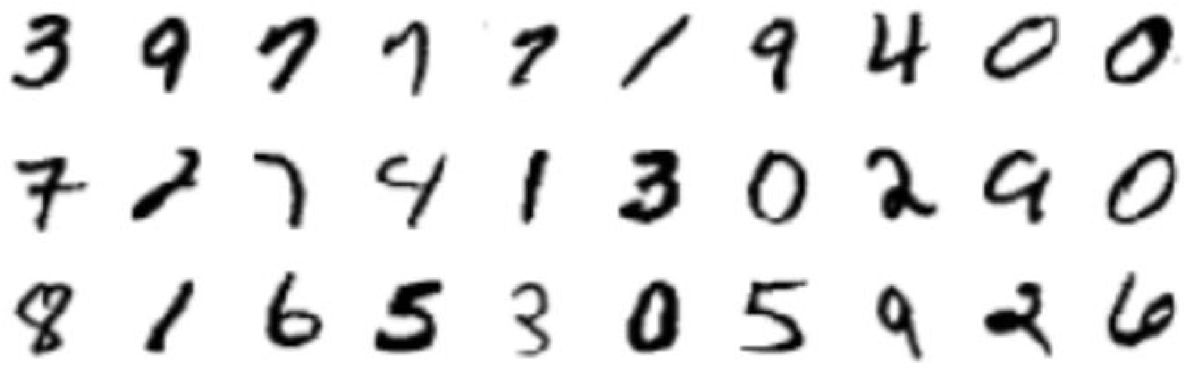
Build and train a CNN from scratch

Goal is to build a simple convolutional neural network (CNN) in Pytorch and train it from scratch for image classification. Using the MNIST dataset, we train the CNN to recognize handwritten digits to 98+% accuracy.

MNIST dataset:

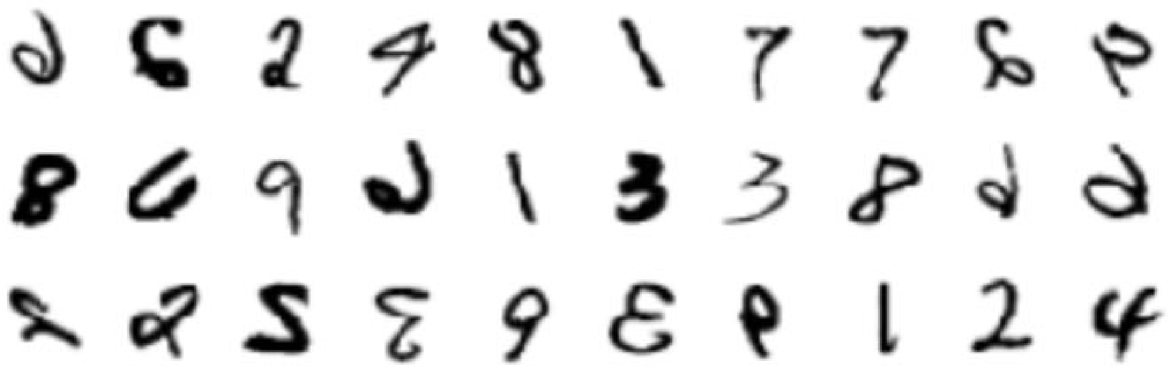
MNIST dataset contains a total of 70,000 images of handwritten digits (labels 0 to 9) with data split of 60,000 for training and 10,000 for testing. The images are grayscale, 28x28 pixels, and centered to reduce preprocessing. We will train and test the CNN on MNIST datasets with and without augmentation.

Visualization of MNIST without augmentation: (= normal)



