PyTorchTrial

March 28, 2025

PyTorch Testing

[294]:
```python
import numpy as np
import torch
```

[295]:
```python
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
```

[296]:
```python
torch.manual_seed(42)
torch.use_deterministic_algorithms(False)
```

[297]:
```python
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of␣
↪x_data
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.8823, 0.9150],
        [0.3829, 0.9593]])
```

[298]:
```python
shape = (2,3)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.3904, 0.6009, 0.2566],
        [0.7936, 0.9408, 0.1332]])
```

```
Ones Tensor:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros Tensor:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

[299]:

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Torch Operations

There are over 1200 tensor operations, they are comprehensively descirbed in this document. They tend to relate to matrix manipulation (transposig, indexing, slicing), sampling, and more:

Docs

Note that tensors are defaultly created on the CPU, We can move tensors using the following command, but memory-wise it is far more efficient to just create the tensor on the correct device in the first place.

[300]:

```
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())
    print(f"Moved to {torch.accelerator.current_accelerator}")
# TODO: Figure out how to determine what device we're on.
```

```
Moved to <function current_accelerator at 0x1246da160>
```

[301]:

```
#Standard numpy-like indicing and slicin
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)
```

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
```

```
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

[302]:
```
#joining tensors
t1 = torch.cat([tensor, tensor, tensor], dim=1) #dim sets the dimension to␣
↪concat along, in this case we add more columns
print(t1)
```

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

[303]:
```
#Standard arithmetic operations

#Matrix multiplication
# This computes the matrix multiplication between two tensors. y1, y2, y3 will␣
↪have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

print(y1)
print(y2)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)


# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

print(z1)
print(z2)

z3 = torch.rand_like(tensor);
torch.mul(tensor, tensor, out=z3);
```

```
tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
tensor([[1., 0., 1., 1.],
```

```
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

[304]:
```
#If you have a signal item in a tensor, you can convert it to a standard python␣
↪number type
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))
```

```
12.0 <class 'float'>
```

[305]:
```
# In-place operations -> think about bubble sort being in-place -> no␣
↪additional memory required.
print(f"{tensor} \n")
new_tensor = tensor.add_(5) # Adding the '_' suffix does this -> And because of␣
↪this suffix, the output of our tensor is
print(new_tensor)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

Bridge with NumPy

[306]:
```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

```
t: tensor([1., 1., 1., 1., 1.])
n: [1. 1. 1. 1. 1.]
```

[307]:
```
# These are stored at the SAME PLACE IN MEMORY
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.])
n: [2. 2. 2. 2. 2.]
```

[308]:
```
#NumPy to Tensor
n = np.ones(5)
t = torch.from_numpy(n)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
n: [1. 1. 1. 1. 1.]
```

[309]:
```
# Again, SAME PLACE IN MEMORY
np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
n: [2. 2. 2. 2. 2.]
```

Datasets and DataLoaders

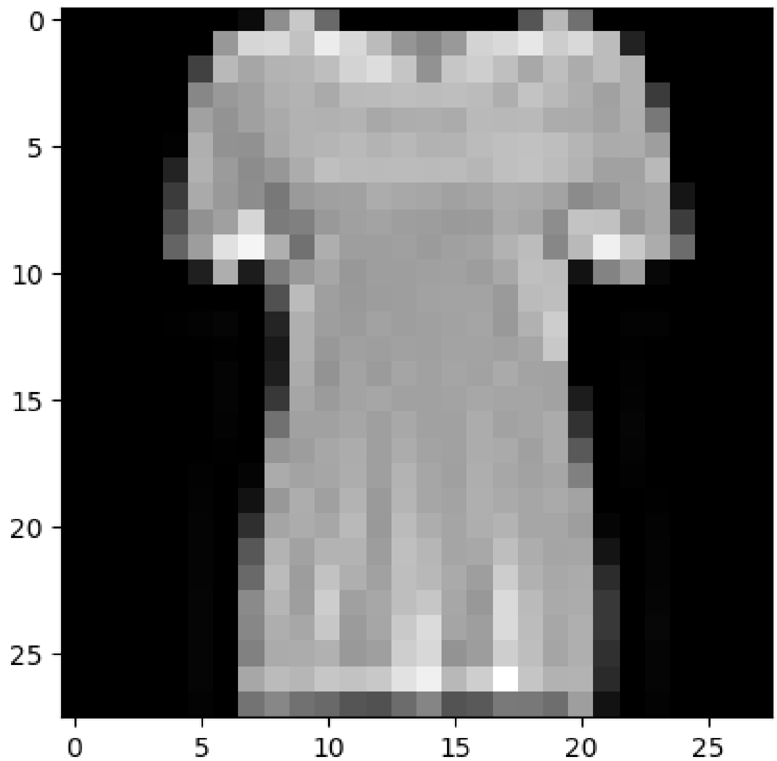
[310]:
```
#Loading a dataset, in this case we can load Fashion-MNIST dataset from␣
↪TorchVision
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
```

