Neural Network Assignment

Group of Aline, Norma, Samuel and Elise

Introduction

Aim: create a NN model or use a pretrained one and explore its behavior.

- Dataset: MNIST image data
- Approaches:
  - Paper PDF (Aline)
  - Google Gemini (Elise)
  - Chat GPT (Sam)
  - Claude (Norma)
- Prompt for (2,3,4):

> I need to create a neural network model in Python for my studies, using the MNIST image data set, to recognise handwritten single digits. Ask me any follow-up questions so you have enough information to create a working code

- Organisation: 4 online meetings + WhatsApp group

Only Book (Aline)

```
import pandas as pd

obj = pd.read_pickle(r'/Users/socialmediaappleid/Desktop/CAS AICP/Python/neural-networks-and-deep-learning-master/da
```

Python

```
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 net = network.Network([784, 100, 10])
      2 net.SGD(training_data, 30, 10, 0.001, test_data=test_data)

NameError: name 'network' is not defined
```

```
ModuleNotFoundError: No module named 'cPickle'
```

```
NameError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 net.SGD(training_data, 30, 10, 3.0, test_data=test_data)

NameError: name 'training_data' is not defined
```

✧ Generate + Code + Markdown

```
NameError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 selection_list = []
----> 2 get_selection_indices(x=embedding[:, 0], y=embedding[:, 1], selection_

NameError: name 'get_selection_indices' is not defined
```

```
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 1
----> 1 import mnist_loader
      2 training_data, validation_data, test_data = \
      3 mnist_loader.load_data_wrapper()

ModuleNotFoundError: No module named 'mnist_loader'
```

```
File /usr/lib/python3.12/random.py:357, in Random.shuffle(self, x)
    354 for i in reversed(range(1, len(x))):
    355     # pick an element in x[:i+1] with which to exchange x[i]
    356     j = randbelow(i + 1)
--> 357     x[i], x[j] = x[j], x[i]

TypeError: 'DataLoader' object is not subscriptable
```

```
print "Training with expanded data, %s neurons in the FC layer, run num %s" % (n, j)
```

```python
class Network(object):
    def __init__(self, sizes):
        """
        The list `sizes` contains the number of neurons in the
        respective layers of the network. For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.
        """
        self.num_layers = len(sizes)
        self.sizes = sizes
        # biases for all layers except input
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        # weights between layers
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if `a` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a) + b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
        """
        Train the neural network using mini-batch stochastic gradient descent.
        `training_data` is a list of (x, y) tuples.
        If `test_data` is provided, the network is evaluated after each epoch.
        """
        if test_data:
            n_test = len(test_data)
        # ensure we can shuffle (in case an iterator is passed in)
        training_data = list(training_data)
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k + mini_batch_size]
                for k in range(0, n, mini_batch_size)
```

Book

```python
class MyModel(nn.Module):
    def __init__(self, n_input, n_hiddens, n_output):
        super(MyModel, self).__init__()

        self.ls = []
        n_prev = n_input
        for i, n_out in enumerate(n_hiddens):
            l = nn.Linear(n_prev, n_out)  # for hidden layer we create a linear projection form n_prev features to n_o
            n_prev = n_out
            self.add_module(f'lin_{i}_{n_out}', l)
            self.ls.append(l)

        self.lout = nn.Linear(n_prev, n_output)  # also we need the output layer

    def forward(self, x):
        h = x
        for li in self.ls:  # for each layer we apply the linear projection and the activation fuinction (ReLU)
            h = li(h)
            h = torch.relu(h)

        logits = self.lout(h)
        # Apply softmax activation per row, to get the class pseudoprobabilities
        probs = F.softmax(logits, dim=1)

        # Prediction: argmax for classification
        pred = torch.argmax(probs, dim=1)  # find the element with highest value in each row

        return logits, probs, pred
```

