PyTorch

Pytorch cheat sheet

```
import torch

from torch import nn

Almost everything in PyTorch is a module

from torch.utils.data import Dataset, DataLoader

Tensors:

  Scalar = torch.tensor(7)

  random_tensor = torch.rand(size=(3,4))

  multiply 2 tensors: *

  torchvision  cv lib  -> from torchvision import datasets, models, transforms
  torch text   text lib -> from torchtext import datasets...
  torch audio  audio   from torch audio...
  torch rec    rec engines


"cuda" > "mps" > "cpu"


device agnostic code

  if torch.cuda is available():
      device = "cuda"
  elif torch.backends.mps.is_available():
      device = "mps"
  else:
      device = "cpu"

  printf(f"using device: {device}")
```

tensors have a device

```
x = torch.tensor ...
print(x.device)

x = x.to(device)

torch.manual_seed(42)

torch.cuda.manual_seed(42)
```

Neural Network Layers

torch.nn package

```
from torch import nn

linear_layer = nn.Linear(in_features=10, out_features=10)

identity_layer = nn.Identity()
```

Convolutional layers:

torch.nn.convXd, X = 1,2,3

X = # of dimensions conv operates for

```
conv1d = nn.Conv1d(in_channels=1, out_channels=10, kernel_size=3)
```

Transformer layers

```
transformer_model = nn.Transformer()        # embedding dim, # of attention heads

transformer_encoder = nn.TransformerEncoderLayer(d_model=768, nhead=12)

trans_decoder = nn.TransformerDecoderLayer(d_model=768, nhead=12)

trans_decode_stack = nn.TransformerDecoder(decoder_layer=transformer_decoder, num_layers=6)
```

Recurrent Neural Networks :

LSTMs & GRUs

```
lstm_cell = nn.LSTMCell(input_size=10, hidden_size=10)
lstm_stack = nn.LSTM(input_size=10, hidden_size=10, num_layers=3)
gru_cell = nn.GRUCell(input_size=10, hidden_size=10)
gru_stack = nn.GRU(input_size=10, hidden_size=10, num_layers=3)
```

Activation Functions

nn.ReLU
nn.Sigmoid
nn.Softmax

```
relu = nn.ReLU()
```

Loss functions

nn.L1Loss — MAE
nn.MSELoss — MSE
nn.BCEWithLogitsLoss — Binary cross entropy
nn.CrossEntropyLoss

optimizers:

torch.optim

torch.optim.SGD(lr = 0.1, params = model.parameters())

torch.optim.Adam(lr = 0.001, params = model.parameters()) Adam optimizer

model = nn...

optimizer = torch.optim...

80% 20% split
using list slicing

Creating a model

torch.nn.Module → forward() function

torch.nn.Sequential

```
from torch import nn

class LRM(nn.Module):

  def __init__(self):
    super().__init__()

    self.linear_layer = nn.Linear(in_features=1, out_features=1)

  def forward(self, x: torch.Tensor) -> torch.Tensor:
      return self.linear_layer(x)

model = LRG()
model, model.state_dict()
```

using nn.Sequential

```
model = torch.nn.Sequential(
      nn.Linear(...)
)
model, model.state_dict()
```

* different state_dict names

!!
Sequential gives default names

```
loss_fn = nn.L1Loss()

optimizer = torch.optim.SGD...)

epochs = 1000
X_train = X_train.to(device)
    :
    :
X_test ...
y_train ...
y_test ...

model = model.to(device)

for epoch in epochs:

    model.train()
    Y_pred = model(X_train)
     loss = loss_fn(y_pred, y_train)
     optimizer.zero_grad()  ???
      loss.backward()
      optimizer.step()
```

Testing:

```
model_1.eval()

with torch.inference_mode():
    test_pred = model_1(X_test)

    test_loss = loss_fn(test_pred, y_test)

if epoch % 100 == 0:
    print(f"Epoch: {epoch} | Train loss: {loss} | Test loss {test_loss}")
```

① $\frac{d\,cost}{d\,log\,probs} \Rightarrow 32 \times 27$

Update log probs properly

$loss = -logprobs[range(n), y_b].mean()$

Averaging across batch size n

for the correct letter, $\frac{d\,loss}{correct\ label\ idx} = -\frac{1}{n}$

The other gradients are zero

② $\frac{d\,cost}{d\,probs} = \frac{d\,cost}{d\,logprobs} \cdot \frac{d\,logprobs}{d\,probs}$

