Deep learning with PyTorch

23 Dec 22

Autograd

x = torch.randn(3, requires_grad = True)
print(x)

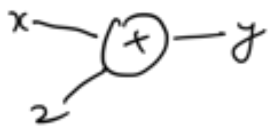
2 Jan 2022

Autograd

- torch.randn → requires_grad = True

e.g. $y = x + 2$

① forward



grad_fn

↳ tensor(, grad_fn<AddBackward

② Add Backward $\frac{dy}{dx}$

↳ ∵ we said requires_grad=True

To calculate gradient
x = torch.randn(3, requires_grad = True)
y = x+2
z = y*y*2 (example)

z = z.mean()

only for scalar outputs

z. backwards ( )   % dz/dx

print(x. grad)

create a vector jacobian product to get gradiants

$$J \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \vdots & & \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_n}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_m} \end{pmatrix}$$

⋮

z=y * y * 2

% z ~~= z~~.mean

example

v= torch. tensor ([0.1, 1.0, 0.001], dtype= torch.float32)

z. backward (v)

print(x. grad)

→ x.requires_grad_(False)
x.detach()
with torch.no_grad().

• x. Functⁿ_( )
will modify the variable

• Ex: with torch.no_grad():

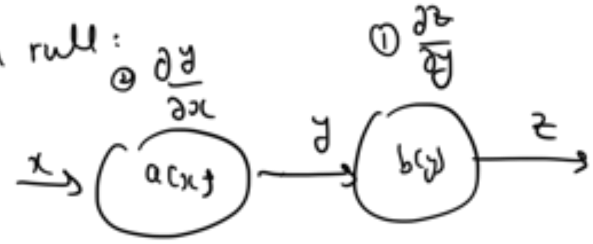
```
with torch.no_grad():
    y = x + 2
    print(y)
```

→ must empty the gradients in training steps.
otherwise sum is printed.

```
for epoch in range(n):
    model_output = (weights * 3).sum()

    model_output.backward()
    print(weights.grad)

    weights.grad.zero_()
```

Backpropagation

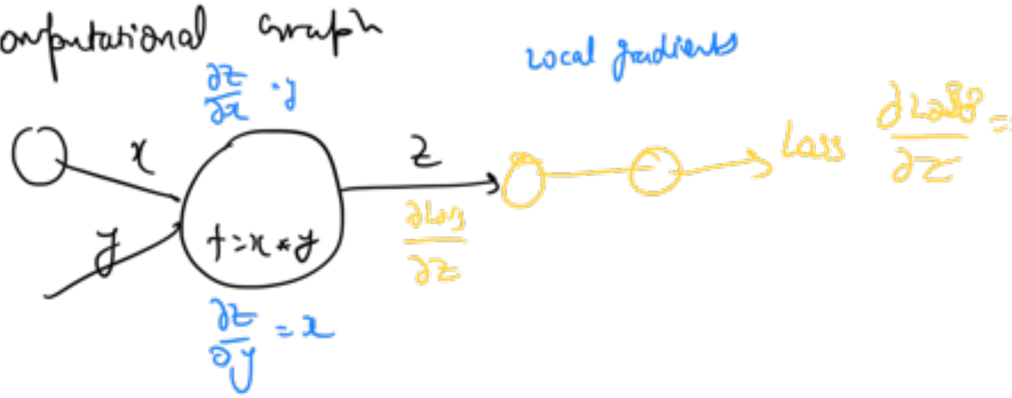
- chain rule :



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

① ②

- computational graph



$$\frac{\partial Loss}{\partial x} = \frac{\partial Loss}{\partial z} \cdot \frac{\partial z}{\partial x}$$

Three steps :

