Pytorch

**Creating tensors in PyTorch**

Random tensors are very important in neural networks. Parameters of the neural networks typically are initialized with random weights (random tensors).

Let us start practicing building tensors in PyTorch library. As you know, tensors are arrays with an arbitrary number of dimensions, corresponding to NumPy's ndarrays. You are going to create a random tensor of sizes 3 by 3 and set it to variable your\_first\_tensor. Then, you will need to print it. Finally, calculate its size in variable tensor\_size and print its value.

*NB: In case you have trouble solving the problems, you can always refer to slides in the bottom right of the screen.*

**Instructions**

**100 XP**

Import PyTorch main library.

Create the variable your\_first\_tensor and set it to a random torch tensor of size 3 by 3.

Calculate its shape (dimension sizes) and set it to variable tensor\_size.

Print the values of your\_first\_tensor and tensor\_size.

# Import torch

import torch

# Create random tensor of size 3 by 3

your\_first\_tensor = torch.rand(3, 3)

# Calculate the shape of the tensor

tensor\_size = your\_first\_tensor.size()

# Print the values of the tensor and its shape

print(your\_first\_tensor)

print(tensor\_size)

**Matrix multiplication**

There are many important types of matrices which have their uses in neural networks. Some important matrices are matrices of ones (where each entry is set to 1) and the identity matrix (where the diagonal is set to 1 while all other values are 0). The identity matrix is very important in linear algebra: any matrix multiplied with identity matrix is simply the original matrix.

Let us experiment with these two types of matrices. You are going to build a matrix of ones with shape 3 by 3 called tensor\_of\_ones and an identity matrix of the same shape, called identity\_tensor. We are going to see what happens when we multiply these two matrices, and what happens if we do an element-wise multiplication of them.

**Instructions**

**100 XP**

Create a matrix of ones with shape 3 by 3, and put it on variable tensor\_of\_ones.

Create an identity matrix with shape 3 by 3, and put it on variable identity\_tensor.

Do a matrix multiplication of tensor\_of\_ones with identity\_tensor and print its value.

Do an element-wise multiplication of tensor\_of\_ones with identity\_tensor and print its value.

# Create a matrix of ones with shape 3 by 3

tensor\_of\_ones = torch.ones(3, 3)

# Create an identity matrix with shape 3 by 3

identity\_tensor = torch.eye(3)

# Do a matrix multiplication of tensor\_of\_ones with identity\_tensor

matrices\_multiplied = torch.matmul(tensor\_of\_ones, identity\_tensor)

print(matrices\_multiplied)

# Do an element-wise multiplication of tensor\_of\_ones with identity\_tensor

element\_multiplication = tensor\_of\_ones \* identity\_tensor

print(element\_multiplication)

**Forward pass**

Let's have something resembling more a neural network. The computational graph has been given below. You are going to initialize 3 large random tensors, and then do the operations as given in the computational graph. The final operation is the mean of the tensor, given by torch.mean(your\_tensor).

Diagram

Description automatically generated

**Instructions**

**100 XP**

**Instructions**

**100 XP**

Initialize random tensors x, y and z, each having shape (1000, 1000).

Multiply x with y, putting the result in tensor q.

Do an elementwise multiplication of tensor z with tensor q, putting the results in f

# Backpropagation by hand

Diagram

Description automatically generated

Given the computational graph above, we want to calculate the derivatives for the leaf nodes (x, y and z). To get you started we already calculated the results of the forward pass (in red) in addition to calculating the derivatives of f and q.

The rules for derivative computations have been given in the table below:

| **Interaction** | **Overall Change** |
| --- | --- |
| Addition | (f+g)′=f′+g′ |
| Multiplication | (f⋅g)′=f⋅dg+g⋅df |
| Powers | (xn)′=ddxxn=nxn−1 |
| Inverse | (1x)′=−1x2 |
| Division | (fg)′=(df⋅1g)+(−1g2dg⋅f) |

##### Answer the question

**50XP**

#### Possible Answers



The Derivative of x is 5, the derivative of y is 5, the derivative of z is 1.

press1



The Derivative of x is 5, the derivative of y is 5, the derivative of z is 5.

press2



The Derivative of x is 8, the derivative of y is -3, the derivative of z is 0.

press3



Derivatives are lame, integrals are cool.

**Backpropagation using PyTorch**

Here, you are going to use automatic differentiation of PyTorch in order to compute the derivatives of x, y and z from the previous exercise.

**Instructions**

**100 XP**

Initialize tensors x, y and z to values 4, -3 and 5.

Put the sum of tensors x and y in q, put the product of q and z in f.

Calculate the derivatives of the computational graph.

Print the gradients of the x, y and z tensors.

# Initialize x, y and z to values 4, -3 and 5

x = torch.tensor(4., requires\_grad=True)

y = torch.tensor(-3., requires\_grad=True)

z = torch.tensor(5., requires\_grad=True)

# Set q to sum of x and y, set f to product of q with z

q = x+y

f = q\*z

# Compute the derivatives

f.backward()

# Print the gradients

print("Gradient of x is: " + str(x.grad))

print("Gradient of y is: " + str(y.grad))

print("Gradient of z is: " + str(z.grad))

**Calculating gradients in PyTorch**

Remember the exercise in forward pass? Now that you know how to calculate derivatives, let's make a step forward and start calculating the gradients (derivatives of tensors) of the computational graph you built back then. We have already initialized for you three random tensors of shape (1000, 1000) called x, y and z. First, we multiply tensors x and y, then we do an elementwise multiplication of their product with tensor z, and then we compute its mean. In the end, we compute the derivatives.

