IX. PyTorch Implementation

Young-geun Kim

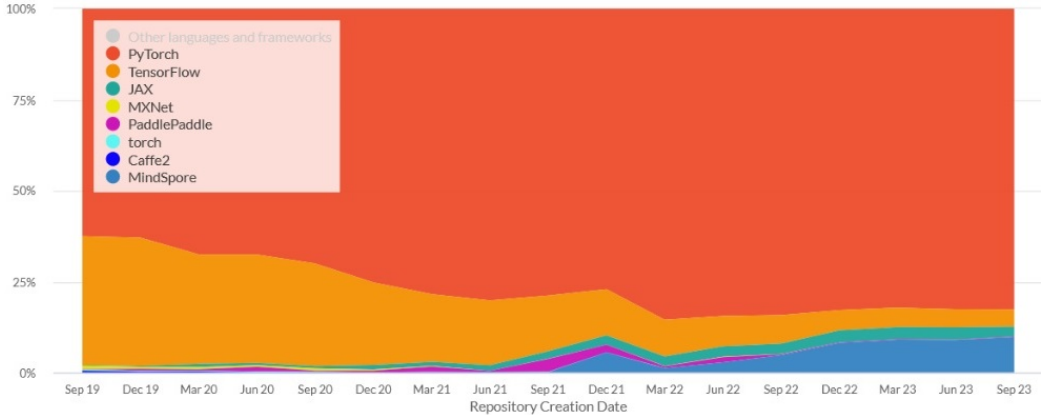
Department of Statistics and Probability

STT 997 (SS 2025)

MICHIGAN STATE
UNIVERSITY

Outline

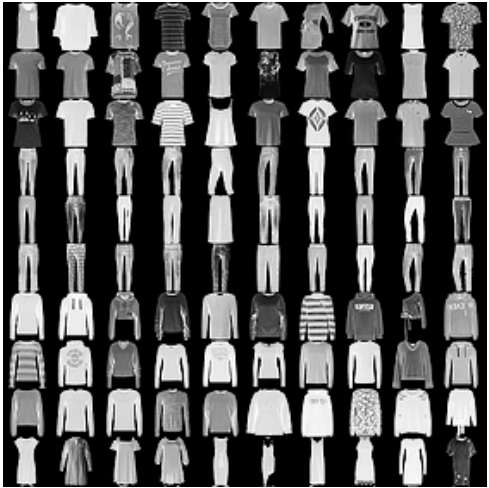
PyTorch



- All codes in this subsection are from `https://pytorch.org/tutorials/beginner/basics/intro.html`.

The figure is from `https://viso.ai/deep-learning/pytorch-vs-tensorflow/`.

Quickstart: Loading and Preprocessing Data



Images are from https://github.com/zalandoresearch/fashion-mnist?tab=readme-ov-file.

Quickstart: Loading and Preprocessing Data I

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
```

Quickstart: Loading and Preprocessing Data II

```
    train=False,
    download=True,
    transform=ToTensor(),
)

batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break

''' Output
```

Quickstart: Loading and Preprocessing Data III

```
Shape of X [N, C, H, W]: torch.Size([64, 1, 28, 28])
Shape of y: torch.Size([64]) torch.int64
'''
```

Quickstart: Building Models I

```
device = (torch.accelerator.current_accelerator().type
          if torch.accelerator.is_available() else "cpu")
print(f"Using {device} device")

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )
```

Quickstart: Building Models II

```
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)

''' Output
Using cuda device
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
```

Quickstart: Building Models III

```
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
'''
```

Quickstart: Model Training I

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

Quickstart: Model Training II

```
        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
```

Quickstart: Model Training III

```
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")

epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")

''' Output
Epoch 1
-------------------------------
loss: 2.303494  [   64/60000]
loss: 2.294637  [ 6464/60000]
loss: 2.277102  [12864/60000]
loss: 2.269977  [19264/60000]
```

Quickstart: Model Training IV

```
loss: 2.254235  [25664/60000]
loss: 2.237146  [32064/60000]
loss: 2.231055  [38464/60000]
loss: 2.205037  [44864/60000]
loss: 2.203240  [51264/60000]
loss: 2.170889  [57664/60000]
Test Error:
 Accuracy: 53.9%, Avg loss: 2.168588

...

Epoch 5
-------------------------------
loss: 1.337803  [   64/60000]
loss: 1.313278  [ 6464/60000]
loss: 1.151837  [12864/60000]
loss: 1.252142  [19264/60000]
```

Quickstart: Model Training V

```
loss: 1.123048  [25664/60000]
loss: 1.159531  [32064/60000]
loss: 1.175011  [38464/60000]
loss: 1.115554  [44864/60000]
loss: 1.160974  [51264/60000]
loss: 1.062730  [57664/60000]
Test Error:
 Accuracy: 64.6%, Avg loss: 1.087374

Done!
'''
```

Quickstart: Saving and Loading Trained Models I

```
torch.save(model.state_dict(), "model.pth")

model = NeuralNetwork().to(device)
model.load_state_dict(torch.load("model.pth", weights_only=True))
```

Quickstart: Prediction with Trained Models I

```
classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]

model.eval()
x, y = test_data[0][0], test_data[0][1]
with torch.no_grad():
    x = x.to(device)
```

Quickstart: Prediction with Trained Models II

```
    pred = model(x)
    predicted, actual = classes[pred[0].argmax(0)], classes[y]
    print(f'Predicted: "{predicted}", Actual: "{actual}"')

''' Output
Predicted: "Ankle boot", Actual: "Ankle boot"
'''
```

Tensor: Initializing a Tensor I

