DL with PyTorch

https://www.youtube.com/watch?v=c36lUUr864M

tensors are just like arrays or vectors from numpy. By default tensors are created with requires grad = False by default. When it's set to true PyTorch knows to calculate gradients for the tensor during optimisation steps. (for when we need a variable to be optimised)

any operations with _ are in place eg. y.add_(x)

Can slice to get certain parts of tensor (useful for when one row of the tensor represents the value you need etc.) eg. x[:, 0] - all rows but only column 0

.item() gets the actual value out of the tensor, only works on single element

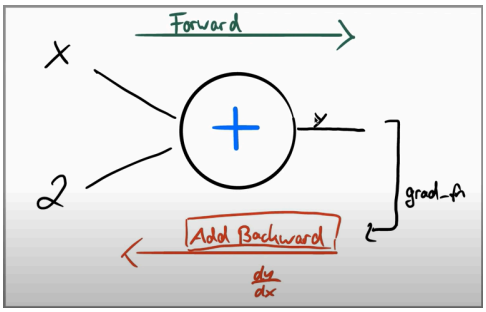
.view() to do a reasonable reshape. Can also do (-1, x) where x is the other dim you want and pytorch replaces -1 to a suitable value
OR .reshape
img = img.reshape(-1, 28*28) is very useful for flattening an image of size 28*28

Numpy cannot handle GPU tensors so we need to send them back to CPU.
z.to("cpu").numpy

AUTOGRAD

We have a tensor x, and we want to calculate the gradients of some function w.r.t x. With requires_grad = True, whenever we do any ops with the tensor, pytorch creates the computation graph for us.

example for x + 2 = y.



For each operation (here +), we have a node with inputs and output. We do the forward pass which is just the operation. Then, for wherever we stated required grad is True, PyTorch automatically creates a 'gradient function' for us which is used in backprop to get the gradients of the variable which we want to optimise. So the output y has an attribute grad_fn which points to the gradient function (here AddBackwards) which is used to calculate gradients in the backward pass (here of y wrt x)

To do it manually: -
y.backward()
x.grad() # stored in gradient attribute of x

In the background, a Jacobian matrix/ chain rule is created to compute this. so .backward() will only work on scalar values, i.e. 1×1 tensor
Otherwise need to pass a vector into backwards

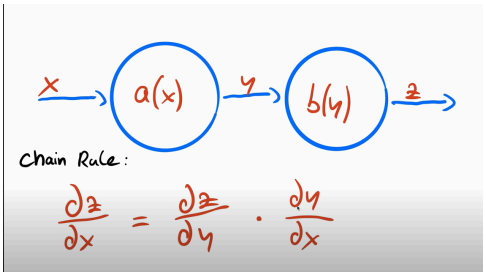
3 ways to stop torch from computing grad or tracking the history of ops
x.requires_grad_(False)
y = x.detach()
with torch.no_grad():
    ...

** Everytime, we call .backward, the gradients are accumulated/ added up into the .grad attribute of the tensors. We need to zero the gradient after every epoch with x.grad.zero_()
Or using torch's built-in optimisers: optimiser.zero_grad()

Backpropogation algorithm

We use it to calculate gradients on our weights

Chain rule: -



(Partial diff has its own diff chain rule)

At each node, we calculate the ‘local gradients’. At the end we compute the loss which we want to minimise so we need the gradient of loss wrt to initial parameter x at the beginning

DOING EVERYTHING MANUALLY

```
import numpy as np

# linear regression, just some linea
r combination of weights and input
s
# ignoring bias here
# f = w * x

# Training samples
# let's keep f = 2 * x
X = np.array([1, 2, 3, 4], dtype=np.
float32)
Y = np.array([2, 4, 6, 8], dtype=np.
float32)

# Initialise weights
w = 0.0


# model prediction
# pytorch convention, we define a
function called forward
def forward(x):
    return w * x

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

USING PYTORCH

```
import torch




X = torch.tensor([1, 2, 3, 4], dtype
=torch.float32)
Y = torch.tensor([2, 4, 6, 8], dtype
=torch.float32)

