DGM_HW1

October 5, 2022

Giuseppe Concialdi

Libraries

[ ]:
```
import numpy as np
from matplotlib import pyplot as plt

import torch
from torch import nn
from torch import distributions
from torch import optim
from torch.nn import functional as F
import torchvision
from torchvision import transforms as T
from torch.utils.data import DataLoader
from torch import optim
from torch.utils.tensorboard import SummaryWriter
```

Utils

[ ]:
```
def draw_grid(imlist, m, n):
  fig, grid = plt.subplots(m,n)
  for i in range(m):
    for j in range(n):
      grid[i,j].axis('off')
      grid[i,j].imshow(np.reshape(imlist[(i-1)*m+j], (28,28)))
```

NADE

[ ]:
```
%reload_ext tensorboard
%tensorboard --logdir /content/CS594-HW1/runs/NADE/
```

```
Reusing TensorBoard on port 6006 (pid 384), started 0:14:09 ago. (Use '!kill␣
↪384' to kill it.)
```

```
<IPython.core.display.Javascript object>
```

[ ]:
```
class NADE(nn.Module):
  def __init__(self, input_dim, hidden_dim):
    super().__init__()
```

```
    self._input_dim = input_dim
    self._hidden_dim = hidden_dim

    self.W = nn.Parameter(torch.zeros(hidden_dim, input_dim))
    self.c = nn.Parameter(torch.zeros(hidden_dim,1))
    self.V = nn.Parameter(torch.zeros(input_dim, hidden_dim, 10))
    self.b = nn.Parameter(torch.zeros(input_dim,10))

    # He initialization
    nn.init.kaiming_normal_(self.W)
    nn.init.kaiming_normal_(self.V)


  def _forward(self, x):
    original_shape = x.shape
    if len(x.shape) > 2:
        x = x.view(original_shape[0], -1)
    flatten_shape = x.shape


    p_hat, x_hat = [], []
    batch_size = 1 if x is None else x.shape[0]

    a = self.c.expand(-1, batch_size).T
    for i in range(self._input_dim):
        h = torch.sigmoid(a) # hxb
        p_i = F.softmax(h @ self.V[i : i + 1, :, :].squeeze() + self.b[i : i + 
↪1, :], dim=1) # bxh @ hx10 + bx10 -> bx10

        p_hat.append(p_i)
        x_i = x[:, i : i + 1]
        if torch.any(x_i < 0):
            x_i = torch.multinomial(p_i, 1)

        x_hat.append(x_i)

        a = a + x_i.float() @ self.W[:, i : i + 1].T

    p_hat, x_hat = torch.stack(p_hat, dim=2), torch.cat(x_hat, dim=1).
↪view(flatten_shape)
    return p_hat, x_hat


  def forward(self, x):
    return self._forward(x)[0]
```

```
  def sample(self, n_samples=16, condition=None):
    with torch.no_grad():
      if not condition:
        condition = torch.ones(n_samples, 1, 28, 28) * -1
      return self._forward(condition)[1]

  def loss(self, x, preds):
      batch_size = x.shape[0]
      x = torch.transpose(F.one_hot(x.view(batch_size, -1), -1), 1, 2).float()
      loss = F.cross_entropy(preds, x, reduction='sum')
      return loss/batch_size

  def evaluation(self, test_loader):
    test_loss = []
    for data, _ in test_loader:
      x = torch.Tensor(data)
      y = data.clone()

      with torch.no_grad():
        output = self(x.float())
        test_loss.append(self.loss(y, output))

    return np.mean(test_loss)


  def checkpoint(self, dir, optim, epoch):
    checkpoint = {
            "model": self.state_dict(),
            "optimizer": optim.state_dict(),
            "epoch": epoch
        }
    torch.save(checkpoint, f'{dir}checkpoint.pth')
```

