CISC3014 Information Retrieval and Web Search (Report)

澳門大學

UNIVERSIDADE DE MACAU

UNIVERSITY OF MACAU

<u>Topic :</u>

Deep Learning Model (Pytorch) Classification on Kuzushiji MNIST Dataset

<u>Members :</u>

1) Wong Kai Yuan (DC026157)
2) Tse Tsz Leong (DB927141)
3) Chan Weng Kin (DB927473)

<u>Prof :</u>

Dr. Pengyang Wang

Table of Content :

1) Abstract
2) Introduction
    2.1 Pytorch - Deep Learning model
    2.2 Datasets (Kuzushiji MNIST) description
    2.3 MLP & LeNet description
    2.4 Goal
3) Implementation on Multilayer Perceptron (MLP)
    3.1 Data Processing
    3.2 Defining the model
    3.3 Training the model
    3.4 Result of the model
    3.5 Final Result on Testing
4) Implementation on LeNet
    4.1 Data Processing
    4.2 Defining the model
    4.3 Training the model
    4.4 Result of the model
    4.5 Final Result on Testing
5) Comparison
    5.1 Test accuracy & error on each model
    5.2 Confusion Matrix
    5.3 Plot incorrect label visualisation
6) Conclusion
7) Reference

1) Abstract

In this project, we target to do classification using deep learning models of Pytorch such as Multilayer Perceptron (MLP) and Lenet (model of CNN) on Kuzushiji MNIST dataset. Then, trying to get some information as a conclusion by doing comparison on each implementation.

2) Introduction

2.1 Pytorch - Deep Learning model

We should first know Torch before we can introduce PyTorch.

Torch is a platform for scientific computing that prioritises GPUs and offers extensive support for machine learning methods. Because of the fast and simple scripting language LuaJIT and the underlying C/CUDA implementation, it is simple to use and effective.

For uses like NLP, PyTorch is an open source machine learning package for Python that is based on Torch. The AI research team at Facebook played a major role in its development.

A Python program called PyTorch offers two sophisticated features:

- Powerful GPU acceleration for tensor computation (e.g. NumPy)
- Deep learning systems with automated derivation

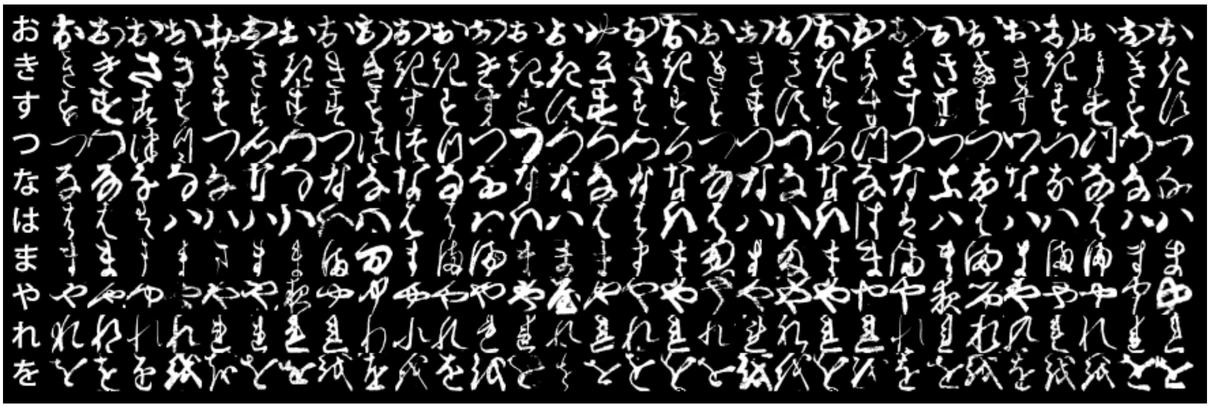
Why are we not using Tensorflow?

While TensorFlow offers better visualization and deployment of trained models to production, PyTorch is more flexible, has better debugging capabilities, and takes less time to train.

2.2 Datasets (Kuzushiji MNIST) description

KMNIST is a dataset, adapted from Kuzushiji(崩し字) Dataset, as a drop-in replacement for MNIST dataset, which is the most famous dataset in the machine learning community.

KMNIST Dataset is created by ROIS-DS Center for Open Data in the Humanities (CODH), based on Kuzushiji Dataset created by National Institute of Japanese Literature.



*The 10 classes of Kuzushiji-MNIST, with the first column showing each character's modern hiragana counterpart.