[311]:
```
# We are looking to grab 60000 training examples and 10000 test examples from␣
↪Fashion-MNIST

# root is the path where the train/test data is stored
# train specifes the training or test dataset
# download=True downlaods the data from the internet if it's not available at␣
↪root
# transform and target_transform specify the feature and label transformation


training_data = datasets.FashionMNIST(root="data",
                                      train=True,
                                      download=True,
                                      transform=ToTensor()
                                     )

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
```

```
)
```

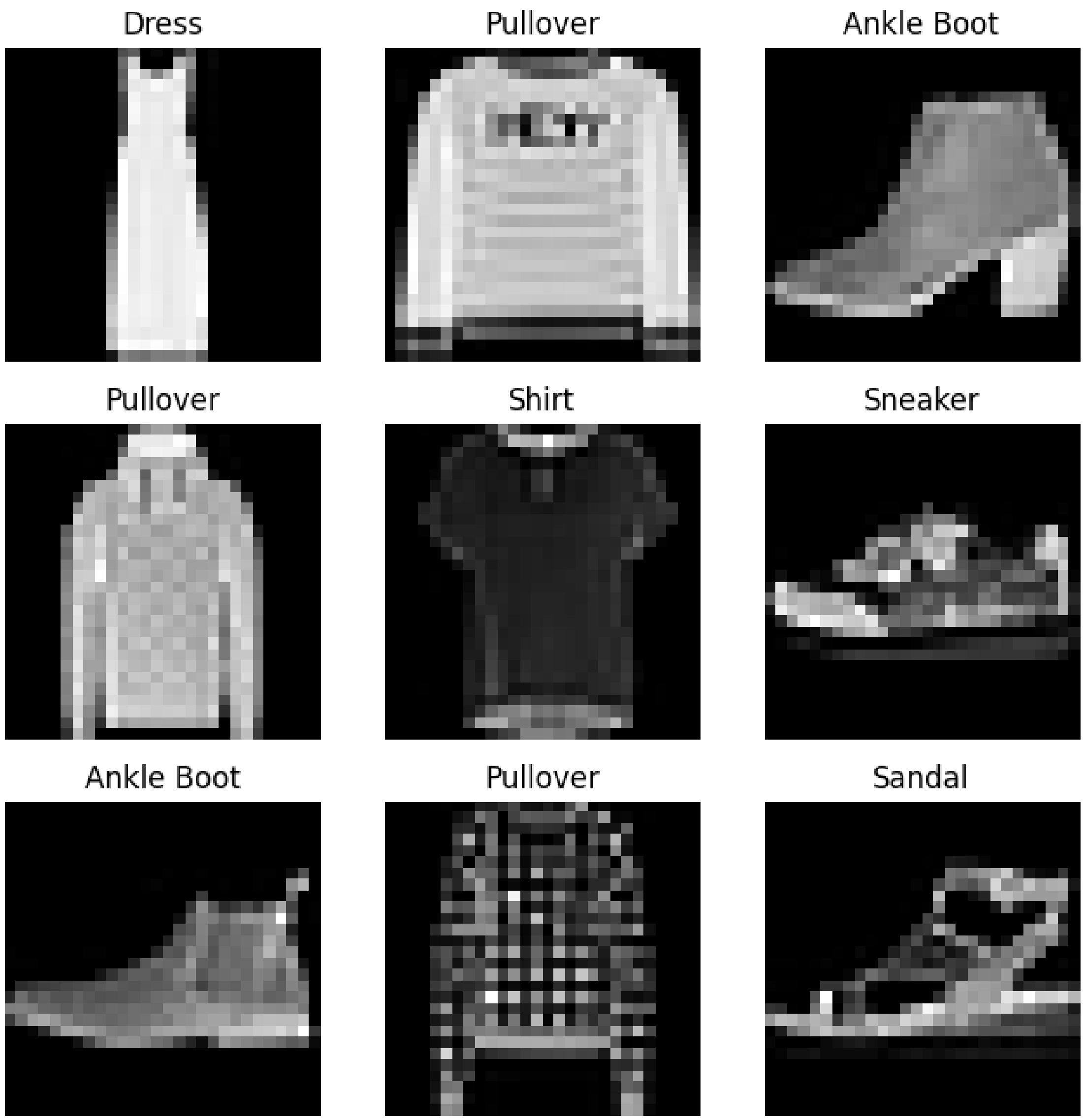
[312]:
```
#print(training_data[10])
plt.imshow(training_data[10][0].squeeze(), cmap='gray')
#This is the standard format of our data.
```

