Modeling in PyTorch

*The content from this deck is taken directly from PyTorch.org tutorial materials by B. Heintz and PyTorch Documentation

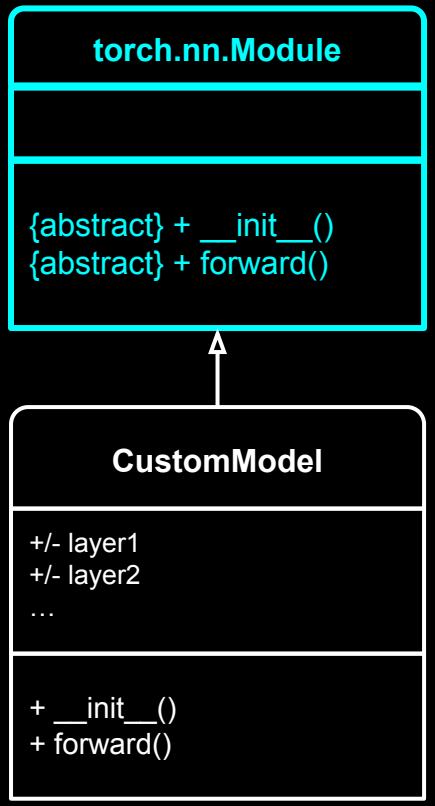
Overview

In this lecture we will cover:

- Defining the Model with PyTorch nn.Module and nn.Parameters
- Common Layer Types
  - linear layers
  - Convolutional layers
  - Recurrent
  - Transformer
- Other Layers and Functions
  - batch norming
  - dropout
  - activation functions
  - loss functiosn
- Training and validation setup with PyTorch Lightning

*The content from this deck is taken directly from PyTorch.org tutorial materials by B. Heintz and PyTorch Documentation

nn.Module and nn.Parameter



nn.Module
- base class that encapsulates models and model components like neural network layers

nn.Parameter
- subclass of torch.Tensor
- represents learning weights

nn.Module and nn.Parameter

```
class MLPerceptron(nn.Module):
def __init__(self):
  super(MLPerceptron, self).__init__()

  self.linear1 = torch.nn.Linear(100,200)
  self.activation = torch.nn.ReLU()
  self.linear2 = torch.nn.Linear(200,10)
  self.softmax = torch.nn.Softmax()


def forward(self, x:torch.Tensor):
  x = self.linear1(x)
  x = self.activation(x)
  x = self.linear2(x)
  x = self.softmax(x)
  return x

mlp= MLPerceptron()

print(“The model:”)
print(mlp)

print(“\n\nJust one layer:”)
print(mlp.linear2)

print(“\n\nModel params:’)
for param in mlp.parameters():
  print(param)

print(“\n\nLayer params:”)
for param in mlp.linear2.parameters():
  print(param)
```

nn.Module

- base class that encapsulates models and model components like neural network layers

nn.Parameter

- subclass of torch.Tensor
- represents learning weights

When a parameter is assigned as an attribute of a module the parameter object gets registered with that module.

If you register an instance of a module subclass as an attribute of a module, the contained module’s parameters are also registered as parameters of the owning class

nn.Module and nn.Parameter

```
class MLPerceptron(nn.Module):
def __init__(self):
  super(MLPerceptron, self).__init__()

  self.linear1 = torch.nn.Linear(100,200)
  self.activation = torch.nn.ReLU()
  self.linear2 = torch.nn.Linear(200,10)
  self.softmax = torch.nn.Softmax()


def forward(self, x:torch.Tensor):
  x = self.linear1(x)
  x = self.activation(x)
  x = self.linear2(x)
  x = self.softmax(x)
  return x

mlp= MLPerceptron()

print("The model:")
print(mlp)

print("\n\nJust one layer:")
print(mlp.linear2)

print("\n\nModel params:')
for param in mlp.parameters():
  print(param)

print("\n\nLayer params:")
for param in mlp.linear2.parameters():
  print(param)
```

`__init__`:

- defines the structure of the model the layers and functions that make it up

`forward`:

- method which composes those layers and functions into the computation

nn.Module and nn.Parameter

```
class MLPerceptron(nn.Module):
def __init__(self):
  super(MLPerceptron, self).__init__()

  self.linear1 = torch.nn.Linear(100,200)
  self.activation = torch.nn.ReLU()
  self.linear2 = torch.nn.Linear(200,10)
  self.softmax = torch.nn.Softmax()


def forward(self, x:torch.Tensor):
  x = self.linear1(x)
  x = self.activation(x)
  x = self.linear2(x)
  x = self.softmax(x)
  return x

mlp= MLPerceptron()

print("The model:")
print(mlp)

print("\n\nJust one layer:")
print(mlp.linear2)

print("\n\nModel params:')
for param in mlp.parameters():
  print(param)

print("\n\nLayer params:")
for param in mlp.linear2.parameters():
  print(param)
```