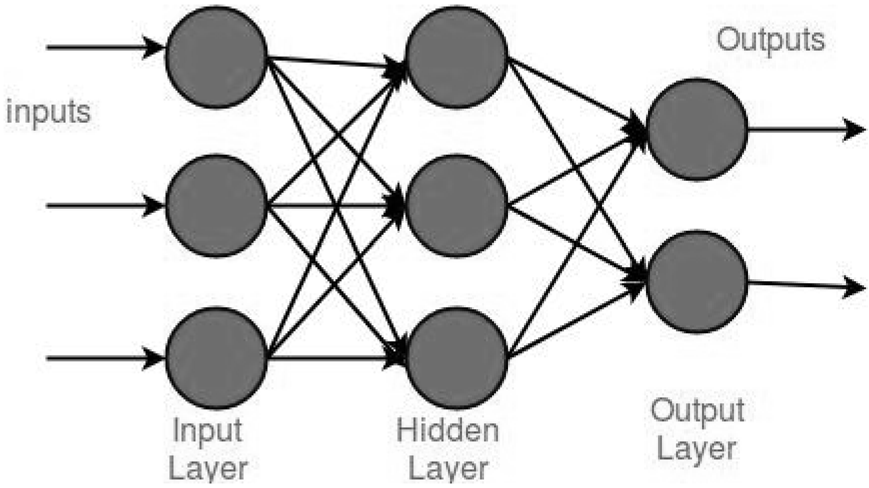
Kuzushiji-MNIST is a drop-in replacement for the MNIST dataset (28x28 grayscale, 70,000 images), provided in the original MNIST format as well as a NumPy format.

Since MNIST restricts us to 10 classes, we chose one character to represent each of the 10 rows of Hiragana(平仮名) when creating Kuzushiji-MNIST.

2.3 MLP & LeNet description

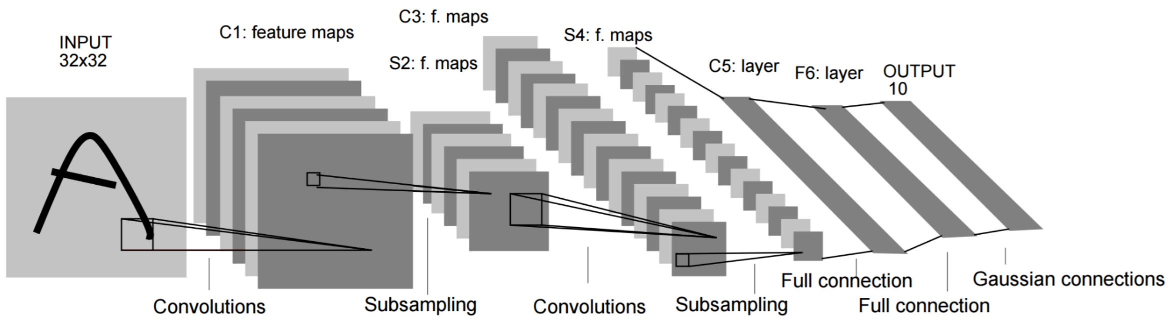
Multilayer Perceptron (MLP)

MLP is a kind of forward-propagation neural networks that employs "backward-propagation" to achieve supervised learning and has at least three structural layers (model learning).
This particular instance is built on a deep neural network(DNN).



LeNet (CNN model)
LeNet is a network architecture proposed by Yann LeCun's team, and is the originator of the convolutional neural network.

The architecture consists of two convolutional layers, a pooling layer, a fully connected layer, and a final Gaussian connection layer to recognize handwritten digital images.



2.4 Goal
We will be building MLP & LeNet models to perform image classification on the Kuzushiji MNIST dataset using Pytorch & Torchvision.

3) Implementation on Multilayer Perceptron (MLP)

Link : https://colab.research.google.com/drive/1NF_3W18yNx-ii_bPPJchOqTzN_gXTemf?usp=sharing

3.1 Data Processing

1) Import all needed modules we need

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets

from sklearn import metrics
from sklearn import decomposition
from sklearn import manifold
from tqdm.notebook import trange, tqdm
import matplotlib.pyplot as plt
import numpy as np

import copy
import random
import time
```

2) Find mean & standard deviation for normalisation later (so that it can be trained more reliable)

```
[15] ROOT = '.data'

     train_data = datasets.KMNIST(root=ROOT,
                                  train=True,
                                  download=True)
```