✓ 0.0s Python

Tutorial

File display

```
# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                                transforms.Lambda(lambda x: torch.flatten(x))])

# Download and load the training data
trainset = datasets.FashionMNIST('~/.pytorch/FMNIST_data/', download=True, train=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# Download and load the test data
testset = datasets.FashionMNIST('~/.pytorch/FMNIST_data/', download=True, train=False, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=True)

# Print shapes of the datasets
print('Train dataset shape:', len(trainset), 'total images and labels')
print('Test dataset shape:', len(testset), 'total images and labels')
```

```
# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                                transforms.Lambda(lambda x: torch.flatten(x))])

# Download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# Download and load the test data
testset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=False, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=True)

# Print shapes of the datasets
print('Train dataset shape:', len(trainset), 'total images and labels')
print('Test dataset shape:', len(testset), 'total images and labels')
```

My Learnings

- I need new Materials and/or the help of friends/Chatgpt
- >>> means it has to go into the Terminal
- In Python 3, cPickle is called Pickle

Starting with Gemini

- Definition with Gemini: deep learning framework (Pytorch), NN architecture (MLP), my experience level with Python (Beginner), specific requirements for the model and how the code should be structured (Jupyter Notebook)

- Approach: using Gemini as my really smart friend and challenge him with the differences that I could see with the exercice during the course

- Play and advise with different parameters: n_hiddens, num_epochs, lr and batch_size

The Neural Network Model

- Input Layer: 784 numbers from 28x28pixels
- Hidden layers: final model is with 128 neurons
- Output layer: 10 (number from 0 to 9)

Training the model

- Data Loading with DataLoader
- Loss function: Cross-Entropy Loss
- Optimiser: Adam optimiser
- Epochs: 20

Testing my own handwriting

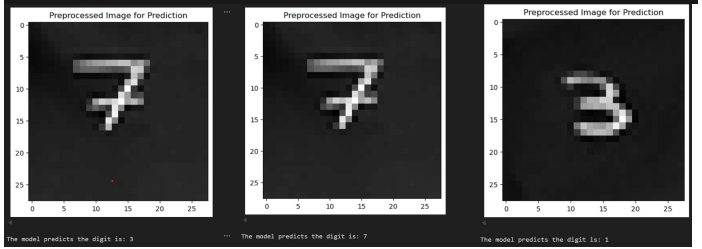
- Integration of my own digits and converted into the same format as the MNIST data
- Image added to the same directory as my notebook
- **Preprocess the image**
- Get a prediction
- **Change parameters**: hidden layers (128), lr (0,005), epoch (20)
- Try with a new digit

```
# Load the image and convert it to grayscale
img = Image.open(image_path).convert('L')

# Resize to 28x28 pixels
img = img.resize((28, 28), Image.LANCZOS)

# Convert to a NumPy array
img_array = np.array(img)

# Invert the image if needed (white background, black digit)
if np.mean(img_array) > 128:
    img_array = 255 - img_array
```



Learnings

1. A model might memorize the training data well but struggle to generalize to new, unseen data

2. The diversity of its training data is so important : MNIST specific handwriting style

3. Hyperparameters will optimise the model’s performance but still can fail with new unseen data

Building a handwritten digit classifier with Chat GPT (Sam)

- I was tasked with creating a model using **ChatGPT** as my coding assistant.
- Chat GPT setup a workflow in PyTorch, running on my MacBook M2 GPU.

Approach:

- **Baseline CNN:** simple convolutional model for a strong starting point.
- **Small ResNet:** deeper model with skip connections, aimed at higher accuracy.

