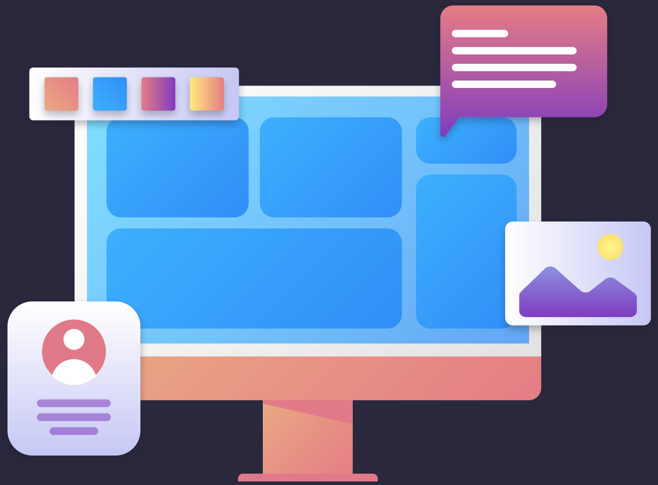
Image Classification using Pytorch

On **KUZUSHIJI MNIST** Dataset

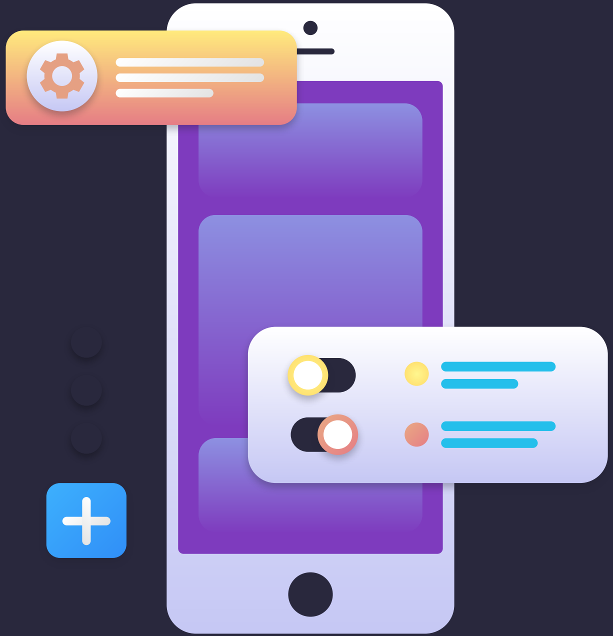


Group 6 : WONG KAI YUAN (DC026157) | TSE TSZ LEONG (DB927141) | CHAN WENG KIN (DB927473) |

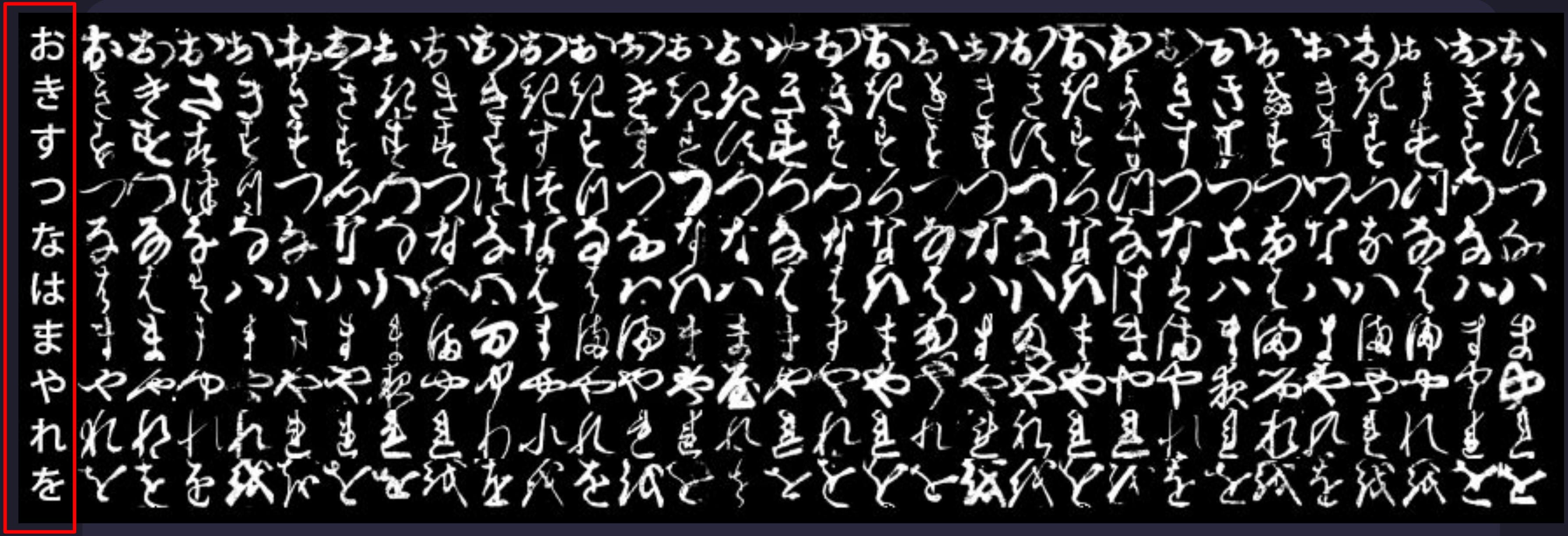
/TABLE OF CONTENTS

/01