1] Forward Pass : Compute Loss

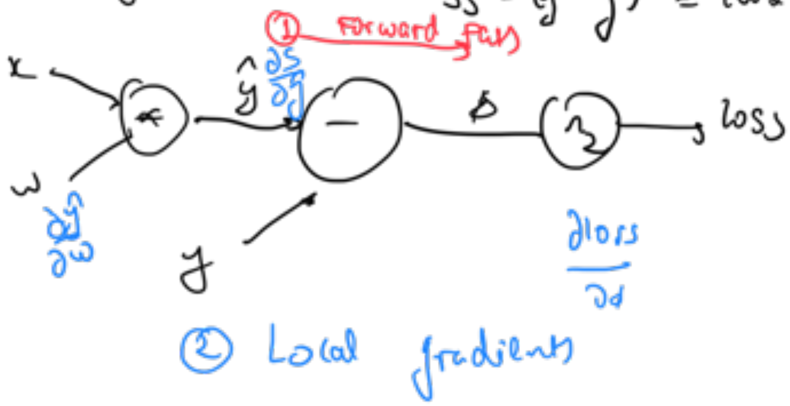
2] Compute local gradients

3] Backward Pass: compute $\frac{d\,Loss}{d\,weights}$

using Chain rule

Linear regression

Eg: $\hat{y} = w \cdot x$ $\quad$ $Loss = (\hat{y} - y)^2 = (wx - y)^2$



③ Backward pass

$\frac{\partial loss}{\partial w} \leftarrow \frac{\partial loss}{\partial \hat{y}} \leftarrow \frac{\partial loss}{\partial s}$

Code:

```
import torch

x = torch.tensor(1.0)
y = torch.tensor(2.0)

w = torch.tensor(1.0, requires_grad = True)

# fwd pass

y_hat = w * x
loss  = (y_hat - y) ** 2
```

bwd pass

loss . backward ( )

w. grad

Update weights
next fwd & bwd pass

Prediction
Gradient Computation
Loss Comp
parameter updates

Gradient Descent

① Numpy

```
import numpy as np
# f = w * x
# f = 2 * x

x = np.array ([1,2,3,4], dtype = np.float32)
y = np.array ([2,4,6,8], dtype = np.float32)

w = 0.0

# model predi-
def forward (x):
    return w * x

# loss MSE
def loss (y, y_predicted):
    return ((y_predicted - y) ** 2). mean( )
```

v

```
# gradient
# MSE = 1/N * (wx - y)^2
# dJ/dw = 1/N 2x (wx - y)

def gradient (x, y, y_predicted):
    return np.dot (2* x, y_predicted - y). mean ( )


print (f'Prediction before training : f (5) = {forward (5):.3f }')


# Training
learning_rate = 0.01
n_iter = 10

for epoch in range (n_iters):
    # fwd
    y_pred = forward (x)

    #Loss
    l = loss (Y, y_pred)

    # grad
    dw = gradient (X, Y, y_pred)

    # update weights
    w -= learning_rate * dw

    if epoch % 1 == 0:
        print (f'epoch {epoch+1}: w = {w:.3f},
                                loss = {l:.8f}')
```

```
print(f'Prediction after train-- -        )
```

 torch

```
x = torch.tensor(        — —, torch.flo— '

y = torch.   -   —   —

w = torch.tensor(0.0, dtype=torch.float32, requires_grad=True)

  :

- ———— :

    #gradients = backward pass
    l.backward()      #dl/dw

    #update weights
    with torch.no_grad():
            w -= learning_rate * w.grad

    #zero grad
    w.grad.zero_()

    :
```

⇒ Training Pipeline

steps:

1) Design model (input, output size, forward pass)
2) construct loss and optimizer
3) Training loop
   - fwd pass: compute predictⁿ
   - bwd pass: gradients
   - update weights

⇒ import torch.nn as nn

```
⋮
lear_rate
n_iter = 100

loss = nn.MSELoss()
optimizer = torch.optim.SGD([w], lr=learning_rate)

for epoch in range(n_iters):
        ⋮
    # update weight
    optimizer.step()

    # zero gradient
    optimizer.zero_grad()
```

forward pass using torch

```
X = torch.tensor([[1],[2],[3],[4]], dtype=torch.float32
Y =        "
X_test = torch.tensor([5], dtype=torch.float)
n_samples, n_features = X.shape

input_size = n_features
output_size = n_features

model = nn.Linear(input_size, output_size)

print(f'Prediction before training: f(5) =
                 {model(X_test).item():.3f}')

    :
    :

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

for epoch in range(n_iter):
    y_pred = model(X)
```

```
if ecoch % 10 ==0:
    [w, b] = model.parameters()
    print(f' epochs {epoch+1} : w =
                {w[0][0].item():.3f},
                    loss  - - )
```

