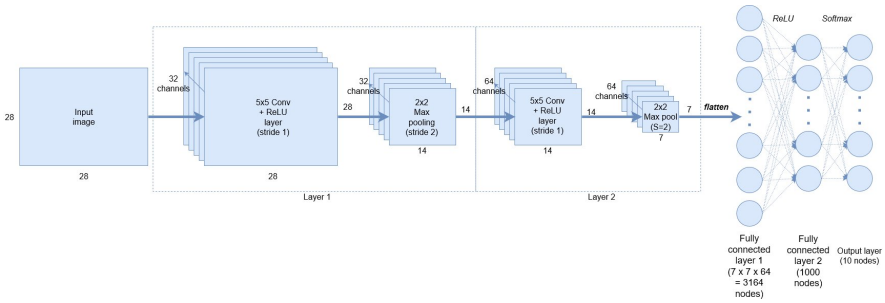
Convolutional Neural Networks Tutorial in PyTorch

By admin | Convolutional Neural Networks

Jun
16
11



In a previous introductory tutorial on neural networks, a three layer neural network was developed to classify the hand-written digits of the MNIST dataset. In the end, it was able to achieve a classification accuracy around 86%. For a simple data set such as MNIST, this is actually quite poor. Further optimizations can bring densely connected networks of a modest size up to 97-98% accuracy. This is significantly better, but still not that great for MNIST. We need something more state-of-the-art, some method which can truly be called *deep learning*. This tutorial will present just such a *deep learning* method that can achieve very high accuracy in image classification tasks – the Convolutional Neural Network. In particular, this tutorial will show you both the theory and practical application of Convolutional Neural Networks in PyTorch.

PyTorch is a powerful deep learning framework which is rising in popularity, and it is thoroughly at home in Python which makes rapid prototyping very easy. This tutorial won't assume much in regards to prior knowledge of PyTorch, but it might be helpful to checkout my previous introductory tutorial to PyTorch. All the code for this Convolutional Neural Networks tutorial can be found on this site's Github repository – found here. Let's get to it.

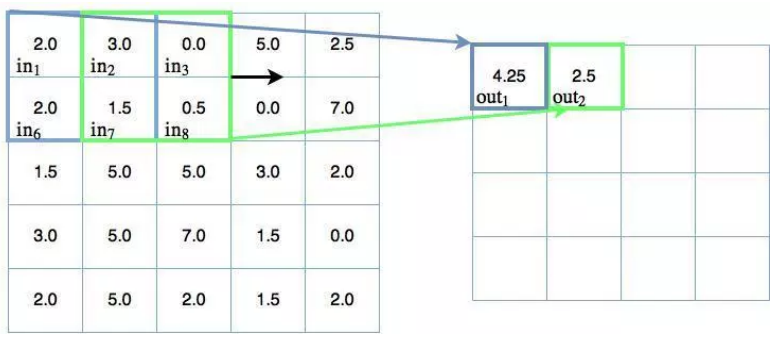
Why Convolutional Neural Networks?

Fully connected networks with a few layers can only do so much – to get close to state-of-the-art results in image classification it is necessary to go *deeper.* In other words, lots more layers are required in the network. However, by adding a lot of additional layers, we come across some problems. First, we can run into the vanishing gradient problem. – however, this can be solved to an extent by using sensible activation functions, such as the ReLU family of activations. Another issue for deep fully connected networks is that the number of trainable parameters in the model (i.e. the weights) can grow rapidly. This means that the training slows down or becomes practically impossible, and also exposes the model to overfitting. So what's a solution?

Convolutional Neural Networks try to solve this second problem by exploiting correlations between adjacent inputs in images (or time series). For instance, in an image of a cat and a dog, the pixels close to the cat's eyes are more likely to be correlated with the nearby pixels which show the cat's nose – rather than the pixels on the other side of the image that represent the dog's nose. This means that not every node in the network needs to be connected to every other node in the next layer – and this cuts down the number of weight parameters required to be trained in the model. Convolution Neural Networks also have some other tricks which improve training, but we'll get to these in the next section.

How does a Convolutional Neural Network work?