$logprobs = log(probs)$ vector to vector

$\frac{d\,log_{probs}}{d\,probs} = \left(\frac{1}{probs}\right)$

$y = log(x)$

③ $\frac{d\,cost}{d\,counts\_sum\_inv}$ Example:

$\frac{d\,cost}{d\,logprobs} \cdots \frac{d\,probs}{d\,counts\_sum\_inv}$ → counts * ...

vector vector (32×27) (32×1)

$probs = counts * counts\_sum\_inv$

$counts.shape$ 32×27

$counts\_sum\_inv.shape$ 32×1

2 operations

$C = a * b$

$a[3\times3] \times b[3\times1]$

$\begin{bmatrix} a_{11} \times b_1 & a_{12} \times b_1 & a_{13} \times b_1 \end{bmatrix}$

④ $\frac{d\,cost}{d\,counts} \Rightarrow 32 \times 27$

$(32 \times 1) * (32 \times 27)$

$\frac{d\,probs}{d\,counts} = counts\_sum\_inv \cdots$

column wise multiplication

⑤ $\frac{d\,cost}{d\,counts\_sum}$

$\frac{d\,counts\_sum\_inv}{d\,counts} \cdot \frac{-1}{(counts\_sum)^2}$

⑥ $\frac{d\,cost}{d\,counts} = \cdots \frac{d\,counts\_sum}{d\,counts}$

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix} \rightarrow \begin{bmatrix} -b_1- \\ -b_2- \\ -b_3- \end{bmatrix}$$

replicate horizontally 27 times

$$32\left(\begin{matrix} 1 \\ \end{matrix}\right) \rightarrow 32\left(\begin{matrix} 27 \\ \end{matrix}\right)$$

(7) $\frac{\partial cost}{\partial norm\_logits} = \cdots \frac{\partial counts}{\partial \partial norm\_logits}$

counts = norm_logits.exp() elementwise

$\frac{\partial counts}{\partial norm} = norm\_logits.exp()$

(8) $\frac{\partial cost}{\partial logit\_maxes} = \cdots \frac{\partial norm\_logits}{\partial logit\_maxes}$

norm_logits = logits − logit_maxes

(32, 27) logit_maxes = (32, 1)

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} - \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

copy $-\frac{\partial \cdots}{\partial \cdots}$

(9) $\frac{\partial cost}{\partial logits} = \frac{\partial logit\_maxes}{\partial logits}$ → need indices

logit_maxes = logits.max(1, ...).values (along columns) => broadcasted to 32×27

32×1

1 at idx

5 at not idx

(10) $\frac{\partial Cost}{\partial h} = \cdots \frac{\partial \text{logits}}{\partial h}$

$\text{logits} = h @ w + b_2$

logits (32, 27)

h (32 × 64)

w_2 (64, 27)

b (1 × 27)

↑ broadcasted column wise

$d = a @ b + c$

$$\begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} + \begin{pmatrix} c_1 & c_2 \\ c_1 & c_2 \end{pmatrix}$$

$d_{11} = a_{11} b_{11} + a_{12} b_{21} + c_1$

$d_{12} = a_{11} b_{12} + a_{12} b_{22} + c_2$

$d_{21} = a_{21} b_{11} + a_{22} b_{21} + c_1$

$d_{22} = a_{21} b_{12} + a_{22} + b_{22} + c_2$

$$\frac{\partial C}{\partial a_{11}} = \frac{\partial C}{\partial d_{11}} b_{11} + \frac{\partial C}{\partial d_{12}} b_{12}$$

$$\frac{\partial C}{\partial a_{12}} = \frac{\partial C}{\partial d_{11}} b_{21} + \frac{\partial C}{\partial d_{12}} b_{22}$$

$$\frac{\partial C}{\partial a_{21}} = \frac{\partial C}{\partial d_{21}} b_{11} + \frac{\partial C}{\partial d_{22}} b_{12}$$

$$\frac{\partial C}{\partial a_{22}} = \frac{\partial C}{\partial d_{21}} b_{21} + \frac{\partial C}{\partial d_{22}} b_{22}$$

$$\begin{pmatrix} \frac{\partial C}{\partial a_{11}} & \frac{\partial C}{\partial a_{12}} \\ \frac{\partial C}{\partial a_{21}} & \frac{\partial C}{\partial a_{22}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial d_{11}} & \frac{\partial C}{\partial d_{12}} \\ \frac{\partial C}{\partial d_{21}} & \frac{\partial C}{\partial d_{22}} \end{pmatrix} \begin{pmatrix} b_{11} & b_{21} \\ b_{12} & b_{22} \end{pmatrix}^T$$

transposed ↑

$\frac{\partial c}{\partial d} \textcircled{a} b^T$

order matters ⭐

(11) $\frac{\partial c}{\partial b} = a^T \textcircled{a} \frac{\partial c}{\partial d}$