```
[16] mean = train_data.data.float().mean() / 255
     std = train_data.data.float().std() / 255
```

3) Define transformation for both train & test data. The transform we use are :

- RandomRotation : randomly rotates the image between (-x, +x) degrees, where we have set x = 5.
- RandomCrop : this first adds padding around our image, 2 pixels here, to artificially make it bigger, before taking a random 28x28 square crop of the image.
- ToTensor() : this converts the image from a PIL image into a PyTorch tensor.

- Normalise : this subtracts the mean and divides by the standard deviations given.

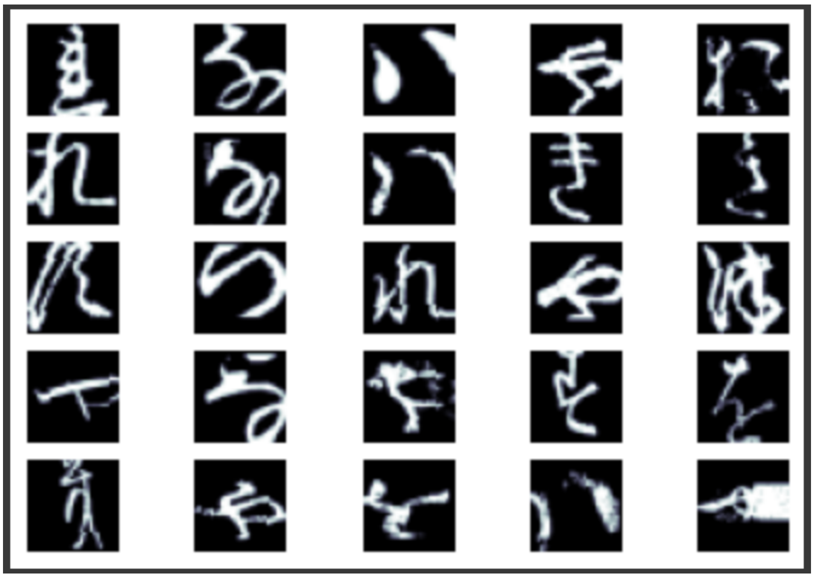
```
[17] train_transforms = transforms.Compose([
                            transforms.RandomRotation(5, fill=(0,)),
                            transforms.RandomCrop(28, padding=2),
                            transforms.ToTensor(),
                            transforms.Normalize(mean=[mean], std=[std])
                                      ])

     test_transforms = transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize(mean=[mean], std=[std])
                                     ])
```

```
train_data = datasets.KMNIST(root=ROOT,
                            train=True,
                            download=True,
                            transform=train_transforms)

test_data = datasets.KMNIST(root=ROOT,
                           train=False,
                           download=True,
                           transform=test_transforms)
```

4) Visualisation of our training data



5) Use random split to further create validation dataset from 10% of training dataset.

```
VALID_RATIO = 0.9

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

```
54000
```

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

6) The number of training, testing and validation datasets are below :

```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 54000
Number of validation examples: 6000
Number of testing examples: 10000
```

7) We define DataLoader as batch of 64 for each datasets

```
BATCH_SIZE = 64

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

valid_iterator = data.DataLoader(valid_data,
                                 batch_size=BATCH_SIZE)

test_iterator = data.DataLoader(test_data,
                                batch_size=BATCH_SIZE)
```

3.2 Defining the model

1) We define our model with 2 hidden layers with linear layer & nonlinear functions such as ReLU.

```
class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.input_fc = nn.Linear(input_dim, 250)
        self.hidden_fc = nn.Linear(250, 100)
        self.output_fc = nn.Linear(100, 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(h_1))

        y_pred = self.output_fc(h_2)

        return y_pred, h_2
```

2) Create an instance of our model and give the correct input & output.

```
INPUT_DIM = 28 * 28
OUTPUT_DIM = 10

model = MLP(INPUT_DIM, OUTPUT_DIM)
```

3) Define our optimizer : Adam optimizer with default parameters to update our model.
4) Define our criterion : CrossEntropyLoss
5) Define our device : place our data & model on GPU, if have one
6) Place our model, criterion on device

```
optimizer = optim.Adam(model.parameters())

criterion = nn.CrossEntropyLoss()

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = model.to(device)
criterion = criterion.to(device)
```

3.3 Training the model

1) Define a function to calculate the accuracy of our model

```
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim=True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
```

2) Define our training loop as follow :

- put our model into train mode
- iterate over our dataloader, returning batches of (image, label)
- place the batch on to our GPU, if we have one
- clear the gradients calculated from the last batch
- pass our batch of images, x, through to model to get predictions, y_pred

- calculate the loss between our predictions and the actual labels
- calculate the accuracy between our predictions and the actual labels
- calculate the gradients of each parameter
- update the parameters by taking an optimizer step
- update our metrics

```python
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in tqdm(iterator, desc="Training", leave=False):

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

3) Define our evaluate loop as follow :

The evaluation loop is similar to the training loop. The differences are:

- we put our model into evaluation mode with model.eval()
- we wrap the iterations inside a with torch.no_grad()
- we do not zero gradients as we are not calculating any
- we do not calculate gradients as we are not updating parameters
- we do not take an optimizer step as we are not calculating gradients

```python
def evaluate(model, iterator, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in tqdm(iterator, desc="Evaluating", leave=False):

            x = x.to(device)
            y = y.to(device)

            y_pred, _ = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

4) Training the model !!

```python
best_valid_loss = float('inf')

for epoch in trange(EPOCHS):

    start_time = time.monotonic()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion, device)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion, device)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut1-model.pt')

    end_time = time.monotonic()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

3.4 Result of the model

As each Epoch, the training loss keeps decreasing and accuracy keeps increasing, this shows that our training is going well.

