Solutions

In order to solve the problem of image classification for two specific data sets, I have implemented two machine learning models, namely a classic feedforward network, commonly known as *multilayer perceptron* (MLP) and a *convolutional neural network* (CNN). Subsequently, I have run tests on both networks, on different settings, and compared the results between settings for each model and then against each other.

Multilayer Perceptron

This network is built from multiple layers of artificial neurons, connected in a linear way. The first layer corresponds to the input data, which, in our case, is comprised of as many nodes as pixel color values per image. There can be one ore more hidden layers in-between the input layer and the output layer, the latter having a number of nodes that represent each label from the given set.

Learning is achieved by the *back-propagation* technique, using the output error to adjust weights between connections repeatedly until convergence (or desirable state). For proper weight adjustment, an optimization algorithm is used, typically *stochastic gradient descent*, but in light of most recent studies, I have implemented the MLP using the **Adam** optimizer, a more efficient, adaptive algorithm.

Furthermore, **ReLU** was used for the neuron activation function and **Cross Entropy Loss** as the loss/error function (named criterion in PyTorch).

Convolutional Neural Network

The CNN expands on the previous layered architecture by introducing **convolutional layers**, which contain a set of filters (or kernels) - matrices that perform a convolution operation on input images, throughputting a set of filtered/convolved images (of generally smaller size) with the aim of extracting patterns from said images as *features*.

In addition, there are **subsampling layers** (also known as pooling layers) that have the role of shrinking the resolution of images given as input, usually after a convolution, so as to reduce the computational cost of convolution operations.

After a sequence of convolution and pooling layers, the throughput can be linearly reduced to the output layer, in order to obtain the estimated label, through zero or more classic hidden layers or other types of layers. This particular implementation uses the same functions and algorithms as the previous one, only differing in layer structure and composition.

Experiments

Configuration

Datasets

- MNIST database
  - handwritten digits, very simple and widely used for image processing
  - labels represented by digits 0-9
  - 60,000 training images; 10,000 test images
  - 28x28 resolution, grayscale (0-255)
- Fashion-MNIST
  - fashion items, a more challenging alternative to MNIST
  - labels represented by 10 types of clothes, boots and bags
  - 60,000 training images; 10,000 test images
  - 28x28 resolution, grayscale (0-255)

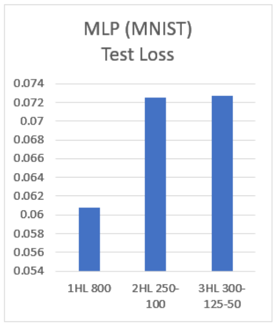
Running settings

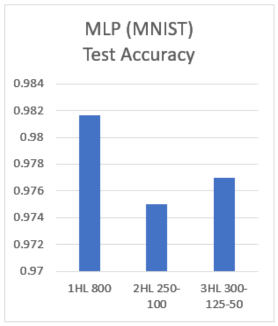
- all data is normalized before training
- training data is augmented through small random cropping (padded back to size) and rotations, to add more randomness in samples
- a random 10% of the train set is chosen as validation set (augmentation does not apply)
- batch size for each iteration equals 1024 (running on CUDA)
- three MLP configurations
  - 1 hidden layer - 800 neurons
  - 2 hidden layers - 250-100 neurons
  - 3 hidden layers - 300-125-50 neurons
- three CNN configurations [1]
  - 1st = [1,28,28] - [6,24,24] - [6,12,12] - [16,8,8] - [16,4,4] - 256-120-84
  - 2nd = [1,28,28] - [2,20,20] - [2,10,10] - [32,8,8] - [32,4,4] - 512-200-64
  - 3rd = [1,28,28] - [32,28,28] - [32,14,14] - [64,14,14] - [64,7,7] - 3136-128
- training iterates through 10 epochs and 20 for CNNs because of slower convergence
- the model with lowest validation loss is stored for the final test set

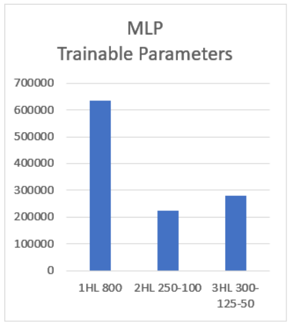
[1] written as [depth,width,height] tuples; operations that alter depth are convolutions while those alter just width and height (by halving) are subsamplings

Results

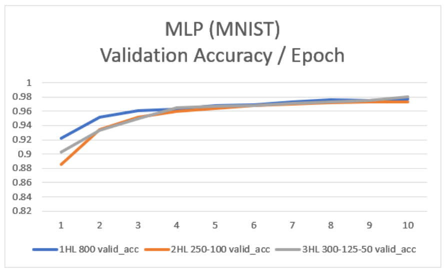
For the standard MNIST dataset, the classic MLP can achieve 97-98% accuracy. Based on the charts below, there is a clear correlation between loss, accuracy and number of trainable parameters. It seems that the strongest network is the one with the most artificial neurons, regardless of hidden layer size and distribution.