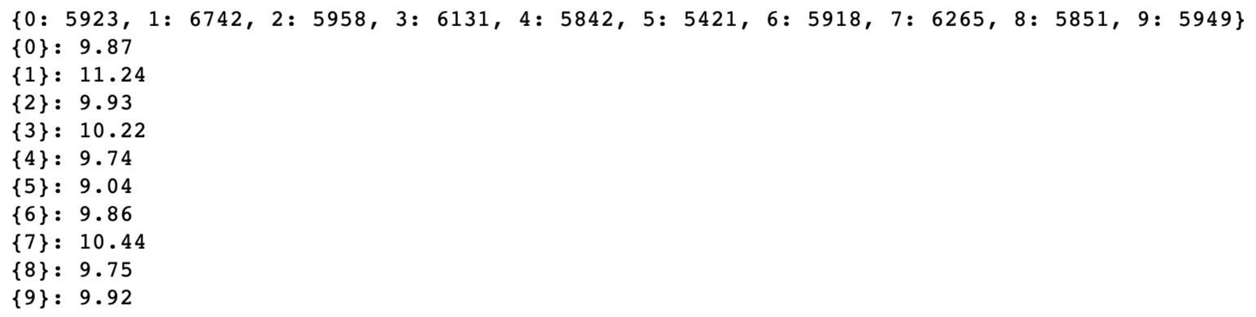
Visualization of MNIST with augmentation: (= augmented)
Numerous transforms were chained together in a list using transforms.Compose() function.

```
transform_aug = transforms.Compose(
                [transforms.Resize(32),
                 transforms.RandomHorizontalFlip(),
                 transforms.RandomCrop(size=32),
                 transforms.ToTensor()])
```



Class bias check:

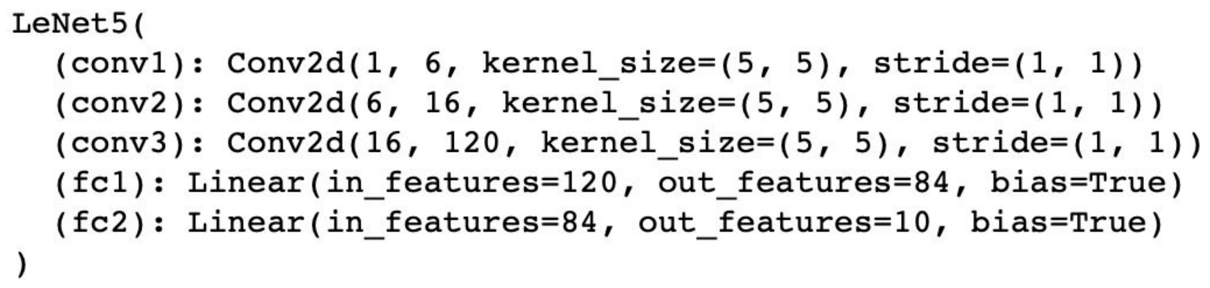
As seen below, the training dataset seems to be equally distributed among the 10 classes. The percentage distribution of training data in each class is roughly between 9% to 10%. This means that the dataset is balanced.



```
{0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7: 6265, 8: 5851, 9: 5949}
{0}: 9.87
{1}: 11.24
{2}: 9.93
{3}: 10.22
{4}: 9.74
{5}: 9.04
{6}: 9.86
{7}: 10.44
{8}: 9.75
{9}: 9.92
```

Building the network architecture:

Use LeNet-5 model which has three 2D convolutional layers followed by two fully-connected (or linear) layers. We create a neural network structure in PyTorch by creating a new class that inherits from the nn.Module superclass.

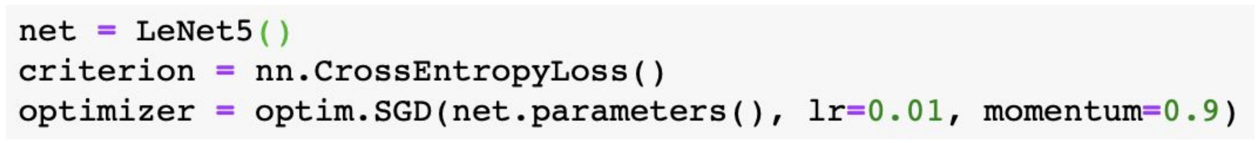


```
LeNet5(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=120, out_features=84, bias=True)
  (fc2): Linear(in_features=84, out_features=10, bias=True)
)
```

Description of training strategy:

Before training our model, we first set up some hyperparameters:
Batch size = 32
Epochs = 10
Learning rate for optimizer = 0.01 (default)