The first thing to understand in a Convolutional Neural Network is the actual *convolution* part. This is a fancy mathematical word for what is essentially a moving window or filter across the image being studied. This moving window applies to a certain neighborhood of nodes as shown below – here, the filter applied is (0.5 × the node value):

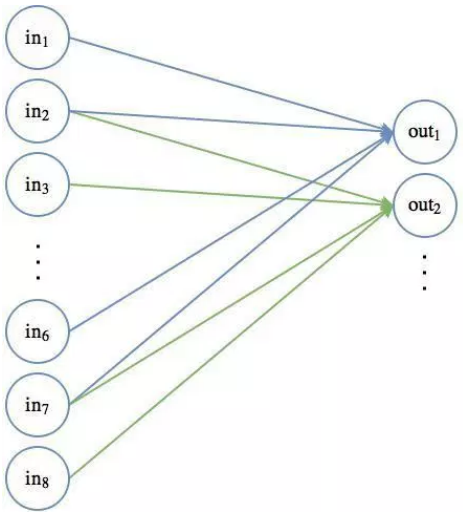


Moving 2×2 filter (all weights = 0.5)

Only two outputs have been shown in the diagram above, where each output node is a map from a 2 x 2 input square. The weight of the mapping of each input square, as previously mentioned, is 0.5 across all four inputs. So the output can be calculated as:

$$
\begin{aligned}
out_1 &= 0.5in_1 + 0.5in_2 + 0.5in_6 + 0.5in_7 \\
&= 0.5 \times 2.0 + 0.5 \times 3.0 + 0.5 \times 2.0 + 0.5 \times 1.5 \\
&= 4.25 \\
out_2 &= 0.5in_2 + 0.5in_3 + 0.5in_7 + 0.5in_8 \\
&= 0.5 \times 3.0 + 0.5 \times 0.0 + 0.5 \times 1.5 + 0.5 \times 0.5 \\
&= 2.5
\end{aligned}
$$

In the convolutional part of the neural network, we can imagine this 2 x 2 moving filter sliding across all the available nodes / pixels in the input image. This operation can also be illustrated using standard neural network node diagrams:



Moving 2×2 filter – node diagram

The first position of the moving filter connections is illustrated by the blue connections, and the second is shown with the green lines. The weights of each of these connections, as stated previously, is 0.5.

There are a few things in this convolutional step which improve training by reducing parameters/weights:

- *Sparse* connections – not every node in the first / input layer is connected to every node in the second layer. This is contrary to fully connected neural networks, where every node is connected to every other in the following layer.
- Constant filter parameters – each filter has constant parameters. In other words, as the filter moves around the image, the same weights are applied to each 2 x 2 set of nodes. Each filter, as such, can be trained to perform a certain specific transformation of the input space. Therefore, each filter has a certain set of weights that are applied for each convolution operation – this reduces the number of parameters.
    - Note – this is not to say that each weight is constant *within* the filter. In the example above, the weights were [0.5, 0.5, 0.5, 0.5] but could have just as easily been something like [0.25, 0.1, 0.8, 0.001]. It all depends on how each filter is trained

These two properties of Convolutional Neural Networks can drastically reduce the number of parameters which need to be trained compared to fully connected neural networks.

The next step in the Convolutional Neural Network structure is to pass the output of the

Cross entropy
Deep learning
Double Q
Dueling Q
gensim
GPUs
Keras
Loss functions
LSTMs
Metrics and summaries
Neural networks
NLP
Optimisation
PyTorch
Recurrent neural networks
Reinforcement learning
TensorBoard
TensorFlow
TensorFlow 2.0
Transfer learning
Weight initialization
Word2Vec

META

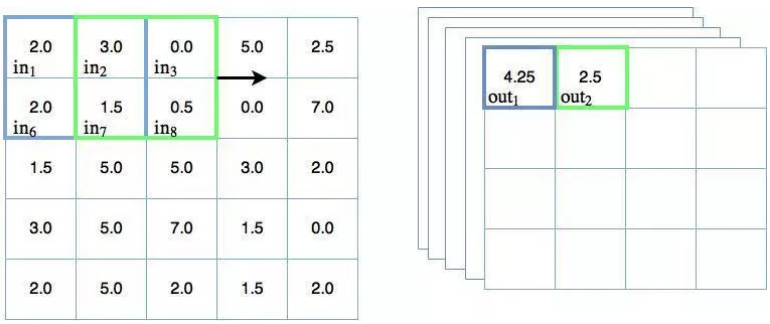
Log in
Entries feed
Comments feed
WordPress.org

of the ReLU activation function. This provides the standard non-linear behavior that neural networks are known for.

