main

October 15, 2024

0.1 Building a Hangul classifier

0.1.1 By Dokyun Kim

This project focuses on developing a machine learning model capable of classifying handwritten Hangul characters from the Korean alphabet. As Korean media and culture continue to gain global popularity, an increasing number of people are learning Korean as a second language. This model can be used as an educational tool to enhance the language learning experience by helping students recognize and practice writing various Hangul characters. It could aid learners in quickly identifying character shapes, reinforcing their familiarity with characters, and improving handwriting skills. This model aims support learners in developing a stronger foundation in the Korean language.

[21]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split, ConcatDataset
from torchsummary import summary
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Currently using {device}")
```

```
Currently using cuda
```

0.2 Handwritten Hangul Dataset

To train the model, we are utilizing the *Handwritten Hangul Characters* dataset from Kaggle, which contains 2,400 images of individual Hangul characters, each sized at 28x28 pixels. Since our model is designed to work exclusively with grayscale images, we first pre-process the dataset by converting all images to grayscale. This step ensures compatibility with the model's input requirements.

To improve the model's robustness, we added an augmented dataset to the original dataset. We applied a random rotation of -20 to 20 degrees to each image, which enhances the model's performance on various handwriting styles and orientations.

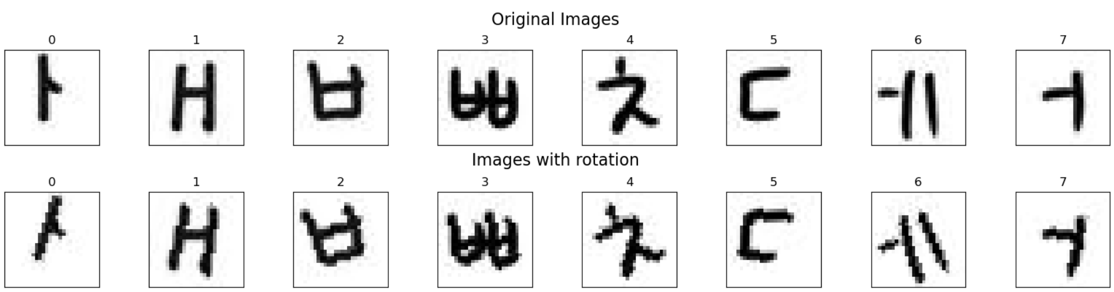
[22]:
```
transform_unaug = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalizing to [-1, 1]
])

transform_aug = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.RandomRotation(degrees=(-20,20), fill=255), # Apply random␣
 ↪rotation
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalizing to [-1, 1]
])

dataset_unaug = datasets.ImageFolder(root='data/', transform=transform_unaug)
dataset_aug = datasets.ImageFolder(root='data/', transform=transform_aug)

# Combine unaugmented & augmented dataset into one
dataset_all = ConcatDataset([dataset_unaug, dataset_aug])

# Split dataset into 70% train, 30% test
train_size, test_size = 0.7, 0.3
batch_size = 40


train_set, test_set = random_split(dataset_all, [train_size, test_size])

train_loader = DataLoader(train_set, batch_size = batch_size, shuffle=True)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=False)

print(f"{len(dataset_all)} total images")
for images, labels in train_loader:
    print('Image Batch Dimension: ', images.shape)
    print('Image Labels Dimension: ', labels.shape)
    break
```

```
4800 total images
Image Batch Dimension:  torch.Size([40, 1, 28, 28])
Image Labels Dimension:  torch.Size([40])
```

0.2.1 Visualizing the dataset

[23]:
```
# Create a 2x8 grid of subplots
fig, axs = plt.subplots(2, 8, figsize=(16, 4))

# Plot original images in the first row
for i in range(8):
```

```
    img, label = dataset_unaug[i*80]
    img = img.squeeze().numpy()
    axs[0, i].imshow(img, cmap='gray')
    axs[0, i].set_title(label)
    axs[0, i].axes.xaxis.set_visible(False)
    axs[0, i].axes.yaxis.set_visible(False)

# Plot augmented images in the second row
for i in range(8):
    img, label = dataset_aug[i*80]
    img = img.squeeze().numpy()
    axs[1, i].imshow(img, cmap='gray')
    axs[1, i].set_title(label)
    axs[1, i].axes.xaxis.set_visible(False)
    axs[1, i].axes.yaxis.set_visible(False)

# Add titles for each row of subplots
fig.text(0.5, 0.98, "Original Images", ha='center', fontsize=16, va='top')
fig.text(0.5, 0.5, "Images with rotation", ha='center', fontsize=16, va='top')

plt.tight_layout()
plt.subplots_adjust(top=0.85, hspace=0.5)
```



0.3 Implementing a MLP solution

In the code block below, we define our MLP model. Since the task is quite simple (classify 30 classes), we will only use one hidden layer of size 203. The model details are printed below the code block.

[24]:
```
# Define MLP object

class Hangul_MLP(nn.Module):
    """
    A model that implements a logistic regression classifier.
    """
    def __init__(self, input_size, num_classes):
```

```python
        """
        Constructor for MLP object

        Args:
            input_size (int): size of input tensor
            num_classes (int): number of classes the model can predict
        """
        super(Hangul_MLP, self).__init__()

        self.linear_stack = nn.Sequential(
            nn.Linear(input_size, (input_size + num_classes) // 4),
            nn.Sigmoid(),
            nn.Linear((input_size + num_classes) // 4, num_classes),
            nn.Sigmoid()
        )

    def forward(self, x):
        """
        Forward pass of the model

        Args:
            x (tensor): Input to the model

        Returns:
            out (tensor): Output of the model

        """
        out = self.linear_stack(x)
        out = F.softmax(out, dim=1)
        return out

print(summary(Hangul_MLP(784, 30).to(device), (1,784)))
```

```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
 Sequential: 1-1                        [-1, 1, 30]               --
|    Linear: 2-1                        [-1, 1, 203]              159,355
|    Sigmoid: 2-2                       [-1, 1, 203]              --
|    Linear: 2-3                        [-1, 1, 30]               6,120
|    Sigmoid: 2-4                       [-1, 1, 30]               --
================================================================================
==========
Total params: 165,475
Trainable params: 165,475
```

```
Non-trainable params: 0
Total mult-adds (M): 0.33
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.63
Estimated Total Size (MB): 0.64
================================================================================
==========
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
 Sequential: 1-1                        [-1, 1, 30]               --
|    Linear: 2-1                        [-1, 1, 203]              159,355
|    Sigmoid: 2-2                       [-1, 1, 203]              --
|    Linear: 2-3                        [-1, 1, 30]               6,120
|    Sigmoid: 2-4                       [-1, 1, 30]               --
================================================================================
==========
Total params: 165,475
Trainable params: 165,475
Non-trainable params: 0
Total mult-adds (M): 0.33
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.63
Estimated Total Size (MB): 0.64
================================================================================
==========
```

0.3.1 MLP with batch training

[25]:
```python
def run_MLP_batch(trainloader, testloader, n_epochs, learning_rate):

    model = Hangul_MLP(input_size = 784, num_classes = 30).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)

    train_losses = np.zeros((n_epochs,))
    test_losses = np.zeros((n_epochs,))
    accuracies = np.zeros((n_epochs,))
```