(12) $\frac{\partial cost}{\partial c} = \frac{\partial cost}{\partial d} \cdot Sum(0)$ across columns

(13) $\frac{\partial cost}{\partial pre\_act} = \cdots \frac{act}{pre\_act}$

$act = \tanh(pre\_act)$

$\frac{\partial act}{\partial pre\_act} = 1 - (pre\_act)^2$

(14) $\frac{\partial cost}{\partial bn\_gain} = \frac{\partial pre\_act}{\partial bn\_gain}$

$hpreact = bngain * bn\_raw + bn\_bias$

(32, 64) (1, 64) (32, 64) (1, 64)

↑ broadcasted

$dbn\_raw = ((1,64) * (32\times64))$
bngamma * dhpreact

bn_mean

$$\mu_B = \frac{1}{m}\sum_{i=1}^{n} x_i$$

bn var

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{n}(x_i - \mu_B)^2$$ bndiff 2

bn raw

$$\hat{x}_i = \frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}}$$ bndiff (32,64), bnvar_inv (1,64)

(32,64)

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$$

Bessels Correction

(15) $dbn\_diff = bnvar\_inv * dbn\_raw$ ✓ (32×64) (1,64) (32×64) wrt bn_raw

(16) $dbn\_var\_inv = bndiff * dbn\_raw$, sum col wrt bn_raw (1,64) (32,64) (32,64)

(17) $$\frac{\partial\, bnvar\_inv}{\partial\, bnvar} = -0.5\,(bnvar + (\epsilon-5))^{-1.5}\,(1)$$

Bessel's correction: use for mini batches to get a better variance

(18) 2bn diff 2 (32, 64)

2 bn var (1, 64)

$\frac{2\,bn\_var}{2\,bndiff}$

Sum in forward pass : broad casting in the backward pass

Broad casting in forward pass : Sum in backward pass

replication

Variable reuse

$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix} = \begin{matrix} b_1 & b_2 \\ | & | \end{matrix}$$

$b_1 = \left(\frac{1}{n-1}\right)(a_{11} + a_{21})$

$b_2 = \left(\frac{1}{n-1}\right)(a_{12} + a_{22})$

$\frac{2\,bn\_var}{2\,bn\_diff\,2} = \left(\frac{1.0}{n-1}\right)(\text{torch.ones\_like}(bn\,diff\,2))$

(19) bndiff (again)

$\frac{2\,bn\,diff2}{2\,bn\,diff} = 2\,bndiff$

(20) $\frac{2\,bndiff}{2\,bn\,meani}$ (1, 64)

Sum in backward pass $= -\text{torch.ones\_like}(bndiff) * d4bndiff$

(21) $\frac{2\,bndiff}{2\,hpre\,bn} = d\,bndiff$

(22) $dbn\_meani = \frac{1}{n} dh\_preact$

$dbn\_meani = dh\_preact.sum(0, keepdim=True)$

↓

broadcasting

$\frac{dh\_preact}{dbn\_meani} = torch.ones\_like(h\_preact) \left(\frac{1}{n}\right)$

Repeat again ....