[312]: <matplotlib.image.AxesImage at 0x300a69950>



[313]:
```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
```

```python
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```



Creating a Custom Dataset for your files

You need to implement the following functions:

init Run once when initializing the Dataset object. Initialize the directory containing the images, the annotations file, and both transforms (covered more later)

len Returns the length of the dataset (number of samples)

getitem Loads and returns a sample from the dataset at the given index idx, generally, we would probably want to return a tensor

[314]:
```
#In this case

#Class that extends the Dataset template

#You can consider this the behind the scenes implementation.
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None,␣
 ↪target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img.labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

[315]:
```
#In the case that we had an EEG dataset based on SSVEP, we would want to define␣
 ↪our database
#with the following attributes

#Our __init__() should have our eeg_data and our labels. This is usually
# samples * channels * (timesteps) * blocks
#And then in this case we can assign our labels as well by translating the␣
 ↪dataset from what we have.
```

[316]:
```
# Here's a sample implementation:

class EEGDataset(Dataset):
    def __init__(self, eeg_signals, labels, window_size=1280, stride=160,␣
 ↪transform=None):
```

```
        # We need to concat each block on top of each other.

        self.data = eeg_signals # Shape: (num_samples, channels)


# TODO: Return to this later
```

Preparing your data for training with DataLoaders

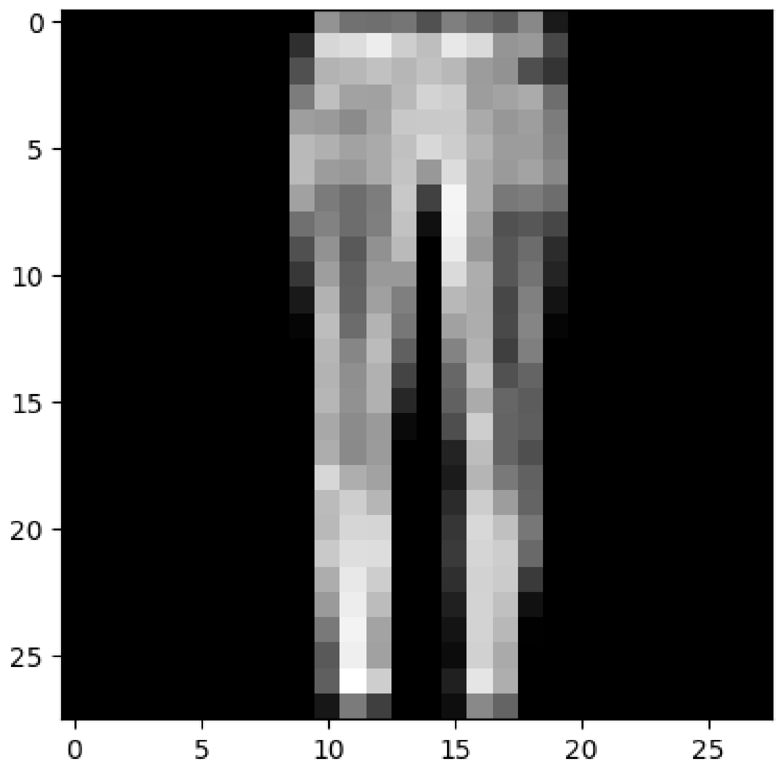
Generally, when training, we want to pass samples in "minibatches", reshuffule the data at every epoch to reduce model overfitting, and use Python's multiprocessing to speed up data retrieval.