/INTRODUCTION

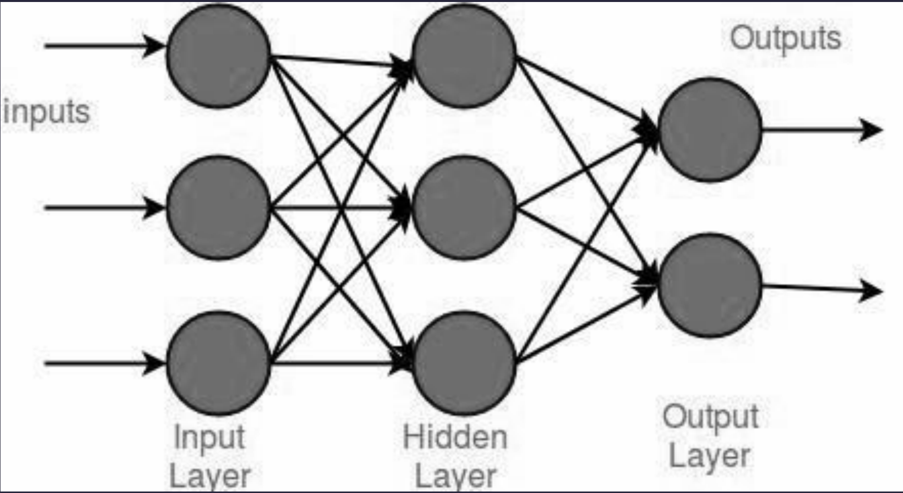


Datasets used : Kuzushiji-MNIST



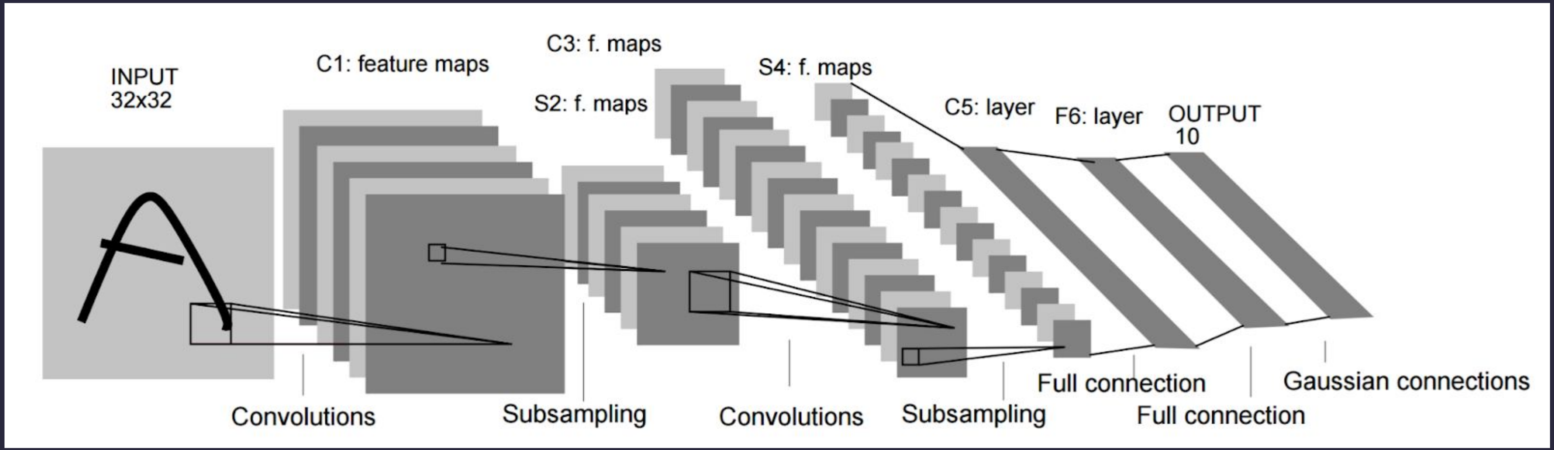
28x28 grayscale, 70,000 images, 10 classes of Hiragana(平仮名)

Model description - MLP & LeNet



Multilayer Perceptron (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).