The main difference from the previous exercise is the scale of the tensors. While before, tensors x, y and z had just 1 number, now they each have 1 million numbers.

Diagram

Description automatically generated

Multiply tensors x and y, put the product in tensor q.

Do an elementwise multiplication of tensors z with q.

Calculate the gradients.

# Multiply tensors x and y

q = torch.matmul(x,y)

# Elementwise multiply tensors z with q

f = z\*q

mean\_f = torch.mean(f)

# Calculate the gradients

mean\_f.backward()

**Your first neural network**

You are going to build a neural network in PyTorch, using the hard way. Your input will be images of size (28, 28), so images containing 784 pixels. Your network will contain an input\_layer (provided for you), a hidden layer with 200 units, and an output layer with 10 classes. The input layer has already been created for you. You are going to create the weights, and then do matrix multiplications, getting the results from the network.

Initialize with random numbers two matrices of weights, called weight\_1 and weight\_2.

Set the result of input\_layer times weight\_1 to hidden\_1. Set the result of hidden\_1 times weight\_2 to output\_layer.

# Initialize the weights of the neural network

weight\_1 = torch.rand(784, 200)

weight\_2 = torch.rand(200, 10)

# Multiply input\_layer with weight\_1

hidden\_1 = torch.matmul(input\_layer, weight\_1)

# Multiply hidden\_1 with weight\_2

output\_layer = torch.matmul(hidden\_1, weight\_2)

print(output\_layer)

**Your first PyTorch neural network**

You are going to build the same neural network you built in the previous exercise, but now using the PyTorch way. As a reminder, you have 784 units in the input layer, 200 hidden units and 10 units for the output layer.

Instantiate two linear layers calling them self.fc1 and self.fc2. Determine their correct dimensions. Implement the .forward() method, using the two layers you defined and returning x.

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        # Instantiate all 2 linear layers

        self.fc1 = nn.Linear(784, 200)

        self.fc2 = nn.Linear(200,10)

    def forward(self, x):

        # Use the instantiated layers and return x

        x = self.fc1(x)

        x = self.fc2(x)

        return x

**Neural networks**

Let us see the differences between neural networks which apply ReLU and those which do not apply ReLU. We have already initialized the input called input\_layer, and three sets of weights, called weight\_1, weight\_2 and weight\_3.

We are going to convince ourselves that networks with multiple layers which do not contain non-linearity can be expressed as neural networks with one layer.

The network and the shape of layers and weights is shown below.

Diagram

Description automatically generated

**Instructions**

**0 XP**

* Calculate the first and second hidden layer by multiplying the appropriate inputs with the corresponding weights.
* Calculate and print the results of the output.
* Set weight\_composed\_1 to the product of weight\_1 with weight\_2, then set weight to the product of weight\_composed\_1 with weight\_3.
* Calculate and print the output.
* # Calculate the first and second hidden layer
* hidden\_1 = torch.matmul(input\_layer, weight\_1)
* hidden\_2 = torch.matmul(hidden\_1, weight\_2)
* # Calculate the output
* print(torch.matmul(hidden\_2, weight\_3))
* # Calculate weight\_composed\_1 and weight
* weight\_composed\_1 = torch.matmul(weight\_1, weight\_2)
* weight = torch.matmul(weight\_composed\_1, weight\_3)
* # Multiply input\_layer with weight
* print(torch.matmul(input\_layer, weight))

**ReLU activation**

In this exercise, we have the same settings as the previous exercise. But now we are going to build a neural network which has non-linearity. By doing so, we are going to convince ourselves that networks with multiple layers and non-linearity functions cannot be expressed as a neural network with one layer.

Diagram

Description automatically generated

*We have already instantiated the ReLU activation function called relu() for you.*

**Instructions**

**100 XP**

* Apply the non-linearity to the two hidden layers and print the result.
* Apply the non-linearity to the product of first two weights.
* Multiply the result of the previous step with weight\_3.
* Multiply input\_layer with weight and print the results.
* # Instantiate non-linearity
* relu = nn.ReLU()
* # Apply non-linearity on the hidden layers
* hidden\_1\_activated = relu(torch.matmul(input\_layer, weight\_1))
* hidden\_2\_activated = relu(torch.matmul(hidden\_1\_activated, weight\_2))
* print(torch.matmul(hidden\_2\_activated, weight\_3))
* # Apply non-linearity in the product of first two weights.
* weight\_composed\_1\_activated = relu(torch.matmul(weight\_1, weight\_2))
* # Multiply `weight\_composed\_1\_activated` with `weight\_3
* weight = torch.matmul(weight\_composed\_1\_activated, weight\_3)
* # Multiply input\_layer with weight
* print(torch.matmul(input\_layer, weight))

**ReLU activation again**

Neural networks don't need to have the same number of units in each layer. Here, you are going to experiment with the ReLU activation function again, but this time we are going to have a different number of units in the layers of the neural network. The input layer will still have 4 features, but then the first hidden layer will have 6 units and the output layer will have 2 units.