The process involved in this convolutional block is often called *feature mapping* – this refers to the idea that each convolutional filter can be trained to "search" for different features in an image, which can then be used in classification. Before we move onto the next main feature of Convolutional Neural Networks, called *pooling*, we will examine this idea of feature mapping and *channels* in the next section.

Feature mapping and multiple channels

As mentioned previously, because the weights of individual filters are held constant as they are applied over the input nodes, they can be trained to select certain features from the input data. In the case of images, it may learn to recognize common geometrical objects such as lines, edges and other shapes which make up objects. This is where the name *feature mapping* comes from. Because of this, any convolution layer needs multiple filters which are trained to detect different features. So therefore, the previous moving filter diagram needs to be updated to look something like this:



Multiple convolutional filters

Now you can see on the right hand side of the diagram above that there are multiple, stacked outputs from the convolution operation. This is because there are multiple trained filters which produce their own 2D output (for a 2D image). These multiple filters are commonly called *channels* in deep learning. Each of these channels will end up being trained to detect certain key features in the image. The output of a convolution layer, for a gray-scale image like the MNIST dataset, will therefore actually have 3 dimensions – 2D for each of the channels, then another dimension for the number of different channels.

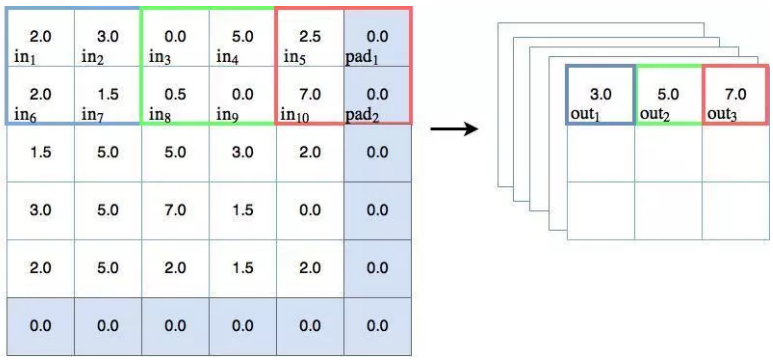
If the input is itself multi-channelled, as in the case of a color RGB image (one channel for each R-G-B), the output will actually be 4D. Thankfully, any deep learning library worth its salt, PyTorch included, will be able to handle all this mapping easily for you. Finally, don't forget that the output of the convolution operation will be passed through an activation for each node.

Now, the next vitally important part of Convolutional Neural Networks is a concept called *pooling*.

Pooling

- It reduces the number of parameters in your model by a process called *down-sampling*
- It makes feature detection more robust to object orientation and scale changes

So what is pooling? It is another sliding window type technique, but instead of applying weights, which can be trained, it applies a statistical function of some type over the contents of its window. The most common type of pooling is called *max pooling*, and it applies the *max()* function over the contents of the window. There are other variants such as *mean pooling* (which takes the statistical mean of the contents) which are also used in some cases. In this tutorial, we will be concentrating on max pooling. The diagram below shows an example of the max pooling operation:



Max pooling example (with padding)

We'll go through a number of points relating to the diagram above:

The basics

In the diagram above, you can observe the max pooling taking effect. For the first window, the blue one, you can see that the max pooling outputs a 3.0 which is the maximum node value in the 2×2 window. Likewise for the green 2×2 window it outputs the maximum of 5.0 and a maximum of 7.0 for the red window. This is pretty straight-forward.

Strides and down-sampling

In the pooling diagram above, you will notice that the pooling window shifts to the right each time by 2 places. This is called a *stride* of 2. In the diagram above, the stride is only shown in the *x* direction, but, if the goal was to prevent pooling window overlap, the stride would also have to be 2 in the *y* direction as well. In other words, the stride is actually specified as [2, 2]. One important thing to notice is that, if during pooling the stride is greater than 1, then the output size will be reduced. As can be observed above, the 5 x 5 input is reduced to a 3 x 3 output. This is a good thing – it is called down-sampling, and it reduces the number of trainable parameters in the model.

