# Handwritten Digits Classification using Neural Networks

**CNN_MNIST-data.ipynb**

## CNN(Convolutional Neural Network)

This report presents the implementation of a Convolutional Neural Network (CNN) classifier on the MNIST dataset using PyTorch. The model was created from scratch (without using high-level libraries like Keras or Scikit-learn) to recognize handwritten digits from 0 to 9. The MNIST dataset is widely used for benchmarking image classification algorithms and consists of 60,000 training images and 10,000 test images.

### Multiple Neural Network Models Implemented

In this project, we implemented and tested multiple types of neural network architectures on the **same MNIST dataset**, using consistent preprocessing and evaluation pipelines. The models were:

1. Fully Connected Neural Network (FCNN)
2. Convolutional Neural Network (CNN)
3. Recurrent Neural Network (RNN)
4. Artificial Neural Network (ANN )

➔ **ANN vs CNN (Artificial Neural Network vs Convolutional Neural Network)**

While both ANN and CNN can be used for image classification tasks like MNIST, CNN significantly outperforms ANN in terms of accuracy and learning capability. ANN is a shallow, fully connected model that flattens the input image and treats all pixels independently, without considering the spatial relationships between them. This leads to a loss of crucial structural information inherent in image data. In contrast, CNNs use convolutional layers to learn localized patterns like edges and textures, followed by pooling layers that reduce dimensionality while retaining essential features. As a result, CNNs achieve much higher test accuracy (99.01%) compared to ANNs (97.83%). Furthermore, while ANNs train faster due to fewer layers and simpler operations, they tend to underperform on complex visual data, making CNNs the preferred choice for image-related tasks.

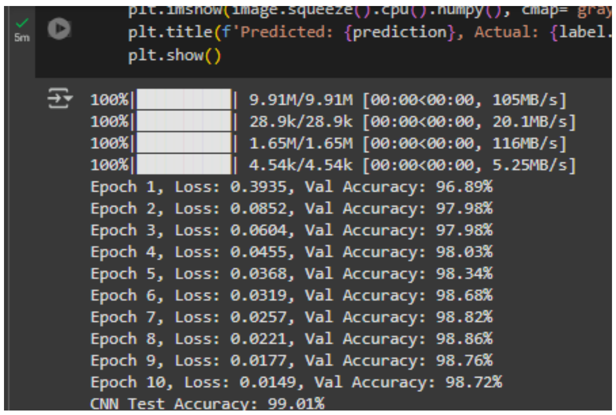
➔ **FCNN vs CNN (Fully Connected Neural Network vs Convolutional Neural Network)**

FCNNs are deeper versions of ANNs with multiple hidden layers, allowing them to learn more complex representations than shallow networks. However, like ANNs, FCNNs still treat images as flat vectors, ignoring the spatial layout of pixels. This results in a large number of parameters, increasing the risk of overfitting and demanding more computational resources. CNNs, on the other hand, are more efficient due to weight sharing in convolutional filters and benefit from spatial feature extraction through hierarchical layers. Although FCNNs can reach reasonable test accuracies (97.80%) on MNIST, they still fall short of the performance and efficiency offered by CNNs. CNNs remain superior in capturing image structure with fewer parameters and better generalization.

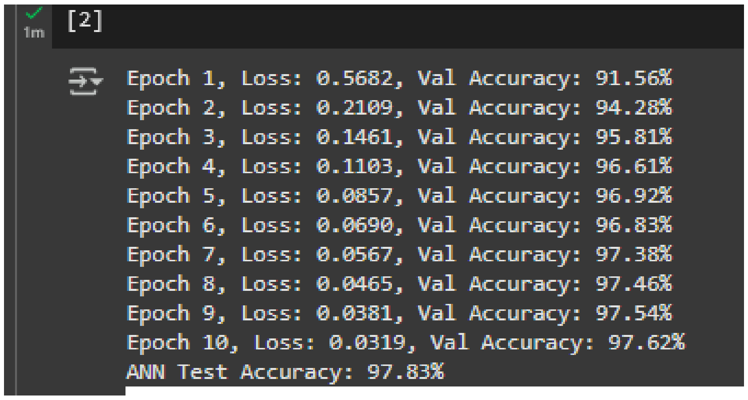
## ➔ RNN vs CNN **(Recurrent Neural Network vs Convolutional Neural Network)**