Diagram

Description automatically generated

**Instructions**

**100 XP**

* Instantiate the ReLU() activation function as relu (the function is part of nn module).
* Initialize weight\_1 and weight\_2 with random numbers.
* Multiply the input\_layer with weight\_1, storing results in hidden\_1.
* Apply the relu activation function over hidden\_1, and then multiply the output of it with weight\_2.
* # Instantiate ReLU activation function as relu
* relu = nn.ReLU()
* # Initialize weight\_1 and weight\_2 with random numbers
* weight\_1 = torch.rand(4, 6)
* weight\_2 = torch.rand(6, 2)
* # Multiply input\_layer with weight\_1
* hidden\_1 = torch.matmul(input\_layer, weight\_1)
* # Apply ReLU activation function over hidden\_1 and multiply with weight\_2
* hidden\_1\_activated = relu(hidden\_1)
* print(torch.matmul(hidden\_1\_activated, weight\_2))

# Calculating loss function by hand

Let's start the exercises by calculating the loss function by hand. Don't do this exercise in PyTorch, it is important to first do it using only pen and paper (and a calculator).

We have the same example as before but now our object is actually a frog, and the predicted scores are -1.2 for class 0 (cat), 0.12 for class 1 (car) and 4.8 for class 2 (frog).

What is the result of the softmax cross-entropy loss function?

| **Class** | **Predicted Score** |
| --- | --- |
| Cat | -1.2 |
| Car | 0.12 |
| Frog | 4.8 |

#### Possible Answers

* 6.0117
* 4.6917
* 0.0117-ans

Score for frog is high, so loss is 0.

**Calculating loss function in PyTorch**

You are going to code the previous exercise, and make sure that we computed the loss correctly. Predicted scores are -1.2 for class 0 (cat), 0.12 for class 1 (car) and 4.8 for class 2 (frog). The ground truth is class 2 (frog). Compute the loss function in PyTorch.

| **Class** | **Predicted Score** |
| --- | --- |
| Cat | -1.2 |
| Car | 0.12 |
| Frog | 4.8 |

**Instructions**

**100 XP**

* Initialize the tensor of scores with numbers [[-1.2, 0.12, 4.8]], and the tensor of ground truth [2].
* Instantiate the cross-entropy loss and call it criterion.
* Compute and print the loss.
* # Initialize the scores and ground truth
* logits = torch.tensor([[-1.2, 0.12, 4.8]])
* ground\_truth = torch.tensor([2])
* # Instantiate cross entropy loss
* criterion = nn.CrossEntropyLoss()
* # Compute and print the loss
* loss = criterion(logits,ground\_truth)
* print(loss)

**Loss function of random scores**

If the neural network predicts random scores, what would be its loss function? Let's find it out in PyTorch. The neural network is going to have 1000 classes, each having a random score. For ground truth, it will have class 111. Calculate the loss function.

**Instructions**

**100 XP**

* Import torch and torch.nn as nn
* Initialize logits with a random tensor of shape (1, 1000) and ground\_truth with a tensor containing the number 111.
* Instantiate the cross-entropy loss in a variable called criterion.
* Calculate and print the loss function.
* # Import torch and torch.nn
* import torch
* import torch.nn as nn
* # Initialize logits and ground truth
* logits = torch.rand(1,1000)
* ground\_truth = torch.tensor([111])
* # Instantiate cross-entropy loss
* criterion=nn.CrossEntropyLoss()
* # Calculate and print the loss
* loss = criterion(logits,ground\_truth)
* print(loss)

**Preparing MNIST dataset**

You are going to prepare dataloaders for MNIST training and testing set. As we explained in the lecture, MNIST has some differences to CIFAR-10, with the main difference being that MNIST images are grayscale (1 channel based) instead of RGB (3 channels).

**Instructions**

**0 XP**

* Transform the data to torch tensors and normalize it to have mean is 0.1307 and std is 0.3081.
* Prepare the trainset and the testset.
* Prepare the dataloaders for training and testing so that only 32 pictures are processed at a time and the training data is shuffled each time.
* # Transform the data to torch tensors and normalize it
* transform = transforms.Compose([transforms.ToTensor(),
* transforms.Normalize((0.1307), ((0.3081)))])
* # Prepare training set and testing set
* trainset = torchvision.datasets.MNIST('mnist', train=True,
* download=True, transform=transform)
* testset = torchvision.datasets.MNIST('mnist', train=False,
* download=True, transform=transform)
* # Prepare training loader and testing loader
* trainloader = torch.utils.data.DataLoader(trainset, batch\_size=32,
* shuffle=True, num\_workers=0)
* testloader = torch.utils.data.DataLoader(testset, batch\_size=32,
* shuffle=False, num\_workers=0)

**Inspecting the dataloaders**

Now you are going to explore a bit the dataloaders you created in the previous exercise. In particular, you will compute the shape of the dataset in addition to the minibatch size.