(4)

```
class LinearRegression(nn.Modoll):
    def _init_(self, input_dim, output_dim):
        super(LinearRegression, self)._init_()
        # define layers
        self.lin = nn.Linear(input_dim,
                             O/p_dim)

    def forward(self, x):
        return self.lin(x)

model = LinearRegression(input_size, output_size)
```

⇒ 7. Linear Regression

```
import torch
... as nn
```

```
import torch.....
import numpy as np
~~from~~ sklearn import datasets
import matplotlib.pyplot as plt


#0) prepare data
x_numpy, y_numpy = datasets. make_regression
                   (n_samples=100, n_features=1,
                   noise=20, random_state = 1)


X = torch.from_numpy( X_numpy . astype (np.float32))
y =  -  " —            -    -

reshape tensor
Y= y. view( y.shape [0], 1)

n_samples, n_features = X.shape


# 1) model
input_size   = n_features
output_size  = 1
model = nn.Linear( input_size, output_size)


#2) loss & optimizer
criterion = nn. MSELoss( )
learning_rate = 0.01
optimizer = torch. optim. SGD( model . parameters(),
                              lr=learning_rate)
```

```
# 3) training loop
num_epochs = 100
for epoch in range(num_epochs):
    # fwd pass & loss
    y_predicted = model(X)
    loss = criterion(y_predicted, y)

    # bwd pass
    loss backward()

    # update
    optimizer.step()

    optimizer.zero_grad()


    if (epoch+1) % 10 == 0:
        print(f'epoch: {epoch+1}, loss =
                {loss.item():.4f}')


#plot
predicted = model(x).detach().numpy()
plt.plot(x_numpy, y_numpy, 'ro')
plt.plot(x_numpy, predicted, 'b')
plt.show
```

→ 8. Logistic regression

i

~~impor~~

```
from sklear import dataset
from sklearn. preprocessing import StandardScal
from sklearn. model_selection import
                              train_test_split


#0) Prepare data
bc = datasets. load_ breast_cancer()
x,y = bc. data, bc. target
n_samples, n_features = x. shape

x_train, x_test, y_train, y_test =
          train_test_split( x, y, test_size=0.2,
                              random_state=1234


#scale
sc = standard Scalar ()
X_train = sc. fit_transform (X_train)
x_test = sc. transform (x_test)

x_train = torch. from_numpy (x_train.
                         astype(np.float

x_test    =    "
y_train   =    "
y_test    =    "


y_train = y_train .view (y_train.shape[0]
```

```
Y_test =        "

# 1) model
# f = wx + b , sigmoid at end

class LogisticRegression (nn.Module):

    def _init_ (self, n_input_features)
        super (LogisticRegression, self)._init_(
        self.linear = nn.Linear (n_inputfeatur

    def forward (self, x):
        y_predicted = torch.sigmoid (
                           self.linear(x))

        return y_predicted


model = LogisticRegression (n_features)


#2) Loss & optimizer
learnin_rate = 0.01
criterion = nn.BCELoss( )
              ← Binary cross Entropy

optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning rate)


#3) training loop

num_epochs = 100
```

```
for epoch in range (num_epochs):

    y_pred = model ( x_train )
    loss = criterion( y_pred, y_train

    # bwd pass
    loss. backward()

    #updates
    optimizer.step()

    # zero grad
    optimizer.zero_grad ( )

    if (epoch+1) % 10 == 0:
        print (f'epoch : {epoch+1},
                loss = {loss.item() : .4f}')


with torch.no_grad():
    y_predicted =  model (X_test)
    y_predicted_cls =  y_predicted.round(
    acc = y_predicted_cls.eq(y_test).sum()
                          float(y_test.shape
```
accuracy

```
    print (f' accuracy = {acc : .4f}')
```