Training NADE

[ ]:
```
model_nade = NADE(input_dim=784, hidden_dim=500)
optimizer = optim.Adam(model_nade.parameters())
batch_size=100
n_epochs = 10
drive = '/content/CS594-HW1/runs/NADE/'

writer = SummaryWriter(drive)



transforms = T.Compose([T.Lambda(lambda t : np.round((np.array(t) / 27)).
 ↪astype(int)), T.ToTensor()])
```

```
training_data = torchvision.datasets.FashionMNIST("dataset", download=True,␣
↪train=True, transform=transforms)
test_data = torchvision.datasets.FashionMNIST("dataset", download=True,␣
↪train=False, transform=transforms)

train_loader = DataLoader(training_data, shuffle=True, batch_size=batch_size,␣
↪num_workers=8)
test_loader = DataLoader(test_data, shuffle=True, batch_size=1, num_workers=8)

for epoch in range(n_epochs):
  ## Training
  train_loss = []
  model_nade.train()
  for i, (data, _) in enumerate(train_loader):
    x = torch.Tensor(data)
    y = data.clone()

    optimizer.zero_grad()
    ## 1. forward propagation
    output = model_nade(x.float())

    ## 2. loss
    loss = model_nade.loss(y, output)
    #writer.add_scalar('NLL/train/iter', loss, (epoch + 1) * (i + 1))

    ## 3. backward propagation
    loss.backward()

    ## 4. weight optimization
    optimizer.step()

    train_loss.append(loss.item())

  model_nade.checkpoint(drive, optimizer, epoch)
  train_loss_epoch = np.mean(train_loss)
  #writer.add_scalar('NLL/train/epoch', train_loss_epoch, epoch + 1)
  test_loss_epoch = model_nade.evaluation(test_loader)
  #writer.add_scalar('NLL/test/epoch', test_loss_epoch, epoch + 1)

  print (f"Epoch: {epoch}, Training Loss: {train_loss_epoch:.2f}, Evaluation␣
↪Loss: {test_loss_epoch:.2f}")
```

Samples NADE

[ ]:
```
n_samples = 16
samples = model_nade.sample(n_samples=n_samples)
draw_grid(samples, 4,4)
```

PixelRNN

[ ]:
```
%reload_ext tensorboard
%tensorboard --logdir /content/CS594-HW1/runs/PixelRNN/
```

```
<IPython.core.display.Javascript object>
```