Goal 1: Train, evaluate, and compare both models on the MNIST dataset.
Goal 2: Create additional code to ingest my own handwritten digits and recognize them.

```python
class BasicBlock(nn.Module):                                    # defines a basic residual block class for a ResNet architecture,
                                                                    # ResNet means Residual Network, a type of neural network that uses skip connections
                                                                # to allow gradients to flow more easily during training, helping to mitigate the vanishing gradient problem in deep networks.
    def __init__(self, in_ch, out_ch, stride=1):                # initializes the BasicBlock class with input channels, output channels, and stride for the convolutional layers,
                                                                # meaning that it can change the number of channels
                                                                # and the spatial dimensions of the input
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, stride=stride, padding=1, bias=False)  # first convolutional layer with specified input/output channels, kernel size of 3, stride,
                                                                                        # padding of 1, and no bias
        self.bn1   = nn.BatchNorm2d(out_ch)                                             # batch normalization layer for the output of the first convolutional layer, meaning that it normalizes
                                                                                        # the activations to improve training stability
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1, bias=False)                # second convolutional layer with output channels as both input and output, kernel size of 3, padding of 1,
                                                                                        # and no bias
        self.bn2   = nn.BatchNorm2d(out_ch)                                             # batch normalization layer for the output of the second convolutional layer.
                                                                                        # Batch normalization helps stabilize and accelerate training by normalizing the inputs to each layer,
        self.down  = None                                                       # initializes a downsampling layer as None, which will be used if the input and output dimensions do not match
        if stride != 1 or in_ch != out_ch:                                      # checks if downsampling is needed based on stride or channel mismatch
            self.down = nn.Sequential(                                          # if the stride is not 1 or the number of input channels does not match the number of output channels,
                                                                                # defines a downsampling layer to match dimensions
                nn.Conv2d(in_ch, out_ch, 1, stride=stride, bias=False),         # 1x1 convolutional layer to adjust the number of channels and spatial dimensions
                nn.BatchNorm2d(out_ch)                                          # batch normalization layer for the downsampled output
            )
    def forward(self, x):                                       # defines the forward pass of the BasicBlock class
        identity = x                                            # stores the input x as the identity (skip connection)
        out = F.relu(self.bn1(self.conv1(x)))                   # applies the first convolutional layer, followed by batch normalization and ReLU activation
        out = self.bn2(self.conv2(out))                         # applies the second convolutional layer, followed by batch normalization
        if self.down is not None:                               # if downsampling is needed
            identity = self.down(identity)                      # applies the downsampling layer to the identity
        out = F.relu(out + identity)                            # adds the identity (skip connection) to the output and applies ReLU activation
        return out                                              # returns the output of the block
```

```python
class SmallResNet(nn.Module):                                   # defines a small ResNet architecture for image classification
                                                                # A lean ResNet: channels [32, 64, 128], one block per stage
    def __init__(self, num_classes=10):                         # initializes the SmallResNet class with a specified number of output classes (default is 10 for MNIST digits 0–9)
        super().__init__()                                      # initializes the parent class (nn.Module)
        self.stem = nn.Sequential(                              # defines a sequential container for the initial layers of the ResNet (the stem)
            nn.Conv2d(1, 32, 3, padding=1, bias=False),         # initial convolutional layer with 1 input channel (grayscale), 32 output channels, kernel size of 3, padding of 1, and no bias
            nn.BatchNorm2d(32),                                 # batch normalization layer for the output of the initial convolutional layer
            nn.ReLU(inplace=True),                              # ReLU activation function applied in-place to save memory
        )
        self.layer1 = BasicBlock(32, 32, stride=1)  # 28x28; # first residual block with 32 input and output channels, stride of 1 (no downsampling)
        self.layer2 = BasicBlock(32, 64, stride=2)  # 14x14 # second residual block with 32 input channels, 64 output channels, and stride of 2 (downsampling to reduce spatial dimensions)
        self.layer3 = BasicBlock(64, 128, stride=2) # 7x7 # third residual block with 64 input channels, 128 output channels, and stride of 2 (further downsampling)
        self.head   = nn.Sequential(                            # defines a sequential container for the final classification layers of the ResNet
            nn.AdaptiveAvgPool2d(1),                            # adaptive average pooling layer that reduces the spatial dimensions to 1x1, meaning that it computes the average of each feature map
            nn.Flatten(),                                       # flattens the 1x1 feature maps into a 1D vector
            nn.Linear(128, num_classes)                         # final fully connected layer with 128 input features and num_classes output features (logits for each class)
        )
    def forward(self, x):                       # defines the forward pass of the SmallResNet class
        x = self.stem(x)                        # applies the initial stem layers to the input x
        x = self.layer1(x)                      # applies the first residual block to the output of the stem
        x = self.layer2(x)                      # applies the second residual block to the output of the first block
        x = self.layer3(x)                      # applies the third residual block to the output of the second block
        x = self.head(x)                        # applies the final classification layers to the output of the third block
        return x                                # returns the logits for each class
```