---

4 Jan 2023

• Dataloader

9. Datasets and Dataloader

epoch = 1 forward & bwd pass of ALL training samples

batch_size = no of training samples in one fwd + bwd pass

no of iter = no of passes, each pass using [batch_size] no of samples

Eg: 100 samples, batch_size = 20 → $\frac{100}{20}$ = 5 iterations for one epoch

```python
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
import numpy as np
import math


class WineDataset(Dataset):

    def __init__(self):
        # data loading
        xy = np.loadtxt('./data/wine/wine.csv',
                        delimiter=",", dtype=np.float32,
```

skiprows =1)

```
self.x = torch.from_numpy (xy[:, 1:])
self.y = torch.from_numpy (xy [:, [0]])
self.n_samples = xy.shape [0]

def __getitem__(self, index):
    return self.x [index], self.y [index]

def __len__ (self).
    return self.n_samples

dataset = WineDataset ()
first_data = dataset [0]
features , labels = first_data
```

% to check

```
dataloader = Dataloader (dataset = dataset ,
        batch_size = 4, shuffle = true,
            num_workers = 2)

dataiter = iter (dataloader)

data = dataiter.next ()
features, labels = data
print (features, labels)
```

% to check

```
#Dummy training loop:
num_epochs = 2
total_samples = len(dataset)
n_iterations = math.ceil(total_samples/4)   → batch size
print(total_samples, n_iteration)

for epoch in range(num_epochs):
    for i, (input, labels) in enumerate(
                                   dataloader):
        # fwd bwd, update
```

10. Dataset transform

```
class WineDataset(Dataset):
    def __init__(self, transform=None):
        # data loading
        xy = np.loadtxt('./data/wine/wine.csv',
                        delimiter=",", dtype=np.float32,
                        skiprows=1)

        self.x = xy[:, 1:]
        self.y = xy[:, [0]]
        self.n_samples = xy.shape[0]
        self.transform = transform
```

```
def __getitem__(self, index):
    sample = self.x[index], self.y[index]
    if self.transform:
        sample = self.transform(sample)
    return sample

def __len__(self).
    return self.n_samples
```

custom tensor class

```
class ToTensor :
    def __call__(self, sample):
        inputs, targets = sample
        return torch.from_numpy(inputs),
                     torch.from_numpy(targets)


dataset = WineDataset(transform= ToTensor())

first_data = dataset[0]
features, labels = first_data
print( type(features), type(labels))
```

another eg:

```
class MulTransform :
    def __init__(self, factor):
        self.factor = factor
```

```
def __call__(self, sample):
    inputs, target = sample
    inputs *= self.factor
    return inputs, target
```

Using composed transform

```
composed = torchvision.transforms.Compose([ToTensor(),
                                            MulTransform(2)])
```

```
dataset = WineDataset(transform=composed)
```

$\boxed{\text{5 Jan 2023}}$

$\boxed{11.}$ Softmax and Cross-Entropy

<u>Softmax</u>

$$S(y_i) = \frac{e^{y_i}}{\sum e^{y_i}}$$

o/p b/w 0 to 1 ← we get probabilities

```
import torch
import torch.nn as nn
import numpy as np
```

def softmax (x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)

x = np.array ([2.0, 1.0, 0.1])
outputs = softmax (x)
print (outputs)

---

x = torch.tensor ([2.0, 1.0, 0.1])
outputs = torch.softmax (x, dim=0)
print (outputs).