```
import torch
import numpy as np

# Directly from data
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

# From a NumPy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

Tensor: Attributes of a Tensor I

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")

''' Output
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
'''
```

Tensor: Operations on Tensors I

```
# We move our tensor to the current accelerator if available
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())

# Standard numpy-like indexing and slicing
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)

''' Output
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
```

Tensor: Operations on Tensors II

```
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
'''

# Joining tensors
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)

''' Output
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
'''

# Arithmetic operations
```

Tensor: Operations on Tensors III

```
# This computes the matrix multiplication between two tensors. y1, y2, y3
# will have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)

# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)

''' Output
```

Tensor: Operations on Tensors IV

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
'''

# Single-element tensors
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))

''' Output
12.0 <class 'float'>
'''

# In-place operations
print(f"{tensor} \n")
```

Tensor: Operations on Tensors V

```
tensor.add_(5)
print(tensor)

''' Output
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
'''
```

Datasets and DataLoaders I

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
```

Datasets and DataLoaders II

```
    download=True,
    transform=ToTensor()
)

# Preparing your data for training with DataLoaders
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)

# Iterate through the DataLoader
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
print(f"Label: {label}")
```

Datasets and DataLoaders III

```
''' Output
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
Label: 5
'''
```

Build Model I

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

device = (torch.accelerator.current_accelerator().type
          if torch.accelerator.is_available() else "cpu")

# Define the Class
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
```

Build Model II

```
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

# Create an instance of NeuralNetwork
model = NeuralNetwork().to(device)
print(model)

''' Output
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
```

Build Model III

```
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
'''

# Call the model and predict a label
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Build Model IV

```
''' Output
Predicted class: tensor([7], device='cuda:0')
'''

# Model Layers
input_image = torch.rand(3,28,28)
print(input_image.size())

''' Output
torch.Size([3, 28, 28])
'''

flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())

''' Output
```

Build Model V

```
torch.Size([3, 784])
'''

layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())

''' Output
torch.Size([3, 20])
'''

print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")

''' Output
Before ReLU: tensor([[ 0.4158, -0.0130, -0.1144,  0.3960,  0.1476, -0.0690, -0.0269,  0.2690,
```

Build Model VI

```
        0.1353,  0.1975,  0.4484,  0.0753,  0.4455,  0.5321, -0.1692,  0.4504,
        0.2476, -0.1787, -0.2754,  0.2462],
      [ 0.2326,  0.0623, -0.2984,  0.2878,  0.2767, -0.5434, -0.5051,  0.4339,
        0.0302,  0.1634,  0.5649, -0.0055,  0.2025,  0.4473, -0.2333,  0.6611,
        0.1883, -0.1250,  0.0820,  0.2778],
      [ 0.3325,  0.2654,  0.1091,  0.0651,  0.3425, -0.3880, -0.0152,  0.2298,
        0.3872,  0.0342,  0.8503,  0.0937,  0.1796,  0.5007, -0.1897,  0.4030,
        0.1189, -0.3237,  0.2048,  0.4343]], grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.4158, 0.0000, 0.0000, 0.3960, 0.1476, 0.0000, 0.0000, 0.2690, 0.1353,
         0.1975, 0.4484, 0.0753, 0.4455, 0.5321, 0.0000, 0.4504, 0.2476, 0.0000,
         0.0000, 0.2462],
        [0.2326, 0.0623, 0.0000, 0.2878, 0.2767, 0.0000, 0.0000, 0.4339, 0.0302,
         0.1634, 0.5649, 0.0000, 0.2025, 0.4473, 0.0000, 0.6611, 0.1883, 0.0000,
         0.0820, 0.2778],
        [0.3325, 0.2654, 0.1091, 0.0651, 0.3425, 0.0000, 0.0000, 0.2298, 0.3872,
```

Build Model VII

```
        0.0342, 0.8503, 0.0937, 0.1796, 0.5007, 0.0000, 0.4030, 0.1189, 0.0000,
        0.2048, 0.4343]], grad_fn=<ReluBackward0>)
'''

seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)

softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)

# Model parameters
```

Build Model VIII

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")

''' Output
Model structure: NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Build Model IX

```
Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[ 0.0273,
        [-0.0188, -0.0354,  0.0187,  ..., -0.0106, -0.0001,  0.0115]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([-0.0155, -0.0327]

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[ 0.0116,
        [ 0.0095,  0.0038,  0.0009,  ..., -0.0365, -0.0011, -0.0221]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([ 0.0148, -0.0256]

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[-0.0147, -
        [-0.0202, -0.0417, -0.0279,  ..., -0.0441,  0.0185, -0.0268]],
       device='cuda:0', grad_fn=<SliceBackward0>)
```

Build Model X

```
Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([ 0.0070, -0.0411],
'''
```

Optimization I

```
# Prerequisite Code
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
```

Optimization II

```
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
```

Optimization III

```
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()

# Hyperparameters
learning_rate = 1e-3
batch_size = 64
epochs = 5

# Loss function
# Initialize the loss function
loss_fn = nn.CrossEntropyLoss()
```

Optimization IV

```python
# Optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Full implementation
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
```

Optimization V

```
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")


def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are computed
```

Optimization VI

```
    # during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for
    # tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
```

Optimization VII

```
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")

''' Output
Epoch 1
-------------------------------
loss: 2.298730  [   64/60000]
loss: 2.289123  [ 6464/60000]
loss: 2.273286  [12864/60000]
loss: 2.269406  [19264/60000]
loss: 2.249603  [25664/60000]
loss: 2.229407  [32064/60000]
loss: 2.227368  [38464/60000]
loss: 2.204261  [44864/60000]
```

Optimization VIII