```
    start = time.time()
    for epoch in range(n_epochs):

        model.train()
        for images, labels in trainloader:
            images, labels = images.to(device), labels.to(device)
            images = images.view(images.size(0), -1) # Reshape to [batch_size,␣
↪784] from [batch_size, 1, 28, 28]

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        model.eval()
        with torch.no_grad():

            total_test_loss, total_train_loss, correct_preds = 0.0, 0.0, 0.0

            for images, labels in testloader:
                images, labels = images.to(device), labels.to(device)
                images = images.view(images.size(0), -1) # Reshape to␣
↪[batch_size, 784] from [batch_size, 1, 28, 28]

                test_outputs = model(images)
                test_loss = criterion(test_outputs, labels)
                total_test_loss += test_loss.item()

                _, preds = torch.max(test_outputs, 1)
                correct_preds += (preds == labels).sum().item()

            for images, labels in trainloader:
                images, labels = images.to(device), labels.to(device)
                images = images.view(images.size(0), -1) # Reshape to␣
↪[batch_size, 784] from [batch_size, 1, 28, 28]

                train_outputs = model(images)
                train_loss = criterion(train_outputs, labels)
                total_train_loss += train_loss.item()

            accuracies[epoch] = correct_preds / len(testloader.dataset) * 100
            test_losses[epoch] = total_test_loss / len(testloader)
            train_losses[epoch] = total_train_loss / len(trainloader)
```

```
            print('Epoch: %03d/%03d | Accuracy: %.3f%%' %(epoch + 1, n_epochs,␣
↪accuracies[epoch]))

    print("Total Train Time: %.2f min" % ((time.time() - start)/60))
    return train_losses, test_losses, accuracies, model
```

[26]:
```
train_losses, test_losses, accuracies, model = run_MLP_batch(train_loader,␣
↪test_loader, 300, 0.8)

torch.save(model.state_dict(), "mlp.pth")
```

```
Epoch: 001/300 | Accuracy: 14.236%
Epoch: 002/300 | Accuracy: 32.431%
Epoch: 003/300 | Accuracy: 45.903%
Epoch: 004/300 | Accuracy: 61.944%
Epoch: 005/300 | Accuracy: 63.750%
Epoch: 006/300 | Accuracy: 67.361%
Epoch: 007/300 | Accuracy: 63.681%
Epoch: 008/300 | Accuracy: 67.847%
Epoch: 009/300 | Accuracy: 66.528%
Epoch: 010/300 | Accuracy: 70.833%
Epoch: 011/300 | Accuracy: 69.653%
Epoch: 012/300 | Accuracy: 70.694%
Epoch: 013/300 | Accuracy: 71.319%
Epoch: 014/300 | Accuracy: 71.944%
Epoch: 015/300 | Accuracy: 73.611%
Epoch: 016/300 | Accuracy: 73.264%
Epoch: 017/300 | Accuracy: 73.333%
Epoch: 018/300 | Accuracy: 73.889%
Epoch: 019/300 | Accuracy: 73.889%
Epoch: 020/300 | Accuracy: 73.819%
Epoch: 021/300 | Accuracy: 75.972%
Epoch: 022/300 | Accuracy: 75.139%
Epoch: 023/300 | Accuracy: 75.764%
Epoch: 024/300 | Accuracy: 75.972%
Epoch: 025/300 | Accuracy: 76.806%
Epoch: 026/300 | Accuracy: 77.014%
Epoch: 027/300 | Accuracy: 75.486%
Epoch: 028/300 | Accuracy: 77.708%
Epoch: 029/300 | Accuracy: 77.569%
Epoch: 030/300 | Accuracy: 78.542%
Epoch: 031/300 | Accuracy: 78.472%
Epoch: 032/300 | Accuracy: 79.444%
Epoch: 033/300 | Accuracy: 79.444%
Epoch: 034/300 | Accuracy: 79.722%
Epoch: 035/300 | Accuracy: 79.653%
Epoch: 036/300 | Accuracy: 80.833%
```

```
Epoch: 037/300 | Accuracy: 80.556%
Epoch: 038/300 | Accuracy: 80.556%
Epoch: 039/300 | Accuracy: 80.694%
Epoch: 040/300 | Accuracy: 80.903%
Epoch: 041/300 | Accuracy: 80.903%
Epoch: 042/300 | Accuracy: 81.667%
Epoch: 043/300 | Accuracy: 81.389%
Epoch: 044/300 | Accuracy: 81.875%
Epoch: 045/300 | Accuracy: 82.431%
Epoch: 046/300 | Accuracy: 81.944%
Epoch: 047/300 | Accuracy: 82.500%
Epoch: 048/300 | Accuracy: 82.500%
Epoch: 049/300 | Accuracy: 82.292%
Epoch: 050/300 | Accuracy: 83.264%
Epoch: 051/300 | Accuracy: 82.361%
Epoch: 052/300 | Accuracy: 83.194%
Epoch: 053/300 | Accuracy: 83.056%
Epoch: 054/300 | Accuracy: 83.056%
Epoch: 055/300 | Accuracy: 83.333%
Epoch: 056/300 | Accuracy: 83.750%
Epoch: 057/300 | Accuracy: 84.097%
Epoch: 058/300 | Accuracy: 83.125%
Epoch: 059/300 | Accuracy: 84.444%
Epoch: 060/300 | Accuracy: 85.278%
Epoch: 061/300 | Accuracy: 83.333%
Epoch: 062/300 | Accuracy: 84.306%
Epoch: 063/300 | Accuracy: 84.375%
Epoch: 064/300 | Accuracy: 84.653%
Epoch: 065/300 | Accuracy: 84.722%
Epoch: 066/300 | Accuracy: 83.958%
Epoch: 067/300 | Accuracy: 84.861%
Epoch: 068/300 | Accuracy: 85.000%
Epoch: 069/300 | Accuracy: 84.653%
Epoch: 070/300 | Accuracy: 85.625%
Epoch: 071/300 | Accuracy: 85.625%
Epoch: 072/300 | Accuracy: 85.764%
Epoch: 073/300 | Accuracy: 87.222%
Epoch: 074/300 | Accuracy: 86.667%
Epoch: 075/300 | Accuracy: 86.875%
Epoch: 076/300 | Accuracy: 87.083%
Epoch: 077/300 | Accuracy: 87.986%
Epoch: 078/300 | Accuracy: 87.014%
Epoch: 079/300 | Accuracy: 86.389%
Epoch: 080/300 | Accuracy: 86.528%
Epoch: 081/300 | Accuracy: 87.708%
Epoch: 082/300 | Accuracy: 87.083%
Epoch: 083/300 | Accuracy: 87.778%
Epoch: 084/300 | Accuracy: 87.014%
```