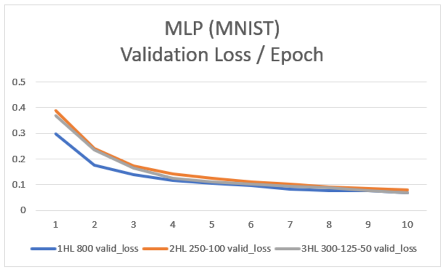




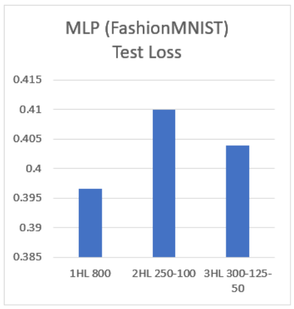


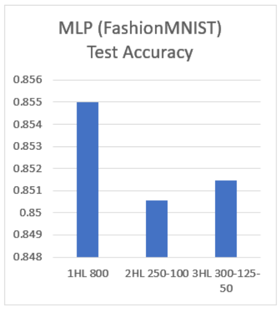
The 800-neuron MLP also stands out as the fastest converging loss, with a corresponding steeper slope of accuracy over 10 epochs.

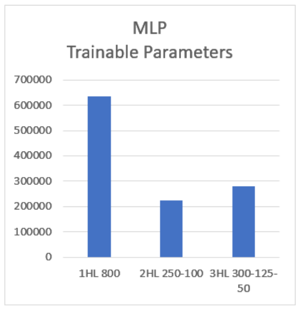


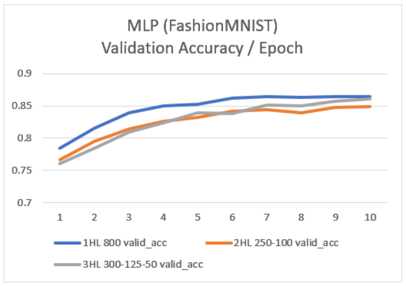


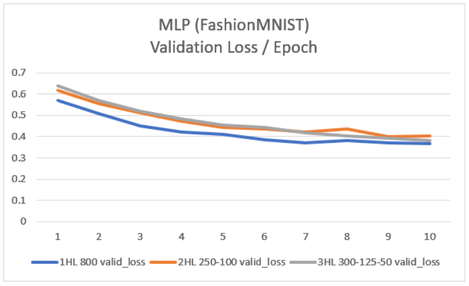
The Fashion-MNIST dataset proved to be a harder task to accomplish for the MLP. The charts show the same correlation as before, although performance is noticeably poorer.







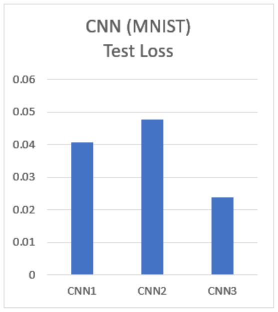


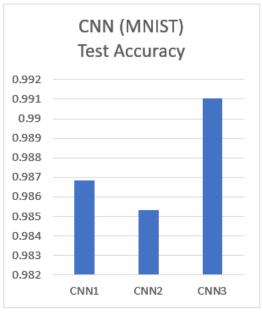


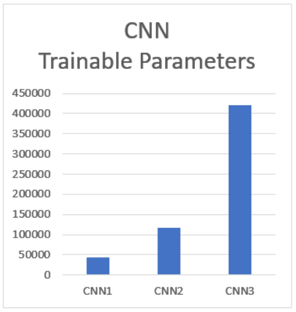
In the case of CNN, it would be of note that overall performance is not dictated by the network total connections, but structure is more important. The 1st CNN clearly outperforms the 2nd, with less than a half of trainable parameters, but the convolutional layers do a better job of extracting patterns by gradually increasing filter size in a 1-6-16 fashion, instead of 1-2-32.

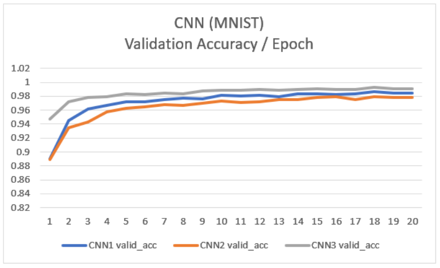
The 3rd CNN expands more and faster to 1-32-64, paired with faster linear 3136-128-10 reduction, at the computational cost of over 400,000 trainable parameters.

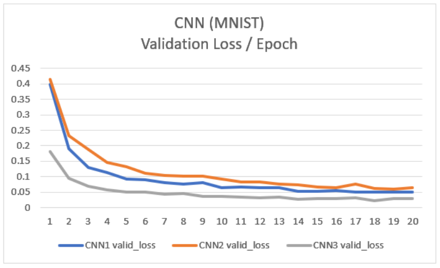
Nevertheless, any implementation of the above outperforms all of the previous MLP variants, with over 98.5% accuracy. The 3rd CNN has an impressive 99.1% accuracy.