**Instructions**

**100 XP**

* Find the shapes of the trainset and testset.
* Print the computed values.
* Find the size of the minibatch for both trainset and testset.
* Print the minibatch size.
* # Compute the shape of the training set and testing set
* trainset\_shape = trainloader.dataset.train\_data.shape
* testset\_shape = testloader.dataset.test\_data.shape
* # Print the computed shapes
* print(trainset\_shape, testset\_shape)
* # Compute the size of the minibatch for training set and testing set
* trainset\_batchsize = trainloader.batch\_size
* testset\_batchsize = testloader.batch\_size
* # Print sizes of the minibatch
* print(trainset\_batchsize, testset\_batchsize)
* <script.py> output:
* torch.Size([60000, 28, 28]) torch.Size([10000, 28, 28])
* 32 32

**Building a neural network - again**

You haven't created a neural network since the end of the first chapter, so this is a good time to build one (practice makes perfect). Build a class for a neural network which will be used to train on the MNIST dataset. The dataset contains images of shape (28, 28, 1), so you should deduct the size of the input layer. For hidden layer use 200 units, while for output layer use 10 units (1 for each class). For activation function, use relu in a functional way (nn.Functional is already imported as F).

For context, the same net will be trained and used to make predictions in the next two exercises.

**Instructions**

**0 XP**

* Define the class called Net which inherits from nn.Module.
* In the \_\_init\_\_() method, define the parameters for the two fully connected layers.
* In the .forward() method, do the forward step.
* # Define the class Net
* class Net(nn.Module):
* def \_\_init\_\_(self):
* # Define all the parameters of the net
* super(Net, self).\_\_init\_\_()
* self.fc1 = nn.Linear(28 \* 28 \* 1, 200)
* self.fc2 = nn.Linear(200, 10)
* def forward(self, x):
* # Do the forward pass
* x = F.relu(self.fc1(x))
* x = self.fc2(x)
* return x

**Training a neural network**

Given the fully connected neural network (called model) which you built in the previous exercise and a train loader called train\_loader containing the MNIST dataset (which we created for you), you're to train the net in order to predict the classes of digits. You will use the Adam optimizer to optimize the network, and considering that this is a classification problem you are going to use cross entropy as loss function.

**Instructions**

**100 XP**

* Instantiate the Adam optimizer with learning rate 3e-4 and instantiate Cross-Entropy as loss function.
* Complete a forward pass on the neural network using the input data.
* Using backpropagation, compute the gradients of the weights, and then change the weights using the Adam optimizer.
* # Instantiate the Adam optimizer and Cross-Entropy loss function
* model = Net()
* optimizer = optim.Adam(model.parameters(), lr=3e-4)
* criterion = nn.CrossEntropyLoss()
* for batch\_idx, data\_target in enumerate(train\_loader):
* data = data\_target[0]
* target = data\_target[1]
* data = data.view(-1, 28 \* 28)
* optimizer.zero\_grad()
* # Complete a forward pass
* output = model(data)
* # Compute the loss, gradients and change the weights
* loss = criterion(output, target)
* loss.backward()
* optimizer.step()

**Using the network to make predictions**

Now that you have trained the network, use it to make predictions for the data in the testing set. The network is called model (same as in the previous exercise), and the loader is called test\_loader. We have already initialized variables total and correct to 0.

**Instructions**

**100 XP**

* Set the network in testing (eval) mode.
* Put each image into a vector using inputs.view(-1, number\_of\_features) where the number of features should be deducted by multiplying spatial dimensions (shape) of the image.
* Do the forward pass and put the predictions in output variable.
* # Set the model in eval mode
* model.eval()
* for i, data in enumerate(test\_loader, 0):
* inputs, labels = data
* # Put each image into a vector
* inputs = inputs.view(-1, 28 \* 28)
* # Do the forward pass and get the predictions
* outputs = model(inputs)
* \_, outputs = torch.max(outputs.data, 1)
* total += labels.size(0)
* correct += (outputs == labels).sum().item()
* print('The testing set accuracy of the network is: %d %%' % (100 \* correct / total))

**Convolution operator - OOP way**

Let's kick off this chapter by using convolution operator from the torch.nn package. You are going to create a random tensor which will represent your image and random filters to convolve the image with. Then you'll apply those images.

The torch library and the torch.nn package have already been imported for you.

**Instructions**

**70 XP**

* Create 10 images with shape (1, 28, 28).
* Build 6 convolutional filters of size (3, 3) with stride set to 1 and padding set to 1.
* Apply the filters in the image and print the shape of the feature map.
* # Create 10 random images of shape (1, 28, 28)
* images = torch.rand(10, 1, 28, 28)
* # Build 6 conv. filters
* conv\_filters = torch.nn.Conv2d(
* in\_channels=1,
* out\_channels=6,
* kernel\_size=3,
* stride=1,
* padding=1)
* # Convolve the image with the filters
* output\_feature = conv\_filters(images)
* print(output\_feature.shape)

**Convolution operator - Functional way**

While I and most of PyTorch practitioners love the torch.nn package (OOP way), other practitioners prefer building neural network models in a more functional way, using torch.nn.functional. More importantly, it is possible to mix the concepts and use both libraries at the same time (we have already done it in the previous chapter). You are going to build the same neural network you built in the previous exercise, but this time using the functional way.

As before, we have already imported the torch library and torch.nn.functional as F.