```
loss: 2.206193  [51264/60000]
loss: 2.166651  [57664/60000]
Test Error:
 Accuracy: 50.9%, Avg loss: 2.166725

Epoch 2
-------------------------------
loss: 2.176750  [   64/60000]
loss: 2.169595  [ 6464/60000]
loss: 2.117500  [12864/60000]
loss: 2.129272  [19264/60000]
loss: 2.079674  [25664/60000]
loss: 2.032928  [32064/60000]
loss: 2.050115  [38464/60000]
loss: 1.985236  [44864/60000]
loss: 1.987887  [51264/60000]
loss: 1.907162  [57664/60000]
```

Optimization IX

```
Test Error:
 Accuracy: 55.9%, Avg loss: 1.915486

Epoch 3
-------------------------------
loss: 1.951612  [   64/60000]
loss: 1.928685  [ 6464/60000]
loss: 1.815709  [12864/60000]
loss: 1.841552  [19264/60000]
loss: 1.732467  [25664/60000]
loss: 1.692914  [32064/60000]
loss: 1.701714  [38464/60000]
loss: 1.610632  [44864/60000]
loss: 1.632870  [51264/60000]
loss: 1.514263  [57664/60000]
Test Error:
 Accuracy: 58.8%, Avg loss: 1.541525
```

Optimization X

```
Epoch 4
-------------------------------
loss: 1.616448  [   64/60000]
loss: 1.582892  [ 6464/60000]
loss: 1.427595  [12864/60000]
loss: 1.487950  [19264/60000]
loss: 1.359332  [25664/60000]
loss: 1.364817  [32064/60000]
loss: 1.371491  [38464/60000]
loss: 1.298706  [44864/60000]
loss: 1.336201  [51264/60000]
loss: 1.232145  [57664/60000]
Test Error:
 Accuracy: 62.2%, Avg loss: 1.260237

Epoch 5
```

Optimization XI

```
-------------------------------
loss: 1.345538  [   64/60000]
loss: 1.327798  [ 6464/60000]
loss: 1.153802  [12864/60000]
loss: 1.254829  [19264/60000]
loss: 1.117322  [25664/60000]
loss: 1.153248  [32064/60000]
loss: 1.171765  [38464/60000]
loss: 1.110263  [44864/60000]
loss: 1.154467  [51264/60000]
loss: 1.070921  [57664/60000]
Test Error:
 Accuracy: 64.1%, Avg loss: 1.089831

Epoch 6
-------------------------------
loss: 1.166889  [   64/60000]
```

Optimization XII

```
loss: 1.170514  [ 6464/60000]
loss: 0.979435  [12864/60000]
loss: 1.113774  [19264/60000]
loss: 0.973411  [25664/60000]
loss: 1.015192  [32064/60000]
loss: 1.051113  [38464/60000]
loss: 0.993591  [44864/60000]
loss: 1.039709  [51264/60000]
loss: 0.971077  [57664/60000]
Test Error:
 Accuracy: 65.8%, Avg loss: 0.982440

Epoch 7
-------------------------------
loss: 1.045165  [   64/60000]
loss: 1.070583  [ 6464/60000]
loss: 0.862304  [12864/60000]
```

Optimization XIII

```
loss: 1.022265  [19264/60000]
loss: 0.885213  [25664/60000]
loss: 0.919528  [32064/60000]
loss: 0.972762  [38464/60000]
loss: 0.918728  [44864/60000]
loss: 0.961629  [51264/60000]
loss: 0.904379  [57664/60000]
Test Error:
 Accuracy: 66.9%, Avg loss: 0.910167

Epoch 8
-------------------------------
loss: 0.956964  [   64/60000]
loss: 1.002171  [ 6464/60000]
loss: 0.779057  [12864/60000]
loss: 0.958409  [19264/60000]
loss: 0.827240  [25664/60000]
```

Optimization XIV

```
loss: 0.850262  [32064/60000]
loss: 0.917320  [38464/60000]
loss: 0.868384  [44864/60000]
loss: 0.905506  [51264/60000]
loss: 0.856353  [57664/60000]
Test Error:
 Accuracy: 68.3%, Avg loss: 0.858248

Epoch 9
-------------------------------
loss: 0.889765  [   64/60000]
loss: 0.951220  [ 6464/60000]
loss: 0.717035  [12864/60000]
loss: 0.911042  [19264/60000]
loss: 0.786085  [25664/60000]
loss: 0.798370  [32064/60000]
loss: 0.874939  [38464/60000]
```

Optimization XV

```
loss: 0.832796  [44864/60000]
loss: 0.863254  [51264/60000]
loss: 0.819742  [57664/60000]
Test Error:
 Accuracy: 69.5%, Avg loss: 0.818780