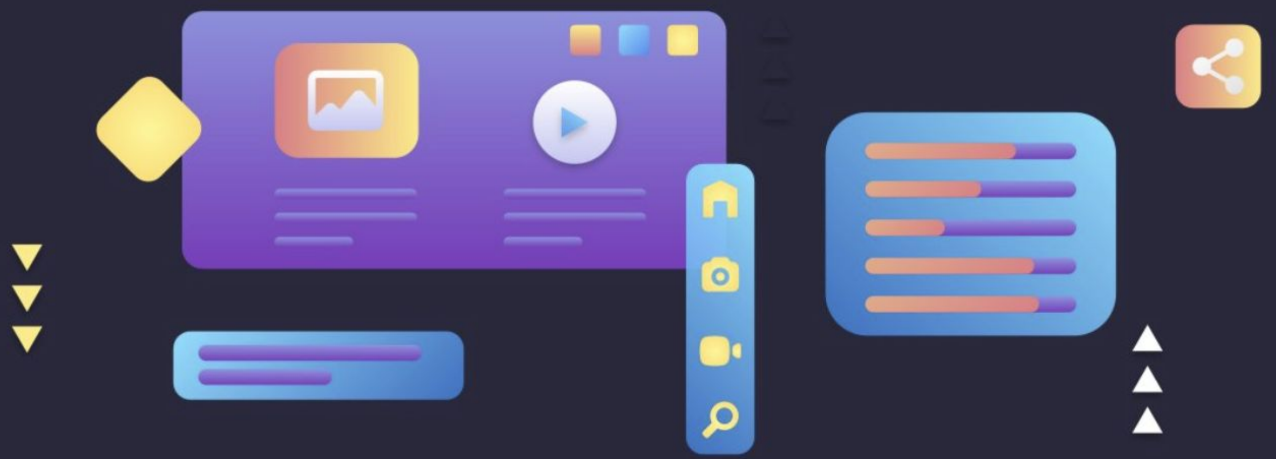
Model description - MLP & LeNet



LeNet is one of the most popular **convolutional neural network (CNN) architecture**. It consists of two convolutional layers, a pooling layer, a fully connected layer, and a final Gaussian connection layer to recognize handwritten digital images.

<GOAL!>

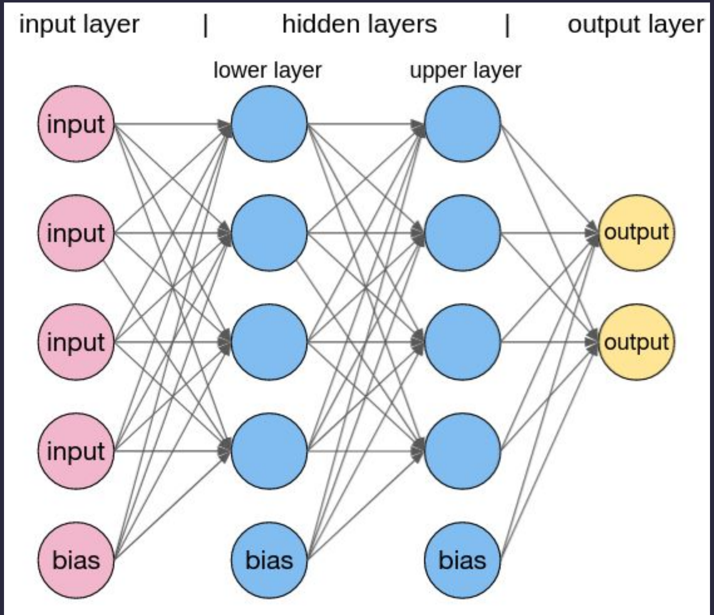
Building **MLP & LeNet** models to perform image classification on the **Kuzushiji MNIST dataset** using Pytorch & Torchvision.