```
Epoch: 085/300 | Accuracy: 87.569%
Epoch: 086/300 | Accuracy: 89.028%
Epoch: 087/300 | Accuracy: 88.542%
Epoch: 088/300 | Accuracy: 88.403%
Epoch: 089/300 | Accuracy: 87.847%
Epoch: 090/300 | Accuracy: 87.500%
Epoch: 091/300 | Accuracy: 88.333%
Epoch: 092/300 | Accuracy: 88.542%
Epoch: 093/300 | Accuracy: 88.472%
Epoch: 094/300 | Accuracy: 88.472%
Epoch: 095/300 | Accuracy: 88.264%
Epoch: 096/300 | Accuracy: 89.444%
Epoch: 097/300 | Accuracy: 89.097%
Epoch: 098/300 | Accuracy: 89.653%
Epoch: 099/300 | Accuracy: 89.514%
Epoch: 100/300 | Accuracy: 89.514%
Epoch: 101/300 | Accuracy: 89.444%
Epoch: 102/300 | Accuracy: 90.000%
Epoch: 103/300 | Accuracy: 88.750%
Epoch: 104/300 | Accuracy: 89.792%
Epoch: 105/300 | Accuracy: 90.069%
Epoch: 106/300 | Accuracy: 90.000%
Epoch: 107/300 | Accuracy: 89.931%
Epoch: 108/300 | Accuracy: 89.514%
Epoch: 109/300 | Accuracy: 90.417%
Epoch: 110/300 | Accuracy: 89.444%
Epoch: 111/300 | Accuracy: 89.653%
Epoch: 112/300 | Accuracy: 90.417%
Epoch: 113/300 | Accuracy: 90.347%
Epoch: 114/300 | Accuracy: 89.722%
Epoch: 115/300 | Accuracy: 90.278%
Epoch: 116/300 | Accuracy: 90.625%
Epoch: 117/300 | Accuracy: 90.000%
Epoch: 118/300 | Accuracy: 89.583%
Epoch: 119/300 | Accuracy: 90.208%
Epoch: 120/300 | Accuracy: 90.139%
Epoch: 121/300 | Accuracy: 90.139%
Epoch: 122/300 | Accuracy: 90.417%
Epoch: 123/300 | Accuracy: 90.347%
Epoch: 124/300 | Accuracy: 90.417%
Epoch: 125/300 | Accuracy: 90.764%
Epoch: 126/300 | Accuracy: 89.861%
Epoch: 127/300 | Accuracy: 90.208%
Epoch: 128/300 | Accuracy: 90.347%
Epoch: 129/300 | Accuracy: 90.139%
Epoch: 130/300 | Accuracy: 90.486%
Epoch: 131/300 | Accuracy: 90.347%
Epoch: 132/300 | Accuracy: 90.486%
```

```
Epoch: 133/300 | Accuracy: 91.181%
Epoch: 134/300 | Accuracy: 90.833%
Epoch: 135/300 | Accuracy: 91.458%
Epoch: 136/300 | Accuracy: 91.250%
Epoch: 137/300 | Accuracy: 90.417%
Epoch: 138/300 | Accuracy: 91.181%
Epoch: 139/300 | Accuracy: 90.833%
Epoch: 140/300 | Accuracy: 91.389%
Epoch: 141/300 | Accuracy: 91.528%
Epoch: 142/300 | Accuracy: 90.347%
Epoch: 143/300 | Accuracy: 91.181%
Epoch: 144/300 | Accuracy: 90.903%
Epoch: 145/300 | Accuracy: 90.764%
Epoch: 146/300 | Accuracy: 90.833%
Epoch: 147/300 | Accuracy: 91.667%
Epoch: 148/300 | Accuracy: 91.667%
Epoch: 149/300 | Accuracy: 91.528%
Epoch: 150/300 | Accuracy: 90.903%
Epoch: 151/300 | Accuracy: 91.250%
Epoch: 152/300 | Accuracy: 91.806%
Epoch: 153/300 | Accuracy: 90.972%
Epoch: 154/300 | Accuracy: 90.764%
Epoch: 155/300 | Accuracy: 91.042%
Epoch: 156/300 | Accuracy: 91.458%
Epoch: 157/300 | Accuracy: 91.597%
Epoch: 158/300 | Accuracy: 90.903%
Epoch: 159/300 | Accuracy: 91.250%
Epoch: 160/300 | Accuracy: 92.292%
Epoch: 161/300 | Accuracy: 91.319%
Epoch: 162/300 | Accuracy: 91.736%
Epoch: 163/300 | Accuracy: 91.667%
Epoch: 164/300 | Accuracy: 91.597%
Epoch: 165/300 | Accuracy: 92.222%
Epoch: 166/300 | Accuracy: 91.597%
Epoch: 167/300 | Accuracy: 92.708%
Epoch: 168/300 | Accuracy: 91.736%
Epoch: 169/300 | Accuracy: 92.292%
Epoch: 170/300 | Accuracy: 91.944%
Epoch: 171/300 | Accuracy: 92.639%
Epoch: 172/300 | Accuracy: 92.500%
Epoch: 173/300 | Accuracy: 92.361%
Epoch: 174/300 | Accuracy: 91.944%
Epoch: 175/300 | Accuracy: 91.806%
Epoch: 176/300 | Accuracy: 92.153%
Epoch: 177/300 | Accuracy: 92.361%
Epoch: 178/300 | Accuracy: 92.222%
Epoch: 179/300 | Accuracy: 92.500%
Epoch: 180/300 | Accuracy: 92.222%
```

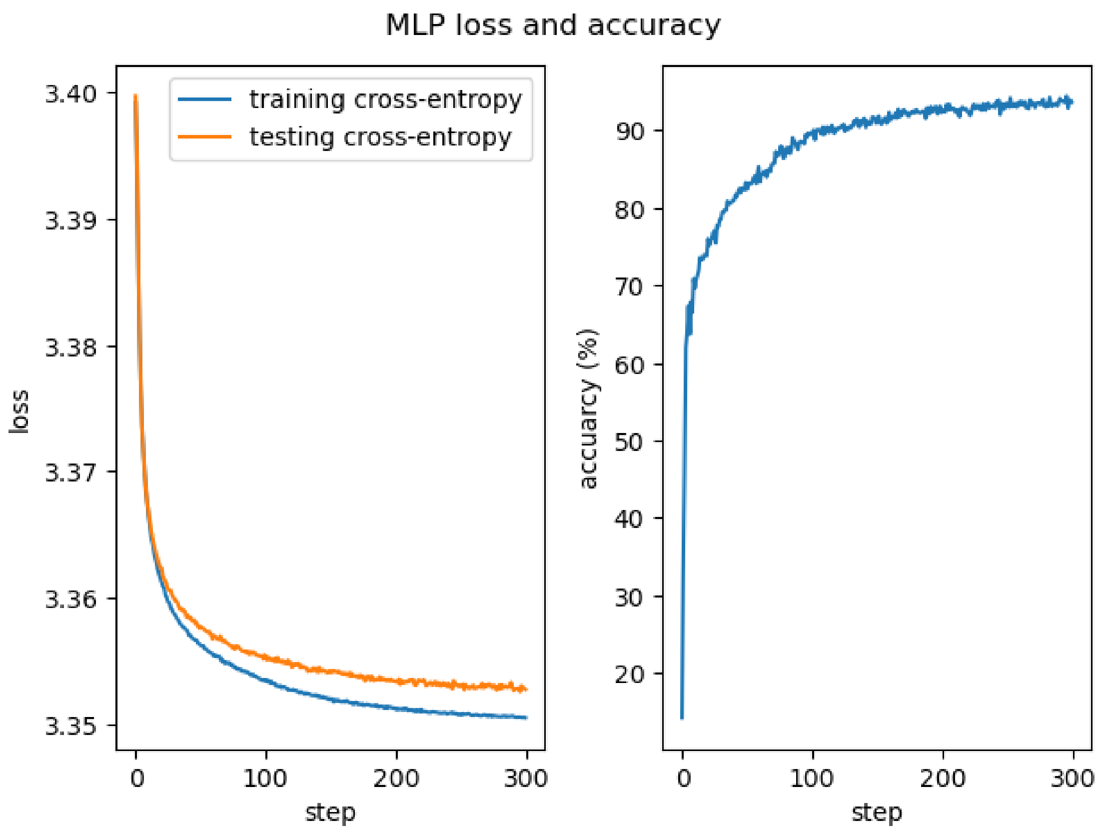
```
Epoch: 181/300 | Accuracy: 92.639%
Epoch: 182/300 | Accuracy: 92.361%
Epoch: 183/300 | Accuracy: 92.292%
Epoch: 184/300 | Accuracy: 92.222%
Epoch: 185/300 | Accuracy: 92.361%
Epoch: 186/300 | Accuracy: 92.500%
Epoch: 187/300 | Accuracy: 92.222%
Epoch: 188/300 | Accuracy: 91.528%
Epoch: 189/300 | Accuracy: 92.917%
Epoch: 190/300 | Accuracy: 92.500%
Epoch: 191/300 | Accuracy: 92.083%
Epoch: 192/300 | Accuracy: 93.056%
Epoch: 193/300 | Accuracy: 92.361%
Epoch: 194/300 | Accuracy: 92.431%
Epoch: 195/300 | Accuracy: 92.986%
Epoch: 196/300 | Accuracy: 93.056%
Epoch: 197/300 | Accuracy: 92.083%
Epoch: 198/300 | Accuracy: 92.361%
Epoch: 199/300 | Accuracy: 93.125%
Epoch: 200/300 | Accuracy: 92.292%
Epoch: 201/300 | Accuracy: 92.431%
Epoch: 202/300 | Accuracy: 92.569%
Epoch: 203/300 | Accuracy: 92.361%
Epoch: 204/300 | Accuracy: 93.264%
Epoch: 205/300 | Accuracy: 92.639%
Epoch: 206/300 | Accuracy: 93.333%
Epoch: 207/300 | Accuracy: 92.639%
Epoch: 208/300 | Accuracy: 92.708%
Epoch: 209/300 | Accuracy: 92.847%
Epoch: 210/300 | Accuracy: 92.778%
Epoch: 211/300 | Accuracy: 92.292%
Epoch: 212/300 | Accuracy: 92.639%
Epoch: 213/300 | Accuracy: 92.569%
Epoch: 214/300 | Accuracy: 91.736%
Epoch: 215/300 | Accuracy: 92.778%
Epoch: 216/300 | Accuracy: 92.986%
Epoch: 217/300 | Accuracy: 92.708%
Epoch: 218/300 | Accuracy: 92.708%
Epoch: 219/300 | Accuracy: 92.778%
Epoch: 220/300 | Accuracy: 92.431%
Epoch: 221/300 | Accuracy: 92.292%
Epoch: 222/300 | Accuracy: 91.875%
Epoch: 223/300 | Accuracy: 92.361%
Epoch: 224/300 | Accuracy: 93.056%
Epoch: 225/300 | Accuracy: 93.333%
Epoch: 226/300 | Accuracy: 92.778%
Epoch: 227/300 | Accuracy: 92.847%
Epoch: 228/300 | Accuracy: 93.194%
```