[ ]:
```
def _padding(i, o, k, s=1, d=1, mode='same'):
  if mode == 'same':
    return ((o-1) * s + (k-1)*(d-1) + k - i) // 2
  else:
    raise RuntimeError('Not implemented')


class MaskedConv2d(nn.Conv2d):
  def __init__(self, *args, mask='B', **kargs):
    super(MaskedConv2d, self).__init__(*args, **kargs)
    self.mask_type = mask
    self.register_buffer('mask', self.weight.data.clone())
    self.mask.fill_(1)

    _, _, H, W = self.mask.size()

    self.mask[:, :, H//2,W//2 + (self.mask_type == 'B'):] = 0
    self.mask[:, :, H//2+1:, :] = 0

  def forward(self, x):
    self.weight.data *= self.mask
    return super(MaskedConv2d, self).forward(x)

class MaskedConv1d(nn.Conv1d):
  def __init__(self, *args, mask='B', **kargs):
    super(MaskedConv1d, self).__init__(*args, **kargs)
    self.mask_type = mask
    self.register_buffer('mask', self.weight.data.clone())
    self.mask.fill_(1)

    _, _, W = self.mask.size()

    self.mask[:, :, W//2 + (self.mask_type == 'B'):] = 0

  def forward(self, x):
    self.weight.data *= self.mask
    return super(MaskedConv1d, self).forward(x)

class RowLSTMCell(nn.Module):
  def __init__(self, hidden_dims, image_size, channel_in, *args, **kargs):
    super(RowLSTMCell, self).__init__(*args, **kargs)
```

```
    self._hidden_dims = hidden_dims
    self._image_size = image_size
    self._channel_in = channel_in
    self._num_units = self._hidden_dims * self._image_size
    self._output_size = self._num_units
    self._state_size = self._num_units * 2

    self.conv_i_s = MaskedConv1d(self._hidden_dims, 4 * self._hidden_dims, 3, mask='B', padding=_padding(image_size, image_size, 3))
    self.conv_s_s = nn.Conv1d(channel_in, 4 * self._hidden_dims, 3, padding=_padding(image_size, image_size, 3))

  def forward(self, inputs, states):
    c_prev, h_prev = states

    h_prev = h_prev.view(-1, self._hidden_dims,  self._image_size)
    inputs = inputs.view(-1, self._channel_in, self._image_size)

    s_s = self.conv_s_s(h_prev)
    i_s = self.conv_i_s(inputs)


    s_s = s_s.view(-1, 4 * self._num_units)
    i_s = i_s.view(-1, 4 * self._num_units)

    lstm = s_s + i_s

    lstm = torch.sigmoid(lstm)

    i, g, f, o = torch.split(lstm, (4 * self._num_units)//4, dim=1)

    c = f * c_prev + i * g
    h = o * torch.tanh(c)

    new_state = (c, h)
    return h, new_state

class RowLSTM(nn.Module):
  def __init__(self, hidden_dims, input_size, channel_in, *args, **kargs):
    super(RowLSTM, self).__init__(*args, **kargs)
    self._hidden_dims = hidden_dims
    self.init_state = (torch.zeros(1, input_size * hidden_dims), torch.zeros(1, input_size * hidden_dims))

    self.lstm_cell = RowLSTMCell(hidden_dims, input_size, channel_in)
```

```python
  def forward(self, inputs, initial_state=None):

    n_batch, channel, n_seq, width = inputs.size()
    if initial_state is None:
        hidden_init, cell_init = self.init_state

    else:
        hidden_init, cell_init = initial_state

    states = (hidden_init.repeat(n_batch,1), cell_init.repeat(n_batch, 1))

    steps = []
    for seq in range(n_seq):
        h, states = self.lstm_cell(inputs[:, :, seq, :], states)
        steps.append(h.unsqueeze(1))

    return torch.cat(steps, dim=1).view(-1, n_seq, width, self._hidden_dims).
 ↪permute(0,3,1,2)

class PixelRNN(nn.Module):
    def __init__(self, num_layers, hidden_dims, input_size, *args, **kargs):
      super(PixelRNN, self).__init__(*args, **kargs)
      pad_conv1 = _padding(input_size, input_size, 7)
      self.conv1 = MaskedConv2d(1, hidden_dims, (7,7), mask='A',␣
 ↪padding=(pad_conv1, pad_conv1))
      self.lstm_list = nn.ModuleList([RowLSTM(hidden_dims, input_size,␣
 ↪hidden_dims) for _ in range(num_layers)])
      self.linear = nn.Linear(hidden_dims, 10)


    def forward(self, inputs):
      x = self.conv1(inputs)
      for lstm in self.lstm_list:
        x = lstm(x)
      x = self.linear(x.transpose(1, 3))
      x = torch.sigmoid(x.transpose(1,3))
      return x


    def loss(self, x, preds):
      batch_size = x.shape[0]
      x = torch.transpose(F.one_hot(x.view(batch_size, -1), -1), 1, 2).float()
      preds = preds.reshape(batch_size, 10, -1)
      loss = F.cross_entropy(preds, x, reduction='sum')
      return loss/batch_size
```

```
    def evaluation(self, test_loader):
      test_loss = []
      for data, _ in test_loader:
        x = torch.Tensor(data)
        y = data.clone()

        with torch.no_grad():
          output = self(x.float())
          test_loss.append(self.loss(y, output))

      return np.mean(test_loss)

    def checkpoint(self, dir, optim, epoch):
      checkpoint = {
              "model": self.state_dict(),
              "optimizer": optim.state_dict(),
              "epoch": epoch
          }
      torch.save(checkpoint, f'{dir}checkpoint.pth')

    def sample(self, n_samples=16, condition=None):
      with torch.no_grad():
        condition = torch.ones(n_samples, 1, 28, 28)
        return self.forward(condition)
```