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

# model prediction


def forward(x):
    return w * x

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

```
# gradient
# loss→ MSE = 1/N * (w*x - y)**2
# derivative J or objective function
is
# dJ/dw = 1/N * 2(x) * (wx-y)
def gradient(x, y, y_predicted):
    return np.dot(2*x, y_predicted -
y).mean()
    # dot product as our x and y are
vectors (tensors)

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

# Training
learning_rate = 0.01
n_iters = 10

for epoch in range(n_iters):
    # prediction = forward pass
    y_pred = forward(X)
    # loss
    l = loss(Y, y_pred)
    # gradients
    dw = gradients(X, Y, y_pred)
    # update weights
    w -= learning_rate * dw
# ** The gradient descent alogorit
hm states we go in the opposite dir
ection of gradient as it points in th
e steepest direction of increase in l
oss

    if epoch % 1 ==0:
        print(epoch+1, w, loss)
```

```
def gradient(x, y, y_predicted):
    pass



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

# Training
learning_rate = 0.01
n_iters = 10

for epoch in range(n_iters):
    # prediction = forward pass
    y_pred = forward(X)
    # loss
    l = loss(Y, y_pred)
    # gradients
    l.backward()  # dl/dw
    # update weights
with torch.no_grad():  # we don't w
ant this to be part of the comp gra
ph
    dw = w.grad
    w -= learning_rate * dw

# zero gradients
    w.grad.zero_()

    if epoch % 1 ==0:
        print(epoch+1, w, loss)
```

```
print("Prediction after training: f(5) = {:.3f}".format(forward(5))
```


```
print("Prediction after training: f(5) = {:.3f}".format(forward(5))
```


PyTorch general training pipeline

0) data
1) design model → no of inputs and outputs/ input size etc.

Also the forward pass with all the different operations or layers
2) design/ construct the loss and optimiser
3) training loop → forward pass: computer prediction
backward pass: gradients
update weights + empty gradients
(iterate)

```
import torch
import torch.nn


X = torch.tensor([1, 2, 3, 4], dtype=torch.float32)
Y = torch.tensor([2, 4, 6, 8], dtype=torch.float32)


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

# model prediction
def forward(x):
    return w * x
```

```
import torch
import torch.nn as nn  # neural network module

X = torch.tensor([[1], [2], [3], [4]], dtype=torch.float32)
Y = torch.tensor([[2], [4], [6], [8]], dtype=torch.float32)
n_samples, n_features = X.shape

# Initialise weights
# pytorch handles this now in the model

# takes n_features while defining
input_size = n_features
output_size = n_features
model == nn.Linear(input_size, output_size)
```

```
print("Prediction before training: f
(5) = {:.3f}".format(forward(5))


# Training
learning_rate = 0.01
n_iters = 10

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

for epoch in range(n_iters):
    # prediction = forward pass
    y_pred = forward(X)
    # loss
    l = loss(Y, y_pred)
    # gradients
    l.backward()  # dl/dw
    # update weights
    optimizer.step()
    # zero gradients
    opimizer.zero_grad()

    if epoch % 1 ==0:
        print(epoch+1, w, loss)

print("Prediction after training: f(5)
= {:.3f}".format(forward(5))
```

```
print("Prediction before training: f
(5) = {:.3f}".format(model(torch.te
nsor([5]).item()))

# Training
learning_rate = 0.01
n_iters = 10

loss = nn.MSELoss()  # callable fu
nction
optimizer = torch.optim.SGD(mode
l.parameters, lr=learning_rate)

for epoch in range(n_iters):
    # prediction = forward pass
    y_pred = model(X)
    # loss
    l = loss(Y, y_pred)
    # gradients
    l.backward()  # dl/dw
    # update weights
    optimizer.step()
    # zero gradients
    optimizer.zero_grad()

    if epoch % 1 ==0:
        [w, b] = model.parameters()
        print(epoch+1, w[0][0].item(),
loss)

print("Prediction after training: f(5)
= {:.3f}".format(model(torch.tensor
([5]).item()))
```

```
# !! Now weights are inited random
ly not 0, optimiser is little different;
can play around with these
```