```
Epoch: 229/300 | Accuracy: 93.125%
Epoch: 230/300 | Accuracy: 92.639%
Epoch: 231/300 | Accuracy: 92.569%
Epoch: 232/300 | Accuracy: 93.403%
Epoch: 233/300 | Accuracy: 92.708%
Epoch: 234/300 | Accuracy: 92.569%
Epoch: 235/300 | Accuracy: 93.333%
Epoch: 236/300 | Accuracy: 93.472%
Epoch: 237/300 | Accuracy: 92.847%
Epoch: 238/300 | Accuracy: 93.125%
Epoch: 239/300 | Accuracy: 93.125%
Epoch: 240/300 | Accuracy: 93.194%
Epoch: 241/300 | Accuracy: 92.917%
Epoch: 242/300 | Accuracy: 93.681%
Epoch: 243/300 | Accuracy: 93.264%
Epoch: 244/300 | Accuracy: 93.889%
Epoch: 245/300 | Accuracy: 92.014%
Epoch: 246/300 | Accuracy: 92.917%
Epoch: 247/300 | Accuracy: 93.403%
Epoch: 248/300 | Accuracy: 92.708%
Epoch: 249/300 | Accuracy: 93.125%
Epoch: 250/300 | Accuracy: 93.472%
Epoch: 251/300 | Accuracy: 93.333%
Epoch: 252/300 | Accuracy: 92.986%
Epoch: 253/300 | Accuracy: 94.167%
Epoch: 254/300 | Accuracy: 93.056%
Epoch: 255/300 | Accuracy: 93.056%
Epoch: 256/300 | Accuracy: 93.403%
Epoch: 257/300 | Accuracy: 93.125%
Epoch: 258/300 | Accuracy: 92.986%
Epoch: 259/300 | Accuracy: 93.611%
Epoch: 260/300 | Accuracy: 93.264%
Epoch: 261/300 | Accuracy: 93.750%
Epoch: 262/300 | Accuracy: 93.403%
Epoch: 263/300 | Accuracy: 93.264%
Epoch: 264/300 | Accuracy: 93.681%
Epoch: 265/300 | Accuracy: 92.917%
Epoch: 266/300 | Accuracy: 93.056%
Epoch: 267/300 | Accuracy: 92.778%
Epoch: 268/300 | Accuracy: 93.889%
Epoch: 269/300 | Accuracy: 93.333%
Epoch: 270/300 | Accuracy: 93.611%
Epoch: 271/300 | Accuracy: 93.333%
Epoch: 272/300 | Accuracy: 92.917%
Epoch: 273/300 | Accuracy: 93.472%
Epoch: 274/300 | Accuracy: 93.542%
Epoch: 275/300 | Accuracy: 93.542%
Epoch: 276/300 | Accuracy: 93.333%
```