DataLoader is an interable that abstracts this complexity for us in an easy API

[317]:
```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

[318]:
```
#Iterating through the dataloader

#Display image and label
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Label batch shape: {train_labels.size()}")
img = train_features[0].squeeze() # Removes all redudant dimensions (1)
label = train_labels[0]
plt.imshow(img, cmap='gray')
plt.show()
label_here = labels_map[label.item()]
print(f"Label: {label_here}")

# This progressively proceeds through the dataset.
```

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Label batch shape: torch.Size([64])
```




```
Label: Trouser
```

Transforms

Data doesn't always come in final processed form, we may need to put the data through a few transforms first. In our case, we likely want to pass our EEG signals through some band pass filter instead of instantly choosing to try and process it through the model. There is the transform parameter of the dataset that modifies the data, and the target_transform that modifies the labels. Remember before that we had a map from numbers to labels, and maybe we would prefer it if those labels were more closely aligned to the actual names, so we could define a target_transform function that would do that for us

[319]:
```
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    #Taking numbers from 0-9 and converting them to a One-Hot Encoded binary␣
 ↪representation of the label.
    #This makes sense, as this is what we expect from our data as an output␣
 ↪from a deep network.
```

```
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.float).
 ↪scatter_(0, torch.tensor(y), value=1))
)

#Note that scatter copies eleemnts
```

Bulding the neural network

Neural networks comrpise of layers that perform operations on data. The torch.nn namespace gives all the building blocks we need to build one. Every module in PyTorch subclasses the nn.Module. A neural network is a module itself that consists of other modules (layers). This nested strtucture allows for building and manging complex architectures easily

In the following sections, we'll build a neural network to classify images in the FasionMNIST dataset.

[320]:
```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Training Device

We want to train our model on some accelertor such as CUDE, MPS, MTIA, or XPU. If it is available, we will use it, but otherwise we can just use the CPU

[321]:
```
device = torch.accelerator.current_accelerator().type if torch.accelerator.
 ↪is_available() else "cpu"
print(f"Using {device} device")
```

```
Using mps device
```

Define the class

Generally, we have class for our neural network which has a few functions that we typically use. The first is **init** for setting up the layers in our neural network stack. Inside is forward(), which does a forward pass of the network

[322]:
```
class ImageClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28 * 28, 512), # 512 nodes that each take in 28 * 28␣
 ↪inputs.
            nn.ReLU(), #Rectified linear unit, basically our cutoff function␣
 ↪for activation.
            nn.Linear(512, 512), # 512 nodes that each take in 512 inputs
            nn.ReLU(),
            nn.Linear(512, 10), # Our final class of 10
```

```python
        )

    def forward(self, x):
            x = self.flatten(x)
            logits = self.linear_relu_stack(x)
            return logits
```

[340]:
```python
model = ImageClassifier().to(device)
print(model, device)
```