**Instructions**

**100 XP**

* Create 10 random images with shape (1, 28, 28).
* Create 6 random filters with shape (1, 3, 3).
* Convolve the images with the filters.
* # Create 10 random images
* image = torch.rand(10, 1, 28, 28)
* # Create 6 filters
* filters = torch.rand(6, 1, 3, 3)
* # Convolve the image with the filters
* output\_feature = F.conv2d(image, filters, stride=1, padding=1)
* print(output\_feature.shape)

# Max-pooling operator

Here you are going to practice using max-pooling in both OOP and functional way, and see for yourself that the produced results are the same. We have already created and printed the image for you, and imported torch library in addition to torch.nn and torch.nn.Functional as F packages.

**Instructions**

**100 XP**

* Build a max-pooling operator with size 2.
* Apply the max-pooling operator in the image (loaded as im).
* Use a max-pooling operator in functional way in the image.
* Print the results of both cases.
* # Build a pooling operator with size `2`.
* max\_pooling = torch.nn.MaxPool2d(2)
* # Apply the pooling operator
* output\_feature = max\_pooling(im)
* # Use pooling operator in the image
* output\_feature\_F = F.max\_pool2d(im, 2)
* # print the results of both cases
* print(output\_feature)
* print(output\_feature\_F)

# Average-pooling operator

After coding the max-pooling operator, you are now going to code the average-pooling operator. You just need to replace max-pooling with average pooling.

##### Instructions

**100 XP**

* Build an average-pooling operator with size 2.
* Apply the average-pooling operator in the image.
* Use an average-pooling operator in functional way in the image, called im.
* Print the results of both cases.
* # Build a pooling operator with size `2`.
* avg\_pooling = torch.nn.AvgPool2d(2)
* # Apply the pooling operator
* output\_feature = avg\_pooling(im)
* # Use pooling operator in the image
* output\_feature\_F = F.avg\_pool2d(im, 2)
* # print the results of both cases
* print(output\_feature)
* print(output\_feature\_F)

# Your first CNN - \_\_init\_\_ method

You are going to build your first convolutional neural network. You're going to use the MNIST dataset as the dataset, which is made of handwritten digits from 0 to 9. The convolutional neural network is going to have 2 convolutional layers, each followed by a ReLU nonlinearity, and a fully connected layer. We have already imported torch and torch.nn as nn. Remember that each pooling layer halves both the height and the width of the image, so by using 2 pooling layers, the height and width are 1/4 of the original sizes. MNIST images have shape (1, 28, 28)

For the moment, you are going to implement the \_\_init\_\_ method of the net. In the next exercise, you will implement the .forward() method.

NB: We need ***2*** pooling layers, but we only need to ***instantiate a pooling layer once***, because each pooling layer will have the same configuration. Instead, we will use *self.pool* twice in the next exercise.

##### Instructions

**100 XP**

* Instantiate two convolutional filters: the first one should have 5 channels, while the second one should have 10 channels. The kernel\_size for both of them should be 3, and both should use padding=1. Use the names of the arguments (instead of using 1, use padding=1).
* Instantiate a ReLU() nonlinearity.
* Instantiate a max pooling layer which halves the size of the image in both directions.
* Instantiate a fully connected layer which connects the units with the number of classes (we are using MNIST, so there are 10 classes).
* class Net(nn.Module):
* def \_\_init\_\_(self):
* super(Net, self).\_\_init\_\_()
* # Instantiate two convolutional layers
* self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1)
* self.conv2 = nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1)
* # Instantiate the ReLU nonlinearity
* self.relu = nn.ReLU()
* # Instantiate a max pooling layer
* self.pool = nn.MaxPool2d(2, 2)
* # Instantiate a fully connected layer
* self.fc = nn.Linear(7 \* 7 \* 10, 10)

# Your first CNN - forward() method

Now that you have declared all the parameters of your CNN, all you need to do is to implement the net's forward() method, and voila, you have your very first PyTorch CNN.

Note: for evaluation purposes, the entire code of the class needs to be in the script. We are using the \_\_init\_\_ method as you have coded it on the previous exercise, while you are going to code the .forward() method here.

##### Instructions

**100 XP**

* Apply the first convolutional layer, followed by the relu nonlinearity, then in the next line apply max-pooling layer.
* Apply the second convolutional layer, followed by the relu nonlinearity, then in the next line apply max-pooling layer.
* Transform the feature map from 4 dimensional to 2 dimensional space. The first dimension contains the batch size (-1), deduct the second dimension, by multiplying the values for height, width and depth.
* Apply the fully-connected layer and return the result.
* class Net(nn.Module):
* def \_\_init\_\_(self, num\_classes):
* super(Net, self).\_\_init\_\_()
* # Instantiate the ReLU nonlinearity
* self.relu = nn.ReLU()
* # Instantiate two convolutional layers
* self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1)
* self.conv2 = nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1)
* # Instantiate a max pooling layer
* self.pool = nn.MaxPool2d(2, 2)
* # Instantiate a fully connected layer
* self.fc = nn.Linear(7 \* 7 \* 10, 10)
* def forward(self, x):
* # Apply conv followd by relu, then in next line pool
* x = self.relu(self.conv1(x))
* x = self.pool(x)
* # Apply conv followd by relu, then in next line pool
* x = self.relu(self.conv2(x))
* x = self.pool(x)
* # Prepare the image for the fully connected layer
* x = x.view(-1,  7\* 7 \*10 )
* # Apply the fully connected layer and return the result
* return self.fc(x)

# Training CNNs

Similarly to what you did in Chapter 2, you are going to train a neural network. This time however, you will train the CNN you built in the previous lesson, instead of a fully connected network. The packages you need have been imported for you and the network (called net) instantiated. The cross-entropy loss function (called criterion) and the Adam optimizer (called optimizer) are also available. We have subsampled the training set so that the training goes faster, and you are going to use a single epoch.