→ Cross - Entropy
↳ measures performance of classification probl
↳ O/P : probab b/w 0 & 1

$$D(\hat{Y}, Y) = -\frac{1}{N} \sum Y_i \cdot \log(\hat{Y}_i)$$

Y ← one-hot encoded class labels
Ŷ ← predicted ← probab

def cross_entropy (actual, predicted):
    loss = np. sum (actual * np.log (predicted))
    return loss # / float (predicted.shape[0])

y ← one hot encoded

Y = np.array([1,0,0])

y_pred_good = np.array([0.7, 0.2, 0.1])

l_1 = cross_entropy (Y, Y_pred_good)

print (f'Loss1 numpy : {l_1: 0.4f}')

torch

loss = nn.CrossEntropyLoss( )

Careful:
already applies:
nn.LogSoftmax + nn.NLLLoss
negative log likelihood loss

hence don't implement soft max in last layer

Y has class Labels, not One-Hot,
Y_pred has raw scores (logits), no softmax

Y = torch.tensor([0])

```
Y = torch.tensor ([0])
# nsamples x nclasses
Y_pred_good = torch.tensor ([[2.0, 1.0, 0.1]])
            array of
            arrays

Y_pred_bad = torch.tensor ([[ 0.5, 2.0, 0.3]])

l1 = loss (Y_pred_good, Y)
l2 = loss (Y_pred_bad, Y)

print (l1.item())
print (l2.item())

_, predictions1 = torch.max(Y_pred_good, 1)

print (predictions1)
```

=> multiple sample.

```
# 3 samples
Y = torch.tensor ([2, 0, 1])

Y_pred_good = torch.tensor (
        [[0.1, 1.0, 2.1], [2.0, 1.0, 0.1],
         [0.1, 3.0, 0.1]])
```

3) Neural network with softmax
pytorch → don't implement softmax at end

↳ binary problem : Sigmoid

pytorch : nn.BCELoss() →

$\boxed{12}$ Activation Function

↳ apply non-linear transformation

↳ decide whether a neuron should be activated or not

without activation function is same as linear-regression

a) Step Function

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Not used in practice

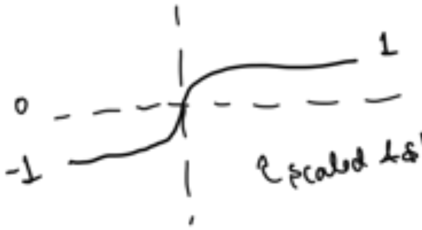
b) Sigmoid function $f(x) = \dfrac{1}{1+e^{-x}}$

Typically used in last layer of binary classification problem.

(c) TanH Funct^n

$$f(x) = \frac{2}{1+e^{-2x}} - 1$$

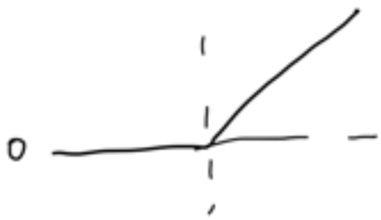
↳ used in hidden layer



(scaled & shifted sigmoid funct^n)

(d) ReLU funct^n

$$f(x) = \max(0, x)$$

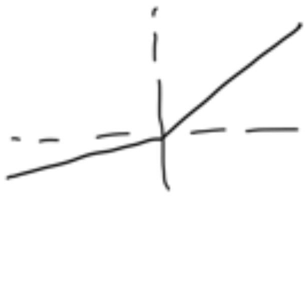


↳ if you don't know what to use, just use a ReLU for hidden layers

(e) Leaky ReLU

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ a \cdot x & \text{otherwise} \end{cases}$$

↳ a is typically very small



↳ improved version of ReLU.

tries to solve vanishing gradient problem

∵ ReLU → 0 y –
ie weights will
not update
there

(f) Softmax

$$f(x) = \frac{e^{y_i}}{\sum e^{y_i}}$$

→ Good choice
in last layer of
multi-class classification
problem

---

13 Feed Forward Network

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# device config
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# hyper parameters
input_size = 784    # 28x28
hidden_size = 100
num_classes = 10
```