Epoch 10
-------------------------------
loss: 0.836395  [   64/60000]
loss: 0.910220  [ 6464/60000]
loss: 0.668506  [12864/60000]
loss: 0.874338  [19264/60000]
loss: 0.754805  [25664/60000]
loss: 0.758453  [32064/60000]
loss: 0.840451  [38464/60000]
loss: 0.806153  [44864/60000]
loss: 0.830360  [51264/60000]
```

Optimization XVI

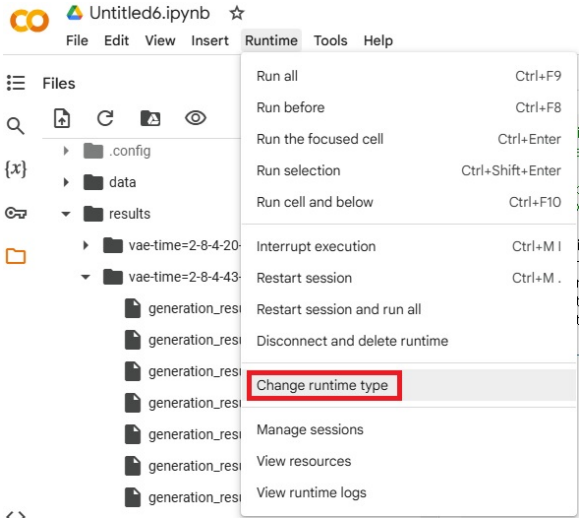
```
loss: 0.790281  [57664/60000]
Test Error:
 Accuracy: 71.0%, Avg loss: 0.787271

Done!
'''
```

Outline

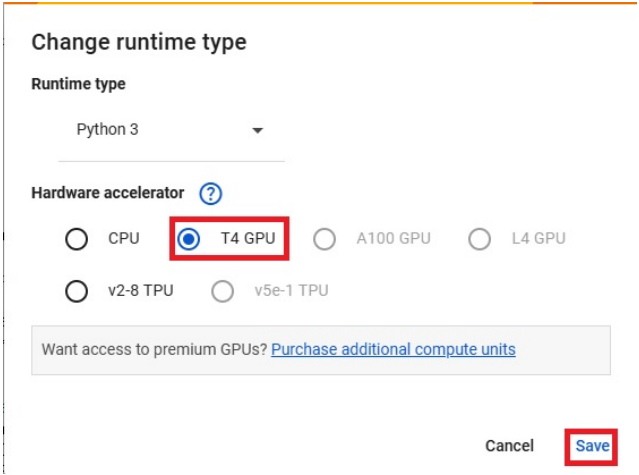
Variational Autoencoder

- The implementation code in this subsection is shared on D2L.
- The code took about 2,200 seconds to run in Google Colab using a T4 GPU.

Running Code using Google Colab GPU



Running Code using Google Colab GPU



Model: Network Architecture

Encoder architecture:

$$
\begin{aligned}
x \in \mathcal{R}^{28\times 28} &\to \text{Conv}_{128} \to \text{BN} \to \text{ReLU} \\
&\to \text{Conv}_{256} \to \text{BN} \to \text{ReLU} \\
&\to \text{Conv}_{512} \to \text{BN} \to \text{ReLU} \\
&\to \text{Conv}_{1024} \to \text{BN} \to \text{ReLU} \to \text{FC}_{8}
\end{aligned}
$$

Decoder architecture:

$$
\begin{aligned}
z \in \mathcal{R}^{8} &\to \text{FC}_{7\times 7\times 1024} \\
&\to \text{FSConv}_{512} \to \text{BN} \to \text{ReLU} \\
&\to \text{FSConv}_{256} \to \text{BN} \to \text{ReLU} \to \text{FSConv}_{1}
\end{aligned}
$$

We utilize encoder-decoder architectures used in Tolstikhin et al. (2017).

Model: nn.Conv2d

Conv2d

CLASS torch.nn.Conv2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.

Model: nn.BatchNorm2d

BatchNorm2d

CLASS torch.nn.BatchNorm2d(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*) [SOURCE]

Applies Batch Normalization over a 4D input.

4D is a mini-batch of 2D inputs with additional channel dimension. Method described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html.

Model: nn.ReLU

ReLU

CLASS torch.nn.ReLU(*inplace=False*) [SOURCE]

Applies the rectified linear unit function element-wise.

$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html.

Model: nn.ConvTranspose2D

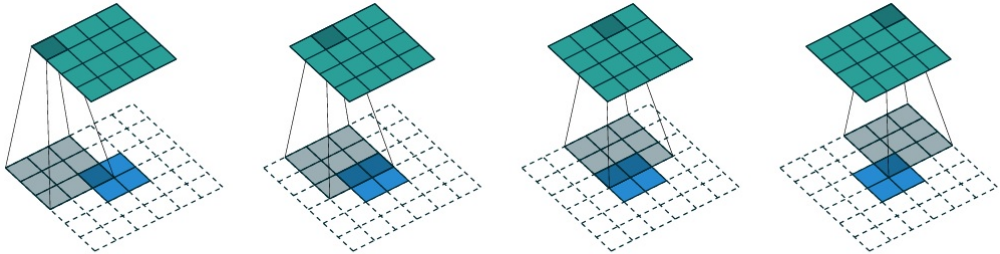


Figure 4.1: The transpose of convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$). It is equivalent to convolving a 3×3 kernel over a 2×2 input padded with a 2×2 border of zeros using unit strides (i.e., $i' = 2$, $k' = k$, $s' = 1$ and $p' = 2$).

The figure is from Dumoulin and Visin (2016).

Model: nn.ConvTranspose2D

ConvTranspose2d

CLASS torch.nn.ConvTranspose2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html.

Model: Encoder Class I

```
class Encoder(nn.Module):
    def __init__(self, dim_x, dim_z, nf=128):
        super(Encoder, self).__init__()
        self.dim_z = dim_z
        nc, resolution, _ = dim_x

        # input size: nc x 28 x 28
        self.c1 = nn.Sequential(
                nn.Conv2d(nc, nf, 4, 2, 1),
                nn.BatchNorm2d(nf),
                nn.ReLU(inplace=True),
                )
        # input size: nf x 14 x 14
        self.c2 = nn.Sequential(
                nn.Conv2d(nf, nf*2, 4, 2, 1),
                nn.BatchNorm2d(nf*2),
                nn.ReLU(inplace=True),
```

Model: Encoder Class II