```
Epoch: 01 | Epoch Time: 0m 20s
        Train Loss: 0.554 | Train Acc: 82.45%
         Val. Loss: 0.229 |  Val. Acc: 93.28%
Epoch: 02 | Epoch Time: 0m 20s
        Train Loss: 0.295 | Train Acc: 90.87%
         Val. Loss: 0.180 |  Val. Acc: 94.54%
Epoch: 03 | Epoch Time: 0m 21s
        Train Loss: 0.236 | Train Acc: 92.64%
         Val. Loss: 0.163 |  Val. Acc: 94.89%
Epoch: 04 | Epoch Time: 0m 20s
        Train Loss: 0.209 | Train Acc: 93.51%
         Val. Loss: 0.155 |  Val. Acc: 95.32%
Epoch: 05 | Epoch Time: 0m 21s
        Train Loss: 0.190 | Train Acc: 94.08%
         Val. Loss: 0.124 |  Val. Acc: 96.59%
Epoch: 06 | Epoch Time: 0m 20s
        Train Loss: 0.172 | Train Acc: 94.74%
         Val. Loss: 0.117 |  Val. Acc: 96.54%
Epoch: 07 | Epoch Time: 0m 20s
        Train Loss: 0.160 | Train Acc: 95.00%
         Val. Loss: 0.123 |  Val. Acc: 96.39%
Epoch: 08 | Epoch Time: 0m 20s
        Train Loss: 0.153 | Train Acc: 95.22%
         Val. Loss: 0.110 |  Val. Acc: 96.56%
Epoch: 09 | Epoch Time: 0m 20s
        Train Loss: 0.146 | Train Acc: 95.37%
         Val. Loss: 0.110 |  Val. Acc: 96.96%
Epoch: 10 | Epoch Time: 0m 20s
        Train Loss: 0.137 | Train Acc: 95.66%
         Val. Loss: 0.102 |  Val. Acc: 96.95%
Epoch: 11 | Epoch Time: 0m 21s
        Train Loss: 0.133 | Train Acc: 95.82%
         Val. Loss: 0.101 |  Val. Acc: 96.62%
Epoch: 12 | Epoch Time: 0m 20s
        Train Loss: 0.133 | Train Acc: 95.92%
         Val. Loss: 0.117 |  Val. Acc: 96.46%
Epoch: 13 | Epoch Time: 0m 20s
        Train Loss: 0.129 | Train Acc: 96.01%
         Val. Loss: 0.109 |  Val. Acc: 96.81%
Epoch: 14 | Epoch Time: 0m 20s
        Train Loss: 0.126 | Train Acc: 96.05%
         Val. Loss: 0.102 |  Val. Acc: 97.03%
Epoch: 15 | Epoch Time: 0m 20s
        Train Loss: 0.117 | Train Acc: 96.36%
         Val. Loss: 0.101 |  Val. Acc: 97.05%
Epoch: 16 | Epoch Time: 0m 20s
        Train Loss: 0.120 | Train Acc: 96.33%
         Val. Loss: 0.098 |  Val. Acc: 97.14%
Epoch: 17 | Epoch Time: 0m 20s
        Train Loss: 0.112 | Train Acc: 96.34%
         Val. Loss: 0.104 |  Val. Acc: 97.07%
Epoch: 18 | Epoch Time: 0m 20s
        Train Loss: 0.109 | Train Acc: 96.53%
         Val. Loss: 0.103 |  Val. Acc: 97.25%
Epoch: 19 | Epoch Time: 0m 20s
        Train Loss: 0.109 | Train Acc: 96.53%
         Val. Loss: 0.096 |  Val. Acc: 97.10%
Epoch: 20 | Epoch Time: 0m 20s
        Train Loss: 0.107 | Train Acc: 96.53%
         Val. Loss: 0.108 |  Val. Acc: 97.01%
```

3.5 Final Result on Testing Data

1) Load our parameters of the model that have the best performance on validation set to evaluate our model on testing dataset
2) Print Test loss & Test accuracy

```
model.load_state_dict(torch.load('tut1-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion, device)


print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
Test Loss: 0.286 | Test Acc: 92.24%
```

Our MLP model is able to achieve Test Loss of 0.286 & Test accuracy of 92.24% !

4) Implementation on LeNet

Link : https://colab.research.google.com/drive/11oXqMG7wJwviNJ2eg45isFpb6sxIapub?usp=sharing

4.1 Data Processing

1. Import all needed modules we need

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets

from sklearn import decomposition
from sklearn import manifold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from tqdm.notebook import tqdm, trange
import matplotlib.pyplot as plt
import numpy as np

import copy
import random
import time
```

2. Find mean & standard deviation for normalization later

```
ROOT = '.data'

train_data = datasets.KMNIST(root=ROOT,
                                                        train=True,
                                                        download=True)

mean = train_data.data.float().mean() / 255
std = train_data.data.float().std() / 255
```

3. Define transformation for both train & test data. The transform we use are :
    - RandomRotation : randomly rotates the image between (-x, +x) degrees, where we have set x = 5.