```
ImageClassifier(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
) mps
```

[341]:
```python
for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

```
Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values :
tensor([[-0.0121, -0.0167,  0.0239,  …, -0.0106,  0.0011, -0.0227],
        [-0.0199, -0.0037,  0.0109,  …,  0.0123,  0.0099, -0.0354]],
       device='mps:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values :
tensor([-0.0287,  0.0055], device='mps:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values :
tensor([[-0.0136,  0.0041, -0.0354,  …, -0.0263, -0.0246,  0.0384],
        [-0.0288,  0.0024, -0.0169,  …,  0.0304, -0.0076,  0.0319]],
       device='mps:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values :
tensor([0.0330, 0.0107], device='mps:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values :
tensor([[-0.0122, -0.0052,  0.0375,  …, -0.0128,  0.0198, -0.0279],
        [-0.0260, -0.0189, -0.0428,  …,  0.0231,  0.0302,  0.0026]],
       device='mps:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values :
tensor([0.0007, 0.0186], device='mps:0', grad_fn=<SliceBackward0>)
```

Automatic Differentiation with torch.autograd

We went through this with minigrad -> it automatically can find the gradient of nodes that need to be optimized. This general works by calling loss.backward() and then accessing w.grad and b.grad. in practice, we want to calculate this as a tensor, multiply it by some training coefficient (alpha) and then apply that to the entire value tensor.

Note that we need to mark attributes with requires_grad as True before PyTorch actually decices to take the gradient when it backpropogates. In other words, for it to be considered as a variable and not as a constant.

For performance reasons, we can only call backward() once on a given graph. If we want to do it again on the same graph, we need to pass retain_graph = True to the backward call.

What is happening here?

As with micrograd, we had each numerical value stored as its own class:

This numerical value stores the out values (for backpropogation later) and the operation that created it. When you backprogogate,

Remember that the partial of the loss is equal to the sum of all the paths that the change in that variable has an influence on.

So to calcuate the gradient of a variable, we need to have the information on all the other nodes influenced by this value.

If we proceed backwards starting from the output node, then we have the change in the loss from the preceding nodes that the output goes to, and then we can calculate the gradient of that node. Since we do BFT, we know that we will encounter each output node before we get to the node that requires all their values

[325]:
```
# Moving on
# We have a function type for our loss.
# This is the simplest neural network that we can have:

import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

[326]:
```
loss.backward()
```

[327]:
```
w.grad
```

[327]:
```
tensor([[0.3121, 0.3190, 0.0224],
        [0.3121, 0.3190, 0.0224],
        [0.3121, 0.3190, 0.0224],
```

```
        [0.3121, 0.3190, 0.0224],
        [0.3121, 0.3190, 0.0224]])
```

[328]:
```
b.grad
```

[328]:
```
tensor([0.3121, 0.3190, 0.0224])
```

[329]:
```
z = torch.matmul(x, w)+b
print(z.requires_grad)
# There are cases where we want to disable gradient tracking, usually when the
↪computational history and gradient computation is not required, like when
↪the model is already trained
# We can declare the following:
with torch.no_grad():
    z=torch.matmul(x,w)+b
    print(z.requires_grad)

# You can achieve the same
```

```
True
False
```

From the PyTorch Docs

More on Computational Graphs

Conceptually, autograd keeps a record of data (tensors) and all executed operations (along with the resulting new tensors) in a directed acyclic graph (DAG) consisting of Function objects. In this DAG, leaves are the input tensors, roots are the output tensors. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

In a forward pass, autograd does two things simultaneously:

```
run the requested operation to compute a resulting tensor

maintain the operation’s gradient function in the DAG.
```

The backward pass kicks off when .backward() is called on the DAG root. autograd then:

```
computes the gradients from each .grad_fn,

accumulates them in the respective tensor’s .grad attribute

using the chain rule, propagates all the way to the leaf tensors.
```

Optional: Jacobian Product

Optional Reading: Tensor Gradients and Jacobian Products

In many cases, we have a scalar loss function, and we need to compute the gradient with respect to some parameters. However, there are cases when the output function is an arbitrary tensor. In this case, PyTorch allows you to compute so-called Jacobian product, and not the actual gradient.

For a vector function y=f(x)y =f(x), where x= x1,…,xn x = x1,…,xn and y= y1,…,ym y = y1,…,ym , a gradient of yy with respect to xx

is given by Jacobian matrix: J=(y1 x1 y1 xn   ym x1 ym xn) J= x1 y1 x1 ym   xn y1 xn ym

Instead of computing the Jacobian matrix itself, PyTorch allows you to compute Jacobian Product vT JvT J for a given input vector v=(v1…vm)v=(v1…vm). This is achieved by calling backward with vv as an argument. The size of vv should be the same as the size of the original tensor, with respect to which we want to compute the product:

Optimizing Model Parameters

Hyperparameters

Hyperparameters are adjustable parameters that control the training process. They can also be considered to be factors such as model size, receptive field for CNNs, and more parameters specific to a particular architecture. In our case, we have:

learning rate (gradient scalar multiplier)

batch_size (Number of data samples propogated through the network before the parameters are updated.)

epochs (Number of times to iterate over the dataset)

[330]:
```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

Optimization Loop

We have the train loop and the test/validation loop.

Picking our loss

Standard loss functions are nn.MSELoss (mean squared error) for regression, and nn.NLLLoss (negative log likelihood) for classification. nn.CrossEntropyLoss combines nn.LogSoftMax and nn.NLLLoss

In this cass, we'll choose cross entropy loss

[331]:
```
loss_fn = nn.CrossEntropyLoss()
```

[353]:
```
if not torch.backends.mps.is_available():
    if not torch.backends.mps.is_built():
        print("MPS not available because PyTorch wasn't built with MPS support")
    else:
        print("MPS not available - requires macOS 12.3+ and an MPS-enabled␣
 ↪device")
else:
    device = torch.device("mps")
    print(f"Using device: {device}")
```

```
Using device: mps
```

Optimizer

Optimization algorithms define how we perform the process of adjusting the model parameters to reduce model error in each step.

[342]:
```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

[ ]:
```
```

[ ]:
```
```

[ ]:
```
```

[346]:
```
#torch.set_default_device(torch.device('mps'))
```

[354]:
```
# We usually define some function that does training for us having been fed our␣
↪dataloader.
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    #Set model to training mode - important for batch normalization and dropout␣
↪layers
    #This is unnecessary in this case but its best practice
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X = X.to(device)
        y = y.to(device)
        # Compute predictions
        pred = model(X)
        loss = loss_fn(pred, y)

        #Backpropogation
        loss.backward()
        optimizer.step() # does a step at the rate of our learning rate.
        optimizer.zero_grad()

        #Give an update every 100 batches
        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

[355]:
```
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and␣
↪dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0
```

```
    # Evaluating the model with torch.no_grad() ensures that no gradients are␣
↪computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage␣
↪for tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            X = X.to(device)
            y = y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:␣
↪{test_loss:>8f} \n")
```

[356]:
```
epochs = 10 #Go through the dataset 10 times
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

```
Epoch 1
-------------------------------
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[356], line 4
      2 for t in range(epochs):
      3     print(f"Epoch {t+1}\n-------------------------------")
----> 4     train_loop(train_dataloader, model, loss_fn, optimizer)
      5     test_loop(test_dataloader, model, loss_fn)
      6 print("Done!")

Cell In[354], line 7, in train_loop(dataloader, model, loss_fn, optimizer)
      4 #Set model to training mode - important for batch normalization and␣
↪dropout layers
      5 #This is unnecessary in this case but its best practice
      6 model.train()
----> 7 for batch, (X, y) in enumerate(dataloader):
      8     X = X.to(device)
      9     y = y.to(device)
```

```
File /opt/anaconda3/envs/eeg-env/lib/python3.13/site-packages/torch/utils/data/
 ↪dataloader.py:708, in _BaseDataLoaderIter.__next__(self)
    705 if self._sampler_iter is None:
    706     # TODO(https://github.com/pytorch/pytorch/issues/76750)
    707     self._reset()  # type: ignore[call-arg]
--> 708 data = self._next_data()
    709 self._num_yielded += 1
    710 if (
    711     self._dataset_kind == _DatasetKind.Iterable
    712     and self._IterableDataset_len_called is not None
    713     and self._num_yielded > self._IterableDataset_len_called
    714 ):