```
        )
# input size: (nf*2) x 7 x 7
self.c3 = nn.Sequential(
        nn.Conv2d(nf*2, nf*4, 4, 2, 2),
        nn.BatchNorm2d(nf*4),
        nn.ReLU(inplace=True),
        )
# input size: (nf*4) x 4 x 4
self.c4 = nn.Sequential(
        nn.Conv2d(nf*4, nf*8, 4, 2, 1),
        nn.BatchNorm2d(nf*8),
        nn.ReLU(inplace=True),
        )
# input size: (nf*8) x 2 x 2
self.mu_net = nn.Conv2d(nf*8, dim_z, 2, 2, 0)
self.log_var_net = nn.Conv2d(nf*8, dim_z, 2, 2, 0)
```

Model: Encoder Class III

```
def forward(self, x_input):
    h1 = self.c1(x_input)
    h2 = self.c2(h1)
    h3 = self.c3(h2)
    h4 = self.c4(h3)
    mu, log_var = self.mu_net(h4), self.log_var_net(h4)
    return mu.view(-1, self.dim_z), log_var.view(-1, self.dim_z)
```

Model: Decoder Class I

```
class Decoder(nn.Module):
    def __init__(self, dim_z, dim_x, nf=128, final_activation='none'):
        super(Decoder, self).__init__()
        self.dim_z = dim_z
        nc, h, _ = dim_x

        # input size: dim_z x 1 x 1
        self.upc1 = nn.Sequential(
                  nn.ConvTranspose2d(dim_z, nf*8, 7, 1, 0),
                  )
        # input size: (nf*8) x 7 x 7
        self.upc2 = nn.Sequential(
                  nn.ConvTranspose2d(nf*8, nf*4, 4, 2, 1),
                  nn.BatchNorm2d(nf*4),
                  nn.ReLU(inplace=True),
                  )
        # input size: (nf*4) x 14 x 14
```

Model: Decoder Class II

```
self.upc3 = nn.Sequential(
          nn.ConvTranspose2d(nf*4, nf*2, 4, 2, 1),
          nn.BatchNorm2d(nf*2),
          nn.ReLU(inplace=True),
          )
# input size: (nf*2) x 28 x 28
if final_activation == 'sigmoid':
    self.mu_net = nn.Sequential(nn.ConvTranspose2d(nf*2, nc, 3, 1, 1),
                                nn.Sigmoid())
elif final_activation == 'tanh':
    self.mu_net = nn.Sequential(nn.ConvTranspose2d(nf*2, nc, 3, 1, 1),
                                nn.Tanh())
elif final_activation == 'none':
    self.mu_net = nn.Sequential(nn.ConvTranspose2d(nf*2, nc, 3, 1, 1))
else:
    print('available choices for final_activation: sigmoid, tanh, and none')
```

Model: Decoder Class III

```
        self.obs_log_var_net = nn.ConvTranspose2d(1, nc, h, 1, 0)

    def forward(self, z_input):
        d1 = self.upc1(z_input.view(-1, self.dim_z, 1, 1))
        d2 = self.upc2(d1)
        d3 = self.upc3(d2)
        o = self.mu_net(d3)

        device = z_input.get_device()
        if device == -1:
            one_tensor = torch.ones((1,1,1,1))
        else:
            one_tensor = torch.ones((1,1,1,1)).cuda()

        obs_log_var = self.obs_log_var_net(one_tensor)
        return o, obs_log_var
```

Basic Utility Functions

```
def sampling(mean, log_var):
    device = mean.get_device()
    if device == -1:
        epsilon = torch.randn(mean.shape)
        return mean + torch.exp(0.5 * log_var) * epsilon
    else:
        epsilon = torch.randn(mean.shape).cuda()
        return mean + torch.exp(0.5 * log_var).cuda() * epsilon

def kl_criterion(mu1, log_var1, mu2, log_var2):
    sigma1 = log_var1.mul(0.5).exp()
    sigma2 = log_var2.mul(0.5).exp()
    kld = torch.log(sigma2/sigma1) + ((sigma1**2) + (mu1 - mu2)**2)/(2.0*(sigma2**2)) - 0.5
    return torch.sum(kld, dim=-1)