- RandomCrop : this first adds padding around our image, 2 pixels here, to artificially make it bigger, before taking a random 28x28 square crop of the image.
- ToTensor() : this converts the image from a PIL image into a PyTorch tensor.
- Normalise : this subtracts the mean and divides by the standard deviations given.

```
train_transforms = transforms.Compose([
                                       transforms.RandomRotation(5, fill=(0,)),
                                       transforms.RandomCrop(28, padding=2),
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=[mean], std=[std])
                                      ])

test_transforms = transforms.Compose([
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[mean], std=[std])
                                     ])
```

```
train_data = datasets.KMNIST(root=ROOT,
                             train=True,
                             download=True,
                             transform=train_transforms)

test_data = datasets.KMNIST(root=ROOT,
                            train=False,
                            download=True,
                            transform=test_transforms)
```

4. Use random split to further create validation dataset from 10% of training dataset.

```
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])
```

5. The number of training, testing and validation datasets are below :

```
Number of training examples: 54000
Number of validation examples: 6000
Number of testing examples: 10000
```

6. We define DataLoader as batch of 64 for each datasets

```
BATCH_SIZE = 64

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

valid_iterator = data.DataLoader(valid_data,
                                 batch_size=BATCH_SIZE)

test_iterator = data.DataLoader(test_data,
                                batch_size=BATCH_SIZE)
```

4.2 Defining the model

1. We will define a standard linear layer, which includes a convolutional layer and a pooling layer.

```
class LeNet(nn.Module):
    def __init__(self, output_dim):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels=1,
                               out_channels=6,
                               kernel_size=5)

        self.conv2 = nn.Conv2d(in_channels=6,
                               out_channels=16,
                               kernel_size=5)

        self.fc_1 = nn.Linear(16 * 4 * 4, 120)
        self.fc_2 = nn.Linear(120, 84)
        self.fc_3 = nn.Linear(84, output_dim)
```

```
def forward(self, x):

    # x = [batch size, 1, 28, 28]

    x = self.conv1(x)

    # x = [batch size, 6, 24, 24]

    x = F.max_pool2d(x, kernel_size=2)

    # x = [batch size, 6, 12, 12]

    x = F.relu(x)

    x = self.conv2(x)
```

```
# x = batch size, 84]

x = F.relu(x)

x = self.fc_3(x)

# x = [batch size, output dim]

return x, h
```

2. Create an instance of our model and give the correct input & output.

```
[ ]  OUTPUT_DIM = 10

     model = LeNet(OUTPUT_DIM)
```

3. Define our optimizer : Adam optimizer with default parameters to update our model.
4. Define our criterion : CrossEntropyLoss
5. Define our device : place our data & model on GPU, if have one
6. Place our model, criterion on device

```
optimizer = optim.Adam(model.parameters())
```

```
criterion = nn.CrossEntropyLoss()
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
model = model.to(device)
criterion = criterion.to(device)
```

4.3 Training the model

1. Define a function to calculate the accuracy of our model
2. Define our training loop as follow :
    - put our model into train mode
    - iterate over our dataloader, returning batches of (image, label)
    - place the batch on to our GPU, if we have one
    - clear the gradients calculated from the last batch
    - pass our batch of images, x, through to model to get predictions, y_pred

- calculate the loss between our predictions and the actual labels
- calculate the accuracy between our predictions and the actual labels
- calculate the gradients of each parameter
- update the parameters by taking an optimizer step
- update our metrics

```
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in tqdm(iterator, desc="Training", leave=False):

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

3. Define our evaluate loop as follow :

The evaluation loop is similar to the training loop. The differences are:

- we put our model into evaluation mode with model.eval()
- we wrap the iterations inside a with torch.no_grad()
- we do not zero gradients as we are not calculating any

- we do not calculate gradients as we are not updating parameters
- we do not take an optimizer step as we are not calculating gradients