__init__:

- defines the structure of the model the layers and functions that make it up

forward:

- method which composes those layers and functions into the computation

When we create an instance of this model and print it we see that it not only knows its own layers and attributes they're assigned to but also the order in which we registered them. When we print out just one of the layers we get a description of just that layer

nn.Module and nn.Parameter

```
class MLPerceptron(nn.Module):
def __init__(self):
  super(MLPerceptron, self).__init__()

  self.linear1 = torch.nn.Linear(100,200)
  self.activation = torch.nn.ReLU()
  self.linear2 = torch.nn.Linear(200,10)
  self.softmax = torch.nn.Softmax()


def forward(self, x:torch.Tensor):
  x = self.linear1(x)
  x = self.activation(x)
  x = self.linear2(x)
  x = self.softmax(x)
  return x

mlp= MLPerceptron()

print("The model:")
print(mlp)

print("\n\nJust one layer:")
print(mlp.linear2)

print("\n\nModel params:')
for param in mlp.parameters():
  print(param)

print("\n\nLayer params:")
for param in mlp.linear2.parameters():
  print(param)
```

__init__:

- defines the structure of the model the layers and functions that make it up

forward:

- method which composes those layers and functions into the computation

When we create an instance of this model and print it we see that it not only knows its own layers and attributes they're assigned to but also the order in which we registered them. When we print out just one of the layers we get a description of just that layer

The model registers parameters of submodules it owns recursively. This is important because the model has to pass all of these parameters to the optimizer during training

PyTorch Layer Types

Classes encapsulate common layer types, so what are those classes?

Linear Layers

```
lin = torch.nn.Linear(3,2) # 3 element input -> 2 element output
x = torch.rand(1,3)

print(f “Input:\n{x}”)
print(“\n\nWeight and Bias parameters:”)
for param in lin.parameters():
  print(param)

y = lin(x)
print(“\n\nOutput:”)
print(y)
```

```
Input:
tensor([[0.2807, 0.5842, 0.7967]])

Weight and Bias parameters:
Parameter containing:
tensor([[-0.1719, 0.4691, -0.0654],
        [-0.2522, 0.5453, -0.5438]], requires_grad=True)
Parameter containing:
tensor([0.2956, 0.2001], requires_grad=True)

Output:
tensor([[0.4693, 0.0146, grad_fn=<AddmmBackward>)
```

The fully connected layer

- every input influences every output
- That influences the layer’s weights
- in a layer has m inputs and n outputs it’s weights will be an mxn matrix

When we print the parameters it lets us know that they require gradients, they are tracking computation history so we can compute gradients for learning

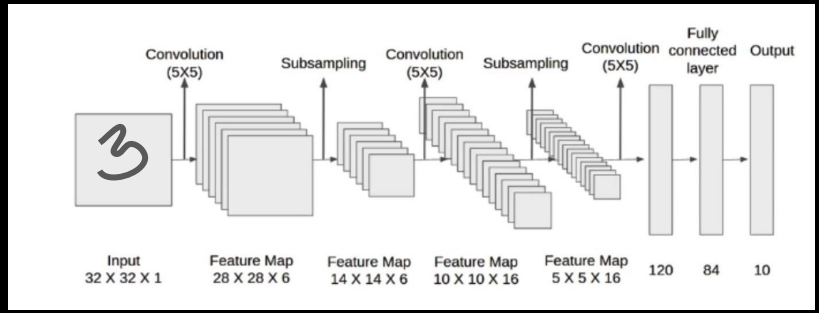
This behavior of setting auto_grad to true is different from what the tensor class does despite the parameter class being a subclass of torch.Tensor

Linear layers are common to classifiers, where they are placed in the last or last few layers