def kl_annealing_weight(epoch, total_epochs):
    return min(1, epoch / (0.1*total_epochs))
```

Loss: Negative Evidence Lower Bound

- The evidence lower bound (ELBO) can be expressed as:

$$\begin{aligned}\text{ELBO}(\theta, \phi; \vec{x}) &:= \log p_\theta(\vec{x}) - \text{KL}(q_\phi(\vec{z}|\vec{x})||p_\theta(\vec{z}|\vec{x})) \\ &= -\int (\log p_\theta(\vec{x}|\vec{z}))\, q_\phi(\vec{z}|\vec{x}) d\vec{z} + \text{KL}(q_\phi(\vec{z}|\vec{x})||p(\vec{z})),\end{aligned} \tag{1}$$

- The reconstruction error can be computed via the Monte Carlo method:

$$-\int (\log p_\theta(\vec{x}|\vec{z}))\, q_\phi(\vec{z}|\vec{x}) d\vec{z} \approx -M^{-1} \sum_{m=1}^{M} \log p_\theta(\vec{x}|\vec{z}_m) \tag{2}$$

where $\vec{z}_m \sim q_\phi(\vec{z}|\vec{x})$, and

$$\log p_\theta(\vec{x}|\vec{z}_m) = -\frac{1}{2} \sum_{j=1}^{p} \log(2\pi\sigma^2_{X_j|\vec{Z}}(\vec{z}_m)) - \sum_{j=1}^{p} (x_j - \mu_{X_j|\vec{Z}}(\vec{z}_m))^2 / 2\sigma^2_{X_j|\vec{Z}}(\vec{z}_m). \tag{3}$$

Loss: Negative Evidence Lower Bound

- In sampling $\vec{z}_m \sim q_\phi(\vec{z}|x)$, the reparametrization (Kingma and Welling, 2014) technique is applied:

$$\vec{z}_m = \vec{\mu}_{\vec{Z}|\vec{X}}(\vec{x}) + \text{diag}((\sigma_{\vec{Z}|\vec{X}}(\vec{x}))_j)\vec{\epsilon}. \tag{4}$$

- The KL penalty term can be expressed as:

$$\frac{1}{2}\left(\sum_{j=1}^{p} \mu^2_{\vec{Z}_j|\vec{X}}(\vec{x}) + \sum_{j=1}^{p} \sigma^2_{Z_j|\vec{X}}(\vec{x}) - \sum_{j=1}^{p} \log \sigma^2_{Z_j|\vec{X}}(\vec{x}) - p\right). \tag{5}$$

Loss: Negative Evidence Lower Bound

```
z_mean, z_log_var = encoder(x_batch)
z_sample = sampling(z_mean, z_log_var)
fire_rate, obs_log_var = decoder(z_sample)

kl_weight = kl_annealing_weight(epoch, num_epoch)
if recon_error=='mse':
    obs_loglik = -0.5*torch.sum((fire_rate - x_batch)**2, dim=(1, 2, 3))
elif recon_error=='likelihood':
    obs_loglik = torch.sum(-0.5 * (obs_log_var + (x_batch - fire_rate)**2
                                   / torch.exp(obs_log_var)), dim=(1, 2, 3))
kl_post_prior = kl_criterion(z_mean, z_log_var,
                             torch.zeros_like(z_mean), torch.zeros_like(z_log_var))

elbo = beta_recon*obs_loglik - beta_kl*kl_weight*kl_post_prior
loss = torch.mean(-elbo)
```

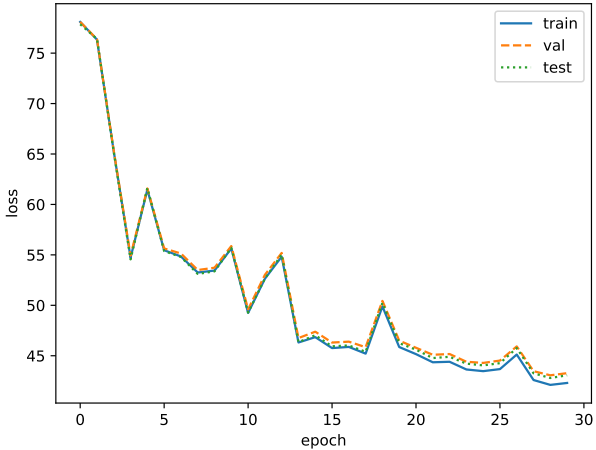
Implementation Detail I

```
# variables for data
p_train = 0.80
p_val = 1.0 - p_train

# variables for VAE architectures
dim_z = 8 # dimension of representations
nf = 64 # a factor to control the overall filter sizes in encoder and decoder networks
decoder_final_activation='none' # the last activation layer in decoder.