RNNs are designed for sequential data and are typically used in tasks involving time series, speech, or natural language processing. When applied to images like MNIST, each row of pixels can be treated as a time step, making it possible to train an RNN by viewing the image as a sequence of 28 vectors (one per row). While this allows RNNs to process images in a stepwise fashion, they are not inherently suited for spatial pattern recognition. As a result, they tend to be slower and slightly less accurate (95.25%) compared to CNNs. CNNs excel at detecting local spatial features and translating them across an image, making them faster, more accurate, and better optimized for image classification. Although RNNs offer a unique way to model sequential data, CNNs are still the better choice for most vision-related tasks.

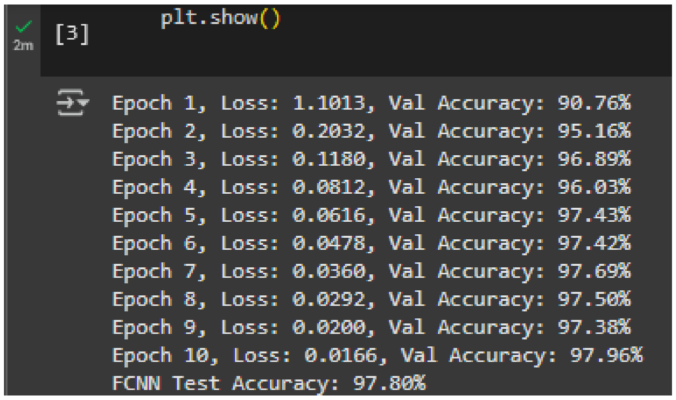
# OUTPUT



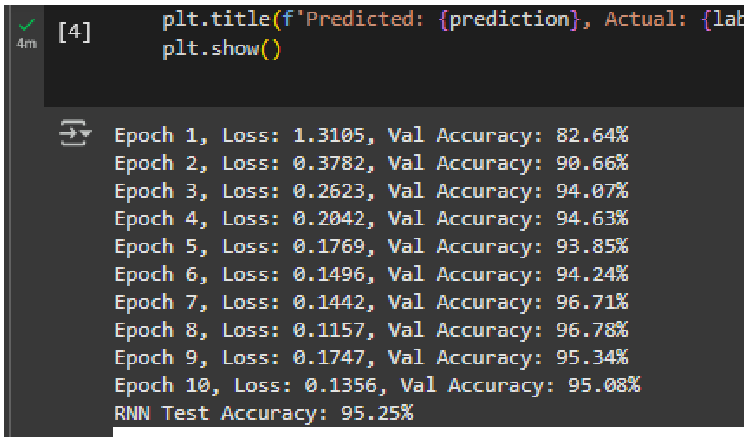
```
    plt.title(f'Predicted: {prediction}, Actual: {label.
    plt.show()

100%|██████████| 9.91M/9.91M [00:00<00:00, 105MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 20.1MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 116MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 5.25MB/s]
Epoch 1, Loss: 0.3935, Val Accuracy: 96.89%
Epoch 2, Loss: 0.0852, Val Accuracy: 97.98%
Epoch 3, Loss: 0.0604, Val Accuracy: 97.98%
Epoch 4, Loss: 0.0455, Val Accuracy: 98.03%
Epoch 5, Loss: 0.0368, Val Accuracy: 98.34%
Epoch 6, Loss: 0.0319, Val Accuracy: 98.68%
Epoch 7, Loss: 0.0257, Val Accuracy: 98.82%
Epoch 8, Loss: 0.0221, Val Accuracy: 98.86%
Epoch 9, Loss: 0.0177, Val Accuracy: 98.76%
Epoch 10, Loss: 0.0149, Val Accuracy: 98.72%
CNN Test Accuracy: 99.01%
```