```
Epoch: 277/300 | Accuracy: 93.472%
Epoch: 278/300 | Accuracy: 93.958%
Epoch: 279/300 | Accuracy: 93.125%
Epoch: 280/300 | Accuracy: 92.986%
Epoch: 281/300 | Accuracy: 93.403%
Epoch: 282/300 | Accuracy: 93.056%
Epoch: 283/300 | Accuracy: 94.236%
Epoch: 284/300 | Accuracy: 93.472%
Epoch: 285/300 | Accuracy: 93.056%
Epoch: 286/300 | Accuracy: 93.264%
Epoch: 287/300 | Accuracy: 93.333%
Epoch: 288/300 | Accuracy: 93.472%
Epoch: 289/300 | Accuracy: 93.125%
Epoch: 290/300 | Accuracy: 93.611%
Epoch: 291/300 | Accuracy: 94.097%
Epoch: 292/300 | Accuracy: 93.819%
Epoch: 293/300 | Accuracy: 93.681%
Epoch: 294/300 | Accuracy: 93.750%
Epoch: 295/300 | Accuracy: 93.611%
Epoch: 296/300 | Accuracy: 94.375%
Epoch: 297/300 | Accuracy: 92.847%
Epoch: 298/300 | Accuracy: 93.611%
Epoch: 299/300 | Accuracy: 93.889%
Epoch: 300/300 | Accuracy: 93.611%
Total Train Time: 15.65 min
```

[27]:

```
n_epochs = 300

plt.figure()
plt.subplot(1,2,1)
plt.plot(range(n_epochs), train_losses, label='training cross-entropy')
plt.plot(range(n_epochs), test_losses, label='testing cross-entropy')
plt.xlabel('step')
plt.ylabel('loss')
plt.legend()


plt.subplot(1,2,2)
plt.plot(range(n_epochs), accuracies)
plt.xlabel('step')
plt.ylabel('accuarcy (%)')
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.suptitle("MLP loss and accuracy")
plt.show()
```



The MLP did converge to an accuracy of ~93% in 300 epochs, but the training time was over 7 minutes, which is fairly long for a lightweight model like ours. As shown in the codeblock where we declared the MLP class, the network has over 160,000 parameters with just a single hiddne layer. In the next section, we introduce a different model architecture that resolves these issues.

0.4 Convolutional Neural Network (CNN) - LeNet5

While an MLP can perform image classification, it typically requires the image data to be flattened, which results in the loss of spatial information. Consequently, MLPs generally perform worse on image data compared to CNNs, as they cannot effectively capture spatial features.

CNNs, however, excel at image classification because they preserve the spatial structure of images through convolutional layers. These layers also reduce the number of parameters by sharing weights, enhancing the model's ability to learn intricate patterns. As a result, CNNs tend to achieve higher accuracy and efficiency in image classification tasks, especially when dealing with large-scale datasets and complex visual patterns.

For the purpose of Hangul classification, we will recreate the LeNet-5 architecture, a CNN structure proposed by LeCun et al. We will assess whether LeNet-5 can outperform the MLP shown in the previous section.

The LeNet-5 Architecture is shown below:

By Cmglee - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=104937230

The code block below implements the LeNet5 architecture shown in the diagram above. The model details are also printed below the code block.

[28]:
```
# Define LeNet object

class LeNet(nn.Module):
    """
    A model that implements a logistic regression classifier.
    """
    def __init__(self, num_classes):
        """
        Constructor for LeNet object

        Args:
            input_size (int): size of input tensor
            num_classes (int): number of classes the model can predict
        """
        super(LeNet, self).__init__()

        self.layers = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2),
            nn.ReLU(),

            nn.AvgPool2d(kernel_size=2, stride=2),

            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
            nn.ReLU(),

            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),

            nn.Linear(in_features= 400, out_features=120),
            nn.ReLU(),

            nn.Linear(in_features=120, out_features=84),
            nn.ReLU(),

            nn.Linear(in_features=84, out_features=num_classes)
        )

    def forward(self, x):
        """
        Forward pass of the model

        Args:
            x (tensor): Input to the model
```

```python
        Returns:
            out (tensor): Output of the model

        """
        # First Conv Layer

        return self.layers(x)

print(summary(LeNet(30).to(device), (1,28,28)))
```

```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
 Sequential: 1-1                        [-1, 30]                  --
|    Conv2d: 2-1                        [-1, 6, 28, 28]           156
|    ReLU: 2-2                          [-1, 6, 28, 28]           --
|    AvgPool2d: 2-3                     [-1, 6, 14, 14]           --
|    Conv2d: 2-4                        [-1, 16, 10, 10]          2,416
|    ReLU: 2-5                          [-1, 16, 10, 10]          --
|    AvgPool2d: 2-6                     [-1, 16, 5, 5]            --
|    Flatten: 2-7                       [-1, 400]                 --
|    Linear: 2-8                        [-1, 120]                 48,120
|    ReLU: 2-9                          [-1, 120]                 --
|    Linear: 2-10                       [-1, 84]                  10,164
|    ReLU: 2-11                         [-1, 84]                  --
|    Linear: 2-12                       [-1, 30]                  2,550
================================================================================
==========
Total params: 63,406
Trainable params: 63,406
Non-trainable params: 0
Total mult-adds (M): 0.48
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 0.24
Estimated Total Size (MB): 0.29
================================================================================
==========
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
```

```
==========
 Sequential: 1-1                        [-1, 30]                  --
|    Conv2d: 2-1                        [-1, 6, 28, 28]           156
|    ReLU: 2-2                          [-1, 6, 28, 28]           --
|    AvgPool2d: 2-3                     [-1, 6, 14, 14]           --
|    Conv2d: 2-4                        [-1, 16, 10, 10]          2,416
|    ReLU: 2-5                          [-1, 16, 10, 10]          --
|    AvgPool2d: 2-6                     [-1, 16, 5, 5]            --
|    Flatten: 2-7                       [-1, 400]                 --
|    Linear: 2-8                        [-1, 120]                 48,120
|    ReLU: 2-9                          [-1, 120]                 --
|    Linear: 2-10                       [-1, 84]                  10,164
|    ReLU: 2-11                         [-1, 84]                  --
|    Linear: 2-12                       [-1, 30]                  2,550
================================================================================
==========
Total params: 63,406
Trainable params: 63,406
Non-trainable params: 0
Total mult-adds (M): 0.48
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 0.24
Estimated Total Size (MB): 0.29
================================================================================
==========
```

[29]:
```python
def train_LeNet(trainloader, testloader, n_epochs, learning_rate):
    model = LeNet(num_classes = 30).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    train_losses = np.zeros((n_epochs,))
    test_losses = np.zeros((n_epochs,))
    accuracies = np.zeros((n_epochs,))

    start = time.time()
    for epoch in range(n_epochs):

        model.train()
        for images, labels in trainloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            optimizer.zero_grad()
            loss = criterion(outputs, labels)
            loss.backward()
```

```python
            optimizer.step()


        model.eval()
        with torch.no_grad():

            total_test_loss, total_train_loss, correct_preds = 0.0, 0.0, 0.0

            for images, labels in testloader:
                images, labels = images.to(device), labels.to(device)
                test_outputs = model(images)
                test_loss = criterion(test_outputs, labels)
                total_test_loss += test_loss.item()