Convolutional Layers

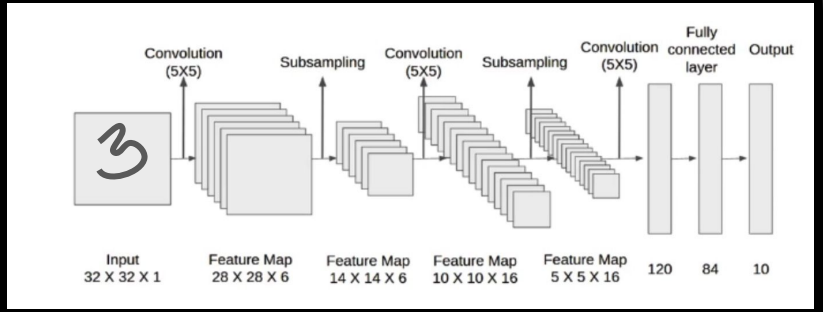
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_features *= s
  return num_features
```

Convolutional Layers address data that is strongly correlated in space and are common in computer vision and natural language processing where they can be used to detect close clusters of interesting features and compose them into larger features or recognized objects



Convolutional Layers

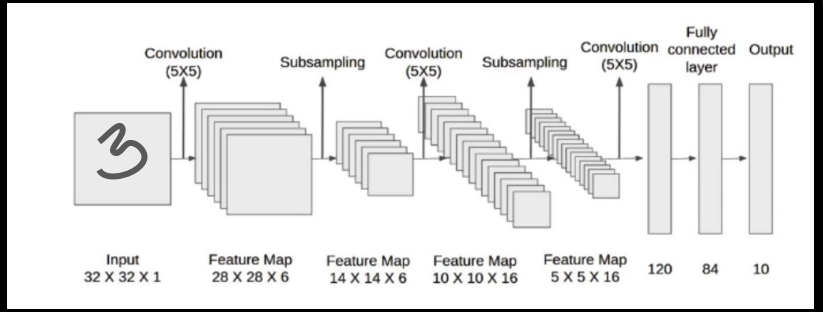
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x)),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x)), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_feaTurs *= s
  return num_features
```



number of channels

Convolutional Layers

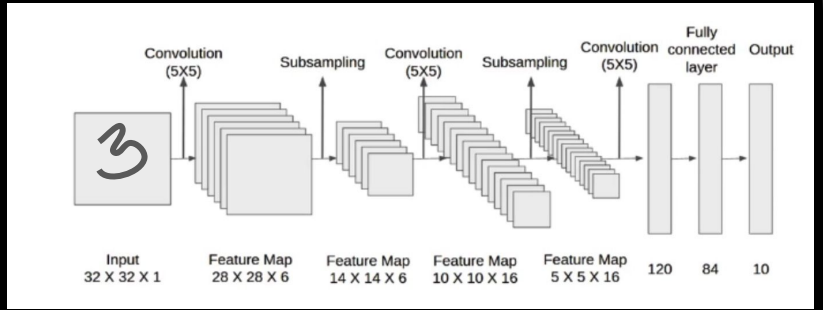
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 5)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_feaTurs *= s
  return num_features
```



number of features we want the layer to learn so it can recognize up to 6 different arrangements of pixels in the input

Convolutional Layers

```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x)),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x)), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_features *= s
  return num_features
```



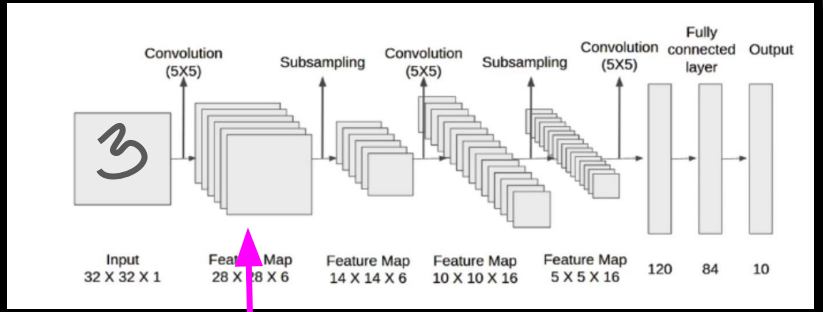
size of convolution kernel (sliding window)

Input: volume of size W_1 x H_1 x D_1

- Requires 4 hyperparameters
  - Number of filters **K**
  - The kernel size/spatial extent **F**
  - The stride **S**
  - The amount of zero padding **P**
- **Output:** volume of size W_2 x H_2 x D_2 where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$

Convolutional Layers

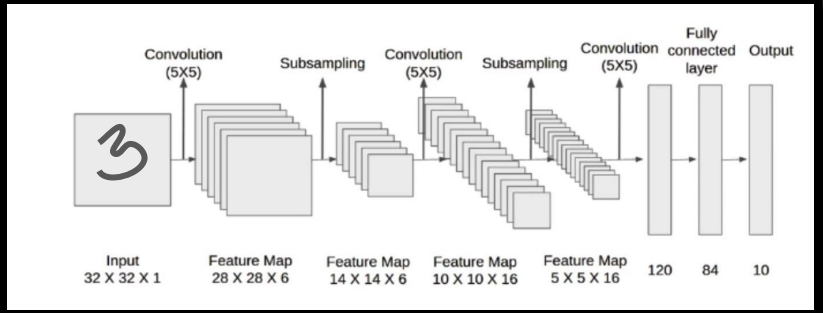
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_featurs *= s
  return num_features
```



The output of the Conv2d layer is an activation map that is a spatial map of where it found certain features