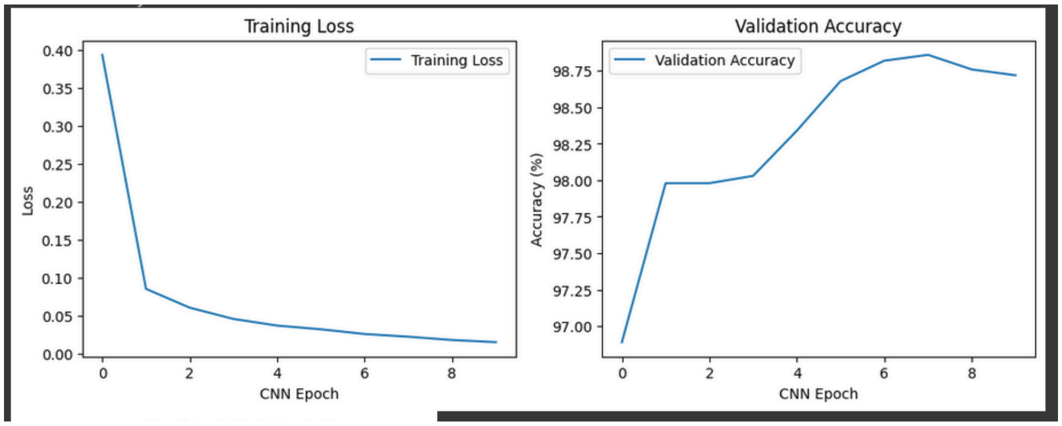
```
[2]

Epoch 1, Loss: 0.5682, Val Accuracy: 91.56%
Epoch 2, Loss: 0.2109, Val Accuracy: 94.28%
Epoch 3, Loss: 0.1461, Val Accuracy: 95.81%
Epoch 4, Loss: 0.1103, Val Accuracy: 96.61%
Epoch 5, Loss: 0.0857, Val Accuracy: 96.92%
Epoch 6, Loss: 0.0690, Val Accuracy: 96.83%
Epoch 7, Loss: 0.0567, Val Accuracy: 97.38%
Epoch 8, Loss: 0.0465, Val Accuracy: 97.46%
Epoch 9, Loss: 0.0381, Val Accuracy: 97.54%
Epoch 10, Loss: 0.0319, Val Accuracy: 97.62%
ANN Test Accuracy: 97.83%
```

