Python and Pytorch basics

oerview of today's lecture

1. Tensor basics:
    - Create, Operations, Numpy
2. Autograd:
    - Linear regression example
3. Training loop:
    - Model, Loss, Optimizer and Scheduler
4. Nueral network:
    - Datasets, DataLoader, GPU (optional)

1. Tensors

Definition:

A multidimenisonal matrix containing a single data type.

Tensors are simply mathematical objects that can be used to describe physical properties , just like scalars and vectors. In fact tensors are merely a generalisation of scalars and vectors; a scalar is a zero rank tensor, and a vector is a first rank tensor.

Example - Hook's law in 3D:

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}$$ -$\sigma, \epsilon \in \mathbb{R}^{3 \times 3}$ are second order tensors.
-$C \in \mathbb{R}^{3 \times 3 \times 3 \times 3}$ is a fourth order tensor.

Pytorch Tensors

Everything in Pytorch is based on tensors.

Use torch.tensor() to generate tensors with known entries.

In [ ]:
```
import torch # This is the first line you should write in your Pytorch script
import numpy as np

# Create Pytorch tensor for scalar 1
a = torch.tensor([1.])
print("a = ", a)

# Create pytorch tensor for vector [1,-0.5]
b = torch.tensor([1.0,-0.5])
```

```
print("b = ", b)

# Create Pytorch tensor for matrix [1,-0.5,-0.5,1]
c = torch.tensor([[1.,-0.5],[-0.5,1.]])
print("c = ", c)
```

Use torch.rand() to generate tensors with random entries between (0,1)

In [ ]:
```
# We can also generate random numbers
x = torch.rand(2,2,3) # tensor, 3 dimensional
print (x)

y = torch.rand(2,3,4,5) # tensor, 4 dimensional
print(y)
```

We can use torch.ones() to generate constant tensor

In [ ]:
```
# Constant second order tensor of 1
x = torch.ones(2,2)
print(x)

# Constant third order tensor of 1.5
x = torch.ones(2,2,2)*1.5
print(x)
```

Slicing of tensor entries is similar to numpy

In [ ]:
```
x = torch.rand(3,3)
print(x)
# To access the first row
a = x[0,:]
print(a)
# To access the first column
b = x[:,0]
print(b)
```

Check size of tensors with tensor.shape

In [ ]:
```
a = torch.rand(3,4)
print(a.shape)
# You can also specify the dimension of the tensor that you wish to query
print(a.shape[0])
```

Check tensor type with tensor.dtype - default float32 (single precision)

In [ ]:
```
a = torch.rand(3,4)
print(a.dtype)
b = torch.tensor([0,1])
print(b.dtype)
c = torch.tensor([0.,1.])
```

```
print(c.dtype)
d = torch.tensor([0,1], dtype=torch.float64)
print(d.dtype)

d = c.double()# this changes the data type to double precision
print(d.dtype)
```

Reshape tensors with torch.view()

In [ ]:
```
a = torch.ones(2,2)
# Reshape a 2x2 matrix into a 4x1 vector
b = a.view(4,1)
print(b)
print(b.shape)
# You can put -1 in one dimension, to force Pytorch to infer from the rest dimensio
c = a.view(-1,2)
d = a.view(-1,4)

print(c.shape)
print(d.shape)
```

Tensor Linear Algebra

Adding two tensors torch.add() or simply +

In [ ]:
```
a = torch.rand(2,2)
b = torch.ones(2,2)

c = a+b
print(c)
d = torch.add(a,b)
print(d)
```

Similarly Substraction "-" Multiplication "*" Division "/"

In [ ]:
```
a = torch.ones(2,2)
b = torch.ones(2,2)+1

print("a-b = ", a-b)
print("a*b", a*b)
print("a/b", a/b)  # Note this is element wise division!
```

Converting torch.tensor to Numpy array and vice versa

In [ ]:
```
# this is important for exmaple in visualization - Matplotlib does not accept tenso
a = torch.ones(2,2)
b = a.numpy()
print(b)
print(type(b))
```

In [ ]:
```
# To do the inverse, just put torch.tensor(nparray)
a = np.eye(3)
b = torch.tensor(a)
print(a)
print(b)
```

Autograd

Finding function derivatives

Autograd is a key enabler of Pytorch. The package uses automatic differentiation for all operations on Tensors. Generally speaking,"torch.autograd" is an engine for computing the vector-Jacobian product. It computes partial derivates while applying the chain rule.