##### Instructions

**0 XP**

* Compute the predictions from the net.
* Using the predictions and the labels, compute the loss function.
* Compute the gradients for each weight.
* Update the weights using the optimizer.
* for i, data in enumerate(train\_loader, 0):
* inputs, labels = data
* optimizer.zero\_grad()
* # Compute the forward pass
* outputs = net(inputs)
* # Compute the loss function
* loss = criterion(outputs, labels)
* # Compute the gradients
* loss.backward()
* # Update the weights
* optimizer.step()

# Using CNNs to make predictions

Building and training neural networks is a very exciting job (trust me, I do it every day)! However, the main utility of neural networks is to make predictions. This is the entire reason why the field of deep learning has bloomed in the last few years, as neural networks predictions are extremely accurate. On this exercise, we are going to use the convolutional neural network you already trained in order to make predictions on the MNIST dataset.

Remember that torch.max() takes two arguments: -output.data - the tensor which contains the data.

* Either 1 to do argmax or 0 to do max.

##### Instructions

**100 XP**

* Iterate over the given test\_loader, saving the results of each iteration in data.
* Get the image and label from the data tuple, storing the results in image and label.
* Make a forward pass in the net using your image.
* Get the net prediction using torch.max() function.
* # Iterate over the data in the test\_loader
* for i, data in enumerate(test\_loader):
* # Get the image and label from data
* image, label = data
* # Make a forward pass in the net with your image
* output = net(image)
* # Argmax the results of the net
* \_, predicted = torch.max(output.data, 1)
* if predicted == label:
* print("Yipes, your net made the right prediction " + str(predicted))
* else:
* print("Your net prediction was " + str(predicted) + ", but the correct label is: " + str(label))
* Yipes, your net made the right prediction tensor([9])
* Yipes, your net made the right prediction tensor([0])
* Yipes, your net made the right prediction tensor([7])
* Yipes, your net made the right prediction tensor([0])
* Yipes, your net made the right prediction tensor([1])
* Yipes, your net made the right prediction tensor([3])
* Yipes, your net made the right prediction tensor([1])
* Yipes, your net made the right prediction tensor([6])
* Yipes, your net made the right prediction tensor([5])
* Yipes, your net made the right prediction tensor([7])

# Sequential module - init method

Having learned about the sequential module, now is the time to see how you can convert a neural network that doesn't use sequential modules to one that uses them. We are giving the code to build the network in the usual way, and you are going to write the code for the same network using sequential modules.

class Net(nn.Module):

def \_\_init\_\_(self, num\_classes):

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1)

self.conv2 = nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1)

self.conv3 = nn.Conv2d(in\_channels=10, out\_channels=20, kernel\_size=3, padding=1)

self.conv4 = nn.Conv2d(in\_channels=20, out\_channels=40, kernel\_size=3, padding=1)

self.relu = nn.ReLU()

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

self.fc1 = nn.Linear(7 \* 7 \* 40, 1024)

self.fc2 = nn.Linear(1024, 2048)

self.fc3 = nn.Linear(2048, 10)

We want the pooling layer to be used after the second and fourth convolutional layers, while the relu nonlinearity needs to be used after each layer except the last (fully-connected) layer. For the number of filters (kernels), stride, passing, number of channels and number of units, use the same numbers as above.

##### Instructions 1/2

**50 XP**

* [2](javascript:void(0))
* Declare all the layers needed for feature extraction in the self.features.
* class Net(nn.Module):
* def \_\_init\_\_(self):
* super(Net, self).\_\_init\_\_()
* # Declare all the layers for feature extraction
* self.features = nn.Sequential(
* nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1),
* nn.ReLU(inplace=True),
* nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1),
* nn.MaxPool2d(2, 2),
* nn.ReLU(inplace=True),
* nn.Conv2d(in\_channels=10, out\_channels=20, kernel\_size=3, padding=1),
* nn.ReLU(inplace=True),
* nn.Conv2d(in\_channels=20, out\_channels=40, kernel\_size=3, padding=1),
* nn.MaxPool2d(kernel\_size=2, stride=2),
* nn.ReLU(inplace=True))
* Declare the three linear layers in self.classifier.
* class Net(nn.Module):
* def \_\_init\_\_(self):
* super(Net, self).\_\_init\_\_()
* # Declare all the layers for feature extraction
* self.features = nn.Sequential(nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1),
* nn.ReLU(inplace=True),
* nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1),
* nn.MaxPool2d(2, 2), nn.ReLU(inplace=True),
* nn.Conv2d(in\_channels=10, out\_channels=20, kernel\_size=3, padding=1),
* nn.ReLU(inplace=True),
* nn.Conv2d(in\_channels=20, out\_channels=40, kernel\_size=3, padding=1),
* nn.MaxPool2d(2, 2), nn.ReLU(inplace=True))
* # Declare all the layers for classification
* self.classifier = nn.Sequential(nn.Linear(7 \* 7 \* 40, 1024, 1024), nn.ReLU(inplace=True),
* nn.Linear(1024, 2048), nn.ReLU(inplace=True),
* nn.Linear(2048, 10))