(23) $\frac{dh\_preact}{dbnraw} =>$

(23) $\frac{dh\_preact}{dbn_1} =>$

(24) $\frac{dh\_preact}{db_1} =>$

(25) $\frac{demb}{demb\_cat} \overset{unflatten}{=>} dembcat.view(emb.shape)$

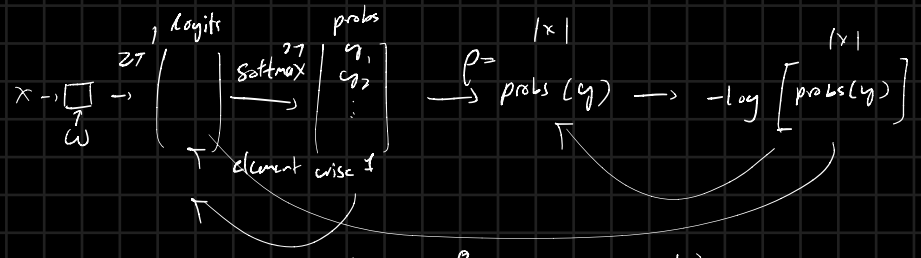
(26) plucking out rows

undo indexing

Gradients for 2+ occurrences must add

```
dC = torch.zeros_like(C)
for k in range(Xb.shape[0]):
    for j in range(Xb.shape[1]):
        ix = Xb[k,j]
        dC[ix] += demb[k,j]
```

All together:



$$\text{loss} = -\log p_y$$

$$= -\log\left(\frac{e^{l_y}}{\sum_j e^{l_j}}\right) \qquad p_i = \frac{e^{l_i}}{\sum_j e^{l_j}}$$

$$\frac{\partial C}{\partial l_i} = \frac{\partial}{\partial l_i}\left(-\log\left(\frac{e^{l_y}}{\sum_j e^{l_j}}\right)\right)$$

$$\begin{bmatrix} 0 \\ \vdots \\ -\frac{1}{p_y} \\ \vdots \\ 0 \end{bmatrix} \quad \text{(derivative of softmax)}$$

$$\frac{\partial C}{\partial \text{prob}} = \begin{bmatrix} \vdots \\ -\frac{1}{p_y} \\ \vdots \end{bmatrix}$$

$$\frac{\partial C}{\partial \text{logits}} = \left(\frac{\partial C}{\partial \text{prob}}\right)\left(\frac{\partial \text{prob}}{\partial \text{logits}}\right)$$

$$\text{prob}_i = \frac{e^{l_i}}{\sum_j e^{l_j}}$$

for all i, $\frac{\partial \text{prob}_i}{\partial \text{logits}} = \left(\frac{1}{\sum e^{l_j}}\right)\left(e^{l_i}\right)$ (constant)

$$i = 1 \qquad \frac{e^{l_1}}{\left(e^{l_1} + e^{l_2} \cdots e^{l_{xx}}\right)}$$

$$\frac{\partial C}{\partial x} = \frac{1}{x}$$

$$\left(\frac{1}{y}\right)\begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \text{(1 at } y\text{th idx)} \qquad \begin{bmatrix} \vdots \\ \frac{1}{x} \\ \vdots \end{bmatrix}$$

$$\text{prob}_i = 1$$

$$\frac{\partial C}{\partial l_i} = \frac{\partial}{\partial l_i}\left(-\log\left(\frac{e^{l_y}}{\sum_j e^{l_j}}\right)\right)$$

$$= -\frac{\sum_j e^{l_j}}{e^{l_y}} \frac{\partial}{\partial i}\left(\frac{e^{l_y}}{\sum_j e^{l_j}}\right)$$

$i \neq y$:

$$-\frac{\sum_j e^{l_j}}{e^{l_y}}\left[0 \times \frac{1}{\sum_j e^{l_j}} + -\frac{e^{l_y}}{\left(\sum_j e^{l_j}\right)^2}\right]$$

$$= \frac{e^{l_i}}{\sum_j e^{l_j}} = p_i$$

$i = y$

$$= \frac{\sum_j e^{l_y}}{e^{l_y}}\left(\frac{e^{l_y}}{\sum_j e^{l_j}} + -\frac{e^{l_y} e^{l_i}}{\left(\sum_j e^{l_j}\right)^2}\right)$$

$$= p_i - 1$$

For batches, average all examples, and divide by n

Batch normalization:

Calculate dh prebn, given dh preact

$$\mu = \frac{1}{m}\sum x_i$$

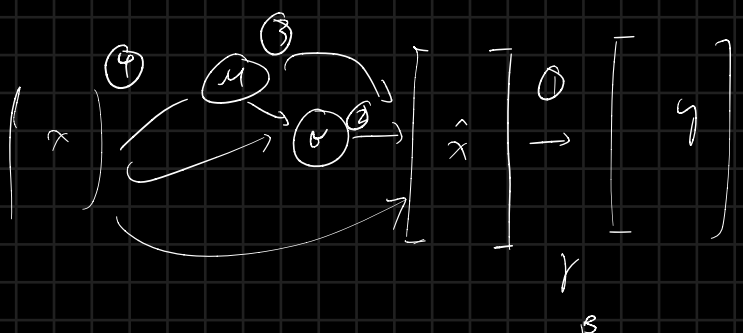
$$\sigma^2 = \frac{1}{m-1}\sum (x_i - \mu)^2$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