Let us consider the function $$y = 2x + 1\quad for \ x \in [0,1]$$ Then $$\frac{dy}{dx} = 2 $$

Let us now use "torch.autograd" to find this.

In [ ]:
```
import matplotlib.pyplot as plt
# requires_grad = True -> tracks all operations on the tensor.
x = torch.linspace(0,1,10,requires_grad=True)
y = 2*x + 1
dydx = torch.autograd.grad(y, x, grad_outputs=torch.ones_like(y),create_graph=True)
plt.plot(x.detach().numpy(),y.detach().numpy()) # You need to detach the tensor fir
plt.plot(x.detach().numpy(),dydx.detach().numpy())
plt.show()
```

Now suppose we further define $$z = y^2 $$ and $$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} =2y \cdot 2 = 8x+4$$

In [ ]:
```
x = torch.linspace(0,1,10,requires_grad=True)
y = 2*x + 1
z = y**2
dzdx = torch.autograd.grad(z, x, grad_outputs=torch.ones_like(y),create_graph=True)
plt.plot(x.detach().numpy(),dzdx.detach().numpy())
plt.show()
```

Back propagation with "scalar_tensor.backward()"

A convenient way of computing the derivatives of a scalar function with respect to inputs. Commonly used to compute the back propagation of Loss functions. Note here Loss function is a scalar function by default.

In [ ]:
```
# Let's compute the gradients with backpropagation
# When we finish our computation we can call .backward() and have all the gradients
# The gradient for this tensor will be accumulated into .grad attribute.
# It is the partial derivate of the scalar function w.r.t. the input tensor

x = torch.tensor([0., 0.5, 1],requires_grad=True)
```

```
y = x + 2
z = torch.sum(y) # z a scalar function

print(x.grad)
z.backward()
print(x.grad) # dz/dx
```

Note: backward() accumulates the gradient for this tensor into .grad attribute

Use optimizer.zero_grad() to clear the gradient in optimization!

Stop a tensor from tracking history:

For example during the training loop when we want to update our weights, or after training during evaluation. These operations should not be part of the gradient computation. To prevent this, we can use:

- `x.requires_grad_(False)`
- `x.detach()`
- wrap in `with torch.no_grad():`

In [ ]:
```
# .requires_grad_(...) changes an existing flag in-place.
a = torch.randn(2, 2)
b = (a * a).sum()
print(a.requires_grad)
print(b.requires_grad)
```

In [ ]:
```
a.requires_grad_(True)
b = (a * a).sum()
print(a.requires_grad)
print(b.requires_grad)
```

In [ ]:
```
# .detach(): get a new Tensor with the same content but no gradient computation:
a = torch.randn(2, 2, requires_grad=True)
b = a.detach()
print(a.requires_grad)
print(b.requires_grad)
```

In [ ]:
```
# wrap in 'with torch.no_grad():'
a = torch.randn(2, 2, requires_grad=True)
print(a.requires_grad)
with torch.no_grad():
    b = a ** 2
    print(b.requires_grad)
```

Gradient Descent Method

Linear Regression example with mean squared loss function:

$f(x) = w * x + b$

here $$\hat{f}(x) = 2*x$$

$$loss(f,\hat{f}) = \frac{1}{N_x} \sum_{k=0}^{N_x}(f(x_k) - \hat{f}(x_k))^2 $$

In [ ]:
```python
# Define the prediction model class
def forward(x):
    return w * x

# Define the loss function
def loss(y, y_pred):
    return ((y_pred - y)**2).mean()
```

In [ ]:
```python
X = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8], dtype=torch.float32)
Y = torch.tensor([2, 4, 6, 8, 10, 12, 14, 16], dtype=torch.float32)

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

X_test = 5.0

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

In [ ]:
```python
# Training
learning_rate = 0.01 # Important hyper parameter!
n_epochs = 100

for epoch in range(n_epochs):
    # predict = forward pass
    y_pred = forward(X)

    # loss
    l = loss(Y, y_pred)

    # calculate gradients = backward pass
    l.backward()

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

    # zero the gradients after updating
    w.grad.zero_()

    if (epoch+1) % 10 == 0:
        print(f'epoch {epoch+1}: w = {w.item():.3f}, loss = {l.item():.3f}')

print(f'Prediction after training: f({X_test}) = {forward(X_test).item():.3f}')
```

3. Model, Loss & Optimizer

A typical PyTorch pipeline looks like this:

1. Design model (input, output, forward pass with different layers)
2. Specify train and test data
3. Construct loss and optimizer
4. Training loop:
    - Forward = compute prediction and loss
    - Backward = compute gradients
    - Update weights

In [ ]:
```python
import torch
import torch.nn as nn