Convolutional Layers

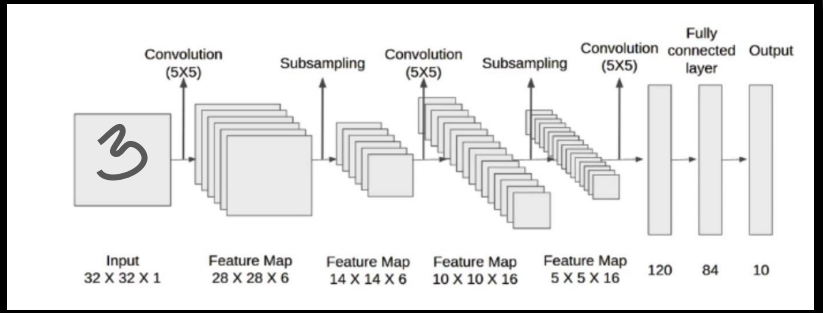
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 5)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_feaTurs *= s
  return num_features
```



The second convolutional layer (conv2) takes the first layer's output (conv1) as input (channels)

Convolutional Layers

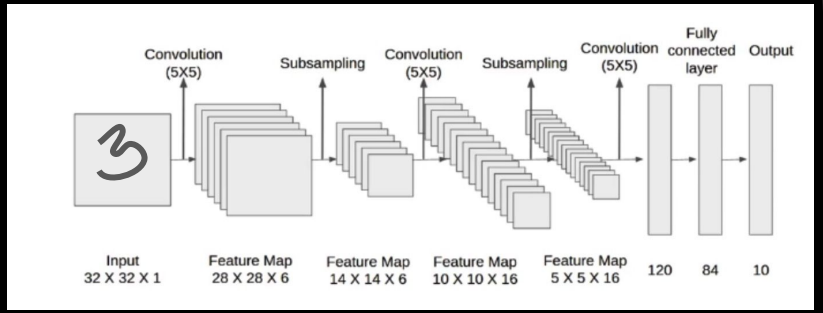
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_feaTurs *= s
  return num_features
```



We want the second layer to learn 16 different features which it makes by composing the features from the first layer

Convolutional Layers

```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_features *= s
  return num_features
```

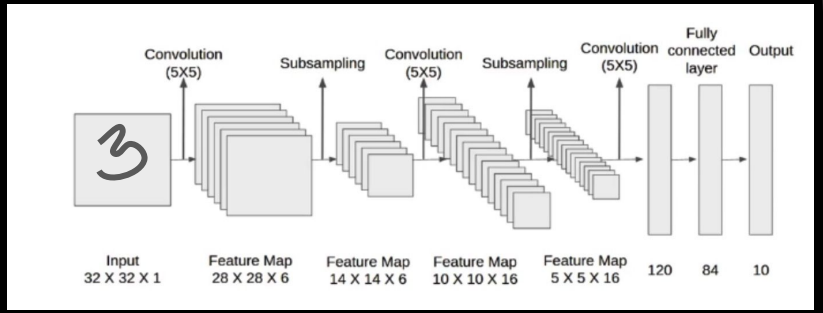


Using a 3x3 kernel

After the second convolution layer has composed its features into a higher level activation map we pass the output to a set of linear layers that act as a classifier with final layer having 10 outputs representing probabilities

Convolutional Layers

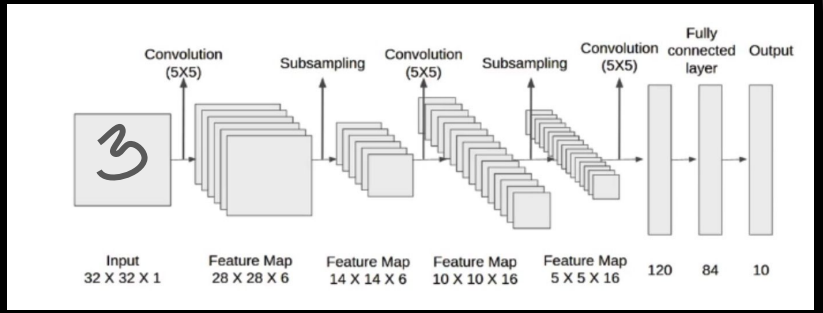
```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_featurs *= s
  return num_features
```



After the second convolution layer has composed its features into a higher level activation map we pass the output to a set of linear layers that act as a classifier with final layer having 10 outputs representing probabilities over the labels 0-9

