# Case Study: Convolutional Neural Network (CNN) for Handwritten Image Classification using Pytorch

The steps involved in building, training, and validating a model are listed below.

1. **Import necessary libraries**

   Pytorch is an open source library used for Deep Learning applications. Import "*torch*" library to use necessary algorithms to implement a CNN.

```
import torch
import torchvision
from torch import nn, optim
```

2. **Initialize hyperparameters**

We must initialise the values epoch, batch size, and learning rate.

**Epoch** is the number of passes of the entire training dataset through the neural network. A pair of forward and backward propagation indicates a single pass.

**Batch size** is the number of samples to work through before updating the weights and biases associated with the model.

**Learning rate** controls how much to change the model parameters in response to the prediction error each time the model weights are updated.

```
batch_size = 64
epoch = 5
learning_rate = 0.01
```

3. **Load the training set and validation set using Dataset and DataLoader.**

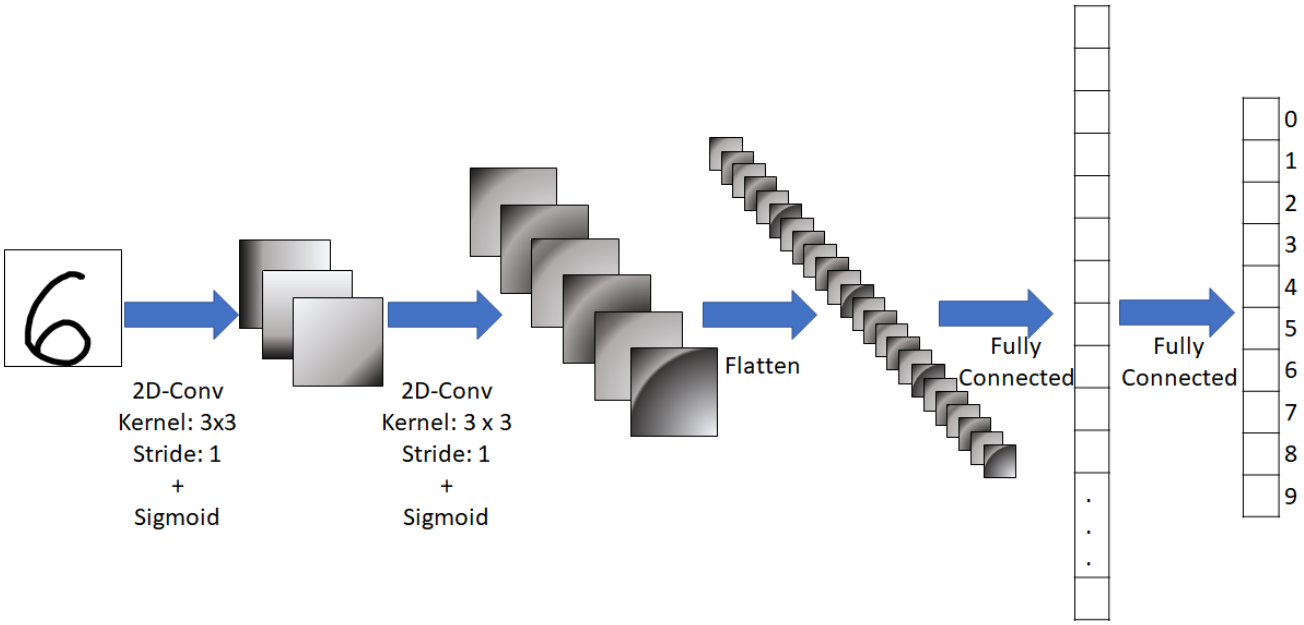
Using *ToTensor()*, convert the images into numbers, then convert the pixels of each image to a range between 0 and 255. These values are then scaled down to a range between 0 and 1.

Load dataset as training set and validation set using DataLoader

```
trans = torchvision.transforms.ToTensor()
```

```
train_data = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST(
        'mnist_data', train=True, download=True, transform=trans
        ), batch_size=batch_size
        )
val_data = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST(
        'mnist_data', train=False, download=True, transform=trans
        ), batch_size=batch_size)
```

### 4. Define the CNN for image classification




A sample CNN is defined here. Handwritten digits are the input to the model. Two convolution layers is followed by fully connected layers. Sigmoid activation is applied on the convolution results. The result from the CNN will be a set of 10 values corresponding to the 10 digits. The maximum value among these 10 values decides the class of the handwritten digit.

```
class ConvNet(nn.Module):
  def __init__(self):
    super(ConvNet, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=3, stri
de=1)
    self.conv2 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=3, stri
de=1)
```

```
    self.sigmoid = nn.Sigmoid()
    self.linear1 = nn.Linear(3456, 10)
  def forward(self, x):
    x = self.sigmoid(self.conv1(x))
    x = self.sigmoid(self.conv2(x))
    x = x.view(x.shape[0], -1)
    x = self.linear1(x)
    return x
```

5. **Define a function for validating the model**

To check whether the model is learning properly we can use a validation set. The accuracy of the model after each epoch can be verified using the following function.

```
def validate(model, data):
  total = 0
  correct = 0
  for i, (images, labels) in enumerate(data):
    images = images.cuda()
    labels = labels.cuda()
    y_pred = model(images)
    value, pred = torch.max(y_pred, 1)
    total += y_pred.size(0)
    correct += torch.sum(pred == labels)
    return correct * 100 / total
```

6. **Initialize the neural network and optimizer**

Initialize the model and transfer the network parameters to the appropriate device. The optimization algorithm chosen for this model is Adam optimizer. Pass model parameters and learning rate to the optimizer.

```
convnet = ConvNet().cuda()
optimizer = optim.Adam(convnet.parameters(), lr=learning_rate)
```

7. **Define loss function**

To penalize the misclassifications made by the model, cross entropy loss is used here.

```
cross_entropy = nn.CrossEntropyLoss()
```

8. **Training the model**
    a. Move the image-label pairs to the device
    b. During training, we need to explicitly set the gradients to zero using *optimizer.zero_grad()* before starting to do backpropragation (i.e., updation of Weights and biases) because PyTorch accumulates the gradients on subsequent backward passes.
    c. Do forward propagation by passing the image to the model and compute the loss based on the prediction.
    *d.* *loss.backward()* accumulates the gradient for each parameter.
    e. *optimizer.step()* performs a parameter update based on the current gradient and the update rule (here it uses Adam optimizer).
    f. Repeat these steps for all the batches in the training set, the entire training set will be processed "*epoch*" times.
    g. Compute the accuracy using the *validate()* function defined in step 5.

```
for n in range(epoch):
  for i, (images, labels) in enumerate(train_data):
    images = images.cuda()
    labels = labels.cuda()
    optimizer.zero_grad()
    prediction = convnet(images)
    loss = cross_entropy(prediction, labels)
    loss.backward()
    optimizer.step()
  accuracy = float(validate(convnet, val_data))
  print("Epoch:", n+1, "Loss: ", float(loss.data), "Accuracy:", accuracy)
```