Training PixelRNN

[ ]:
```
model_pixelrnn = PixelRNN(num_layers=2, hidden_dims=16, input_size=28)
optimizer = optim.RMSprop(model_pixelrnn.parameters())
batch_size=100
n_epochs = 10
drive = '/content/CS594-HW1/runs/PixelRNN/'

writer = SummaryWriter(drive)


transforms = T.Compose([T.Lambda(lambda t : np.round((np.array(t) / 27)).
 ↪astype(int)), T.ToTensor()])

training_data = torchvision.datasets.FashionMNIST("dataset", download=True,␣
 ↪train=True, transform=transforms)
test_data = torchvision.datasets.FashionMNIST("dataset", download=True,␣
 ↪train=False, transform=transforms)

train_loader = DataLoader(training_data, shuffle=True, batch_size=batch_size,␣
 ↪num_workers=8)
test_loader = DataLoader(test_data, shuffle=True, batch_size=1, num_workers=8)
```

```
for epoch in range(n_epochs):
  ## training part
  train_loss = []
  model_pixelrnn.train()
  for i, (data, _) in enumerate(train_loader):
    x = torch.Tensor(data)
    y = data.clone()

    optimizer.zero_grad()
    ## 1. forward propagation
    output = model_pixelrnn(x.float())

    ## 2. loss calculation
    loss = model_pixelrnn.loss(y, output)

    #writer.add_scalar('NLL/train/iter', loss, (epoch + 1) * (i + 1))

    ## 3. backward propagation
    loss.backward()

    ## 4. weight optimization
    optimizer.step()

    train_loss.append(loss.item())

  model_pixelrnn.checkpoint(drive, optimizer, epoch)
  train_loss_epoch = np.mean(train_loss)
  #writer.add_scalar('NLL/train/epoch', train_loss_epoch, epoch + 1)
  test_loss_epoch = model_pixelrnn.evaluation(test_loader)
  #writer.add_scalar('NLL/test/epoch', test_loss_epoch, epoch + 1)

  print (f"Epoch: {epoch}, Training Loss: {train_loss_epoch:.2f}, Evaluation␣
↪Loss: {test_loss_epoch:.2f}")
```

Transformer

[ ]:
```
%reload_ext tensorboard
%tensorboard --logdir '/content/CS594-HW1/runs/PixelRNN/'
```

```
<IPython.core.display.Javascript object>
```