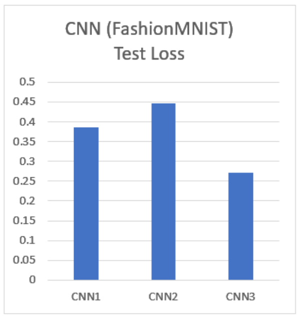


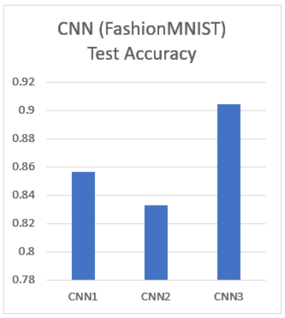


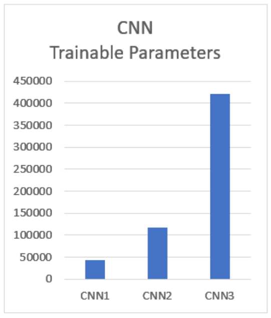


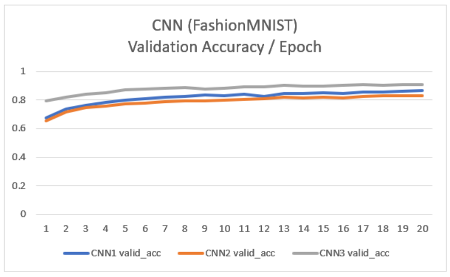


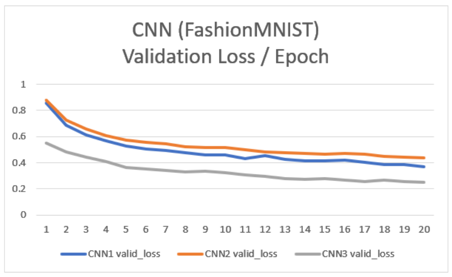












The more challenging dataset provides a clear distinction of loss convergence and overall performance of the 3rd CNN implementation.

As a concluding comparison of this experiment, CNN is a more complex and more suited ML model for image classification than the MLP.

Usage and Tweaking

Code was written using the *Jupyter Notebook* environment, which allows running cells of lines of code individually, in a sequential manner, making it easier to test different configurations without running the entire script over and over again.

Thus, small changes in some sections can made to test the networks on different settings. The implementation of both scripts (MLP and CNN) mainly use the same cells of code, except for the specific architecture.

The ratio of samples that remain in the train set can be changed to a different fraction to ensure a lower chance of overfitting.

```
VALID_RATIO = 0.9

n_train_examples = int(len(train_data) * VALID_RATIO)
n_valid_examples = len(train_data) - n_train_examples

train_data, valid_data = data.random_split(train_data,
                                           [n_train_examples,
                                           n_valid_examples])
```

A different batch size can be considered in case of poor computational performance of the device or other logic reasons.

```
BATCH_SIZE = 1024

train_iterator = data.DataLoader(train_data,
                                 shuffle = True,
                                 batch_size = BATCH_SIZE)
valid_iterator ...
...
```

Moreover, one can easily experiment with some other criterion (loss function) and optimizer algorithm, as well as run the training for more epochs in case of slow convergence. Note that a device with CUDA capability could run noticeably faster.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
criterion = criterion.to(device)
EPOCHS = 10
for epoch in range(EPOCHS):
...
```

It is also very easy and fast to change the structure of the implemented network, as PyTorch provides a simple and flexible API to do so. In the specific case below, we define two layers in-between input and output with 250 and, respectively, 100, then implement the feed-forward function that passes on activated nodes from layer to layer.

```
H_1_DIM = 250
H_2_DIM = 100
class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.input_fc = nn.Linear(input_dim, H_1_DIM)
        self.hidden_fc_1 = nn.Linear(H_1_DIM, H_2_DIM)
        self.output_fc = nn.Linear(H_2_DIM, output_dim)
    def forward(self, x):
        batch_size = x.shape[0]
        x = x.view(batch_size, -1)
        h_1 = F.relu(self.input_fc(x))
        h_2 = F.relu(self.hidden_fc_1(h_1))
        y_pred = self.output_fc(h_2)
        return y_pred, h_2
```

Changing the structure of the CNN implementation might prove more difficult, as it is important to keep track of image size and number of filters that convolved the images, as well as the same dimensions that result after each layer. In the end, we want to linearly flatten all the convolved images into a fixed dimension (the label set size).

An important note of computational value is that activation is always better to be called after a pooling layer, because the results are the same as vice-versa, but there are less computations to be done after shrinking images.

```
class CNN(nn.Module):
    def __init__(self, output_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels = 1,
                               out_channels = 32,
                               kernel_size = 3,
                               padding = 1)
        self.conv2 = nn.Conv2d(in_channels = 32,
                               out_channels = 64,
                               kernel_size = 3,
                               padding = 1)
        self.fc_1 = nn.Linear(3136, 128)
        self.fc_2 = nn.Linear(128, output_dim)
    def forward(self, x):
        x = self.conv1(x)
        x = F.max_pool2d(x, kernel_size = 2)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.max_pool2d(x, kernel_size = 2)
        x = F.relu(x)
        x = x.view(x.shape[0], -1)
        h = x
        x = self.fc_1(x)
        x = F.relu(x)
        x = self.fc_2(x)
        return x, h
```