Next, we create an instance of our CNN class model which is randomly initialized by PyTorch, define our loss function and optimizer. We use the cross entropy loss function defined as the criterion to calculate the loss per epoch and use a SGD optimizer to minimize the loss. PyTorch keeps track of all the parameters within our model which are required to be trained and optimized and via net.parameters() these parameters can be passed into the optimizer function. The default learning rate and momentum is also passed into the optimizer function.



```
net = LeNet5()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

Note: the images from MNIST dataset have been flattened using .view() function, converted into tensors and loaded into data loader which is used as an iterator and used to shuffle, batch and load data in parallel.

Inside the train function, we first ensure that our model is set to training mode by running model.train(). Next, the training loop is created for one epoch in which we iterate through all the batches by looping over the train_loader. The data from train_loader is a tuple and thus can be separated into X (images) and y (target or ground truth labels).

Forward pass:
The X (images) obtained from the train_loader are passed through our model to generate outputs. Then the model outputs and ground truth labels are passed into the cross entropy loss function and the value is stored in a loss variable. The loss for each batch is summed per iteration and at the end of the for loop we return the cumulative loss divided by the number of batches which corresponds to the loss over one epoch.

Backprop and Optimize:
The next step is to perform back-propagation and an optimized training step. We first manually set the gradients to zero using the optimizer.zero_grad() since PyTorch by default accumulates gradients. Next, we call .backward() function on the loss variable to perform the back-propagation which fine-tunes the weights of our model based on the loss obtained over the epoch. Once the gradients have been calculated in the back-propagation, we call optimizer.step() to perform the SGD optimizer training step.

Track Accuracy:
The next step involves keeping track of the accuracy of the training dataset. The model outputs and ground truth labels are passed into our accuracy function. For each sample in the batch, the predictions of the model can be determined by using the argmax function, which returns the index of the maximum value in a tensor. The model prediction is then compared with the ground truth labels (predicted == labels) returning a boolean which is summed per batch to determine the number of correct predictions. To obtain the accuracy, the number of correct predictions is divided by the batch size (number of training data in the batch). The accuracy of per batch is summed per iteration and at the end of the for loop we return the cumulative accuracy divided by the number of batches which corresponds to the accuracy over one epoch.

Training over epochs:
Finally, we loop over the number of epochs (epochs = 10) and train our model by running the train function (training loop over one epoch). The loss per epoch is appended to a list and used to plot the progress of the training. After every iteration, the loss and the accuracy of the model is printed per epoch. The stopping criteria of the training process is both loss and epochs. We trained two different datasets of MNIST - one without any augmentation (train-normal) and one with augmentation (train-augmented). Below are the training outputs over 10 epochs for both datasets.

Training Outputs over 10 epochs:

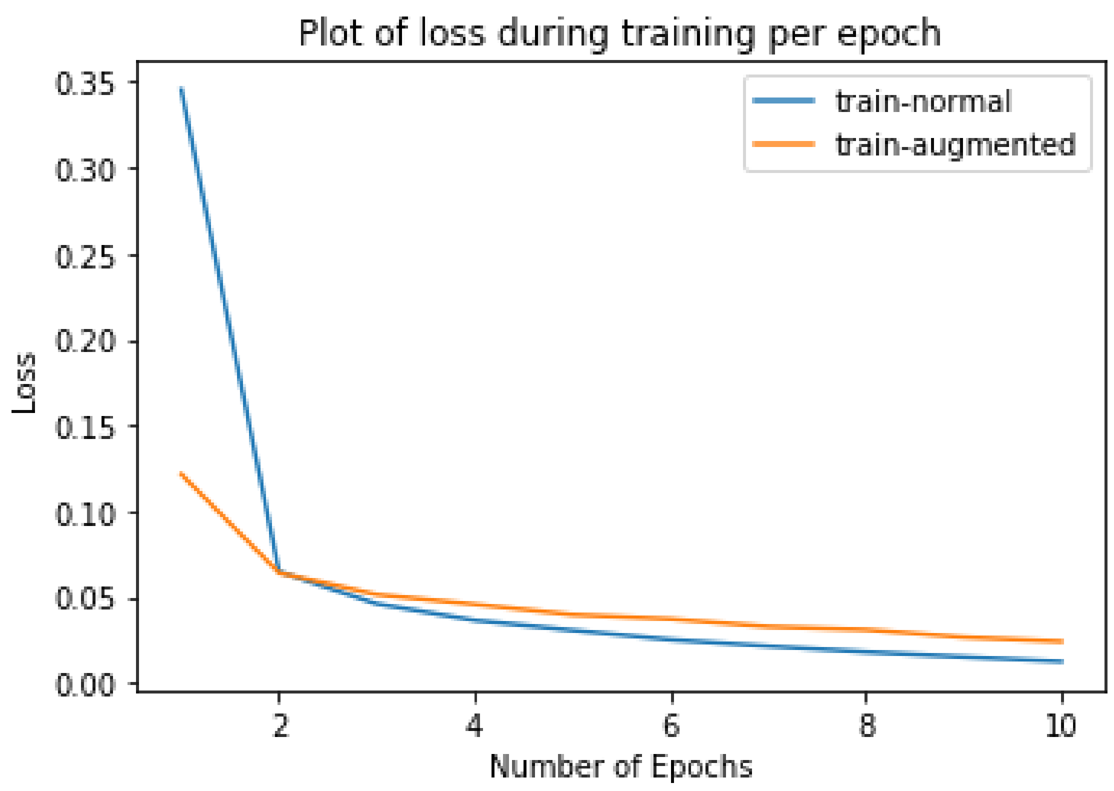
No Augmentation:

```
Epoch: 01 | Epoch Time: 0m 23s
        Train Loss: 0.345 | Train Acc: 88.57%
Epoch: 02 | Epoch Time: 0m 25s
        Train Loss: 0.065 | Train Acc: 98.00%
Epoch: 03 | Epoch Time: 0m 25s
        Train Loss: 0.046 | Train Acc: 98.53%
Epoch: 04 | Epoch Time: 0m 25s
        Train Loss: 0.036 | Train Acc: 98.83%
Epoch: 05 | Epoch Time: 0m 25s
        Train Loss: 0.031 | Train Acc: 99.01%
Epoch: 06 | Epoch Time: 0m 24s
        Train Loss: 0.025 | Train Acc: 99.20%
Epoch: 07 | Epoch Time: 0m 23s
        Train Loss: 0.021 | Train Acc: 99.32%
Epoch: 08 | Epoch Time: 0m 25s
        Train Loss: 0.018 | Train Acc: 99.44%
Epoch: 09 | Epoch Time: 0m 25s
        Train Loss: 0.015 | Train Acc: 99.54%
Epoch: 10 | Epoch Time: 0m 25s
        Train Loss: 0.012 | Train Acc: 99.60%
```

Augmentation:

```
Epoch: 01 | Epoch Time: 0m 26s
        Train Loss: 0.121 | Train Acc: 96.31%
Epoch: 02 | Epoch Time: 0m 26s
        Train Loss: 0.064 | Train Acc: 97.97%
Epoch: 03 | Epoch Time: 0m 26s
        Train Loss: 0.051 | Train Acc: 98.36%
Epoch: 04 | Epoch Time: 0m 26s
        Train Loss: 0.046 | Train Acc: 98.55%
Epoch: 05 | Epoch Time: 0m 27s
        Train Loss: 0.040 | Train Acc: 98.70%
Epoch: 06 | Epoch Time: 0m 26s
        Train Loss: 0.037 | Train Acc: 98.84%
Epoch: 07 | Epoch Time: 0m 26s
        Train Loss: 0.033 | Train Acc: 98.95%
Epoch: 08 | Epoch Time: 0m 27s
        Train Loss: 0.031 | Train Acc: 99.02%
Epoch: 09 | Epoch Time: 0m 26s
        Train Loss: 0.026 | Train Acc: 99.17%
Epoch: 10 | Epoch Time: 0m 25s
        Train Loss: 0.024 | Train Acc: 99.21%
```

Training curve of loss against epochs:



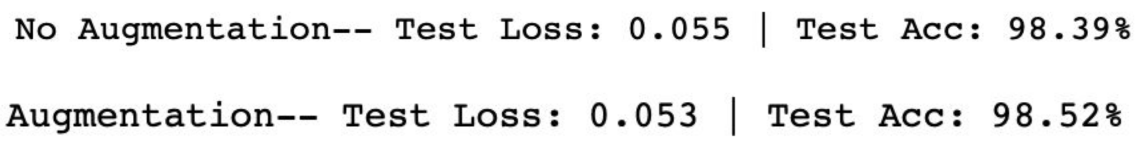
Description of testing strategy:

Inside the test function, we first ensure our model is set to evaluation mode by running model.eval(). The torch.no_grad() statement disables the autograd functionality in the model. Note: both back-propagation step and optimizer step is skipped during the model testing and the model is tested only once, therefore we do not iterate over epochs. The rest is the same as the training, except that we iterate over test_loader. Another difference is that we created two lists, correct_idx and wrong_idx, which store the indices of the correct and wrong predictions made by the model. This information is later used to visualize correct and wrong predictions.