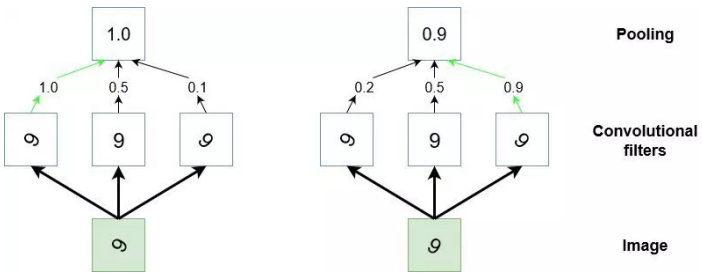
Padding

Another thing to notice in the pooling diagram above is that there is an extra column and row added to the 5 x 5 input – this makes the effective size of the pooling space equal to 6 x 6. This is to ensure that the 2 x 2 pooling window can operate correctly with a stride of [2, 2] and is called *padding*. These nodes are basically dummy nodes – because the values of these dummy nodes is 0, they are basically invisible to the max pooling operation. Padding will need to be considered when constructing our Convolutional Neural Network in PyTorch.

Ok, so now we understand how pooling works in Convolutional Neural Networks, and how it is useful in performing down-sampling, but what else does it do? Why is max pooling used so frequently?

Why is pooling used in convolutional neural networks?

In addition to the function of down-sampling, pooling is used in Convolutional Neural Networks to make the detection of certain features somewhat invariant to scale and orientation changes. Another way of thinking about what pooling does is that it generalizes over lower level, more complex information. Let's imagine the case where we have convolutional filters that, during training, learn to detect the digit "9" in various orientations within the input images. In order for the Convolutional Neural Network to learn to classify the appearance of "9" in the image correctly, it needs to in some way "activate" whenever a "9" is found anywhere in the image, no matter what the size or orientation the digit is (except for when it looks like "6", that is). Pooling can assist with this higher level, generalized feature selection, as the diagram below shows:



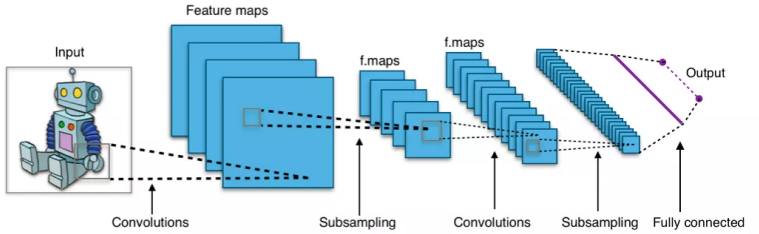
Stylized representation of pooling

The diagram is a stylized representation of the pooling operation. If we consider that a small region of the input image has a digit "9" in it (green box) and assume we are trying to detect such a digit in the image, what will happen is that, if we have a few convolutional filters, they will learn to activate (via the ReLU) when they "see" a "9" in the image (i.e. return a large output). However, they will activate more or less strongly depending on what orientation the "9" is. We want the network to detect a "9" in the image regardless of what the orientation is and this is where the pooling comes it. It "looks" over the output of these three filters and gives a high output so long as *any one* of these filters has a high activation.

Therefore, pooling acts as a *generalizer* of the lower level data, and so, in a way, enables the network to move from high resolution *data* to lower resolution *information.* In other words, pooling coupled with convolutional filters attempts to detect *objects* within an image.

The final picture

The image below from Wikipedia shows the structure of a fully developed Convolutional Neural Network:



Full convolutional neural network – By Aphex34 (Own work) [CC BY-SA 4.0], via Wikimedia Commons

If you work the image above from left to right, we first see that there is an image of a robot. Then "scanning" over this image are a series of convolutional filters or feature maps. The output of these filters is then sub-sampled by pooling operations. After this, there is another set of convolutions and pooling on the output of the first convolution-pooling operation. Finally, at the output there is "attached" a fully connected layer. The purpose of this fully connected layer at the output of the network requires some explanation.