Convolutional Layers

```
import torch.functional as F

class LeNet(nn.Module):
def __init__(self):
  super(LeNet, self).__init__()
  # 1 input image channel (black and white), 6 output
  # channels, 3x3 square kernel
  self.conv1 = torch.nn.Conv2d(1, 6, 5)
  self.conv2 = torch.nn.Conv2d(6, 16, 3)
  # an affine operation: y=Wx+b
  self.fc1 = torch.nn.Linear(16*6*6, 120) #6x6 image dimension
  self.fc2 = torch.nn.Linear(120, 84)
  self.fc3 = torch.nn.Linear(84, 10)


def forward(self, x:torch.Tensor):
  # Max pooling over a (2,2) window
  x = F.max_pool2d(F.relu(self.conv1(x),(2,2))
  # if the size is a square you can specify a single number
  x = F.max_pool2d(F.relu(self.conv2(x), 2)
  x = x.view(-1, self.num_flat_features(x))
  x = F.relu(self.fc1(x))
  x = F.relu(self.fc2(x))
  x = self.fc3(x)
  return x

def num_flat_features(self, x):
  size = x.size()[1:] # all dimensions except batch dimensions
  num_features = 1
  for s in size:
    num_featurs *= s
  return num_features
```



PyTorch has

- Conv1D
- Conv2D
- Conv3D

Additional Layers Available

Recurrent Layers: Used for Time Series and NLP

- LSTM
- GRU
- ConvGRU

Transformers: Common in NLP and Complex in Architecture

- Encoder
- Decoder

} Transformer has architectures built into it so it is not really a singular layer

Other Layers and Functions

Data Manipulation Layers

- Max pooling: reduce a tensor by combining cells and assigning the maximum value of the input cells to the output cell

```
my_tensor = torch.rand(1,6,6)
print(my_tensor)

maxpool_layer = torch.nn.MaxPool2D(3)
print(maxpool_layer(my_tensor))

my_tensor = torch.rand(1,6,6)
print(my_tensor)

maxpool_layer = torch.nn.MaxPool2D(3)
print(maxpool_layer(my_tensor))
```

```
tensor([[[0.8160, 0.1406, 0.5950, 0.0883, 0.5464, 0.3993],
         [0.0623, 0.6626, 0.3991, 0.4878, 0.7548, 0.2426],
         [0.9081, 0.4207, 0.8590, 0.3784, 0.6931, 0.5609],
         [0.6182, 0.8588, 0.3766, 0.9734, 0.9662, 0.9880],
         [0.0599, 0.8338, 0.6750, 0.0829, 0.3554, 0.3998],
         [0.6159, 0.7129, 0.8945, 0.8717, 0.9930, 0.9059]]])
tensor([[[0.9081, 0.7548],
         [0.8945, 0.9930]]])
```

Input: volume of size $W_1 \times H_1 \times D_1$

- Requires two hyperparameters:
  - The spatial extent/kernel size F
  - The stride S
- Produces a volume of size $W_2 \times H_2 \times D_2$
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$

Other Layers and Functions

Data Manipulation Layers

- Normalization layers: re-center and normalize the output of one layer before feeding it to another. Center and scaling the intermediate tensors can allow you to use higher learning rates without exploding/vanishing gradients.

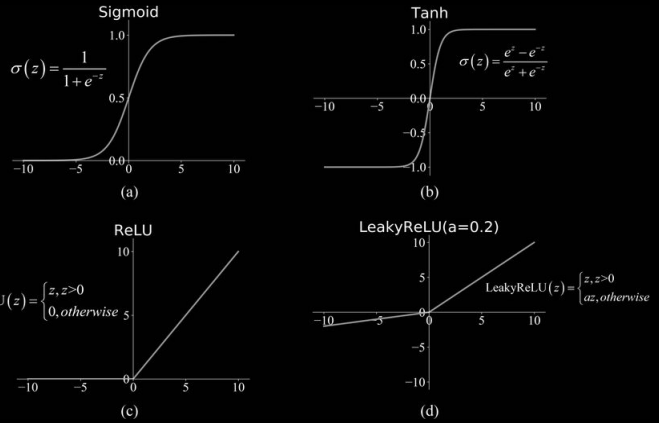
```
my_tensor = torch.rand(1, 4, 4) * 20 + 5
print(my_tensor)

print(my_tensor.mean())

norm_layer = torch.nn.BatchNorm1d(4)
normed_tensor = norm_layer(my_tensor)
print(normed_tensor)

print(normed_tensor.mean())
```

```
tensor([[[18.0634,  5.6720,  5.7805, 12.3243],
         [ 9.3712, 19.7366,  6.4853, 22.8629],
         [14.6223, 21.5803, 17.8267, 20.3997],
         [21.7664,  5.0936, 19.5952, 11.8554]]])
tensor(14.5647)
tensor([[[ 1.4762, -0.9296, -0.9086,  0.3619],
         [-0.7650,  0.7475, -1.1862,  1.2037],
         [-1.4918,  1.1130, -0.2922,  0.6710],
         [ 1.0893, -1.4371,  0.7603, -0.4125]]],
       grad_fn=<NativeBatchNormBackward>)
tensor(1.3039e-08, grad_fn=<MeanBackward0>)
```

smaller and grouped around zero!