```
def evaluate(model, iterator, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in tqdm(iterator, desc="Evaluating", leave=False):

            x = x.to(device)
            y = y.to(device)

            y_pred, _ = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

and define a function that tells us how long an epoch takes

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

4. Training the model !!

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

4.4 Result of the model

As each Epoch, the training loss keeps decreasing and accuracy keeps increasing, this shows that our training is going well.

```
Epoch: 01 | Epoch Time: 0m 35s
	Train Loss: 0.618 | Train Acc: 80.28%
	 Val. Loss: 0.215 |  Val. Acc: 93.45%
Epoch: 02 | Epoch Time: 0m 35s
	Train Loss: 0.248 | Train Acc: 92.27%
	 Val. Loss: 0.147 |  Val. Acc: 95.81%
Epoch: 03 | Epoch Time: 0m 34s
	Train Loss: 0.181 | Train Acc: 94.33%
	 Val. Loss: 0.113 |  Val. Acc: 96.36%
Epoch: 04 | Epoch Time: 0m 34s
	Train Loss: 0.153 | Train Acc: 95.23%
	 Val. Loss: 0.088 |  Val. Acc: 97.52%
Epoch: 05 | Epoch Time: 0m 35s
	Train Loss: 0.133 | Train Acc: 95.85%
	 Val. Loss: 0.079 |  Val. Acc: 97.66%
Epoch: 06 | Epoch Time: 0m 34s
	Train Loss: 0.118 | Train Acc: 96.27%
	 Val. Loss: 0.077 |  Val. Acc: 97.71%
Epoch: 07 | Epoch Time: 0m 35s
	Train Loss: 0.108 | Train Acc: 96.56%
	 Val. Loss: 0.074 |  Val. Acc: 98.03%
Epoch: 08 | Epoch Time: 0m 34s
	Train Loss: 0.099 | Train Acc: 96.78%
	 Val. Loss: 0.063 |  Val. Acc: 98.01%
Epoch: 09 | Epoch Time: 0m 35s
	Train Loss: 0.094 | Train Acc: 97.00%
	 Val. Loss: 0.069 |  Val. Acc: 98.02%
Epoch: 10 | Epoch Time: 0m 34s
	Train Loss: 0.086 | Train Acc: 97.25%
	 Val. Loss: 0.061 |  Val. Acc: 98.04%
Epoch: 11 | Epoch Time: 0m 34s
	Train Loss: 0.081 | Train Acc: 97.44%
	 Val. Loss: 0.069 |  Val. Acc: 97.93%
Epoch: 12 | Epoch Time: 0m 35s
	Train Loss: 0.081 | Train Acc: 97.44%
	 Val. Loss: 0.068 |  Val. Acc: 97.98%
Epoch: 13 | Epoch Time: 0m 35s
	Train Loss: 0.075 | Train Acc: 97.59%
	 Val. Loss: 0.062 |  Val. Acc: 98.37%
Epoch: 14 | Epoch Time: 0m 35s
	Train Loss: 0.070 | Train Acc: 97.79%
	 Val. Loss: 0.059 |  Val. Acc: 98.19%
Epoch: 15 | Epoch Time: 0m 35s
	Train Loss: 0.070 | Train Acc: 97.80%
	 Val. Loss: 0.062 |  Val. Acc: 98.30%
Epoch: 16 | Epoch Time: 0m 34s
	Train Loss: 0.068 | Train Acc: 97.77%
	 Val. Loss: 0.060 |  Val. Acc: 98.14%
Epoch: 17 | Epoch Time: 0m 35s
	Train Loss: 0.062 | Train Acc: 98.02%
	 Val. Loss: 0.061 |  Val. Acc: 98.27%
Epoch: 18 | Epoch Time: 0m 34s
	Train Loss: 0.062 | Train Acc: 98.03%
	 Val. Loss: 0.062 |  Val. Acc: 98.39%
Epoch: 19 | Epoch Time: 0m 34s
	Train Loss: 0.062 | Train Acc: 97.95%
	 Val. Loss: 0.056 |  Val. Acc: 98.36%
Epoch: 20 | Epoch Time: 0m 34s
	Train Loss: 0.057 | Train Acc: 98.15%
	 Val. Loss: 0.062 |  Val. Acc: 98.23%
```

4.5 Final Result on Testing Data

1. Load our parameters of the model that have the best performance on validation set to evaluate our model on testing dataset
2. Print Test loss & Test accuracy

```
model.load_state_dict(torch.load('tut2-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion, device)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.166 | Test Acc: 95.46%
```

Our LeNet model is able to achieve Test Loss of 0.166 & Test accuracy of 95.46% !!

5) Comparison

5.1 Test accuracy & error on each model

	MLP model	CNN model
Test accuracy	92.24%	95.46%
Test error	0.286	0.166
Results screenshot	Test Loss: 0.286 | Test Acc: 92.24%	Test Loss: 0.166 | Test Acc: 95.46%

- According to our test results, the final accuracy of the MLP model is 92.24%, and the accuracy of the CNN model is 95.46%.
- The data results show that the accuracy of the MLP model is lower than that of the CNN model.