Screenshots of the ResNet model

```python
# ---------- 7) Models ----------
class SimpleCNN(nn.Module):                                     # defines a simple convolutional neural network (CNN) class for image classification
    def __init__(self, num_classes=10):                         # initializes the SimpleCNN class with a specified number of output classes (default is 10 for MNIST digits 0–9)
        super().__init__()                                      # initializes the parent class (nn.Module)
        self.features = nn.Sequential(                          # defines a sequential container for the feature extraction layers of the CNN, meaning that the layers will be applied in order
            nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(inplace=True),  # first convolutional layer with 1 input channel (grayscale), 32 output channels, kernel size of 3, and ReLU activation
                                                                    # In a convolutional layer, the kernel (or filter) size determines the dimensions of the sliding window that
                                                                    # moves over the input image to extract features.
                                                                    # A kernel size of 3 means a 3x3 pixels filter is used to scan over the input image.
            nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(2),  # 14x14                             # max pooling layer with a kernel size of 2, which reduces the spatial dimensions by half (from 28x28 to 14x14).
                                                                    # A Max Pooling layer is a downsampling operation that
                                                                    # reduces the spatial dimensions (height and width) of the input feature maps.
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(inplace=True), # second convolutional layer with 32 input channels, 64 output channels, kernel size of 3, and ReLU activation
            nn.Conv2d(64, 64, 3, padding=1), nn.ReLU(inplace=True), # third convolutional layer with 64 input channels, 64 output channels, kernel size of 3, and ReLU activation
            nn.MaxPool2d(2),                                        # 7x7 # reduces spatial dimensions to 7x7
        )
        self.classifier = nn.Sequential(                        # defines a sequential container for the classification layers of the CNN
            nn.Flatten(),                                       # flattens the 2D feature maps into a 1D vector
            nn.Linear(64 * 7 * 7, 128), nn.ReLU(inplace=True),  # fully connected layer with 64*7*7 input features and 128 output features, followed by ReLU activation
            nn.Linear(128, num_classes)                         # final fully connected layer with 128 input features and num_classes output features (logits for each class)
        )

    def forward(self, x):                                       # defines the forward pass of the SimpleCNN class
        x = self.features(x)                                    # applies the feature extraction layers to the input x
        return self.classifier(x)
```

Screenshot of the Convolutional Neural Network model (CNN)

Optimizer: AdamW
Learning rate schedule: cosine decay
Epochs : 15
Batch size : 512
Method: Autocast / mixed precision

Overcoming Training Challenges

- The CNN worked well in the very first try. No code adjustments necessary.
- Initial training of ResNet failed (~10% accuracy). Chat GPT discovered that **EMA + BatchNorm** caused problems on Apple's MPS.
    - Solution: It suggested to turn off AMP and EMA for the ResNet. Then training "stabilized".

Results:

- Baseline CNN: 99.3% accuracy.
- Small ResNet: 99.6% accuracy (slightly better, as expected).

Key lesson from the ResNet fail:
Small details (like mixed precision, normalization layers) can make or break training stability.

```
=== FINAL SUMMARY ===
Model                  Test Loss    Test Accuracy
---------------------  -----------  ---------------
Baseline CNN                0.0265           0.9927
Small ResNet + LS + EMA    28.271            0.0974
```