Having $\frac{\partial L}{\partial y_i}$

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i}\gamma$$



① $\frac{\partial y}{\partial \hat{x}} = \gamma$

② $\sigma^2 = \frac{1}{m-1}\sum (x_i - \mu)^2$

$(2)\left(\frac{\partial \sigma}{\partial x_i}\right)(\sigma) =$

② $\frac{\partial L}{\partial \sigma^2} = \sum_i \frac{\partial L}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial \sigma^2}$

$= \gamma \cdot \frac{\partial L}{\partial y_i}\frac{d}{d\sigma^2}\left((x_i - \mu)(\sigma^2 + \epsilon)^{-\frac{1}{2}}\right)$

$= -\frac{1}{2}\gamma \frac{\partial L}{\partial y_i}\cdot (x - \mu)(\sigma^2 + \epsilon)^{-\frac{3}{2}}$

③ $\frac{\partial L}{\partial \mu} = \sum \left(\frac{\partial L}{\partial \hat{x}_i}\right)\left(\frac{\partial \hat{x}_i}{\partial \mu}\right) + \left(\frac{\partial L}{\partial \sigma^2}\right)\left(\frac{\partial \sigma^2}{\partial \mu}\right)$

$\frac{\partial \hat{x}_i}{\partial \mu} = \frac{\partial}{\partial \mu}\left[\frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\right] = -(\sigma^2 + \epsilon)^{-\frac{1}{2}}$

$\frac{\partial \sigma^2}{\partial \mu} = \frac{\partial}{\partial \mu}\left(\frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu)^2\right)$

$= \frac{-2}{m-1}\sum (x_i - \mu)$

$= \frac{-2}{m-1}\left(\sum x_i - \sum \mu\right)$

$= \left(\frac{2}{m-1}\right)\left(\overset{①}{m\mu} - m\mu\right)$ subbing in

$= 0$

$\frac{\partial L}{\partial \mu} = -\sum \frac{\partial L}{\partial y_i}\cdot \gamma \cdot (\sigma^2 + \epsilon)^{\frac{1}{2}}$

(4)

$\lambda \to x_i \to$ 32 arrows

exactly parallel and $\frac{\partial x_i}{\partial x} = 1$

$$\frac{\partial c}{\partial x_i} = \frac{\partial c}{\partial \hat{x}_i}\frac{\partial \hat{x}_i}{\partial x_i} + \frac{\partial c}{\partial \mu}\frac{\partial \mu}{\partial x_i} + \frac{\partial c}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial x_i}$$

$$\frac{\partial \hat{x}_i}{\partial x_i} = \frac{d}{dx_i}\left(\frac{x_i - \mu}{\sqrt{\sigma^2+\epsilon}}\right) = (\sigma^2+\epsilon)^{\frac{1}{2}}, \quad \frac{d\mu}{dx_i} = \frac{d}{dx_i}\left(\frac{1}{m}\sum_j x_j\right) = \frac{1}{m}$$

$$\frac{\partial \sigma^2}{\partial x_i} = \frac{2}{\partial x_i}\left(\frac{1}{m-1}\sum_j (x_j-\mu)^2\right) = \left(\frac{2}{m-1}x_i - \mu\right)$$

$$\left(\frac{\partial c}{\partial y_i}\right)(\gamma)\left(\sigma^2+\epsilon\right)^{-\frac{1}{2}} + \left(\sum_j \frac{\partial c}{\partial y_j}\cdot\gamma\cdot(\sigma^2+\epsilon)^{-\frac{1}{2}}\right)\cdot\frac{1}{m}$$

placeholder

$$+\left(-\frac{1}{2}\gamma\sum_j \frac{\partial c}{\partial y_j}\cdot(x_j-\mu)(\sigma^2+\epsilon)^{-\frac{3}{2}}\right)\left(\frac{2}{m-1}\right)(x_i-\mu)$$

torch vision => computer vision torch.float Tensor(np.array())