```
# Store index of correct and wrong predictions
pred = output.argmax(dim=1, keepdim=True)
preds.append(pred)
c = [i for i, val in enumerate(pred.eq(target.view_as(pred))) if (val==True)]
w = [i for i, val in enumerate(pred.eq(target.view_as(pred))) if (val==False)]
correct_idx.append(c)
wrong_idx.append(w)
```

Testing Accuracy:

We evaluate our model on the test dataset (10,000 images). We tested two different datasets of MNIST - one without any augmentation and one with augmentation. The loss and accuracy for both datasets are displayed below.



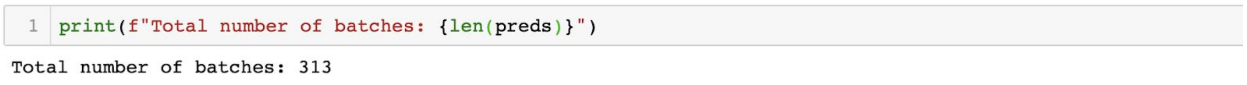
```
No Augmentation-- Test Loss: 0.055 | Test Acc: 98.39%

Augmentation-- Test Loss: 0.053 | Test Acc: 98.52%
```

Conclusion:

The model quickly achieves a high degree of accuracy on the training set and after training over 10 epochs, it achieves a test accuracy of 98+%.

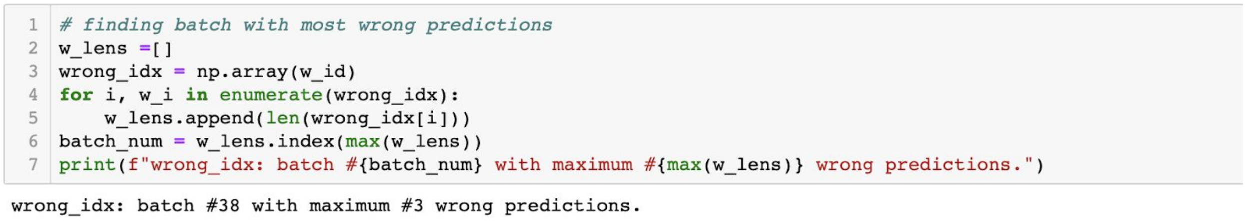
Visualization of correct and wrong predictions:

Visualizations are only provided for the MNIST test dataset without any augmentation.
Since a batch size of 32 was used, the total number of batches were 313. In order to plot the correct and wrong predictions, we write code to find the batch# with most wrong predictions and least correct predictions (this should be the same batch number!).



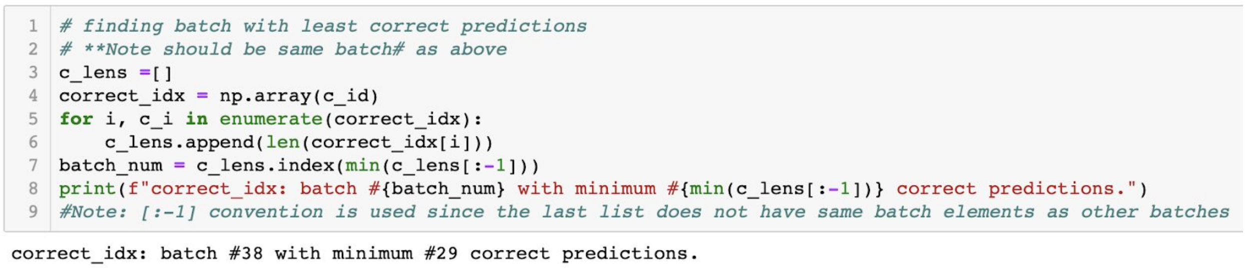
```python
print(f"Total number of batches: {len(preds)}")
```