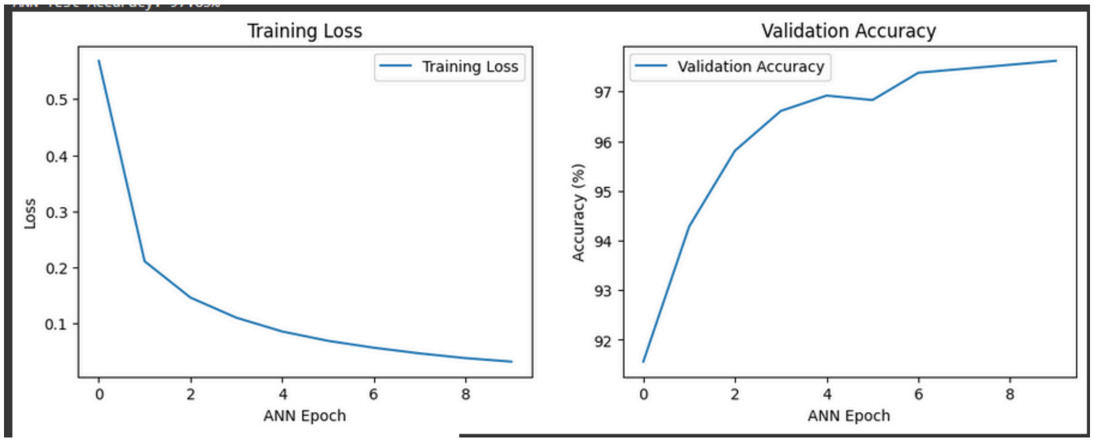


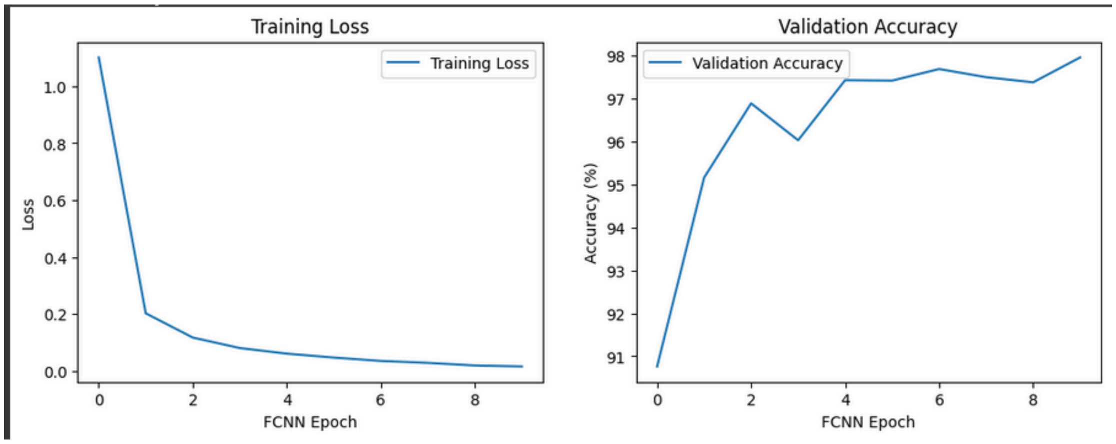
```
[3]     plt.show()

Epoch 1, Loss: 1.1013, Val Accuracy: 90.76%
Epoch 2, Loss: 0.2032, Val Accuracy: 95.16%
Epoch 3, Loss: 0.1180, Val Accuracy: 96.89%
Epoch 4, Loss: 0.0812, Val Accuracy: 96.03%
Epoch 5, Loss: 0.0616, Val Accuracy: 97.43%
Epoch 6, Loss: 0.0478, Val Accuracy: 97.42%
Epoch 7, Loss: 0.0360, Val Accuracy: 97.69%
Epoch 8, Loss: 0.0292, Val Accuracy: 97.50%
Epoch 9, Loss: 0.0200, Val Accuracy: 97.38%
Epoch 10, Loss: 0.0166, Val Accuracy: 97.96%
FCNN Test Accuracy: 97.80%
```