5.2 Confusion Matrix

Coding Part:

- “model.eval”: evaluate model ()
- "with torch.no_grad": The requirements_grad of the calculated tensor is automatically set to False.
- “torch.cat”: Splicing multiple tensors

```
def get_predictions(model, iterator, device):

    model.eval()

    images = []
    labels = []
    probs = []

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)

            y_pred, _ = model(x)

            y_prob = F.softmax(y_pred, dim=-1)

            images.append(x.cpu())
            labels.append(y.cpu())
            probs.append(y_prob.cpu())

    images = torch.cat(images, dim=0)
    labels = torch.cat(labels, dim=0)
    probs = torch.cat(probs, dim=0)

    return images, labels, probs
```

```
[ ] images, labels, probs = get_predictions(model, test_iterator, device)

    pred_labels = torch.argmax(probs, 1)
```

- “figsize”: Specify the width and height of the figure in inches
- "fig.add_subplot": Set the position of the subplot
- “Metrics.confusion_matrix”: Evaluate Classifier Accuracy
  Input true label and pred label
- “cm” is the value from which the confusion matrix is calculated

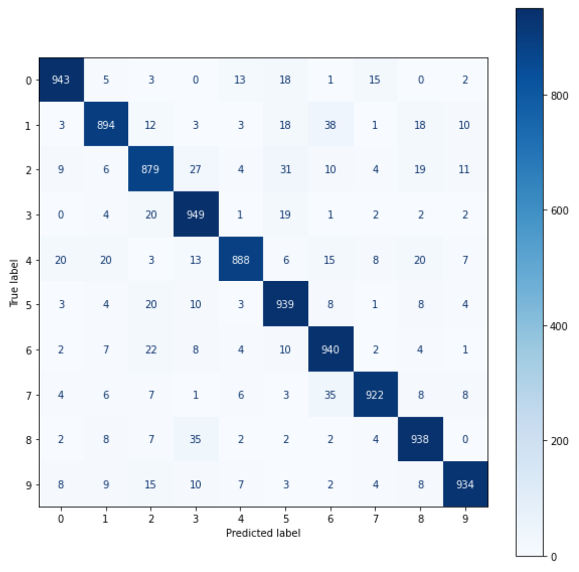
```
[ ] def plot_confusion_matrix(labels, pred_labels):

        fig = plt.figure(figsize=(10, 10))
        ax = fig.add_subplot(1, 1, 1)
        cm = metrics.confusion_matrix(labels, pred_labels)
        cm = metrics.ConfusionMatrixDisplay(cm, display_labels=range(10))
        cm.plot(values_format='d', cmap='Blues', ax=ax)
```

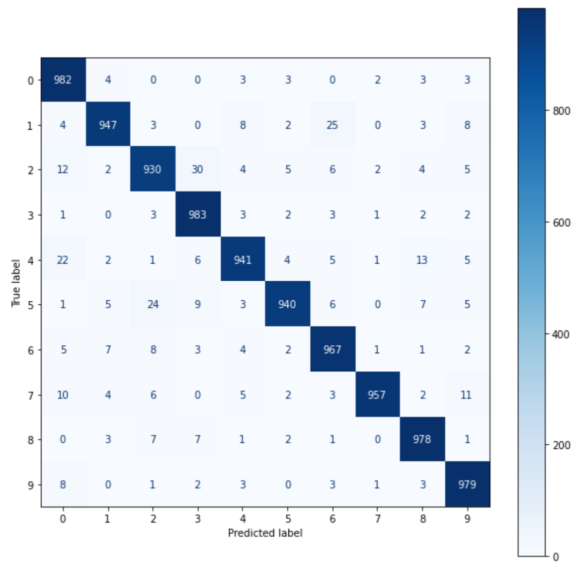
```
[ ] plot_confusion_matrix(labels, pred_labels)
```

The final result of the confusion matrix

<u>MLP model</u>



<u>CNN model</u>



What we can discover from above confusion matrix is that :
Most confused pair in MLP : Class 1 & Class 6
Most confused pair in LeNet : Class 2 & Class 3

5.3 Plot incorrect label visualisation

Coding Part:

- “torch.eq()”: Perform element-by-element comparison between two tensors, if the two elements in the same position are the same, return “True”; if they are different, return “False”.
- If “correct” is “False”, put it in “incorrect_example”

In [44]:
```
corrects = torch.eq(labels, pred_labels)
```

In [45]:
```
incorrect_examples = []

for image, label, prob, correct in zip(images, labels, probs, corrects):
    if not correct:
        incorrect_examples.append((image, label, prob))

incorrect_examples.sort(reverse=True,
                        key=lambda x: torch.max(x[2], dim=0).values)
```