```
Total number of batches: 313
```



```python
# finding batch with most wrong predictions
w_lens =[]
wrong_idx = np.array(w_id)
for i, w_i in enumerate(wrong_idx):
    w_lens.append(len(wrong_idx[i]))
batch_num = w_lens.index(max(w_lens))
print(f"wrong_idx: batch #{batch_num} with maximum #{max(w_lens)} wrong predictions.")
```

```
wrong_idx: batch #38 with maximum #3 wrong predictions.
```

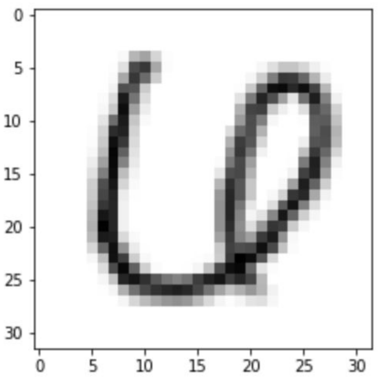


```python
# finding batch with least correct predictions
# **Note should be same batch# as above
c_lens =[]
correct_idx = np.array(c_id)
for i, c_i in enumerate(correct_idx):
    c_lens.append(len(correct_idx[i]))
batch_num = c_lens.index(min(c_lens[:-1]))
print(f"correct_idx: batch #{batch_num} with minimum #{min(c_lens[:-1])} correct predictions.")
#Note: [:-1] convention is used since the last list does not have same batch elements as other batches
```

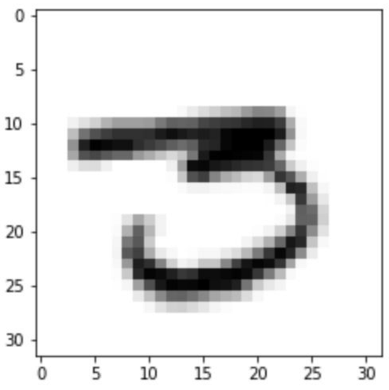
```
correct_idx: batch #38 with minimum #29 correct predictions.
```

Visualization of wrong predictions:

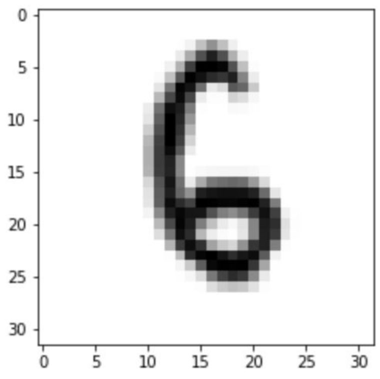
Batch #38 with 3 wrong predictions:



```
The target label is 6.
The prediction label is 0.
```

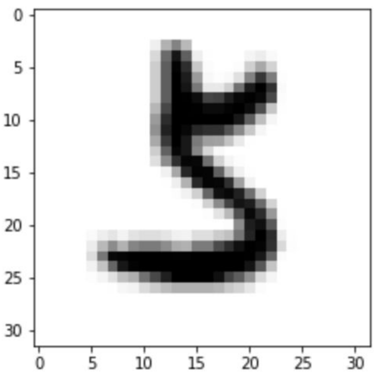


```
The target label is 3.
The prediction label is 5.
```

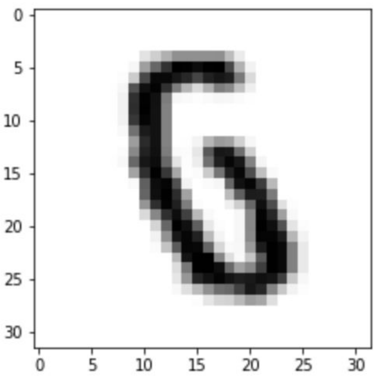