# variables for optimization
seed = 0 # the random seed number
num_epoch = 30 # the number of epoch
batch_size = 64 # the number of samples in each mini-batch
num_worker = 8 # the number of CPU cores
beta_recon = 1.0 # the coefficient of the log-likelihood
beta_kl = 1.0 # the coefficient of the KL-penalty term in ELBOs
Adam_beta1 = 0.5 # beta1 for Adam optimizer
```

Implementation Detail II

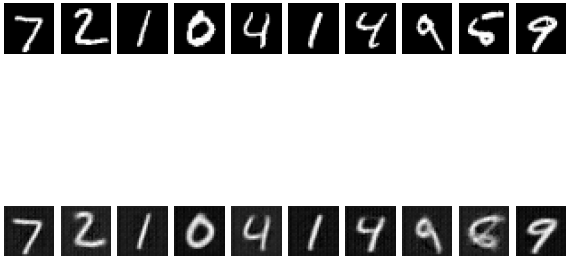
```
Adam_beta2 = 0.999 # beta2 for Adam optimizer
weight_decay = 5e-6 # the coefficient of the half of L2 penalty term
init_lr = 2e-4 # the initial learning rate
lr_milestones = [10, 20] # the epochs to reduce the learning rate
lr_gamma = 0.5 # the multiplier for each time learning rate is reduced
val_period = 1 # the frequency of evaluating trained models
recon_error = 'mse' # the type of reconstruction error (supports 'mse' and 'likelihood')
dtype = torch.float32 # the data type
result_path = None # the directory where results are saved.
# Its default value is results/vae-time=month-day-hour-min-sec.
```

Result: Loss Curve

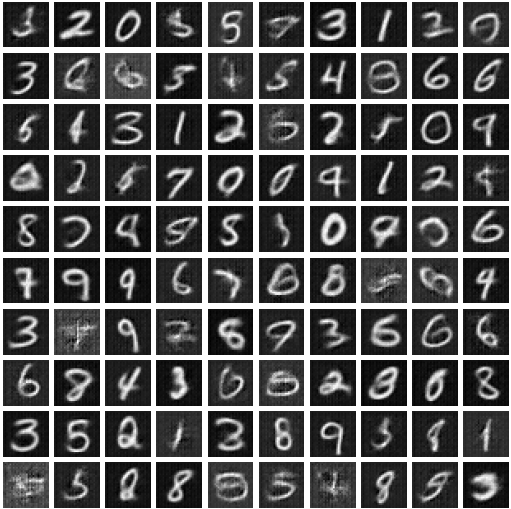


Result: Reconstruction on Training Data

Result: Reconstruction on Test Data

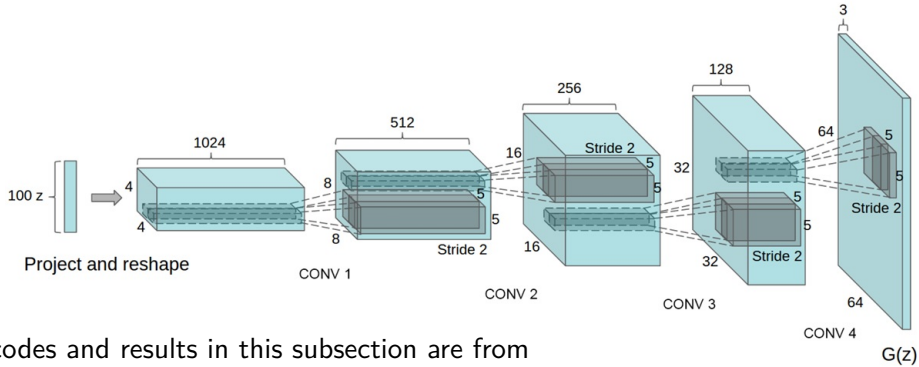


Result: Generation



Outline

Generative Adversarial Network



- All codes and results in this subsection are from `https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html`.

The figure is from Radford (2015).

Dataset Description: CelebA

- CelebFaces Attributes (CelebA) (Liu et al., 2015) is one of the large-scale facial image datasets.
- The dataset contains $202,599$ sample images, each annotated with 40 binary attributes.

The figure is from `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`.

Dataset Description: CelebA

Sample images are from `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`.

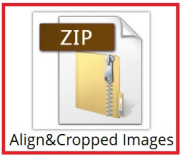
Dataset Description: Downloading CelebA

Oval Face

Smiling

Downloads

In-The-Wild Images



Align&Cropped Images

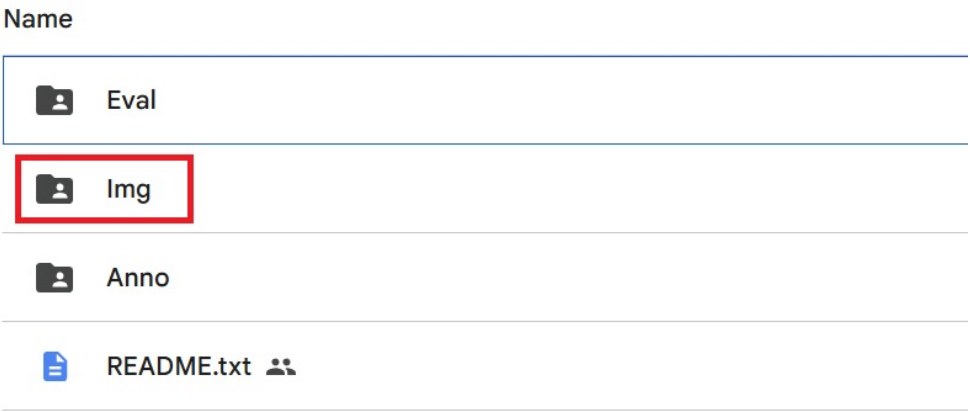
Landmarks Annotations

Attributes Annotations

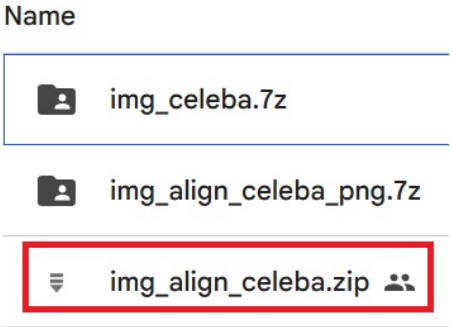
Identities Annotations

Evaluation

Dataset Description: Downloading CelebA



Dataset Description: Downloading CelebA



Loading Data I

```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
```

Building Models I

```
# custom weights initialization called on ``netG`` and ``netD``
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Generator Code
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
```

Building Models II

```
nn.BatchNorm2d(ngf * 8),
nn.ReLU(True),
# state size. ``(ngf*8) x 4 x 4``
nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf * 4),
nn.ReLU(True),
# state size. ``(ngf*4) x 8 x 8``
nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf * 2),
nn.ReLU(True),
# state size. ``(ngf*2) x 16 x 16``
nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
nn.BatchNorm2d(ngf),
nn.ReLU(True),
# state size. ``(ngf) x 32 x 32``
nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
nn.Tanh()
```

Building Models III

```
            # state size. ``(nc) x 64 x 64``
        )

    def forward(self, input):
        return self.main(input)

# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all weights
#  to ``mean=0``, ``stdev=0.02``.
netG.apply(weights_init)
```

Building Models IV

```
# Print the model
print(netG)

''' Output
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fal
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fal
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=Fals
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

Building Models V

```
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False
    (13): Tanh()
  )
)
'''

# Discriminator Code
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
```