Why is centering beneficial?

- Activation functions have their strongest gradients newar 0 but can suffer from vanishing or exploding gradients for inputs that drive them from zero.
- Keeping the data centered around the are of the steepest gradient will tend to mean faster, better learning and higher learning rates



Other Layers and Functions

Data Manipulation Layers

- Dropout layers:
  - Encourages sparse representations in model by pushing it to infer on less data.
  - Randomly sets parts of the input tensor during training to zero forcing the model to learn against a masked dataset

```
my_tensor = torch.rand(1, 4, 4)

dropout = torch.nn.Dropout(p=0.4)
print(dropout(my_tensor))
print(dropout(my_tensor))
```

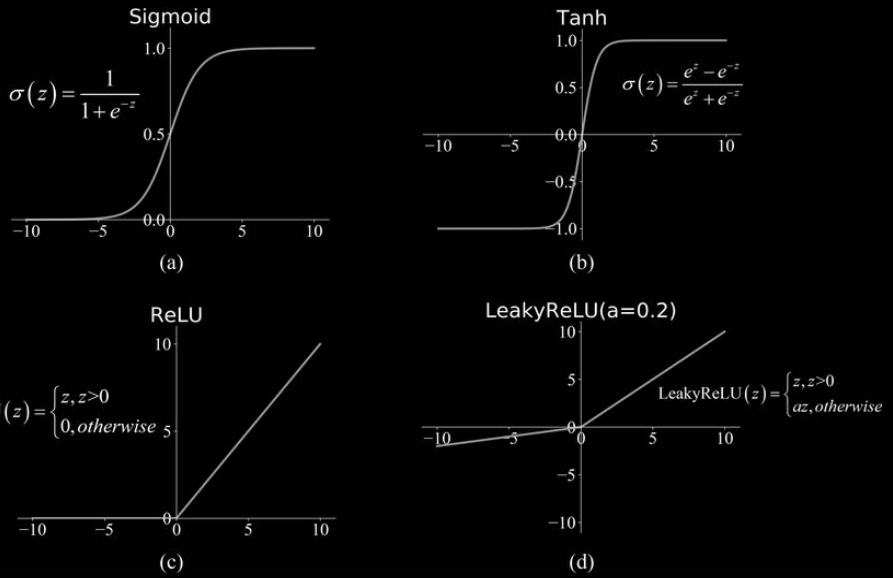
```
tensor([[[0.0000, 1.1702, 0.5911, 0.0000],
         [0.1932, 1.4928, 1.2912, 0.0000],
         [0.1236, 1.3672, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000]]])
tensor([[[1.5033, 1.1702, 0.5911, 0.9341],
         [0.0000, 0.0000, 0.0000, 1.5020],
         [0.1236, 1.3672, 0.0000, 0.0000],
         [0.4993, 0.9576, 0.0000, 1.6664]]])
```

- **p**: sets the probability of an individual weight randomly dropping out
- default **p**: is 50%

Activation Functions

A neural network is a program that simulates mathematical functions. Matrix multiplication can only result in linear functions which are trivial to represent.

non-linear activation functions between layers allows deep learning models to simulate any function rather than just linear ones because they enable the network to learn, complex nonlinear relationships within the data making the model more expressive.



PyTorch Model Training

The Dataset and the DataLoader

```
import torch
import torchvision
import torchvision.transforms as transforms

from datetime import datetime


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

# Create datasets for training & validation, download if
necessary
training_set = torchvision.datasets.FashionMNIST('./data',
train=True, transform=transform, download=True)
validation_set = torchvision.datasets.FashionMNIST('./data',
train=False, transform=transform, download=True)

# Create data loaders for our datasets; shuffle for training, not
for validation
training_loader = torch.utils.data.DataLoader(training_set,
batch_size=4, shuffle=True, num_workers=2)
validation_loader = torch.utils.data.DataLoader(validation_set,
batch_size=4, shuffle=False, num_workers=2)

# Class labels
classes = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot')

# Report split sizes
print('Training set has {} instances'.format(len(training_set)))
print('Validation set has {}
instances'.format(len(validation_set)))
```

```
Training set has 60000 instances
Validation set has 10000 instances
```

- The Dataset and DataLoader classes encapsulate the process of pulling data from storage and exposing it to your training loop in batches
- The Dataset is responsible for accessing and processing single instances of data.
- The DataLoader pulls instances of data from the Dataset (either automatically or with a sampler that you define), collects them in batches, and returns them for consumption by your training loop.