# Linear regression
# f = w * x
# here : f = 2 * x

# 1) Design Model, the model has to implement the forward pass!

# Here we could simply use a built-in model from PyTorch
# model = nn.Linear(input_size, output_size)

class LinearRegression(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearRegression, self).__init__()
        # define different layers
        self.lin = nn.Linear(input_dim, output_dim)

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

In [ ]:
```python
# 2) Specify training samples, watch the shape!
X = torch.tensor([[1], [2], [3], [4], [5], [6], [7], [8]], dtype=torch.float32)
Y = torch.tensor([[2], [4], [6], [8], [10], [12], [14], [16]], dtype=torch.float32)

# 2) A general way of specifying train/test data is to reshape the data as N_sample
n_samples, n_features = X.shape
print(f'n_samples = {n_samples}, n_features = {n_features}')

# 2) create a test sample
X_test = torch.tensor([5], dtype=torch.float32)

# 2) Initialize model
input_size, output_size = n_features, n_features

model = LinearRegression(input_size, output_size)

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

Select your optimizer with torch.optim, and be careful with your learning rate

Tune your learning rate during the training with torch.optim.lr_scheduler

In [ ]:
```
# 3) Define loss and optimizer
learning_rate = 0.01
n_epochs = 100

loss = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1) # D
```

In [ ]:
```
# 4) Training loop
for epoch in range(n_epochs):
    # predict = forward pass with our model
    y_predicted = model(X)

    # loss
    l = loss(Y, y_predicted)

    # calculate gradients = backward pass
    l.backward()

    # update weights
    optimizer.step()

    # zero the gradients after updating
    optimizer.zero_grad()

    # Update learning rate
    scheduler.step()


    if (epoch+1) % 10 == 0:
        w, b = model.parameters() # unpack parameters
        print('epoch ', epoch+1, ': w = ', w[0][0].item(), ' loss = ', l.item())

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

4. First Neural Net

Datasets, DataLoader, Neural Net, Training & Evaluation, GPU(optional)

Datasets: Let us consider the celebrated MNIST data set for ML

Don't forget to normalize your data before training!

In [ ]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Device configuration (Optional)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
# Hyper-parameters
input_size = 784 # 28x28
output_classes = 10
FCNN_arch = [input_size, 500,output_classes]
non_linearity = nn.ReLU
num_epochs = 2
batch_size = 100
learning_rate = 0.001

# Loading the MNIST dataset  600 training data, 100 test data
# Generally you need to define this your self
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

examples = iter(test_loader)
example_data, example_targets = next(examples)

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

Design your neural network architecture and choose nonlinearity

In [ ]:

```
# Fully connected neural network w
class FCNN(nn.Module):
    def __init__(self, layers, nonlinearity):
        super(FCNN, self).__init__()

        self.n_layers = len(layers) - 1

        assert self.n_layers >= 1

        self.layers = nn.ModuleList()

        for j in range(self.n_layers):
            self.layers.append(nn.Linear(layers[j], layers[j + 1]))
```

```
            if j != self.n_layers - 1:
                self.layers.append(nonlinearity())

    def forward(self, x):
        for _, l in enumerate(self.layers):
            x = l(x)

        return x

FCNN_arch = [input_size, 500,output_classes]
non_linearity = nn.ReLU
model = FCNN(FCNN_arch, non_linearity).to(device)
print(model)
```

Loss and optimizer: this is a classification problem - use cross entropy loss

In [ ]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1) # D
```

In [ ]:
```
# Train the model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass and loss calculation
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        scheduler.step()
        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{600}], Loss: {los

    # Test the model: we don't need to compute gradients
    with torch.no_grad():
        n_correct = 0
        n_samples = len(test_loader.dataset)

        for images, labels in test_loader:
            images = images.reshape(-1, 28*28).to(device)
            labels = labels.to(device)

            outputs = model(images)

            # max returns (output_value ,index)
```

```
            _, predicted = torch.max(outputs, 1)
            n_correct += (predicted == labels).sum().item()

        acc = n_correct / n_samples
    print ("end epoch", epoch, "Train loss", loss.item(), "Accuracy", acc)
```