                _, preds = torch.max(test_outputs, 1)
                correct_preds += (preds == labels).sum().item()

            for images, labels in trainloader:
                images, labels = images.to(device), labels.to(device)
                train_outputs = model(images)
                train_loss = criterion(train_outputs, labels)
                total_train_loss += train_loss.item()

            accuracies[epoch] = correct_preds / len(testloader.dataset) * 100
            test_losses[epoch] = total_test_loss / len(testloader)
            train_losses[epoch] = total_train_loss / len(trainloader)

            print('Epoch: %03d/%03d | Accuracy: %.3f%%' %(epoch + 1, n_epochs,␣
↪accuracies[epoch]))

    print("Total Train Time: %.2f min" % ((time.time() - start)/60))
    return train_losses, test_losses, accuracies, model
```

[30]:
```python
train_losses, test_losses, accuracies, model = train_LeNet(train_loader,␣
↪test_loader, 50, 0.001)

torch.save(model.state_dict(), "lenet.pth")
```

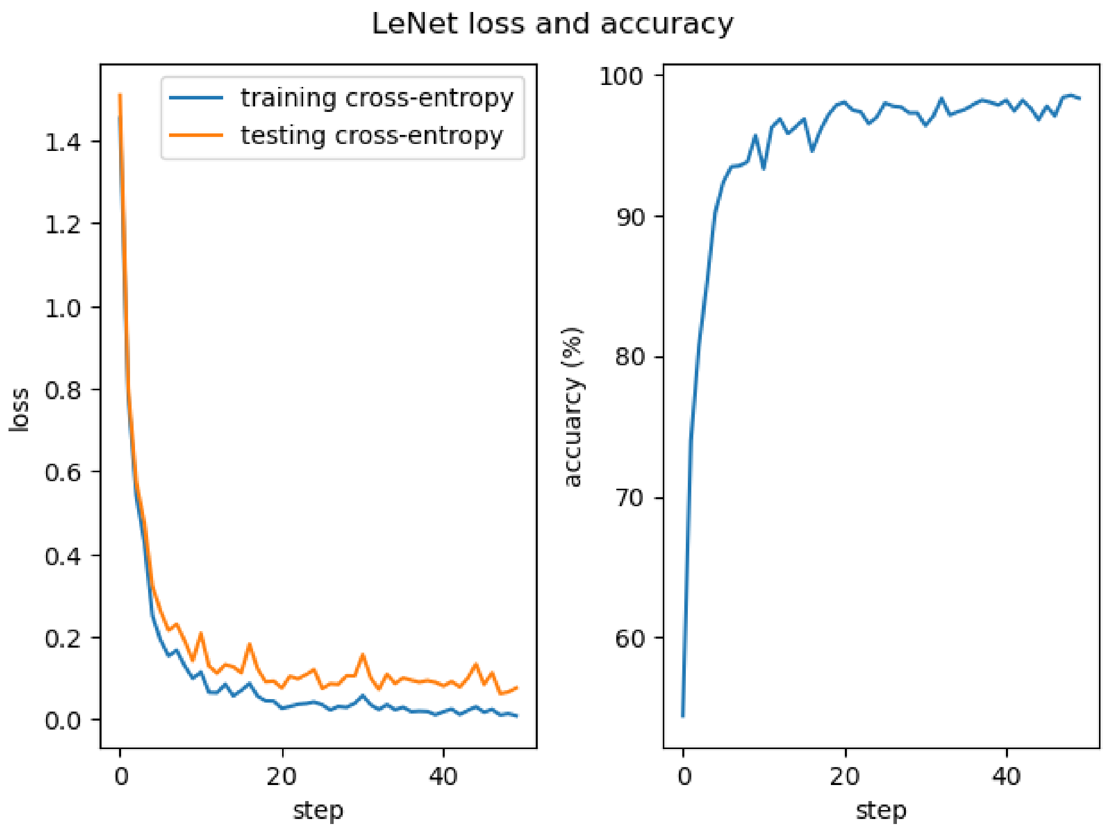
```
Epoch: 001/050 | Accuracy: 54.444%
Epoch: 002/050 | Accuracy: 73.958%
Epoch: 003/050 | Accuracy: 80.694%
Epoch: 004/050 | Accuracy: 85.208%
Epoch: 005/050 | Accuracy: 90.139%
Epoch: 006/050 | Accuracy: 92.292%
Epoch: 007/050 | Accuracy: 93.403%
Epoch: 008/050 | Accuracy: 93.472%
Epoch: 009/050 | Accuracy: 93.750%
```

```
Epoch: 010/050 | Accuracy: 95.625%
Epoch: 011/050 | Accuracy: 93.264%
Epoch: 012/050 | Accuracy: 96.181%
Epoch: 013/050 | Accuracy: 96.806%
Epoch: 014/050 | Accuracy: 95.764%
Epoch: 015/050 | Accuracy: 96.250%
Epoch: 016/050 | Accuracy: 96.806%
Epoch: 017/050 | Accuracy: 94.514%
Epoch: 018/050 | Accuracy: 95.972%
Epoch: 019/050 | Accuracy: 97.083%
Epoch: 020/050 | Accuracy: 97.778%
Epoch: 021/050 | Accuracy: 97.986%
Epoch: 022/050 | Accuracy: 97.431%
Epoch: 023/050 | Accuracy: 97.292%
Epoch: 024/050 | Accuracy: 96.458%
Epoch: 025/050 | Accuracy: 96.944%
Epoch: 026/050 | Accuracy: 97.917%
Epoch: 027/050 | Accuracy: 97.708%
Epoch: 028/050 | Accuracy: 97.639%
Epoch: 029/050 | Accuracy: 97.222%
Epoch: 030/050 | Accuracy: 97.222%
Epoch: 031/050 | Accuracy: 96.319%
Epoch: 032/050 | Accuracy: 97.014%
Epoch: 033/050 | Accuracy: 98.264%
Epoch: 034/050 | Accuracy: 97.083%
Epoch: 035/050 | Accuracy: 97.292%
Epoch: 036/050 | Accuracy: 97.500%
Epoch: 037/050 | Accuracy: 97.847%
Epoch: 038/050 | Accuracy: 98.125%
Epoch: 039/050 | Accuracy: 97.986%
Epoch: 040/050 | Accuracy: 97.778%
Epoch: 041/050 | Accuracy: 98.125%
Epoch: 042/050 | Accuracy: 97.361%
Epoch: 043/050 | Accuracy: 98.125%
Epoch: 044/050 | Accuracy: 97.569%
Epoch: 045/050 | Accuracy: 96.736%
Epoch: 046/050 | Accuracy: 97.708%
Epoch: 047/050 | Accuracy: 97.014%
Epoch: 048/050 | Accuracy: 98.333%
Epoch: 049/050 | Accuracy: 98.472%
Epoch: 050/050 | Accuracy: 98.264%
Total Train Time: 2.68 min
```

[31]:

```
n_epochs = 50

plt.figure()
plt.subplot(1,2,1)
```

```
plt.plot(range(n_epochs), train_losses, label='training cross-entropy')
plt.plot(range(n_epochs), test_losses, label='testing cross-entropy')
plt.xlabel('step')
plt.ylabel('loss')
plt.legend()


plt.subplot(1,2,2)
plt.plot(range(n_epochs), accuracies)
plt.xlabel('step')
plt.ylabel('accuarcy (%)')
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.suptitle("LeNet loss and accuracy")
plt.show()
```



As shown above, LeNet achieved an accuracy of approximately 98% in under a minute using only 50 epochs, which is about six times faster than the MLP architecture. This speed is due to convolutional layers having significantly fewer parameters than fully-connected layers, allowing the model to train faster. Additionally, the structure of convolutional layers is better suited for capturing unique features across different classes, which contributes to the overall improvement in classification accuracy.

0.5 Model Evaluation

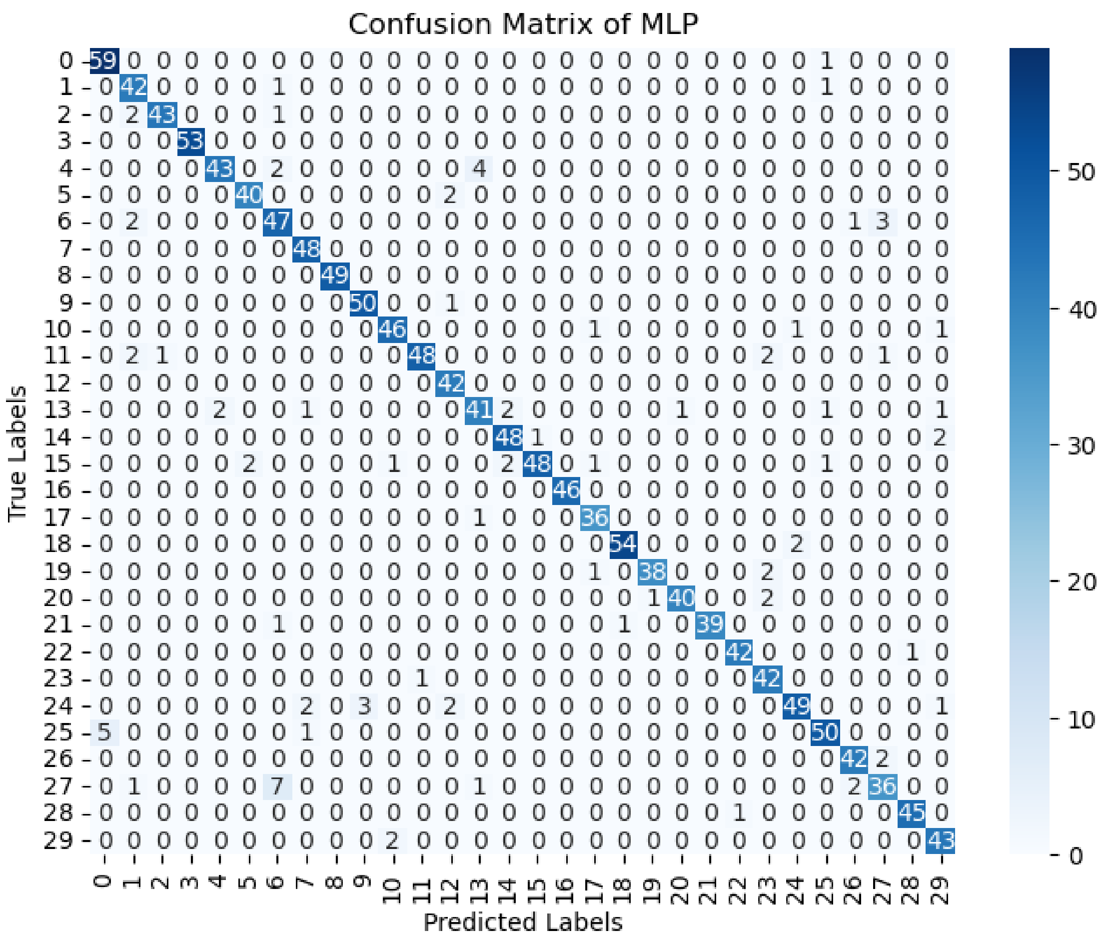
[32]:
```
# Load saved model

mlp = Hangul_MLP(input_size=784, num_classes=30)
mlp.load_state_dict(torch.load('mlp.pth'))

lenet = LeNet(num_classes=30)
lenet.load_state_dict(torch.load('lenet.pth'))
```

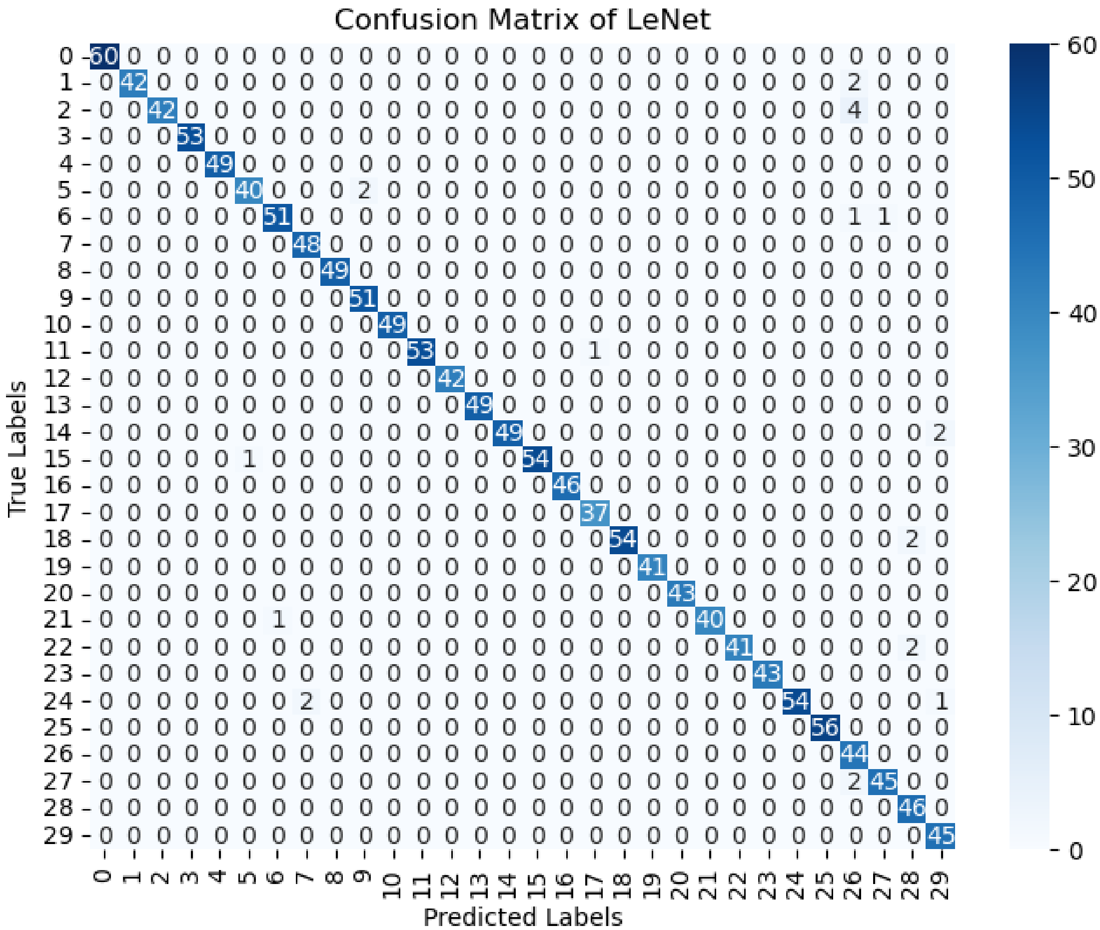
[32]:
```
<All keys matched successfully>
```

A common method for evaluating a classification model is by using a confusion matrix. This matrix enables the calculation of metrics such as True Positives, True Negatives, False Positives, and False Negatives. These metrics can then be used to determine measures like Precision and Recall.

[33]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

mlp_preds = []
mlp_true_labels = []

lenet_preds = []
lenet_true_labels = []

with torch.no_grad():
    for imgs, labels_true in test_loader:

        output_mlp = mlp(imgs.view(imgs.size(0), -1))
        output_lenet = lenet(imgs)

        _, labels_pred_mlp = torch.max(output_mlp, 1)
        _, labels_pred_lenet = torch.max(output_lenet, 1)

        mlp_preds.extend(labels_pred_mlp.numpy())
        mlp_true_labels.extend(labels_true.numpy())

        lenet_preds.extend(labels_pred_lenet.numpy())
        lenet_true_labels.extend(labels_true.numpy())

conf_matrix_mlp = confusion_matrix(mlp_true_labels, mlp_preds)
conf_matrix_lenet = confusion_matrix(lenet_true_labels, lenet_preds)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix_mlp, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
```

```python
plt.title('Confusion Matrix of MLP')
plt.show()

plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix_lenet, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix of LeNet')
plt.show()
```