The fully connected layer

As previously discussed, a Convolutional Neural Network takes high resolution data and effectively resolves that into representations of objects. The fully connected layer can therefore be thought of as attaching a standard classifier onto the information-rich output of the network, to "interpret" the results and finally produce a classification result. In order to attach this fully connected layer to the network, the dimensions of the output of the Convolutional Neural Network need to be flattened.

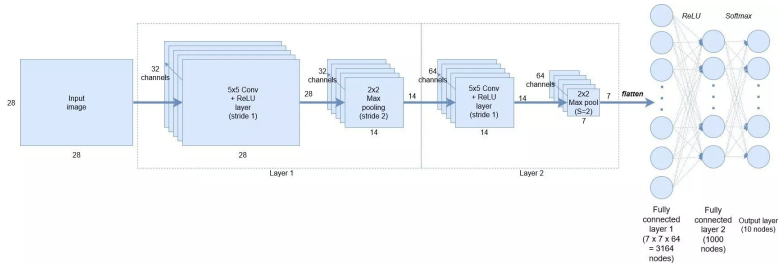
Consider the previous diagram – at the output, we have multiple channels of x x y matrices/tensors. These channels need to be flattened to a single (N X 1) tensor. Consider an example – let's say we have 100 channels of 2 x 2 matrices, representing the output of the final pooling operation of the network. Therefore, this needs to be flattened to 2 x 2 x 100 = 400 rows. This can be easily performed in PyTorch, as will be demonstrated below.

Now the basics of Convolutional Neural Networks has been covered, it is time to show how they can be implemented in PyTorch.

Implementing Convolutional Neural Networks in PyTorch

Any deep learning framework worth its salt will be able to easily handle Convolutional Neural Network operations. PyTorch is such a framework. In this section, I'll show you how to create Convolutional Neural Networks in PyTorch, going step by step. Ideally, you will already have some notion of the basics of PyTorch (if not, you can check out my introductory PyTorch tutorial) – otherwise, you're welcome to wing it. The network we're going to build will perform MNIST digit classification. The full code for the tutorial can be found at this site's Github repository.

The Convolutional Neural Network architecture that we are going to build can be seen in the diagram below:



Convolutional neural network that will be built

First up, we can see that the input images will be 28 x 28 pixel greyscale representations of digits. The first layer will consist of 32 channels of 5 x 5 convolutional filters + a ReLU activation, followed by 2 x 2 max pooling down-sampling with a stride of 2 (this gives a 14 x 14 output). In the next layer, we have the 14 x 14 output of layer 1 being scanned again with 64 channels of 5 x 5 convolutional filters and a final 2 x 2 max pooling (stride = 2) down-sampling to produce a 7 x 7 output of layer 2.

After the convolutional part of the network, there will be a flatten operation which creates 7 x 7 x 64 = 3164 nodes, an intermediate layer of 1000 fully connected nodes and a softmax operation over the 10 output nodes to produce class probabilities. These layers represent the output classifier.

Loading the dataset

PyTorch has an integrated MNIST dataset (in the torchvision package) which we can use via the DataLoader functionality. In this sub-section, I'll go through how to setup the data loader for the MNIST data set. But first, some preliminary variables need to be defined:

```
# Hyperparameters
num_epochs = 5
num_classes = 10
batch_size = 100
learning_rate = 0.001

DATA_PATH = 'C:\\Users\Andy\PycharmProjects\MNISTData'
MODEL_STORE_PATH = 'C:\\Users\Andy\PycharmProjects\pytorch_models\\
```

First off, we set up some training hyperparameters. Next – there is a specification of some local drive folders to use to store the MNIST dataset (PyTorch will download the dataset into this folder for you automatically) and also a location for the trained model parameters once training is complete.

Next, we setup a transform to apply to the MNIST data, and also the data set variables:

```
# transforms to apply to the data
trans = transforms.Compose([transforms.ToTensor(), transforms.Norma

# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=Tru
test_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=Fals
```

the specified dataset. Numerous transforms can be chained together in a list using the Compose() function. In this case, first we specify a transform which converts the input data set to a PyTorch *tensor*. A PyTorch tensor is a specific data type used in PyTorch for all of the various data and weight operations within the network. In its essence though, it is simply a multi-dimensional matrix. In any case, PyTorch requires the data set to be transformed into a tensor so it can be consumed in the training and testing of the network.