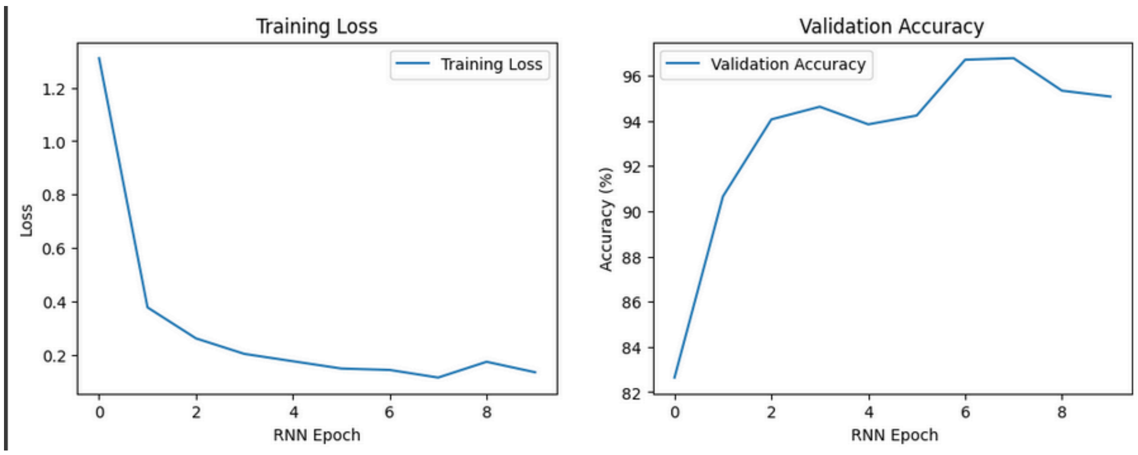


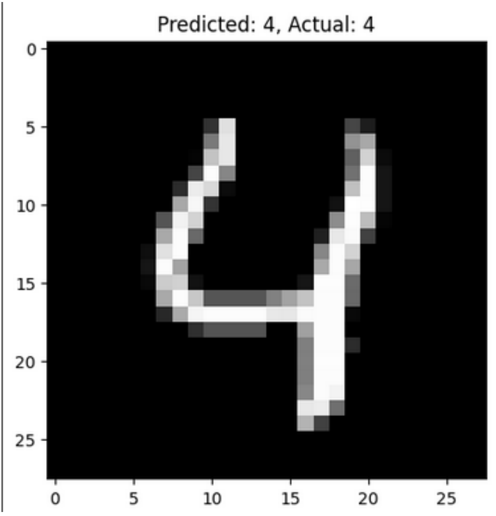
```
[4]     plt.title(f'Predicted: {prediction}, Actual: {lab
        plt.show()

Epoch 1, Loss: 1.3105, Val Accuracy: 82.64%
Epoch 2, Loss: 0.3782, Val Accuracy: 90.66%
Epoch 3, Loss: 0.2623, Val Accuracy: 94.07%
Epoch 4, Loss: 0.2042, Val Accuracy: 94.63%
Epoch 5, Loss: 0.1769, Val Accuracy: 93.85%
Epoch 6, Loss: 0.1496, Val Accuracy: 94.24%
Epoch 7, Loss: 0.1442, Val Accuracy: 96.71%
Epoch 8, Loss: 0.1157, Val Accuracy: 96.78%
Epoch 9, Loss: 0.1747, Val Accuracy: 95.34%
Epoch 10, Loss: 0.1356, Val Accuracy: 95.08%
RNN Test Accuracy: 95.25%
```


Training Loss
Training Loss
Loss
CNN Epoch
Validation Accuracy
Validation Accuracy
Accuracy (%)
CNN Epoch

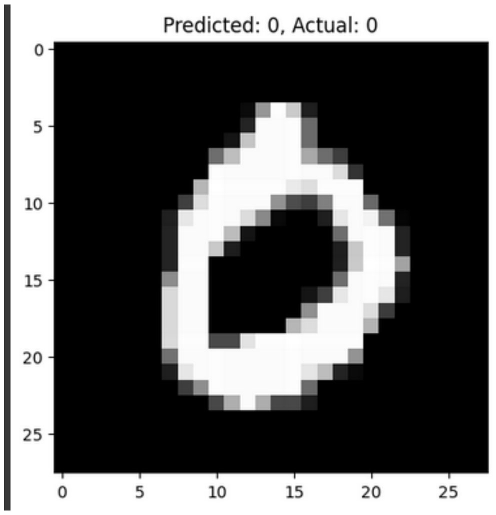

Training Loss
Training Loss
Loss
ANN Epoch
Validation Accuracy
Validation Accuracy
Accuracy (%)
ANN Epoch


Training Loss
Training Loss
Loss
FCNN Epoch
Validation Accuracy
Validation Accuracy
Accuracy (%)
FCNN Epoch

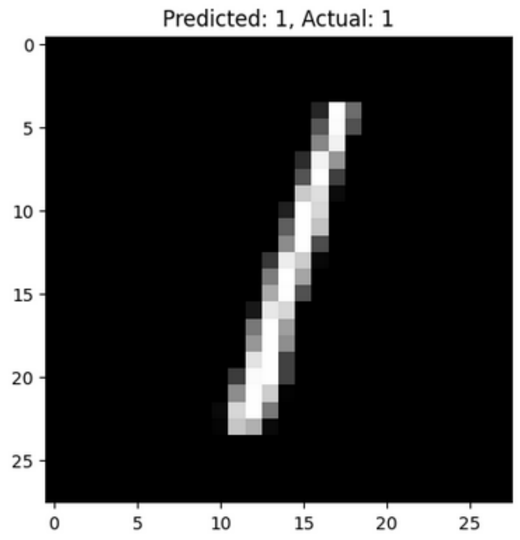

Training Loss
Training Loss
Loss
RNN Epoch
Validation Accuracy
Validation Accuracy
Accuracy (%)
RNN Epoch