The Dataset and the DataLoader

```
import torch
import torchvision
import torchvision.transforms as transforms

from datetime import datetime


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

# Create datasets for training & validation, download if
necessary
training_set = torchvision.datasets.FashionMNIST('./data',
train=True, transform=transform, download=True)
validation_set = torchvision.datasets.FashionMNIST('./data',
train=False, transform=transform, download=True)

# Create data loaders for our datasets; shuffle for training, not
for validation
training_loader = torch.utils.data.DataLoader(training_set,
batch_size=4, shuffle=True, num_workers=2)
validation_loader = torch.utils.data.DataLoader(validation_set,
batch_size=4, shuffle=False, num_workers=2)

# Class labels
classes = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle Boot')

# Report split sizes
print('Training set has {} instances'.format(len(training_set)))
print('Validation set has {}
instances'.format(len(validation_set)))
```

```
Training set has 60000 instances
Validation set has 10000 instances
```

```
import matplotlib.pyplot as plt
import numpy as np

# Helper function for inline image display
def matplotlib_imshow(img, one_channel=False):
    if one_channel:
        img = img.mean(dim=0)
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    if one_channel:
        plt.imshow(npimg, cmap="Greys")
    else:
        plt.imshow(np.transpose(npimg, (1, 2, 0)))

for i, data in enumerate(training_loader):
    images, labels = data
    break

# Create a grid from the images and show them
img_grid = torchvision.utils.make_grid(images)
matplotlib_imshow(img_grid, one_channel=True)
print('  '.join(classes[labels[j]] for j in range(4)))
```

Sandal Shirt Shirt Sandal

FashionMNIST has 10 classes of clothing

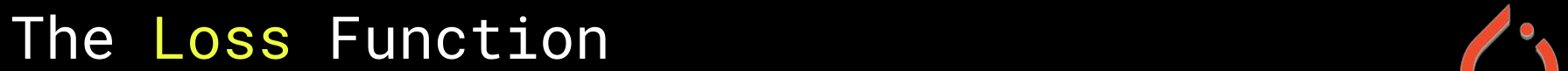
The Model


```
import torch.nn as nn
import torch.nn.functional as F

# PyTorch models inherit from torch.nn.Module
class GarmentClassifier(nn.Module):
    def __init__(self):
        super(GarmentClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


model = GarmentClassifier()
```

Sandal Shirt Shirt Sandal

- We would like to classify a garment based on its image.
- Which architecture does GarmentClassifier use?

The Loss Function



```
loss_fn = torch.nn.CrossEntropyLoss()

# NB: Loss functions expect data in batches, so we're creating
batches of 4
# Represents the model's confidence in each of the 10 classes for
a given input
dummy_outputs = torch.rand(4, 10)
# Represents the correct class among the 10 being tested
dummy_labels = torch.tensor([1, 5, 3, 7])

print(dummy_outputs)
print(dummy_labels)

loss = loss_fn(dummy_outputs, dummy_labels)
print('Total loss for this batch: {}'.format(loss.item()))
```

```
tensor([[0.7915, 0.4766, 0.3735, 0.5340, 0.0799, 0.9948, 0.1870,
0.0507, 0.1183,
         0.9106],
        [0.9666, 0.3765, 0.4324, 0.7354, 0.1953, 0.8906, 0.6882,
0.1925, 0.7076,
         0.8777],
        [0.4412, 0.0325, 0.4886, 0.9350, 0.9792, 0.5580, 0.6199,
0.2478, 0.3619,
         0.8307],
        [0.3287, 0.8571, 0.6046, 0.6719, 0.5982, 0.0540, 0.7193,
0.4764, 0.7451,
         0.8345]])
tensor([1, 5, 3, 7])
Total loss for this batch: 2.196722984313965
```

Sandal Shirt Shirt Sandal

- We will use the Cross-Entropy Loss in this example.

The Optimizer

```
# Optimizers specified in the torch.optim package
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

- We will be using stochastic gradient descent (SGD) with momentum.
- Parameters we can select:
  - lr: the learning rate determines the size of the steps the optimizer takes. What does a different learning rate do to your training results, in terms of accuracy and convergence time?
  - momentum: nudges the optimizer in the direction of the strongest gradient over multiple steps. What does changing this value do to your results?
  - There are also different optimizers:
    - averaged SGD
    - Adagrad
    - Adam

The Training Loop

To keep our code manageable we will do it in steps.