```
num_epoch -
batch_size =100
learning_rate =0.001

#MNIST
train_dataset = torchvision.dataset.MNIST
                    (root='./data', train=True,
                     transform=transforms.ToTensor()
                     download=True)

test_dataset = torchvision - - - - -  (root= - -
                  - - train=False, transfor - - - ,))
                                       download=Tr

train_loader = torch.utils.data.DataLoader
                  (dataset=train_dataset, batchsize=
                                          batch_size,
                  shuffle = True)

test_loader = - - -
                  ( - - =test_dataset, - - - -)
                                     shuffle =False)

examples = iter (train_loader)
samples, labels = examples.next()
print( sample.shape, labels.shape)
```

```
for i in range (6):
    plt.subplot (2, 3, i+1)
    plt . imshow( samples[i][0] , cmap='gray')

plt. show( )


class NeuralNet (nn. Modules):
    def __init__ (self, input_size, hidden_size,
                                    num_classes):

        super (NeuralNet, self). __init__()
        self. l1 = nn. Linear (input_size,
                                    hidden_size)

        self. relu = nn. ReLU( )
        self. l2 = nn. Linear ( hidden_size
                                    num_classes)


    def forward (self, x):
        out = self. l1(x)
        out = self. relu (out)
        out = self. l2 (out)

        return out

model = NeuralNet (input_size, hidden_size
                                    num_classes)


# loss & optimizer
```

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(
                                lr = learning_rate
```

```
#training loop
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in
                    enumerate(train_loader):
```

image:
(100, 1, 28, 28)
our data
(100, 784)

```
        images =
            images.reshape(-1, 28*28).
                        to(device)

        labels = labels.to(device)

        #fwd pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # backwards
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print(f'epoch {epoch+1}/
```

{num_epochs},

step {i+1}/{n_total_steps},
loss = {loss.item():.4f}')

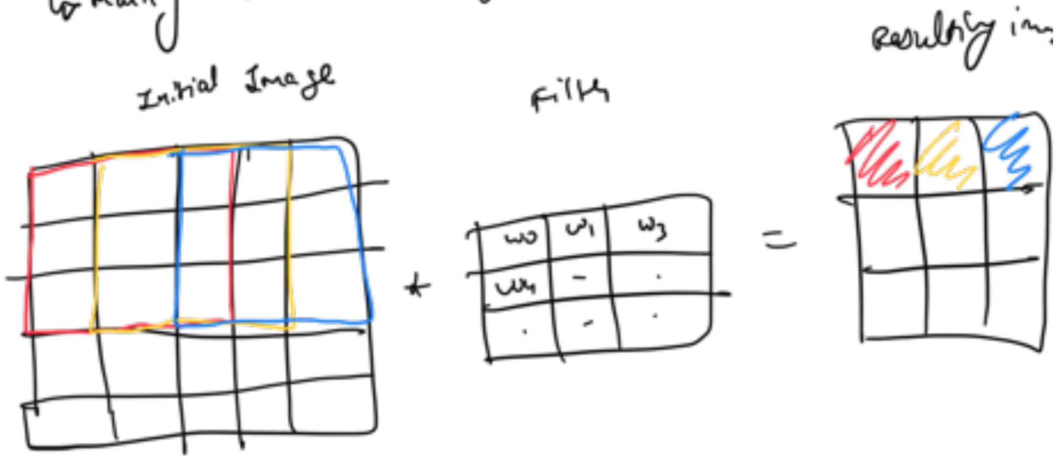
```
#testing loop
with torch.no_grad():
        n_correct = 0
        n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28)
                                .to(device)
        labels = labels.to(device)
        outputs = model(images)
        #value, index
        _, predictions = torch.max(outputs, 1)
        n_samples += labels.shape[0]
        n_correct = (predictions == labels)
                        .sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'accuracy = {acc}')
```

6 Jan 2023

Neural Network

14 Convolution

↳ mainly work on image data



↳ setting correct size is very important

- Pooling
  - max pooling
    - downsample an image by applying max filter in sub region
    - used to reduce computational cost
    - helps avoid overfitting