Building Models VI

```
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
```

Building Models VII

```
        return self.main(input)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all weights
# like this: ``to mean=0, stdev=0.2``.
netD.apply(weights_init)

# Print the model
print(netD)

''' Output
```

Building Models VIII

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
```

Building Models IX

)
'''

Loss Functions and Optimizers I

```
# Initialize the ``BCELoss`` function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

Model Training I

```
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
```

Model Training II

```
## Train with all-real batch
netD.zero_grad()
# Format batch
real_cpu = data[0].to(device)
b_size = real_cpu.size(0)
label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
# Forward pass real batch through D
output = netD(real_cpu).view(-1)
# Calculate loss on all-real batch
errD_real = criterion(output, label)
# Calculate gradients for D in backward pass
errD_real.backward()
D_x = output.mean().item()

## Train with all-fake batch
# Generate batch of latent vectors
noise = torch.randn(b_size, nz, 1, 1, device=device)
```

Model Training III

```
# Generate fake image batch with G
fake = netG(noise)
label.fill_(fake_label)
# Classify all fake batch with D
output = netD(fake.detach()).view(-1)
# Calculate D's loss on the all-fake batch
errD_fake = criterion(output, label)
# Calculate the gradients for this batch, accumulated (summed) with previous gradients
errD_fake.backward()
D_G_z1 = output.mean().item()
# Compute error of D as sum over the fake and the real batches
errD = errD_real + errD_fake
# Update D
optimizerD.step()

############################
# (2) Update G network: maximize log(D(G(z)))
```

Model Training IV

```
############################
netG.zero_grad()
label.fill_(real_label)  # fake labels are real for generator cost
# Since we just updated D, perform another forward pass of all-fake batch through D
output = netD(fake).view(-1)
# Calculate G's loss based on this output
errG = criterion(output, label)
# Calculate gradients for G
errG.backward()
D_G_z2 = output.mean().item()
# Update G
optimizerG.step()

# Output training stats
if i % 50 == 0:
    print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4
          % (epoch, num_epochs, i, len(dataloader),
```

Model Training V

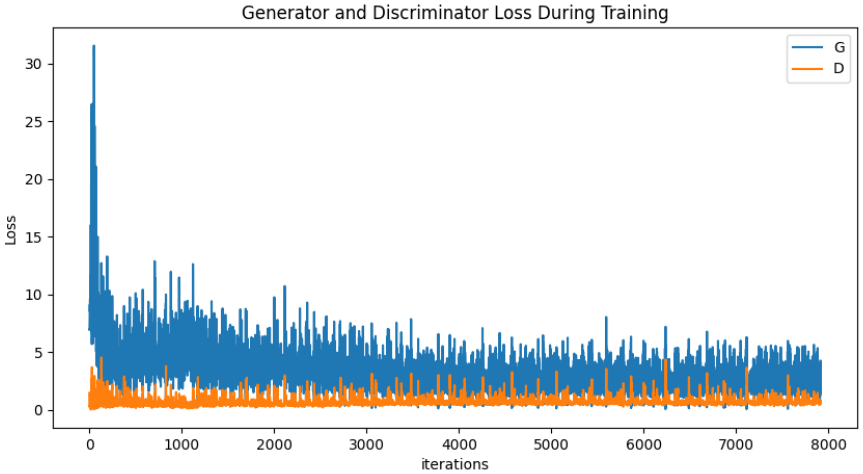
```
                errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

```
''' Output
Starting Training Loop...
[0/5][0/1583]   Loss_D: 1.4639  Loss_G: 6.9356  D(x): 0.7143    D(G(z)): 0.5877 / 0.0017
```

Model Training VI

```
[0/5][50/1583]  Loss_D: 0.3242  Loss_G: 31.5483 D(x): 0.8383    D(G(z)): 0.0000 / 0.0000
[0/5][100/1583] Loss_D: 0.6255  Loss_G: 4.1696  D(x): 0.7227    D(G(z)): 0.0358 / 0.0356
...
[4/5][1450/1583]        Loss_D: 0.9364  Loss_G: 5.0477  D(x): 0.8877    D(G(z)): 0.5022 / 0.01
[4/5][1500/1583]        Loss_D: 0.5947  Loss_G: 1.7611  D(x): 0.7653    D(G(z)): 0.2372 / 0.21
[4/5][1550/1583]        Loss_D: 1.4834  Loss_G: 0.6801  D(x): 0.3084    D(G(z)): 0.0380 / 0.55
'''
```

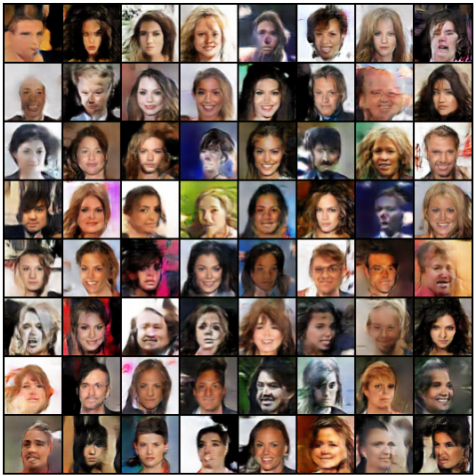
Result: Loss Curves



Result: Generation Result

Real Images

Fake Images



References I

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Liu, Z., Luo, P., Wang, X., and Tang, X. (2015). Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*.

Radford, A. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Tolstikhin, I., Bousquet, O., Gelly, S., and Schoelkopf, B. (2017). Wasserstein auto-encoders. *arXiv preprint arXiv:1711.01558*.