```
def train_one_epoch(epoch_index):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            wandb.log({"Loss/train": last_loss},\
                      step=epoch_index * len(training_loader) + i + 1)
            running_loss = 0.

    return last_loss
```

- Get a batch of data from the DataLoader
- Zero the optimizer's gradients
- Perform "inference" to get predictions from teh model for an input batch
- Calculate the loss for that set of predictions vs the ground truth labels on the dataset
- Calculate the backward gradients over the learning weights
- Tell the optimizer to perform one learning step
    - adjust the model's learning weights based on the observed gradients for the current batch
- Report loss every 1000 batches
- Report the average per-batch loss for the last 1000 batches, for comparison

It is a good idea to modularize your code with PyTorch wrappers like PyTorch Lightning that prevent your main function to get too long and can take advantage of multiple GPU/TPUs.

We will first show how to do the training loop in vanilla PyTorch and clean it up with Lightning later

Per-Epoch Activity

```
import wandb

timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
wandb.init(project="your_project_name", name='fashion_trainer_{}'.format(timestamp))
epoch_number = 0

EPOCHS = 5

best_vloss = 1_000_000.

for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch_number + 1))

    model.train(True)
    avg_loss = train_one_epoch(epoch_number)

    model.train(False)
    running_vloss = 0.0
    for i, vdata in enumerate(validation_loader):
        vinputs, vlabels = vdata
        voutputs = model(vinputs)
        vloss = loss_fn(voutputs, vlabels)
        running_vloss += vloss

    avg_vloss = running_vloss / (i + 1)
    print('LOSS train {} valid {}'.format(avg_loss, avg_vloss))

    wandb.log({"Training vs. Validation Loss": {"Training": avg_loss, "Validation":
avg_vloss}}, step=epoch_number + 1)

    if avg_vloss < best_vloss:
        best_vloss = avg_vloss
        model_path = 'model_{}_{}'.format(timestamp, epoch_number)
        torch.save(model.state_dict(), model_path)

    epoch_number += 1
```

We would like to:

- Perform validation by checking our relative loss on a set of data that was not used for training and report it.
- Save a copy of the model
- Use Weights and Biases to log our losses

Per-Epoch Activity

```
import wandb

timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
wandb.init(project="your_project_name", name='fashion_trainer_{}'.format(timestamp))
epoch_number = 0

EPOCHS = 5

best_vloss = 1_000_000.

for epoch in range(EPOCHS):
    print('EPOCH {}:'.format(epoch_number + 1))

    model.train(True)
    avg_loss = train_one_epoch(epoch_number)

    model.train(False)
    running_vloss = 0.0
    for i, vdata in enumerate(validation_loader):
        vinputs, vlabels = vdata
        voutputs = model(vinputs)
        vloss = loss_fn(voutputs, vlabels)
        running_vloss += vloss

    avg_vloss = running_vloss / (i + 1)
    print('LOSS train {} valid {}'.format(avg_loss, avg_vloss))

    wandb.log({"Training vs. Validation Loss": {"Training": avg_loss, "Validation":
avg_vloss}}, step=epoch_number + 1)

    if avg_vloss < best_vloss:
        best_vloss = avg_vloss
        model_path = 'model_{}_{}'.format(timestamp, epoch_number)
        torch.save(model.state_dict(), model_path)

    epoch_number += 1
```

```
EPOCH 6:
  batch 1000 loss: 0.2712569512435075
  batch 2000 loss: 0.2601859306513652
  batch 3000 loss: 0.259985514376438
  batch 4000 loss: 0.2620664734886377
  batch 5000 loss: 0.25726098109555096
  batch 6000 loss: 0.2829849383759356
  batch 7000 loss: 0.2504337926213739
  batch 8000 loss: 0.28633546594417153
  batch 9000 loss: 0.2615796004622789
  batch 10000 loss: 0.27075869734541264
  batch 11000 loss: 0.26998766335008256
  batch 12000 loss: 0.2614513303764511
  batch 13000 loss: 0.25101113697645633
  batch 14000 loss: 0.2572545314691452
  batch 15000 loss: 0.2646196218387686
LOSS train 0.2646196218387686 valid 0.3120253086090088
EPOCH 7:
  batch 1000 loss: 0.23217408991576668
  batch 2000 loss: 0.26750317257382084
  batch 3000 loss: 0.2423647686961576
  batch 4000 loss: 0.247761928711403
  batch 5000 loss: 0.2550650118602407
  batch 6000 loss: 0.2383200812106561
  batch 7000 loss: 0.24519096856425546
  batch 8000 loss: 0.26013731523246864
  batch 9000 loss: 0.2632041203577719
  batch 10000 loss: 0.2694681866055953
  batch 11000 loss: 0.2497381271280392
  batch 12000 loss: 0.258198305114337
  batch 13000 loss: 0.25165209144220446
  batch 14000 loss: 0.24501102960605475
  batch 15000 loss: 0.2714995371173936
LOSS train 0.2714995371173936 valid 0.29794615507125854
EPOCH 8:
  batch 1000 loss: 0.23320657860705432
  batch 2000 loss: 0.24858402526881765
...
```

Loading the Saved Model

To load the model:

```
saved_model = GarmentClassifier()

saved_model.load_state_dict(torch.load(PATH))
```

If your model has constructor parameters that affect hte model structure, please make sure you provide them to configure the model identically to the parameters it had right before you saved it.