Remember for bigger datasets, we need batches as it is too much to store gradients over the whole epoch. So we loop over each epoch then within it loop over each batch.
This now becomes:

```
epoch = 1 forward and backward pass of ALL training samples

batch_size = number of training samples in one forward & backward pass

number of iterations = number of passes, each pass using [batch_size] number of samples

e.g. 100 samples, batch_size=20 --> 100/20 = 5 iterations for 1 epoch
```

Writing Custom Model PyTorch

```
class LinearReg(nn.Module):

    def __init__(self, input_dim, output_dim):
        super(LinearReg, self).__init__()
        # define layers
        self.lin = nn.Linear(input_dim, output_dim)

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

model = LinearReg(input_size, output_size
```

Writing Custom Transforms Torchvision

```
class ToTensor:
    def __init__(self, factor):
        # optioanls

    def __call__(self, sample):  # make it a callable object
    inputs, targets = sample
    return torch.from_numpy(inputs), torch.from_numpy(targets)
```

Hyperparameters

learning rate, num epochs

Weight init, optimiser, new or remove layers

Evaluation

Here's where we use an unseen test set to report results. (Before this, we use a val set if we just want to see how our model is doing on unseen *by it* data and improve it. We are avoiding data bias) Remember we don't want these computations to be tracked so use torch.no_grad()

Accuracy is one option

ROC/ AOC

Euclidean Distance

Rank K

Writing Custom Dataset PyTorch

```
# we inherit Dataset from torch.utils
# then we must implement 3 methods

class MyDataset(Dataset):
```

```
    def __init__(self):
        # data loading
        # np.loadtxt is great for csvs
        # remember to keep the shape num_samples x num_features/dims

    def __getitem__(self, index):
        # allows for indexing with simply dataset[0]

    def __len__(self):
```

** To access dataloader elements, can write next(iter(dataloader)) → make it iterable
OR for i, (images, labels) in enumerate(dataloader)
OR for images, labels in testloader

MATHS

** ML stuff: Linear reg produces a linear line that fits the data
Logistic reg f = wx+b PLUS sigmoid (returns val between 0-1) function at the end (used for binary classification, so we use BCEloss)
In it, it's recommended to scale data to have 0 mean and unit variance

- Softmax: For each element S(yi) = e^yi / SUM e^yi (bottom half is normalising). It basically squashes the output to be between 0 and 1 so we get probabilities. So when a linear layer gives us an output of say, 3 values ('raw values' we call these scores or logits)
  To get prediction out of it, simply do torch.max(Y_pred, 1) # along dim 1
  Can do that on logits too not just softmax.

** In python, most ops on arrays may be element-wise

- Softmax and CE loss often go hand-in-hand. CE(y', y) = -1/N * SUM yi * log(yi'). Loss increases as predicted probability goes further from actual class. We compare with a 1-hot vector. Here, yi is one-hot and yi' is softmax. PyTorch's CE already applies softmax. Also Y must be class label (which are

still numbers BUT NOT 1-hot) and as seen, Y_pred must be raw/ logits not softmaxed.

PROTIP: - CE for multi class, if there's just 2 classes can reform it to have just one output/ logit and use sigmoid and round up on that, THEN BCE (PyTorce BCE does not implement sigmoid)

- Linear regression or entirely linear models do not suit more complicated tasks. To prevent our entire model from turning linear even though we have many layers, we need to add in activation functions. Some common ones are sigmoid, softmax,
  tanh(scaled sigmoid with a shift of -1) → returns value between -1 and 1,
  Relu → basically linear for values above 0 but it is non-linear
  Leaky Relu → prevents the 0 value which could cause 0 grads and prevent the neuron from training
  ^^ these are used in hidden layers (middle layers)

Matplotlib can plot tensor images I think

https://colab.research.google.com/drive/1XlqPFR6QvdAALt1RsuMFcObx8xIeRaBd?usp=sharing