[ ]:
```
class DecoderLayer(nn.Module):
    def __init__(self, hparams):
        super().__init__()
        self.attn = Attn(hparams)
        self.hparams = hparams
        self.dropout = nn.Dropout(p=hparams.dropout)
```

```
        self.layernorm_attn = nn.LayerNorm([self.hparams.hidden_size],␣
↪eps=1e-6, elementwise_affine=True)
        self.layernorm_ffn = nn.LayerNorm([self.hparams.hidden_size], eps=1e-6,␣
↪elementwise_affine=True)
        self.ffn = nn.Sequential(nn.Linear(self.hparams.hidden_size, self.
↪hparams.filter_size, bias=True),
                                 nn.ReLU(),
                                 nn.Linear(self.hparams.filter_size, self.
↪hparams.hidden_size, bias=True))

    def preprocess_(self, x):
        return x

    def forward(self, x):
        x = self.preprocess_(x)
        y = self.attn(x)
        x = self.layernorm_attn(self.dropout(y) + x)
        y = self.ffn(self.preprocess_(x))
        x = self.layernorm_ffn(self.dropout(y) + x)
        return x

class Attn(nn.Module):
    def __init__(self, hparams):
        super().__init__()
        self.hparams = hparams
        self.kd = self.hparams.total_key_depth or self.hparams.hidden_size
        self.vd = self.hparams.total_value_depth or self.hparams.hidden_size
        self.q_dense = nn.Linear(self.hparams.hidden_size, self.kd, bias=False)
        self.k_dense = nn.Linear(self.hparams.hidden_size, self.kd, bias=False)
        self.v_dense = nn.Linear(self.hparams.hidden_size, self.vd, bias=False)
        self.output_dense = nn.Linear(self.vd, self.hparams.hidden_size,␣
↪bias=False)
        assert self.kd % self.hparams.num_heads == 0
        assert self.vd % self.hparams.num_heads == 0

    def dot_product_attention(self, q, k, v, bias=None):
        logits = torch.einsum("...kd,...qd->...qk", k, q)
        if bias is not None:
            logits += bias
        weights = F.softmax(logits, dim=-1)
        return weights @ v

    def forward(self, x):
        q = self.q_dense(x)
        k = self.k_dense(x)
        v = self.v_dense(x)
```

```
        q = q.view(q.shape[:-1] + (self.hparams.num_heads, self.kd // self.hparams.num_heads)).permute([0, 2, 1, 3])
        k = k.view(k.shape[:-1] + (self.hparams.num_heads, self.kd // self.hparams.num_heads)).permute([0, 2, 1, 3])
        v = v.view(v.shape[:-1] + (self.hparams.num_heads, self.vd // self.hparams.num_heads)).permute([0, 2, 1, 3])
        q *= (self.kd // self.hparams.num_heads) ** (-0.5)

        if self.hparams.attn_type == "global":
            bias = -1e9 * torch.triu(torch.ones(x.shape[1], x.shape[1]), 1).to(x.device)
            result = self.dot_product_attention(q, k, v, bias=bias)
        elif self.hparams.attn_type == "local_1d":
            len = x.shape[1]
            blen = self.hparams.block_length
            pad = (0, 0, 0, (-len) % self.hparams.block_length)
            q = F.pad(q, pad)
            k = F.pad(k, pad)
            v = F.pad(v, pad)

            bias = -1e9 * torch.triu(torch.ones(blen, blen), 1).to(x.device)
            first_output = self.dot_product_attention(
                q[:,:,:blen,:], k[:,:,:blen,:], v[:,:,:blen,:], bias=bias)

            if q.shape[2] > blen:
                q = q.view(q.shape[0], q.shape[1], -1, blen, q.shape[3])
                k = k.view(k.shape[0], k.shape[1], -1, blen, k.shape[3])
                v = v.view(v.shape[0], v.shape[1], -1, blen, v.shape[3])
                local_k = torch.cat([k[:,:,:-1], k[:,:,1:]], 3)
                local_v = torch.cat([v[:,:,:-1], v[:,:,1:]], 3)
                tail_q = q[:,:,1:]
                bias = -1e9 * torch.triu(torch.ones(blen, 2 * blen), blen + 1).to(x.device)
                tail_output = self.dot_product_attention(tail_q, local_k, local_v, bias=bias)
                tail_output = tail_output.view(tail_output.shape[0], tail_output.shape[1], -1, tail_output.shape[4])
                result = torch.cat([first_output, tail_output], 2)
                result = result[:,:,:x.shape[1],:]
            else:
                result = first_output[:,:,:x.shape[1],:]

        result = result.permute([0, 2, 1, 3]).contiguous()
        result = result.view(result.shape[0:2] + (-1,))
        result = self.output_dense(result)
        return result
```

```python
class Transformer(nn.Module):
    def __init__(self, hparams):
        super().__init__()
        self.hparams = hparams
        self.layers = nn.ModuleList([DecoderLayer(hparams) for _ in␣
 ↪range(hparams.nlayers)])
        self.input_dropout = nn.Dropout(p=hparams.dropout)

        self.embeds = nn.Embedding(10, self.hparams.hidden_size)
        self.output_dense = nn.Linear(self.hparams.hidden_size, 10, bias=True)


    def pad(self, x):
        shape = x.shape
        x = x.view(shape[0], shape[1] * shape[2], shape[3])
        x = x[:,:-1,:]
        x = F.pad(x, (0, 0, 1, 0))
        x = x.view(shape)
        return x

    def forward(self, x):
        x = x.permute([0, 2, 3, 1]).contiguous()
        x = x.view(x.shape[0], x.shape[1], x.shape[2] * x.shape[3])
        x = self.embeds(x.int()) * (self.hparams.hidden_size ** 0.5)
        x = self.pad(x)
        shape = x.shape
        x = x.view(shape[0], -1, shape[3])

        x = self.input_dropout(x)
        for layer in self.layers:
            x = layer(x)
        x = self.output_dense(x).view(shape[:3] + (-1,))


        x = x.view(x.shape[0], x.shape[1], x.shape[2] // self.hparams.channels,␣
 ↪self.hparams.channels, x.shape[3])
        x = x.permute([0, 3, 1, 2, 4])

        return F.softmax(x, dim=-1)

    def loss(self, x, preds):
        batch_size = x.shape[0]
        x = torch.transpose(F.one_hot(x.view(batch_size, -1), -1), 1, 2).float()
        preds = preds.squeeze().transpose(3, 1).reshape(batch_size, 10, -1)
        loss = F.cross_entropy(preds, x, reduction='sum')
```

```
        return loss/batch_size

    def sample(self, n_samples=16, condition=None):
      with torch.no_grad():
        if not condition:
          condition = torch.ones(n_samples, 1, 28, 28)
        return self.forward(condition)

    def checkpoint(self, dir, optim, epoch):
      checkpoint = {
              "model": self.state_dict(),
              "optimizer": optim.state_dict(),
              "epoch": epoch
          }
      torch.save(checkpoint, f'{dir}checkpoint.pth')

    def evaluation(self, test_loader):
      test_loss = []
      for data, _ in test_loader:
        x = torch.Tensor(data)
        y = data.clone()

        with torch.no_grad():
          output = self(x.float())
          test_loss.append(self.loss(y, output))

      return np.mean(test_loss)
```