File /opt/anaconda3/envs/eeg-env/lib/python3.13/site-packages/torch/utils/data/
 ↪dataloader.py:763, in _SingleProcessDataLoaderIter._next_data(self)
    762 def _next_data(self):
--> 763     index = self._next_index()  # may raise StopIteration
    764     data = self._dataset_fetcher.fetch(index)  # may raise StopIteratio
    765     if self._pin_memory:

File /opt/anaconda3/envs/eeg-env/lib/python3.13/site-packages/torch/utils/data/
 ↪dataloader.py:698, in _BaseDataLoaderIter._next_index(self)
    697 def _next_index(self):
--> 698     return next(self._sampler_iter)

File /opt/anaconda3/envs/eeg-env/lib/python3.13/site-packages/torch/utils/data/
 ↪sampler.py:344, in BatchSampler.__iter__(self)
    342         yield [*batch_droplast]
    343 else:
--> 344     batch = [*itertools.islice(sampler_iter, self.batch_size)]
    345     while batch:
    346         yield batch

File /opt/anaconda3/envs/eeg-env/lib/python3.13/site-packages/torch/utils/data/
 ↪sampler.py:198, in RandomSampler.__iter__(self)
    196 else:
    197     for _ in range(self.num_samples // n):
--> 198         yield from torch.randperm(n, generator=generator).tolist()
    199     yield from torch.randperm(n, generator=generator).tolist()[
    200         : self.num_samples % n
    201     ]

File /opt/anaconda3/envs/eeg-env/lib/python3.13/site-packages/torch/utils/
 ↪_device.py:104, in DeviceContext.__torch_function__(self, func, types, args,␣
 ↪kwargs)
    102 if func in _device_constructors() and kwargs.get('device') is None:
    103     kwargs['device'] = self.device
--> 104 return func(*args, **kwargs)
```

```
RuntimeError: Expected a 'mps:0' generator device but found 'cpu'
```

[343]:
```
torch.argmax?
```

```
Docstring:
argmax(input) -> LongTensor

Returns the indices of the maximum value of all elements in the :attr:`input`␣
↪tensor.

This is the second value returned by :meth:`torch.max`. See its
documentation for the exact semantics of this method.

.. note:: If there are multiple maximal values then the indices of the first␣
↪maximal value are returned.

Args:
    input (Tensor): the input tensor.

Example::

    >>> a = torch.randn(4, 4)
    >>> a
    tensor([[ 1.3398,  0.2663, -0.2686,  0.2450],
            [-0.7401, -0.8805, -0.3402, -1.1936],
            [ 0.4907, -1.3948, -1.0691, -0.3132],
            [-1.6092,  0.5419, -0.2993,  0.3195]])
    >>> torch.argmax(a)
    tensor(0)

.. function:: argmax(input, dim, keepdim=False) -> LongTensor
   :noindex:

Returns the indices of the maximum values of a tensor across a dimension.

This is the second value returned by :meth:`torch.max`. See its
documentation for the exact semantics of this method.

Args:
    input (Tensor): the input tensor.
    dim (int): the dimension to reduce. If ``None``, the argmax of the flattened␣
↪input is returned.
    keepdim (bool): whether the output tensor has :attr:`dim` retained or not.

Example::
```

```
    >>> a = torch.randn(4, 4)
    >>> a
    tensor([[ 1.3398,  0.2663, -0.2686,  0.2450],
            [-0.7401, -0.8805, -0.3402, -1.1936],
            [ 0.4907, -1.3948, -1.0691, -0.3132],
            [-1.6092,  0.5419, -0.2993,  0.3195]])
    >>> torch.argmax(a, dim=1)
    tensor([ 0,  2,  0,  1])
Type:      builtin_function_or_method
```