Class LinearModel(nn.module):

def __init__(self, input_size, output_size):

super(LinearModel, self).__init__()

self.w = nn.Parameter(torch.randn(output_size, input_size))

def forward(self, x)
return x @ self.w.t()

output.detach() (removes tensor class)

① linear = nn.Linear(inputs, outputs)

linear.weight.data

output = linear(data)

② loss_function = nn.MSELoss()
③ optimizer = torch.optim.SGD(linear.parameters(), lr=0.01)

loss.backward() (calculates gradients)
optimizer.step() (Takes a learning step)

Forward pass
Calculate loss
reset stored gradients to zero
loss.backward()
take learning step with optimizer.step()

loss_function expects 2 arguments of same form

loss_function found in torch.nn

```
class Model(nn.Module):

    def __init__(self, ...):
        super(Model, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        self.h1 = self.linear1(x)
        self.h2 = torch.sigmoid(h1)
        self.h3 = self.linear2(self.h2)
        return self.h3


logistic = Model(input_size=2, output_size=1, hidden_size=2)

loss_function = nn.BCEWithLogitsLoss()

optimizer = torch.optim.Adam(logistic.parameters(), lr=1e-2)
```

Torch dataset + dataloaders

Dataloader object

__init__

__getitem__

__len__

Data / Dataloaders

```
torch.utils.data import Dataloader

torch.utils.data.Dataset import Dataloader

torchvision.datasets import MNIST
```

```
class SmCWave(Dataset):

    def __init__(self, num_data):
        self.x_data = ...
        self.y_data = ...

    def __getitem__(self, index):
        """ Get a new batch, passes in the index
        """
        return self.x_data[index], self.y_data[index]

    def __len__(self)
        return self.x_data.shape[0]
```

Convolutional Neural Networks

PIL - Python Imaging Library

torch.utils.data.random_split(total, [valid], generator=torch.Generator().manual_seed(42))

Seed is important when saving!

LeNet5

transform = transforms.Compose([transforms.Resize(32)
transforms.ToTensor()
transforms.Normalize ...])

nn.Conv2D(channels_in, out_channels, kernel_size)

(batch, in_channels, length, width)

(batch, out_channels, out_len, out_width)

model.train()

model.eval()

=> Disables Dropout, BatchNorm (running mean & std dev)

fx.argmax(1).sum().item()

PIL:

Image.open(...).convert('RGB')

# of filters
(out_channels, in_channels, K height, K width)

(batch, in_channels, im height, im width)

nn.Conv2D(in_channels, out_channels, kernel_size, stride, padding, bias = True)

Transfer Learning

Weight freezing : Freeze pretrained model from being updated in backward pass

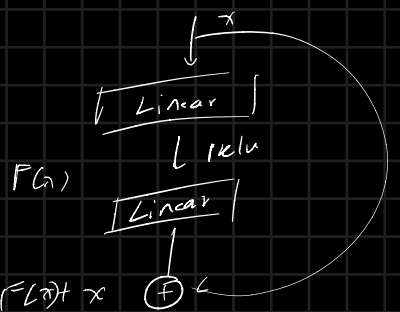
```
torch.vision.models as models

num_ftrs = resnet.fc.in_features
resnet.fc = nn.Linear(num_ftrs, 10).to(device)

torch.load("model.pth")
model = torch.load(....)
model[key]
key

model.load_state_dict(torch.load(save_path[variable]))
optimizer.load_state_dict(torch.load(save_path[variable]))

x = torch.save({'model_state_dict': model.state_dict(), 'optimizer_state_dict': optimizer.state_dict()}, model.pth)
```