Training Transformer

[ ]:
```
class Parameters():
  channels = 1
  image_size = 28
  hidden_size = 200
  nlayers = 2
  num_heads = 4
  total_key_depth = 0
  total_value_depth = 0
  attn_type = "local_1d"
  filter_size = 2048
  dropout = 0.3
  block_length = 256

batch_size = 100
drive = '/content/CS594-HW1/runs/PixelRNN/'

writer = SummaryWriter(drive)
```

```
transforms = T.Compose([T.Lambda(lambda t : np.round((np.array(t) / 27)).astype(int)), T.ToTensor()])

training_data = torchvision.datasets.FashionMNIST("dataset", download=True, train=True, transform=transforms)
test_data = torchvision.datasets.FashionMNIST("dataset", download=True, train=False, transform=transforms)

train_loader = DataLoader(training_data, shuffle=True, batch_size=batch_size, num_workers=8)
test_loader = DataLoader(test_data, shuffle=True, batch_size=1, num_workers=8)

model_transformer = Transformer(Parameters())
optimizer = optim.Adam(model_transformer.parameters())

n_epochs = 3
for epoch in range(n_epochs):
  ## training part
  train_loss = []
  model_transformer.train()
  for i, (data, _) in enumerate(train_loader):
    x = torch.Tensor(data)
    y = data.clone()

    optimizer.zero_grad()
    ## 1. forward propagation
    output = model_transformer(x.float())

    ## 2. loss calculation
    loss = model_transformer.loss(y, output)

    #writer.add_scalar('NLL/train/iter', loss, (epoch + 1) * (i + 1))

    ## 3. backward propagation
    loss.backward()

    ## 4. weight optimization
    optimizer.step()

    train_loss.append(loss.item())

  model_transformer.checkpoint(drive, optimizer, epoch)

  train_loss_epoch = np.mean(train_loss)
  #writer.add_scalar('NLL/train/epoch', train_loss_epoch, epoch + 1)
  test_loss_epoch = model_transformer.evaluation(test_loader)
```

```
    #writer.add_scalar('NLL/test/epoch', test_loss_epoch, epoch + 1)

    print (f"Epoch: {epoch}, Training Loss: {train_loss_epoch:.2f}, Evaluation␣
↪Loss: {test_loss_epoch:.2f}")
```