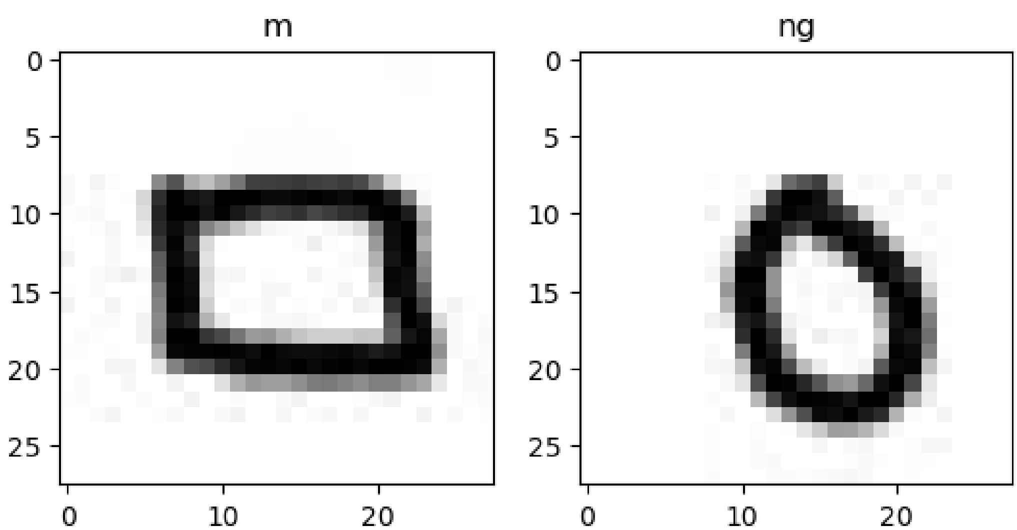


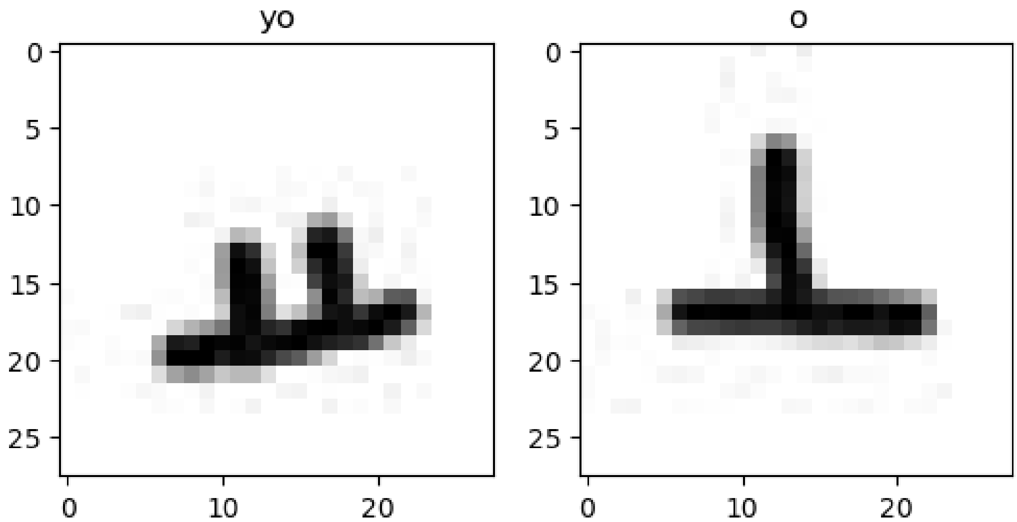


The confusion matrix above shows that both the MLP and LeNet perform extremely well on the test images, which is shown by the diagonal line where the predicted label matches the true label. One thing to note is that there were two classes that the LeNet made a few errors on. There were 13 images predicted to be class #15 that turned out to be class #17 and 9 images predicted to be class 28 that turned out to be class #18. We visualize these classes below.

[34]:
```
classes = dataset_unaug.classes

# class 15 vs class 17

plt.subplot(1,2,1)
img1, label1 = dataset_unaug[15*80]
plt.imshow(img1.squeeze().numpy(), cmap='gray')
plt.title(classes[label1])

plt.subplot(1,2,2)
img2, label2 = dataset_unaug[17*80]
plt.imshow(img2.squeeze().numpy(), cmap='gray')
```

```
plt.title(classes[label2])
plt.show()

# class 28 vs class 18
plt.subplot(1,2,1)
img1, label1 = dataset_unaug[28*80]
plt.imshow(img1.squeeze().numpy(), cmap='gray')
plt.title(classes[label1])

plt.subplot(1,2,2)
img2, label2 = dataset_unaug[18*80]
plt.imshow(img2.squeeze().numpy(), cmap='gray')
plt.title(classes[label2])
plt.show()
```





Upon examining the classes with the most errors, we observed that the two characters share many visual similarities. As a fluent Korean speaker, I find that poorly written `(m, ng)` and `(yo, o)` characters can be difficult to distinguish, even for native speakers. This issue may stem from the limited dataset, which contains only 80 images per class. As a result, the dataset might have failed to capture the small, distinct visual features that differentiate these similar classes.