Skip / Residual Connections

Take output, skip layers, then add to a hidden layer

=> Reduces impact of the vanishing gradient

Input & hidden must be of same size to add



Creating modules in modules

eg. create separate skip/residual blocks

nn.BatchNorm2d
nn.Sequential
nn.DropOut

```
class ConvBlock(nn.Module):

    def __init__(self, channels):
        super(ConvBlock, self).__init__()

        self.conv1 = nn.conv2d(channels, channels//2, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm(channels//2)
        self.conv2 = nn.Conv2d(channels//2, channels, kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        x = F.relu(x)
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.conv2(x)
        return x


class ResBlock(nn.Module):

    def __init__(self, channels):
        self.conv1 = nn.Conv2d(...)
        self.bn1 = nn.BatchNorm2d(...)
        self.conv2 = nn.Conv2d(...)

    def forward(self, x):
        x0 = x
        x = F.relu(x)
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.conv2(x)

        return x + x0
```

```
Res dann Block ( nn. Module ):
    def __init__ (self, in_channels, out_channels):
    super ( ... )
    self.conv1 = nn. conv2d ( in_channels, out_channels, ... )
    self.conv2 = nn. Conv2d ( out_channels, out_channels, ... )
    self. bn1 = nn. conv 2d ( in_channels, out_channels , ... )
    self. conv3 = nn. conv 2d ( in_ channels, out_channels, ... )

        def forward ( self, x )
            x = F .relu (x)
            x0 = self .conv 3(x)
            x = F.relu( self .bn1 (self .con 1(x))
            x= self . conv 2(x)
            return x + x0

class Skip Block ( nn. Module)
    def __init__( ... )
        super ...
        conv1 ( channels , channels//2 , kernel_size=3 ... )
        bn1
        conv 2

    def forward (self , x)
        x = F, relu ( x)
        x1 = self . conv 1(x)
        x2 = F, relu (self . bn1 (x))
        x3 = self. conv 2( x2 )
        return torch. cat ( x1, x3 , 1) # Across depth
```

```
Class DeepCNN(nn.Module):

    def __init__(self, in_channels, num_blocks=2, ch_width=32, layer_type=ResBlock):
        super(...).__init__()

        conv1
        bn1
        conv2
        bn2
        conv3
        maxPool

        self.layers = self.create_blocks(num_blocks, layer_type, channels=ch_width)

        else: nn.Identity()

        dropout
        Linear(...)

        blocks.append(block_type(channels))

        return nn.Sequential(*blocks)
                              ↑
                        unpacks list

    def forward(self, x):

        x = F.relu(self.bn1(self.conv1(x)))
        :
        x = F.relu(self.layers(x))
        :
        :

nn.Sequential([layer1, layer2, ...])
```

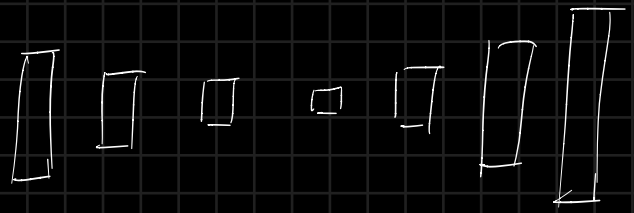
Augmentations:

1. AutoAugment()
2. RandomRotation
3. RandomHorizontalFlip

Learning rate Scheduler

model.set_lr_scheduler(optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs, eta_min=0))

Auto encoders



Down Block

conv1 => C(in, out, 3, 2, 1)

S=2
P=1

Up Block

Conv1(in, out, 3, 1, 1)

ConvTranspose2d(in, out, 3, 2, 1, 1)

ConvTranspose2d(..) is the opposite of Conv2d(...)

ConvTranspose2d(in, out, Kernel, Stride, padding, output_padding)

Auto encoders are useful for denoising

KL divergence: $p(x) = N(\mu_p, \sigma_p)$
$q(x) = N(\mu_q, \sigma_q)$

$$KL(p,q) = \ln\left(\frac{\sigma_q}{\sigma_p}\right) + \frac{\sigma_r^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} - \frac{1}{2}$$

Variational auto encoders: represents complex data distributions in a latent space

latent variables z

image → z , z → output
Encoder Decoder

VAE: make dist learned by encoder close to Gaussian, using Kull back - Leibler divergence

Minimize Negative Evidence Lower Bound

$$ELBO: \; 2(x) = E_{q(z|x)}[\log p(x|z)] - KL(q(z|x) \| p(z))$$

KL between 2 normal dists:

$KL(p\|q)$

$$KL(p\|q) = \ln\left(\frac{\sigma_q}{\sigma_p}\right) + \frac{\sigma_p^2 + (\mu_r - \mu_q)^2}{2\sigma_q^2} - \frac{1}{2}$$

Set $q(z)$ to be Gaussian

$$KL(q(z|x)\|p(z)) = \ln\left(\frac{1}{\sigma}\right) + \frac{\sigma^2 + \mu^2}{2} - \frac{1}{2}$$

KL allows for regularization, and a meaningful encoder structure