```
The target label is 6.
The prediction label is 2.
```

Batch #2 with 2 wrong predictions:



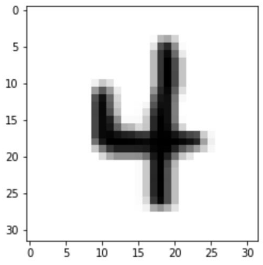
```
The target label is 5.
The prediction label is 2.
```



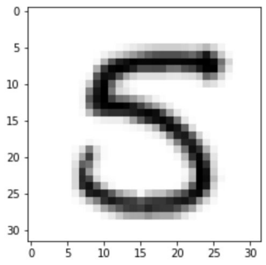
```
The target label is 6.
The prediction label is 0.
```

Visualization of correct predictions:

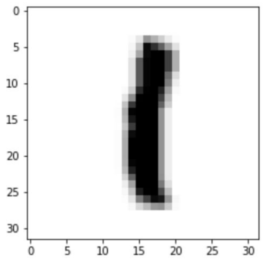
Batch #38 with 29 correct predictions: (only some displayed)



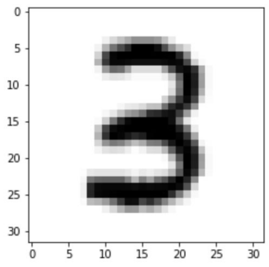
```
The target label is 4.
The prediction label is 4.
```



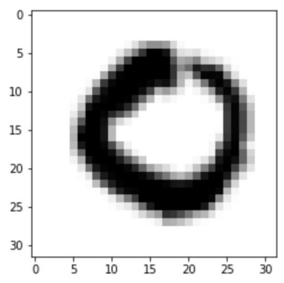
```
The target label is 5.
The prediction label is 5.
```



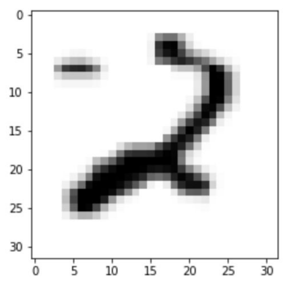
```
The target label is 1.
The prediction label is 1.
```



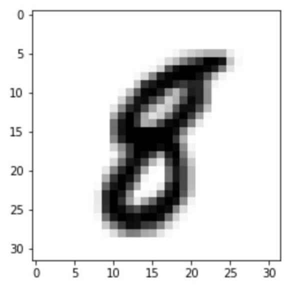
```
The target label is 3.
The prediction label is 3.
```



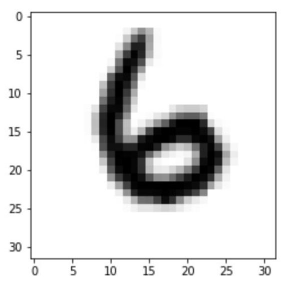
The target label is 0.
The prediction label is 0.



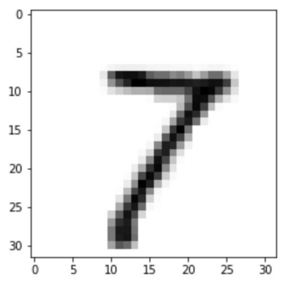
The target label is 2.
The prediction label is 2.



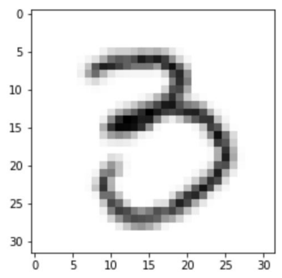
The target label is 8.
The prediction label is 8.



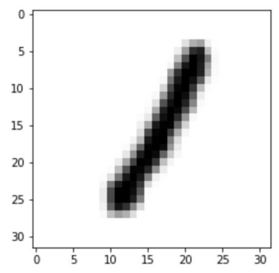
The target label is 6.
The prediction label is 6.



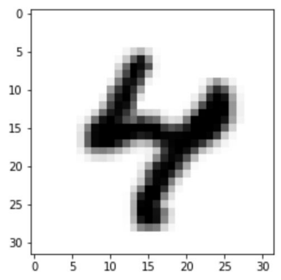
The target label is 7.
The prediction label is 7.



The target label is 3.
The prediction label is 3.



The target label is 1.
The prediction label is 1.



The target label is 4.
The prediction label is 4.