**\**

# Sequential module - forward() method

Now, that you have defined all the modules that the network needs, it is time to apply them in the forward() method. For context, we are giving the code for the forward() method, if the net was written in the usual way.

class Net(nn.Module):

def \_\_init\_\_(self, num\_classes):

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1)

self.conv2 = nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1)

self.conv3 = nn.Conv2d(in\_channels=10, out\_channels=20, kernel\_size=3, padding=1)

self.conv4 = nn.Conv2d(in\_channels=20, out\_channels=40, kernel\_size=3, padding=1)

self.relu = nn.ReLU()

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

self.fc1 = nn.Linear(7 \* 7 \* 40, 1024)

self.fc2 = nn.Linear(1024, 2048)

self.fc3 = nn.Linear(2048, 10)

def forward():

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

x = self.relu(self.pool(self.conv2(x)))

x = self.relu(self.conv3(x))

x = self.relu(self.pool(self.conv4(x)))

x = x.view(-1, 7 \* 7 \* 40)

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

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

x = self.fc3(x)

return x

Note: for evaluation purposes, the entire code of the class needs to be in the script. We are using the \_\_init\_\_ method as you have coded it on the previous exercise, while you are going to code the forward() method here.

##### Instructions

**100 XP**

* Extract the features from the images.
* Squeeze the three spatial dimensions of the feature maps into one using the view() method.
* Classify images based on the extracted features.

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        # Declare all the layers for feature extraction

        self.features = nn.Sequential(nn.Conv2d(in\_channels=1, out\_channels=5, kernel\_size=3, padding=1),

                                      nn.ReLU(inplace=True),

                                      nn.Conv2d(in\_channels=5, out\_channels=10, kernel\_size=3, padding=1),

                                      nn.MaxPool2d(2, 2), nn.ReLU(inplace=True),

                                      nn.Conv2d(in\_channels=10, out\_channels=20, kernel\_size=3, padding=1),

                                      nn.ReLU(inplace=True),

                                      nn.Conv2d(in\_channels=20, out\_channels=40, kernel\_size=3, padding=1),

                                      nn.MaxPool2d(2, 2), nn.ReLU(inplace=True))

        # Declare all the layers for classification

        self.classifier = nn.Sequential(nn.Linear(7 \* 7 \* 40, 1024), nn.ReLU(inplace=True),

                                        nn.Linear(1024, 2048), nn.ReLU(inplace=True),

                                        nn.Linear(2048, 10))

    def forward(self, x):

        # Apply the feature extractor in the input

        x = self.features(x)

        # Squeeze the three spatial dimensions in one

        x = x.view(-1, 7 \* 7 \* 40)

        # Classify the images

        x = self.classifier(x)

        return x

# Validation set

You saw the need for validation set in the previous video. Problem is that the datasets typically are not separated into training, validation and testing. It is your job as a data scientist to split the dataset into training, testing and validation. The easiest (and most used) way of doing so is to do a random splitting of the dataset. In PyTorch, that can be done using SubsetRandomSampler object. You are going to split the training part of MNIST dataset into training and validation. After randomly shuffling the dataset, use the first 55000 points for training, and the remaining 5000 points for validation.

##### Instructions

**100 XP**

* Use numpy.arange() to create an array containing numbers [0, 59999] and then randomly shuffle the array.
* In the train\_loader using SubsetRandomSampler() use the first 55k points for training.
* In the val\_loader use the remaining 5k points for validation.
* # Shuffle the indices
* indices = np.arange(60000)
* np.random.shuffle(indices)
* # Build the train loader
* train\_loader = torch.utils.data.DataLoader(datasets.MNIST('mnist', download=True, train=True,
* transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])),
* batch\_size=64, shuffle=False, sampler=torch.utils.data.SubsetRandomSampler(indices[:55000]))
* # Build the validation loader
* val\_loader = torch.utils.data.DataLoader(datasets.MNIST('mnist', download=True, train=True,
* transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])),
* batch\_size=64, shuffle=False, sampler=torch.utils.data.SubsetRandomSampler(indices[55000:]))

# L2-regularization

You are going to implement each of the regularization techniques explained in the previous video. Doing so, you will also remember important concepts studied throughout the course. You will start with l2-regularization, the most important regularization technique in machine learning. As you saw in the video, l2-regularization simply penalizes large weights, and thus enforces the network to use only small weights

* Instantiate an object called model from class Net(), which is available in your workspace (consider it as a blackbox).
* Instantiate the cross-entropy loss.
* Instantiate Adam optimizer with learning\_rate equals to 3e-4, and l2 regularization parameter equals to 0.001.
* # Instantiate the network
* model = Net()
* # Instantiate the cross-entropy loss
* criterion = nn.CrossEntropyLoss()
* # Instantiate the Adam optimizer
* optimizer = optim.Adam(model.parameters(), lr=3e-4, weight\_decay=0.001)

# Dropout

You saw that dropout is an effective technique to avoid overfitting. Typically, dropout is applied in fully-connected neural networks, or in the fully-connected layers of a convolutional neural network. You are now going to implement dropout and use it on a small fully-connected neural network.

