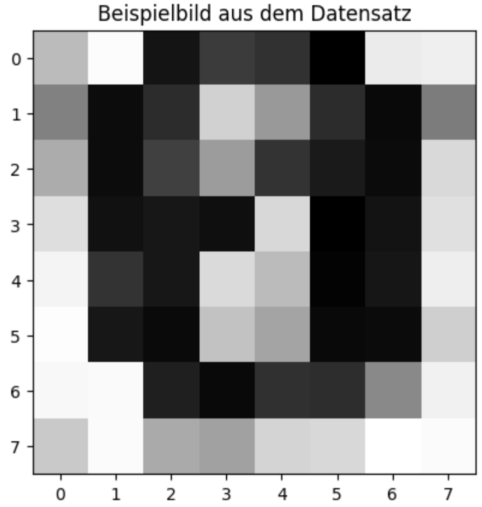
Deep Learning for Handwritten Digit Recognition

The begging of every analysis is Data Exploration




```
# Custom dataset class
class CustomTensorDataset(Dataset):
    def __init__(self, tensors, transform=None):
        self.tensors = tensors
        self.transform = transform

    def __len__(self):
        return len(self.tensors)

    def __getitem__(self, idx):
        image, label = self.tensors[idx]
        # Convert image if necessary
        if isinstance(image, np.ndarray):
            image = Image.fromarray(image)
        elif torch.is_tensor(image):
            image = transforms.ToPILImage()(image)
        # Apply transformations
        if self.transform:
            image = self.transform(image)
        return image, label
```

Data transformation

To use PyTorch functions, images must meet specific format requirements. The original data consists of **tuples** with `torch.Tensor` images of size `(8,8)`, but PyTorch requires tensors in the format `(C, H, W)`, where **C = channels, H = height, W = width**. Normalization is also necessary to improve model stability.

PIL (Python Imaging Library) is used to load and manipulate images in Python. PyTorch uses **Pillow** to convert images from `numpy.ndarray` or `torch.Tensor` into a manageable format before applying transformations.

What does `CustomTensorDataset` do?

The `CustomTensorDataset` class adapts the data for use in PyTorch:

- **Stores the data** as tuples `(image, label)`.
- **Converts images** to PIL if they are in `numpy.ndarray` or `torch.Tensor`.
- **Applies transformations** (resizing, normalization, etc.).
- **Returns** the transformed image and its label.

This ensures that `(8,8)` images are resized and formatted correctly for compatibility with PyTorch models, such as MNIST or aditinal images.

```
# Create transformed datasets
train_set_pk = CustomTensorDataset(train_data_pk, transform=transform_train)
test_set_pk = CustomTensorDataset(test_data_pk, transform=transform_test)

# Create data loaders
batch_size_pk = 16
trainloader_pk = DataLoader(train_set_pk, batch_size=batch_size_pk, shuffle=True)
testloader_pk = DataLoader(test_set_pk, batch_size=batch_size_pk, shuffle=True)
```

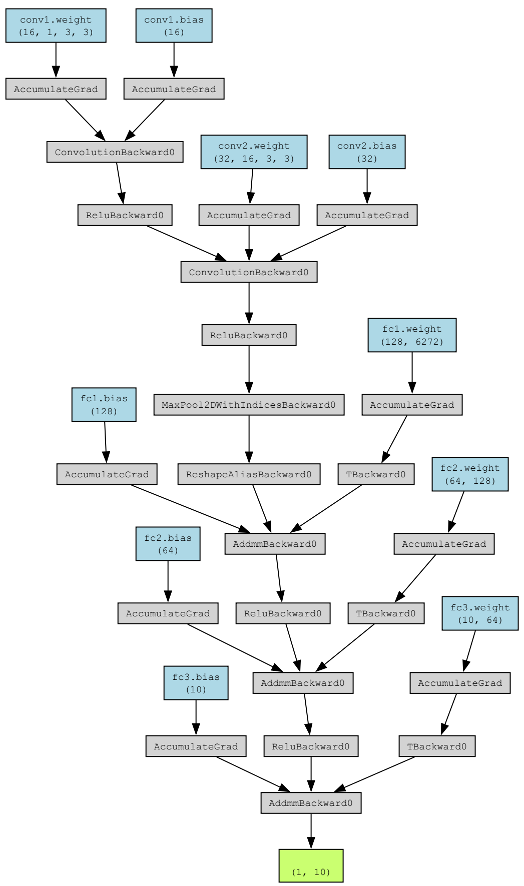
```
# Transformations for training and testing
transform_train = transforms.Compose([
    transforms.Resize((28, 28)),                          # Redimensiona a 28x28
    transforms.RandomRotation(10),                        # Rotación aleatoria
    transforms.RandomAffine(0, translate=(0.1, 0.1)),   # Desplazamiento aleatorio
    transforms.ToTensor(),                                # Convierte a tensor
    transforms.Normalize((0.5,), (0.5,))                 # Normaliza
])

transform_test = transforms.Compose([
    transforms.Resize((28, 28)),                          # Redimensiona a 28x28
    transforms.ToTensor(),                                # Convierte a tensor
    transforms.Normalize((0.5,), (0.5,))                 # Normaliza
])
```

HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Summary of Steps in `ImprovedNet`

`Padding = Reduce total size(ex.28x28->14x14) Stride = Lenght of step(messured on pixels) Kernel = Size of the filter(matrix)`

1. **Conv1** (`Conv2d(1, 16, kernel_size=3, stride=1, padding=1)`)
   - Applies 16 filters **3x3**, `stride=1`, `padding=1`
   - **Output:** `16x28x28` (size preserved due to `padding=1`)
2. **Conv2** (`Conv2d(16, 32, kernel_size=3, stride=1, padding=1)`)
   - Applies 32 filters **3x3**, `stride=1`, `padding=1`
   - **Output:** `32x28x28`
3. **Max Pooling** (`MaxPool2d(kernel_size=2, stride=2)`)
   - Reduces size by half, taking max values from **2x2** blocks, `stride=2`
   - **Output:** `32x14x14`
4. **Flatten** (`torch.flatten(x, 1)`)
   - Converts `32x14x14` into a `6272` -element vector
5. **Fully Connected Layers** (`fc1`, `fc2`, `fc3`)
   - `fc1` : `6272 → 128`, ReLU + Dropout
   - `fc2` : `128 → 64`, ReLU + Dropout
   - `fc3` : `64 → 10`, final output with logits



Final Training Variables

```python
# Create an instance of the model from cero with my handwriting
model_ph = ImprovedNet()
model_ph.eval()
learning_rate=0.001
epochs=4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model_pk = model_pk.to(device=device)
batch_size_ph = 16
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_ph.parameters(), lr=learning_rate)
```

Neural Network Definition

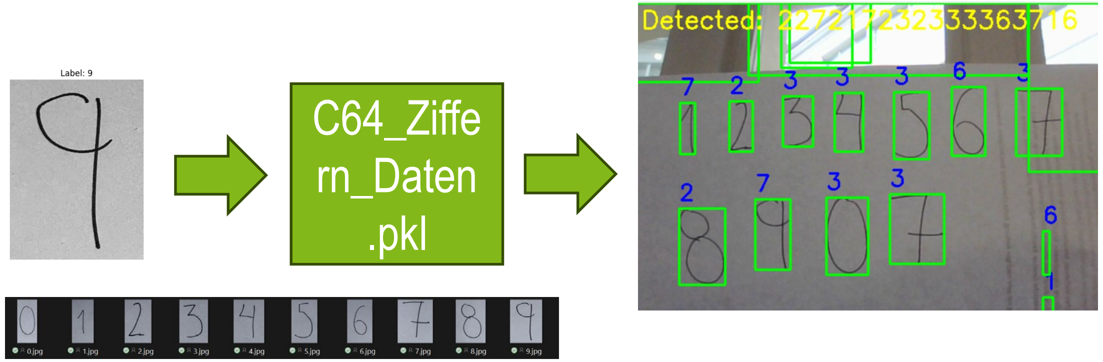
```python
class ImprovedNet(nn.Module):
    def __init__(self):
        super(ImprovedNet, self).__init__()
        # Convolutional layers with fewer filters
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # Max pooling

        # Calculate flattened size dynamically
        self._flattened_size = self._get_flattened_size()

        # Fully connected layers with reduced sizes
        self.fc1 = nn.Linear(self._flattened_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

        # Dropout with reduced probability
        self.dropout = nn.Dropout(0.3)
```