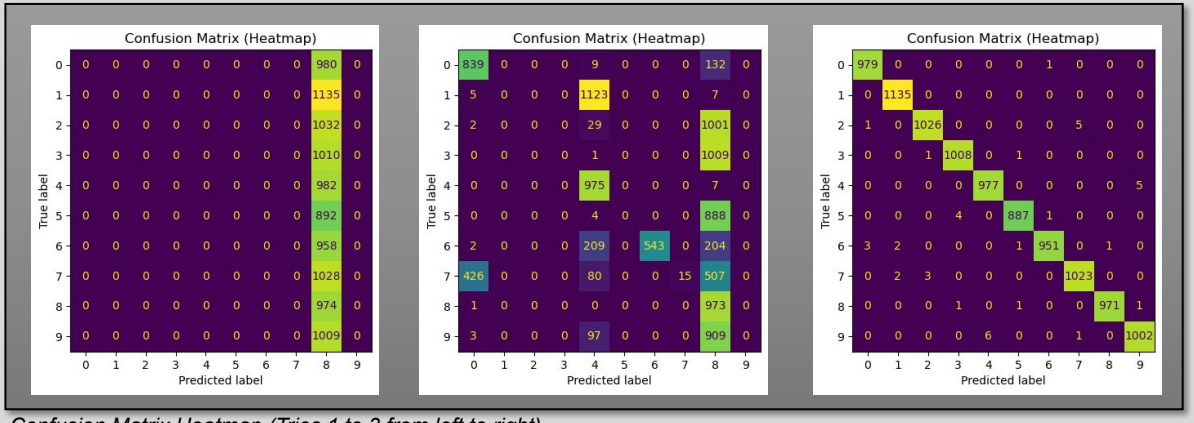
Summary 1st try (untouched ResNet)

```
=== FINAL SUMMARY ===
Model                  Test Loss    Test Accuracy
---------------------  -----------  ---------------
Baseline CNN                0.0265           0.9927
Small ResNet + LS + EMA     3.3315           0.3345
```

Summary 2nd try (turned off AMP)

```
=== FINAL SUMMARY ===
Model                  Test Loss    Test Accuracy
---------------------  -----------  ---------------
Baseline CNN                0.0265           0.9927
Small ResNet + LS + EMA     0.3036           0.9959
```

Summary 3nd try (turned off AMP & EMA)



Confusion Matrix Heatmap (Tries 1 to 3 from left to right)

Unanticipated code running errors:
Error 1: Using "Tabulate" without installing it
Error 2: Dataloader: Issue w/ VSCode/Notebooks

Testing with My Own Handwriting

- Chat GPT generated additional code to collected 10 of my own handwritten digits (0–9).
- It built a preprocessing pipeline to resize, invert, and normalize images to match MNIST style.

First attempt:

- all predictions for each of the 10 digits were “8” (showed preprocessing mismatch).

Second attempt:

- After increasing contrast in Photoshop: models reached ~70% accuracy on my digits.

Next suggested steps:

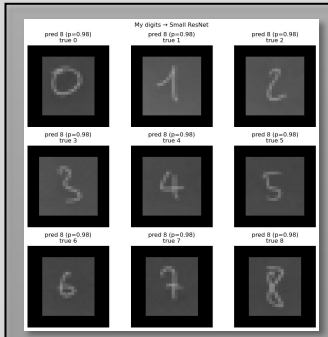
Chat GPT suggests to refine preprocessing code (force inversion, stronger contrast, stroke dilation) for better results.

MY CONCLUSION

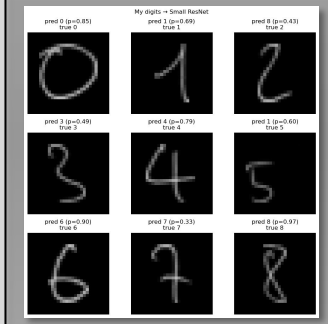
Using ChatGPT5 to build a neural network works astonishingly well.

But: Without knowledge of NNs and Python your project is destined to fail.

→ **Interdependency** between programmer and LLM




```
=== Results for Small ResNet ===
['file', 'true (from name)', 'pred', 'confidence']
['0.png', '0', 8, '0.98']
['1.png', '1', 8, '0.98']
['2.png', '2', 8, '0.98']
['3.png', '3', 8, '0.98']
['4.png', '4', 8, '0.98']
['5.png', '5', 8, '0.98']
['6.png', '6', 8, '0.98']
['7.png', '7', 8, '0.98']
['8.png', '8', 8, '0.98']
['9.png', '9', 8, '0.98']
Mini-set accuracy (from filenames): 1/10 = 0.100
```




```
=== Results for Small ResNet ===
['file', 'true (from name)', 'pred', 'confidence']
['0.png', '0', 0, '0.85']
['1.png', '1', 1, '0.69']
['2.png', '2', 8, '0.43']
['3.png', '3', 3, '0.49']
['4.png', '4', 4, '0.79']
['5.png', '5', 1, '0.60']
['6.png', '6', 6, '0.90']
['7.png', '7', 7, '0.33']
['8.png', '8', 8, '0.97']
['9.png', '9', 1, '0.29']
Mini-set accuracy (from filenames): 7/10 = 0.700
```


```
=== Results for Small ResNet ===
['file', 'true (from name)', 'pred', 'confidence']
['0.png', '0', 0, '0.87']
['1.png', '1', 4, '0.85']
['2.png', '2', 8, '0.48']
['3.png', '3', 8, '0.48']
['4.png', '4', 8, '0.39']
['5.png', '5', 5, '0.79']
['6.png', '6', 6, '0.33']
['7.png', '7', 8, '0.42']
['8.png', '8', 8, '0.98']
['9.png', '9', 8, '0.54']
Mini-set accuracy (from filenames): 4/10 = 0.400
```