The next argument in the Compose() list is a normalization transformation. Neural networks train better when the input data is normalized so that the data ranges from -1 to 1 or 0 to 1. To do this via the PyTorch Normalize transform, we need to supply the mean and standard deviation of the MNIST dataset, which in this case is 0.1307 and 0.3081 respectively. Note, that for each input channel a mean and standard deviation must be supplied – in the MNIST case, the input data is only single channeled, but for something like the CIFAR data set, which has 3 channels (one for each color in the RGB spectrum) you would need to provide a mean and standard deviation for each channel.

Next, the *train_dataset* and *test_dataset* objects need to be created. These will subsequently be passed to the data loader. In order to create these data sets from the MNIST data, we need to provide a few arguments. First, the *root* argument specifies the folder where the train.pt and test.pt data files exist. The *train* argument is a boolean which informs the data set to pickup either the train.pt data file or the test.pt data file. The next argument, *transform*, is where we supply any transform object that we've created to apply to the data set – here we supply the *trans* object which was created earlier. Finally, the download argument tells the MNIST data set function to download the data (if required) from an online source.

Now both the train and test datasets have been created, it is time to load them into the data loader:

```
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_s
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_siz
```

The data loader object in PyTorch provides a number of features which are useful in consuming training data – the ability to shuffle the data easily, the ability to easily batch the data and finally, to make data consumption more efficient via the ability to load the data in parallel using multiprocessing. As can be observed, there are three simple arguments to supply – first the data set you wish to load, second the batch size you desire and finally whether you wish to randomly shuffle the data. A data loader can be used as an iterator – so to extract the data we can just use the standard Python iterators such as enumerate. This will be shown in practice later in this tutorial.

Creating the model

Next, we need to setup our nn.Module class, which will define the Convolutional Neural Network which we are going to train:

```
class ConvNet(nn.Module):
    def __init__(self):
```

```
            nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.drop_out = nn.Dropout()
        self.fc1 = nn.Linear(7 * 7 * 64, 1000)
        self.fc2 = nn.Linear(1000, 10)
```

Ok – so this is where the model definition takes place. The most straight-forward way of creating a neural network structure in PyTorch is by creating a class which inherits from the nn.Module super class within PyTorch. The nn.Module is a very useful PyTorch class which contains all you need to construct your typical deep learning networks. It also has handy functions such as ways to move variables and operations onto a GPU or back to a CPU, apply recursive functions across all the properties in the class (i.e. resetting all the weight variables), creates streamlined interfaces for training and so on. It is worth checking out all the methods available here.

The first step is to create some sequential layer objects within the class _init_ function. First, we create layer 1 (*self.layer1*) by creating a nn.Sequential object. This method allows us to create sequentially ordered layers in our network and is a handy way of creating a convolution + ReLU + pooling sequence. As can be observed, the first element in the sequential definition is the Conv2d nn.Module method – this method creates a set of convolutional filters. The first argument is the number of input channels – in this case, it is our single channel grayscale MNIST images, so the argument is 1. The second argument to Conv2d is the number of output channels – as shown in the model architecture diagram above, the first convolutional filter layer comprises of 32 channels, so this is the value of our second argument.

The *kernel_size* argument is the size of the convolutional filter – in this case we want 5 x 5 sized convolutional filters – so the argument is 5. If you wanted filters with different sized shapes in the *x* and *y* directions, you'd supply a tuple (*x-size, y-size*). Finally, we want to specify the padding argument. This takes a little bit more thought. The output size of any dimension from either a convolutional filtering or pooling operation can be calculated by the following equation:

$$W_{out} = \frac{(W_{in} - F + 2P)}{S} + 1$$

Where W_{in} is the width of the input, *F* is the filter size, *P* is the padding and *S* is the stride. The same formula applies to the height calculation, but seeing as our image and filtering are symmetrical the same formula applies to both. If we wish to keep our input and output dimensions the same, with a filter size of 5 and a stride of 1, it turns out from the above formula that we need a padding of 2. Therefore, the argument for padding in Conv2d is 2.

