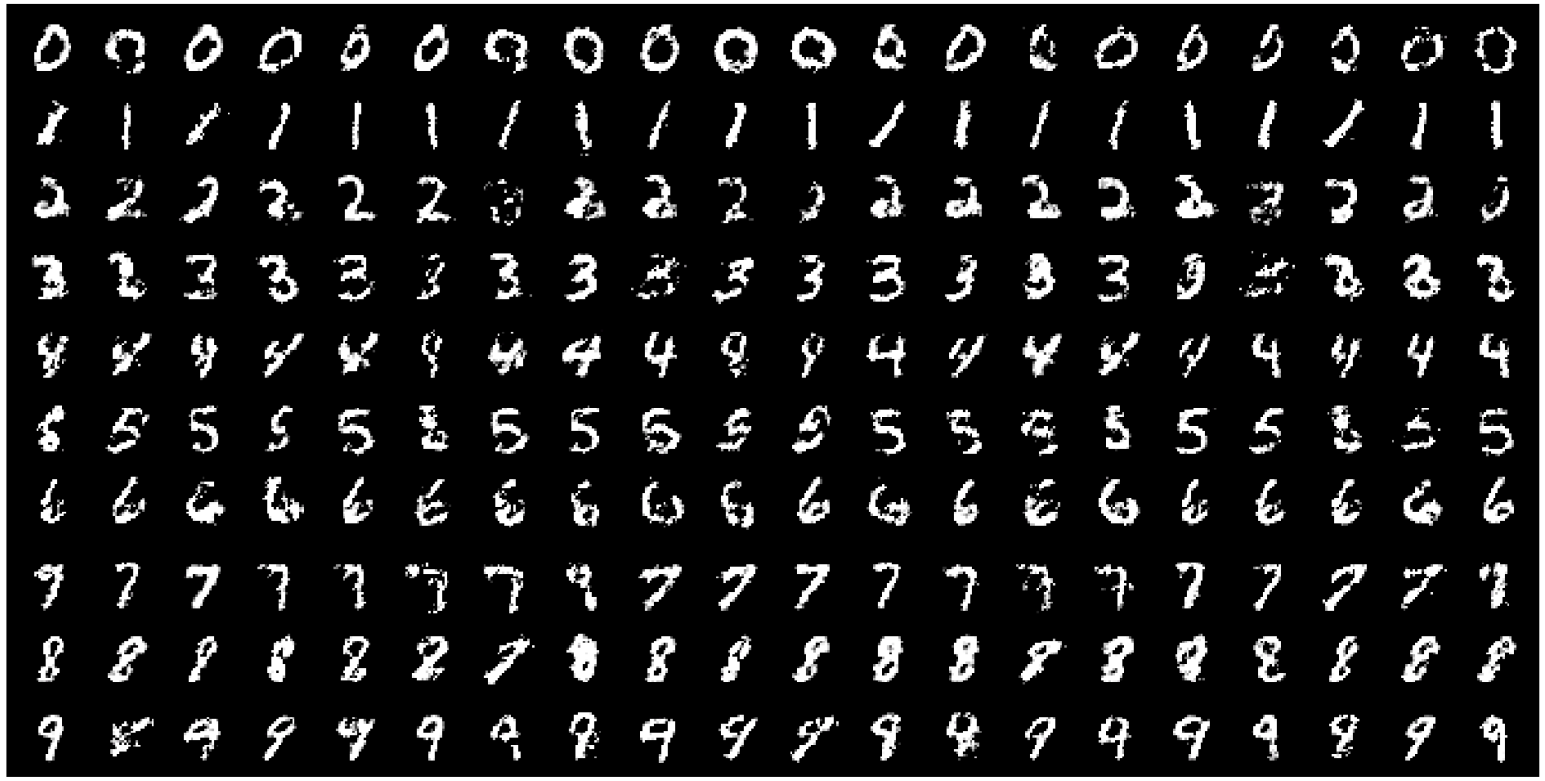
Conditional GAN Implementation: Losses

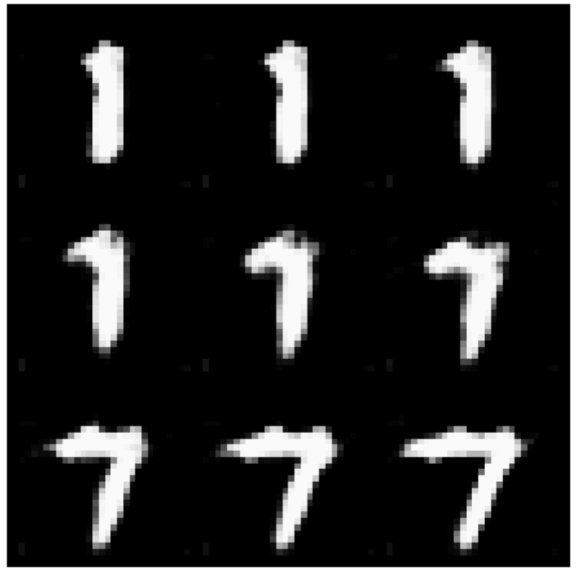
```
def _calc_d_loss(self, real, fake, image_one_hot_labels):

    fake_image_and_labels = t.cat([fake, image_one_hot_labels], dim=1)
    real_image_and_labels = t.cat([real, image_one_hot_labels], dim=1)
    d_fake_pred = self._d(fake_image_and_labels)
    d_real_pred = self._d(real_image_and_labels)

    d_fake_loss = self._criterion(d_fake_pred, t.zeros_like(d_fake_pred))
    d_real_loss = self._criterion(d_real_pred, t.ones_like(d_real_pred))
    d_loss = (d_fake_loss + d_real_loss) / 2


    return d_loss
```

Conditional GAN Implementation: Losses

```
def _calc_g_loss(self, fake, image_one_hot_labels):
    fake_image_and_labels = t.cat([fake, image_one_hot_labels], dim=1)
    d_fake_pred = self._d(fake_image_and_labels)

    g_loss = self._criterion(d_fake_pred, t.ones_like(d_fake_pred))

    return g_loss
```

Conditional GAN Implementation: Results





SOTA ML School

State-Of-The-Art Machine Learning School

Thanks

@sschatty