MNIST Optimization and Modern Deep Learning Techniques with Claude

Feedforward Neural Network

Challenge: Outdated Book Example (2015)

Systematic MNIST Improvement

Original (2015)	Modern Version (2025)	Improvement
Sigmoid + SGD + MSE	ReLU + Adam + CrossEntropy	**+247 predictions**
95.01% (9501/10000)	97.48% (9748/10000)	**(+2.47%)**

python

```python
# Original (outdated standards)
model = nn.Sequential([
    nn.Linear(784, 30),
    nn.Sigmoid(),                # Vanishing Gradient Problem
    nn.Linear(30, 10),
    nn.Sigmoid()
])
optimizer = optim.SGD(lr=3.0)  # Fixed Learning Rate
criterion = nn.MSELoss()       # Suboptimal for classification
```

python

```python
# Modern (current best practices)
model = nn.Sequential([
    nn.Linear(784, 128),
    nn.ReLU(),                       # Solves Vanishing Gradient
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
])
optimizer = optim.Adam(lr=0.001)  # Adaptive Learning Rate
criterion = nn.CrossEntropyLoss() # Optimal for classification
```

Personalization to Own Handwriting

Model	MNIST Performance	Personal Handwriting (10 PNG files)
Original	95.01%	**10%** (1/10)
Improved	97.48%	**30%** (3/10)

Model	Personal Handwriting	Robustness
Fine-tuned	100% (10/10)	100% (Overfitting-Risk)
Augmented	100% (10/10)	90% on variations

- **Automatic extraction failed:** OpenCV recognized 0 valid digits
- **Large dataset impractical:** Writing hundreds of digits manually too time-consuming
- **Solution:** 10 handwritten PNG files (digit_0.png to digit_9.png)

```python
# 10 Originalbilder → 210 Variationen
transforms = [
    transforms.RandomRotation(degrees=12),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomAffine(degrees=0, scale=(0.8, 1.2))
]
```

Synthetic Multi-Digit Pictures

- **First evaluation (78.5% accuracy):** tested on *all* 200 images (training + test combined)
- **Second evaluation (2.5% accuracy):** tested only on 40 test images (20% split)

Why this happens:

- Dataset too small: 200 images are not enough for sequence-to-sequence learning
- No regularization: the model memorizes instead of learning
- Architecture issue: the current CNN→Dense setup is not suitable for variable sequence lengths
- **Actual test accuracy:** ~2.5% (on unseen data)
  Apparent accuracy: ~78.5% (on training data)
  Overfitting gap: ~76%

```
Evaluiere Multi-Digit Modell...
True:   0 | Pred:   0 | ✓
True:   1 | Pred:   9 | ✗
True:   2 | Pred:   0 | ✗
True:   3 | Pred:   0 | ✗
True:   4 | Pred:   0 | ✗
True:   5 | Pred:   0 | ✗
True:   6 | Pred:   0 | ✗
True:   7 | Pred:   0 | ✗
True:   8 | Pred:   7 | ✗
True:   9 | Pred:   0 | ✗

MULTI-DIGIT ACCURACY RESULTS:
Overall: 157/200 = 0.785
1-stellig: 1/10 = 0.100
2-stellig: 79/90 = 0.878
3-stellig: 77/100 = 0.770
```

Learnings

1. Best technique does not always mean improvement

2. Augmentation helps with generalizing and thus have a better accuracy

3. Even if its able to create new synthetic data, it still is not enough for creating meaningful results

Conclusion