<u>code</u>

Same

```
#hyperparameters
num_epochs = 4
batch_size = 4
learning_rate = 0.001
```

```
# dataset
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5),
                          (0.5, 0.5, 0.5))])

train_dataset = torchvision.datasets.CIFAR10
    (root='./data', train=True,
     transform=transform)

test_dataset = _ _ _ ___
    ( _ _, train=False,
     ___ )

train_loader = torch.utils.data.DataLoader
    (train_dataset, batch_size=batch_size,
     shuffle=True)

test_loader = _ . _ _
    ( _ _ . .
     .. shuffle=False

classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

Input Conv+Relu Pooling Conv+Relu Pooling Flatten Fully connected Soft

$\underbrace{\text{Conv+Relu Pooling Conv+Relu Pooling}}_{\text{Feature Learning}}$ $\underbrace{\text{Flatten Fully connected Soft}}_{\text{classification}}$

```
... ConvNet (nn.Module):
```

class ConvNet

def __init__(self):
    super(ConvNet, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)

    self.pool = nn.MaxPool2d(2, 2)

    self.conv2 = nn.Conv2d(6, 16, 5)

    self.fc1 = nn.Linear(16*5*5, 120)

    self.fc2 = nn.Linear(120, 84)

    self.fc3 = nn.Linear(84, 10)

10 diff classes

O/P size:

$(W - F + 2P)/S + 1$

Eg: 5×5 input, 3×3 filter, padding = 0, stride = 1

$(5 - 3 + 0)/1 + 1 = 3 \rightarrow 3\times3$

import torch.nn.functional as F

def forward(self, x)

    x = self.pool(F.relu(self.conv1(x)))

    x = self.pool(— — . conv2(x)

flatten → x = x.view(-1, 16*5*5)

    x = F.relu(self.fc1(x))

    x = F.relu(self.fc2(x))

    x = self.fc3(x)

return x.

---

15 Transfer Learning

↳ ML method where a model developed for a first task is reused as a starting point for the second task

```
model = models.resnet18(pretrained = True)

num_ftrs = model.fc.in_features     ← i/p features of last layer

exchange last fully connected layers

model.fc = nn.Linear(num_ftrs, 2)
model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(),
                      lr = 0.01)


#scheduler
step_lr_scheduler = lr_scheduler.StepLR
```

(optimizer, step-size = ,
gamma = 0.1)

```
for epoch in range (100):
    train()
    evaluate()
    scheduler.step()
```

```
model = train_model ( model, criterion,
        optimizer, scheduler,
        num_epochs = 20)
```

---

16 Tensorboard

↳ tensorflow's visualise to
↳ pytorch tensorboard --logdir = runs

install tensorboard

pip install tensorboard

feed forward code

```
import torch. utils. tensorboard import
                        SummaryWriter
import sys
```

```
~~ob~~ writer = SummaryWriter ("runs /mnist")
```

,

|

/

#plot

```
#plt.show()
img_grid = torchvision.utils.make_grid(example_
                                          dat
writer.add_image('mnist_images', img_grid)
writer.close()
//sys.exit()
```

17 Jan 2023

Loss & optimiser

```
:
.
writer.add_graph(model, example_data.reshape(-1,28*2

.
|
```

Train the model

```
:
running_loss = 0.0
running_correct = 0

.

|
```

running_loss += loss.item()

_, predicted = torch.max(outputs.data, 1)
running_correct += (predicted == labels).sum().item()

if (i+1) % 100 == 0:

:
writer.add_scalar ('training loss',
running_loss /100,
epoch * n_total_steps + i)

" ——— ('accuracy',
running_correct/100
ep — —)

running_loss = 0.0
running_correct = 0

→ checkout torch.utils.tensorboard on pytorch key

" add_pr_curve

---

18 Jan 2022

17: Save and Load Models

```
import torch
import torch.nn as nn
```

1

```
# Save Data
torch.save(arg, PATH)        ← Lazy method)

# Load data
model = torch.load(PATH)
model.eval()
```

2 Recommended:

```
# State dict
torch.save(model.state_dict(), PATH)
```