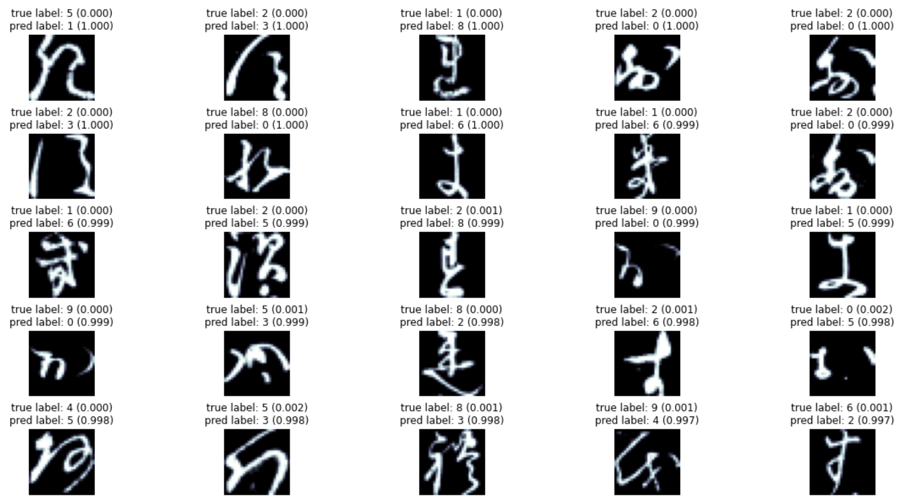
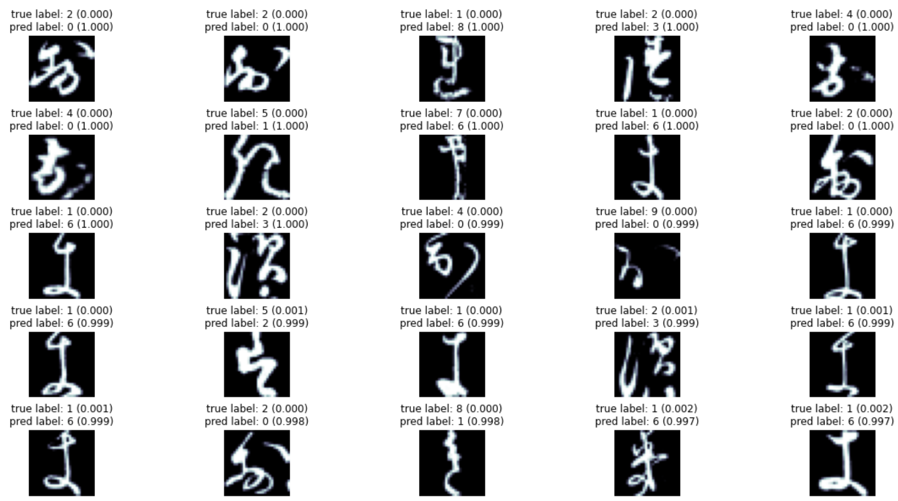
In [46]:
```
def plot_most_incorrect(incorrect, n_images):

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize=(20, 10))
    for i in range(rows*cols):
        ax = fig.add_subplot(rows, cols, i+1)
        image, true_label, probs = incorrect[i]
        true_prob = probs[true_label]
        incorrect_prob, incorrect_label = torch.max(probs, dim=0)
        ax.imshow(image.view(28, 28).cpu().numpy(), cmap='bone')
        ax.set_title(f'true label: {true_label} ({true_prob:.3f})\n'
                     f'pred label: {incorrect_label} ({incorrect_prob:.3f})')
        ax.axis('off')
    fig.subplots_adjust(hspace=0.5)
```

In [47]:
```
N_IMAGES = 25

plot_most_incorrect(incorrect_examples, N_IMAGES)
```

MLP model	
CNN model	

6) Conclusion

We learned MLP & LeNet models and successfully implemented deep learning with Pytorch on the Kuzushiji MNIST dataset. By comparing the implementation results of two models, we found that the accuracy of LeNet models is higher than that of MLP models (20 times epoch). Hence, we discovered that CNN models normally perform better on image processing compared to older networks.

7) Reference

http://naruhodo.weebly.com/blog/introduction-to-kuzushiji

https://github.com/rois-codh/kmnist

https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/

https://chih-sheng-huang821.medium.com/%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92-%E7%A5%9E%E7%B6%93%E7%B6%B2%E8%B7%AF-%E5%A4%9A%E5%B1%A4%E6%84%9F%E7%9F%A5%E6%A9%9F-multilayer-perceptron-mlp-%E5%90%AB%E8%A9%B3%E7%B4%B0%E6%8E%A8%E5%B0%8E-ee4f3d5d1b41

https://medium.com/ching-i/%E5%8D%B7%E7%A9%8D%E7%A5%9E%E7%B6%93%E7%B6%B2%E7%B5%A1-cnn-%E7%B6%93%E5%85%B8%E6%A8%A1%E5%9E%8B-lenet-alexnet-vgg-nin-with-pytorch-code-84462d6cf60c

https://zh.d2l.ai/chapter_convolutional-neural-networks/lenet.html

https://blog.csdn.net/weixin_43624538/article/details/88352409

https://paperswithcode.com/dataset/kuzushiji-mnist