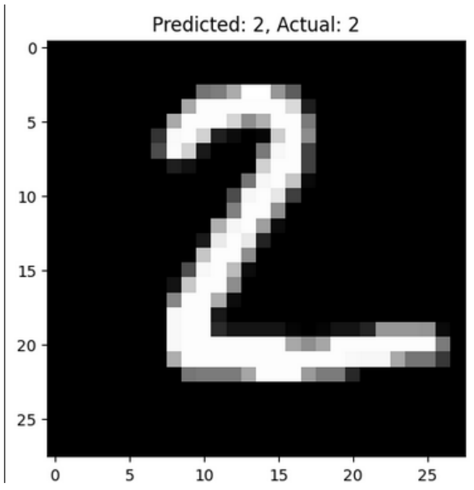
Predicted: 4, Actual: 4



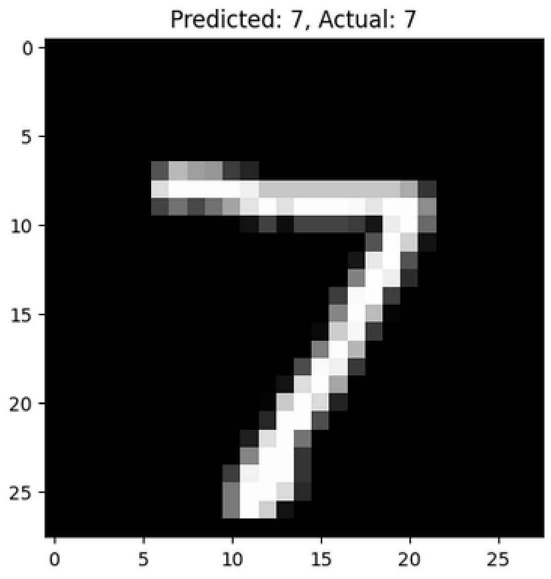
Predicted: 0, Actual: 0



Predicted: 1, Actual: 1



Predicted: 2, Actual: 2



Predicted: 7, Actual: 7

## Architecture Code

(core implementation of each model)

## CNN

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16,
kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32,
kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
      return x
```

## ANN

```
class ANN(nn.Module):
    def __init__(self):
        super(ANN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

## FCNN

```
class FCNN(nn.Module):
    def __init__(self):
        super(FCNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 64)
        self.fc5 = nn.Linear(64, 10)
        self.relu = nn.ReLU()
    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.relu(self.fc4(x))
        x = self.fc5(x)
        return x
```

## RNN

```
class RNN(nn.Module):
    def __init__(self, input_size=28, hidden_size=128,
num_layers=2, num_classes=10):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size,
num_layers, batch_first=True, nonlinearity='relu')
        self.fc = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        x = x.squeeze(1)  # Shape: (batch_size, 28, 28)
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])
        return out
```

- ## Performance Comparison

| **Model Type** | **Architecture Highlights** | **Test Accuracy (%)** |
|---|---|---|
| FCNN | Input → Dense(128) → ReLU → Dense(10) | 97.80% |
| CNN | Conv(16) → ReLU → MaxPool → Conv(32) → ... | 99.01% |
| RNN | Sequence of pixels as input → GRU / LSTM | 95.25% |
| ANN | Input → Dense(128) → ReLU → Dense(10) | 97.83% |

=> Most Accurate model is CNN with 99.01% accuracy

- ## Time Comparison

| **Model Type** | **Time (in min)** |
|---|---|
| CNN | 5 m |
| ANN | 1 m |
| FCNN | 2 m |
| RNN | 4 m |

**=>ANN** is the **fastest** to train and predict , within 1 min,  because it's just matrix math.