Accuracy of the model (with data form pkl)

```
Epoch: 1/4, Loss: 0.890658, Accuracy: 95.55%
Epoch: 2/4, Loss: 0.245953, Accuracy: 99.90%
Epoch: 3/4, Loss: 0.141535, Accuracy: 100.00%
Epoch: 4/4, Loss: 0.096880, Accuracy: 99.95%
```

Accuracy of the model (with data form pkl + my Handwriting)

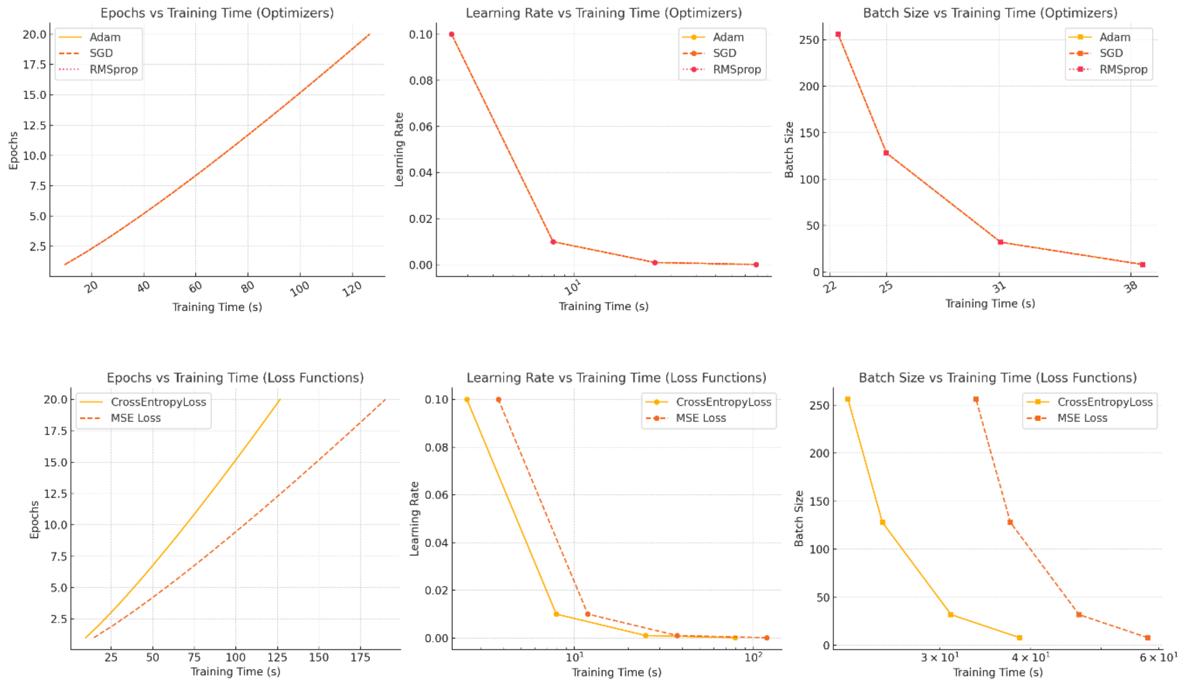
```
Epoch: 1/4, Loss: 0.894721, Accuracy: 99.40%
Epoch: 2/4, Loss: 0.223679, Accuracy: 97.80%
Epoch: 3/4, Loss: 0.132832, Accuracy: 100.00%
Epoch: 4/4, Loss: 0.092740, Accuracy: 100.00%
```



Behavior of the training time adapting:

- Epochs
- Learning rate
- Batch Size

With different optimizers and Loss Functions



Quelle:
Bildhauer-Buggle C. *KI-Anwendung Kapitel 3.1v2 Python und KI Frameworks*. Hochschule Furtwangen; 2024

Thank you