The next element in the sequence is a simple ReLU activation. The last element that is added in the sequential definition for *self.layer1* is the max pooling operation. The first argument is the pooling size, which is 2 x 2 and hence the argument is 2. Second – we want to down-sample our data by reducing the effective image size by a factor of 2. To do this, using the formula above, we set the stride to 2 and the padding to zero. Therefore, the stride argument is equal to 2. The padding argument defaults to 0 if we don't specify

Next, the second layer, *self.layer2*, is defined in the same way as the first layer. The only difference is that the input into the Conv2d function is now 32 channels, with an output of 64 channels. Using the same logic, and given the pooling down-sampling, the output from *self.layer2* is 64 channels of 7 x 7 images.

Next, we specify a drop-out layer to avoid over-fitting in the model. Finally, two two fully connected layers are created. The first layer will be of size 7 x 7 x 64 nodes and will connect to the second layer of 1000 nodes. To create a fully connected layer in PyTorch, we use the nn.Linear method. The first argument to this method is the number of nodes in the layer, and the second argument is the number of nodes in the following layer.

With this _init_ definition, the layer definitions have now been created. The next step is to define how the data flows through these layers when performing the forward pass through the network:

```
def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.reshape(out.size(0), -1)
    out = self.drop_out(out)
    out = self.fc1(out)
    out = self.fc2(out)
    return out
```

It is important to call this function "forward" as this will override the base forward function in nn.Module and allow all the nn.Module functionality to work correctly. As can be observed, it takes an input argument *x*, which is the data that is to be passed through the model (i.e. a batch of data). We pass this data into the first layer (*self.layer1*) and return the output as "out". This output is then fed into the following layer and so on. Note, after *self.layer2*, we apply a reshaping function to *out*, which flattens the data dimensions from 7 x 7 x 64 into 3164 x 1. Next, the dropout is applied followed by the two fully connected layers, with the final output being returned from the function.

Ok – so now we have defined what our Convolutional Neural Network is, and how it operates. It's time to train the model.

Training the model

Before we train the model, we have to first create an instance of our ConvNet class, and define our loss function and optimizer:

```
model = ConvNet()

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

First, an instance of ConvNet() is created called "model". Next, we define the loss operation that will be used to calculate the loss. In this case, we use PyTorch's CrossEntropyLoss() function. You may have noticed that we haven't yet defined a SoftMax

function – winning. Next, we define an Adam optimizer. The first argument passed to this function are the parameters we want the optimizer to train. This is made easy via the nn.Module class which ConvNet derives from – all we have to do is pass model.parameters() to the function and PyTorch keeps track of all the parameters within our model which are required to be trained. Finally, the learning rate is supplied.

Next – the training loop is created:

```
# Train the model
total_step = len(train_loader)
loss_list = []
acc_list = []
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Run the forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())

        # Backprop and perform Adam optimisation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Track the accuracy
        total = labels.size(0)
        _, predicted = torch.max(outputs.data, 1)
        correct = (predicted == labels).sum().item()
        acc_list.append(correct / total)

        if (i + 1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accur
                  .format(epoch + 1, num_epochs, i + 1, total_step,
                          (correct / total) * 100))
```

The most important parts to start with are the two loops – first, the number of epochs is looped over, and within this loop, we iterate over train_loader using enumerate. Within this inner loop, first the outputs of the forward pass through the model are calculated by passing *images* (which is a batch of normalized MNIST images from *train_loader*) to it. Note, we don't have to call *model.forward(images)* as nn.Module knows that *forward* needs to be called when it executes *model(images)*.

The next step is to pass the model outputs and the true image labels to our CrossEntropyLoss function, defined as *criterion*. The loss is appended to a list that will be used later to plot the progress of the training. The next step is to perform back-propagation and an optimized training step. First, the gradients have to be zeroed, which can be done easily by calling zero_grad() on the optimizer. Next, we call .backward() on the *loss* variable to perform the back-propagation. Finally, now that the gradients have been calculated in the back-propagation, we simply call optimizer.step() to perform the Adam optimizer training step. PyTorch makes training the model very easy and intuitive.

The next set of steps involves keeping track of the accuracy on the training set. The predictions of the model can be determined by using the torch.max() function, which returns the index of the maximum value in a tensor. The first argument to this function is the tensor to be examined, and the second argument is the axis over which to determine

maximum value over the 10 output nodes. Each of these will correspond to one of the hand written digits (i.e. output 2 will correspond to digit "2" and so on). The output node with the highest value will be the prediction of the model. Therefore, we need to set the second argument of the torch.max() function to 1 – this points the max function to examine the output node axis (axis=0 corresponds to the batch_size dimension).

This returns a list of prediction integers from the model – the next line compares the predictions with the true labels (predicted == labels) and sums them to determine how many correct predictions there are. Note the output of sum() is still a tensor, so to access it's value you need to call .item(). We divide the number of correct predictions by the batch_size (equivalent to labels.size(0)) to obtain the accuracy. Finally, during training, after every 100 iterations of the inner loop the progress is printed.

The training output will look something like this:

```
Epoch [1/6], Step [100/600], Loss: 0.2183, Accuracy: 95.00%
Epoch [1/6], Step [200/600], Loss: 0.1637, Accuracy: 95.00%
Epoch [1/6], Step [300/600], Loss: 0.0848, Accuracy: 98.00%
Epoch [1/6], Step [400/600], Loss: 0.1241, Accuracy: 97.00%
Epoch [1/6], Step [500/600], Loss: 0.2433, Accuracy: 95.00%
Epoch [1/6], Step [600/600], Loss: 0.0473, Accuracy: 98.00%
Epoch [2/6], Step [100/600], Loss: 0.1195, Accuracy: 97.00%
```

Next, let's create some code to determine the model accuracy on the test set.

Testing the model

To test the model, we use the following code:

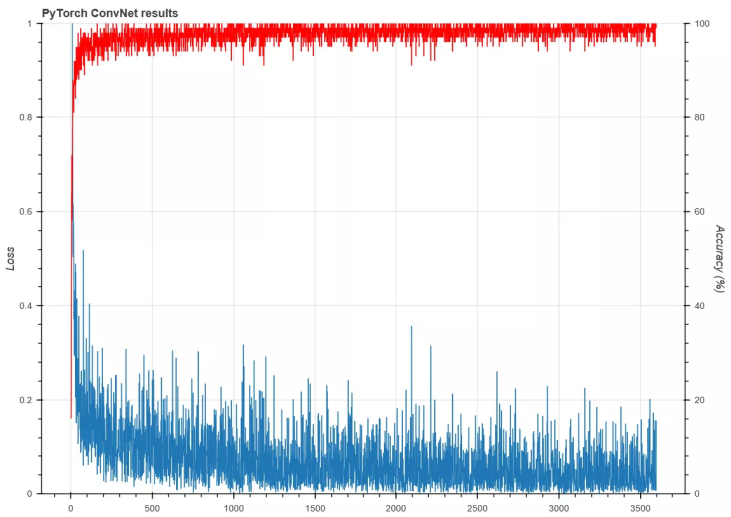
```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %

# Save the model and plot
torch.save(model.state_dict(), MODEL_STORE_PATH + 'conv_net_model.cl
```

As a first step, we set the model to evaluation mode by running model.eval(). This is a handy function which disables any drop-out or batch normalization layers in your model, which will befuddle your model evaluation / testing. The torch.no_grad() statement disables the autograd functionality in the model (see here for more details) as it is not needing in model testing / evaluation, and this will act to speed up the computations. The rest is the same as the accuracy calculations during training, except that in this case, the code iterates through the *test loader*.

Finally, the result is output to the console, and the model is saved using the torch.save() function.

In the the last part of the code on the Github repo, I perform some plotting of the loss and accuracy tracking using the Bokeh plotting library. The final results look like this:

Test Accuracy of the model on the 10000 test images: 99.03 %



PyTorch Convolutional Neural Network results

As can be observed, the network quite rapidly achieves a high degree of accuracy on the training set, and the test set accuracy, after 6 epochs, arrives at 99% – not bad! Certainly better than the accuracy achieved in basic fully connected neural networks.

In summary: in this tutorial you have learnt all about the benefits and structure of Convolutional Neural Networks and how they work. You have also learnt how to implement them in the awesome PyTorch deep learning framework – a framework which, in my view, has a big future. I hope it was useful – have fun in your deep learning journey!

Recommended online course: If you're more of a video learner, check out this inexpensive online course: Practical Deep Learning with PyTorch

---

About the Author