For the first hidden layer use 200 units, for the second hidden layer use 500 units, and for the output layer use 10 units (one for each class). For the activation function, use ReLU. Use .Dropout() with strength 0.5, between the first and second hidden layer. Use the sequential module, with the order being: fully-connected, activation, dropout, fully-connected, activation, fully-connected.

class Net(nn.Module):

    def \_\_init\_\_(self):

        # Define all the parameters of the net

        self.classifier = nn.Sequential(

            nn.Linear(28\*28, 200),

            nn.ReLU(inplace=True),

            nn.Dropout(p=0.5),

            nn.Linear(200, 500),

            nn.ReLU(inplace=True),

            nn.Linear(500, 10))

* Apply the forward pass in the forward() method.
* class Net(nn.Module):
* def \_\_init\_\_(self):
* # Define all the parameters of the net
* self.classifier = nn.Sequential(
* nn.Linear(28\*28, 200),
* nn.ReLU(inplace=True),
* nn.Dropout(p=0.5),
* nn.Linear(200, 500),
* nn.ReLU(inplace=True),
* nn.Linear(500, 10))
* def forward(self, x):
* # Do the forward pass
* return self.classifier(x)

# Batch-normalization

Dropout is used to regularize fully-connected layers. Batch-normalization is used to make the training of convolutional neural networks more efficient, while at the same time having regularization effects. You are going to implement the \_\_init\_\_ method of a small convolutional neural network, with batch-normalization. The feature extraction part of the CNN will contain the following modules (in order): convolution, max-pool, activation, batch-norm, convolution, max-pool, relu, batch-norm.

The first convolutional layer will contain 10 output channels, while the second will contain 20 output channels. As always, we are going to use MNIST dataset, with images having shape (28, 28) in grayscale format (1 channel). In all cases, the size of the filter should be 3, the stride should be 1 and the padding should be 1.

##### Instructions

**0 XP**

* Implement the feature extraction part of the network, using the description in the context.
* Implement the fully-connected (classifier) part of the network.
* class Net(nn.Module):
* def \_\_init\_\_(self):
* super(Net, self).\_\_init\_\_()
* # Implement the sequential module for feature extraction
* self.features = nn.Sequential(
* nn.Conv2d(in\_channels=1, out\_channels=10, kernel\_size=3, stride=1, padding=1),
* nn.MaxPool2d(2,2),
* nn.ReLU(inplace=True), nn.BatchNorm2d(10),
* nn.Conv2d(in\_channels=10,
* out\_channels=20,
* kernel\_size=3,
* stride=1,
* padding=1),
* nn.MaxPool2d(2, 2),
* nn.ReLU(inplace=True),
* nn.BatchNorm2d(20))
* # Implement the fully connected layer for classification
* self.fc = nn.Linear(in\_features=7\*7\*20, out\_features=10)

# Finetuning a CNN

Previously, you trained a model to classify handwritten digits and saved the model parameters to my\_net.pth. Now you're going to classify handwritten letters, but you have a smaller training set.

In the first step, you'll create a new model using this training set, but the accuracy will be poor. Next, you'll perform the same training, but you'll start with the parameters from your digit classifying model. Even though digits and letters are two different classification problems, you'll see that using information from your previous model will dramatically improve this one.

* Create a new model using the Net() module.
* Change the number of output units, to the number of classifications for letters.
* # Create a new model
* model = Net()
* # Change the number of out channels
* model.fc = nn.Linear(7 \* 7 \* 512, 26)
* # Train and evaluate the model
* model.train()
* train\_net(model, optimizer, criterion)
* print("Accuracy of the net is: " + str(model.eval()))
* Repeat the training process, but first load the digit classifier parameters from my\_net.pth.
* # Create a model using
* model = Net()
* # Load the parameters from the old model
* model.load\_state\_dict(torch.load('my\_net.pth'))
* # Change the number of out channels
* model.fc = nn.Linear(7 \* 7 \* 512, 26)
* # Train and evaluate the model
* model.train()
* train\_net(model, optimizer, criterion)
* print("Accuracy of the net is: " + str(model.eval()))

# Torchvision module

You already finetuned a net you had pretrained. In practice though, it is very common to finetune CNNs that someone else (typically the library's developers) have pretrained in ImageNet. Big networks still take a lot of time to be trained on large datasets, and maybe you cannot afford to train a large network on a dataset of 1.2 million images on your laptop.

Instead, you can simply download the network and finetune it on your dataset. That's what you will do right now. You are going to assume that you have a personal dataset, containing the images from all your last 7 holidays. You want to build a neural network that can classify each image depending on the holiday it comes from. However, since the dataset is so small, you need to use the finetuning technique.

##### Instructions

**70 XP**

* Import the module that lets you download state-of-the-art CNNs.
* Download and load a pretrained ResNet18 network.
* Freeze all the layers bar the final one.
* Change the last layer to correspond to the number of classes (7) in your dataset.
* # Import the module
* import torchvision
* # Download resnet18
* model = torchvision.models.resnet18(pretrained=True)
* # Freeze all the layers bar the last one
* for param in model.parameters():
* param.requires\_grad = False
* # Change the number of output units
* model.fc = nn.Linear(512, 7)