/02 /IMPLEMENTATION

/For both implementation, we have similar steps :

1) DATA PROCESSING → 2) DEFINING THE MODEL → 3) TRAINING THE MODEL → 4) RESULT ON TESTING DATA



PPT will just show the brief idea, detailed codes & steps can refer to report

/Implementation On Multilayer Perceptron (MLP)

(1) DATA PROCESSING

1) DATA PROCESSING → 2) DEFINING THE MODEL → 3) TRAINING THE MODEL → 4) RESULT ON TESTING DATA

(1) DATA PROCESSING

- Import all needed modules we need
- Find Mean & Std for normalization
- Define transformation :

1) Random Rotation
2) Random Crop
3) ToTensor()
4) Normalize

- Load Training & Testing dataset with transformation

(1) DATA PROCESSING

```
import torch
import torch.nn as nn
import torch.nn.funct
import
import

import
import

from sk
from sk
from sk
from tc
import
import

import copy
import random
import time
```

```
[15] ROOT =

     train_
```

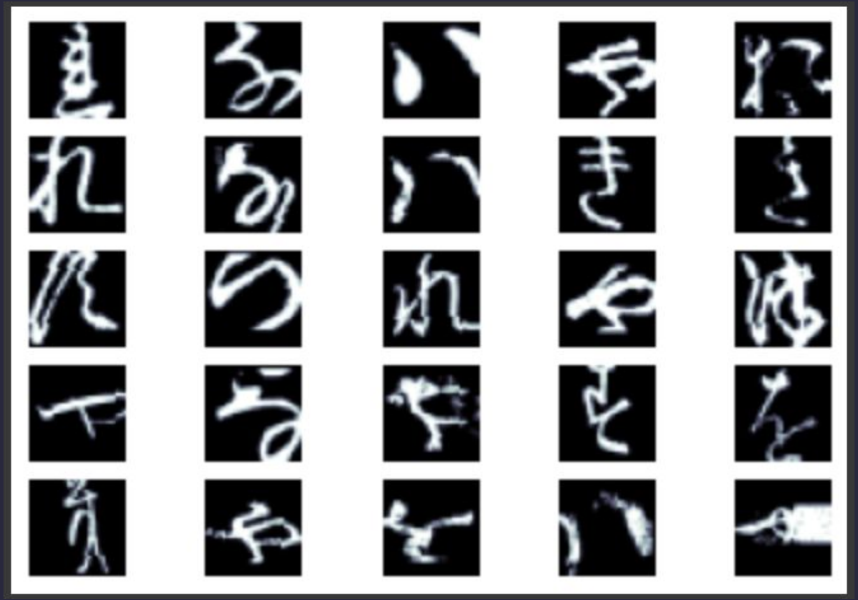
```
[16] mean =
     std =
```

```
[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)
```

(1) DATA PROCESSING

- Visualization of our training data



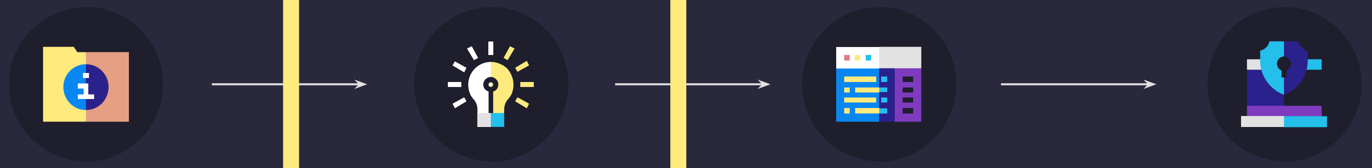
(1) DATA PROCESSING

- Create **validation** dataset from 10% of training data
- Define **DataLoader** as batch of 64 for each dataset
- The number of each 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
```

(2) DEFINING THE MODEL



1) DATA PROCESSING

2) DEFINING THE MODEL

3) TRAINING THE MODEL

4) RESULT ON TESTING DATA

(2) DEFINING THE MODEL

- Define **MLP** with **2 hidden layers** & non-linear function as such **ReLU**
- Create an instance of model and give **correct input & output**

```
INPUT_DIM = 28 * 28
OUTPUT_DIM = 10

model = MLP(INPUT_DIM, OUTPUT_DIM)
```

(2) DEFINING THE MODEL

- Our model (MLP)

Input : 28 * 28
Output : 10
Hidden Layer 1 : 250
Hidden Layer 2 : 100

```python
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) DEFINING THE MODEL

- Define Optimizer : Adam Optimizer with default parameters
- Define Criterion : CrossEntropyLoss
- Define Device : place on GPU if have one
- 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) TRAINING THE MODEL

1) DATA PROCESSING

2) DEFINING THE MODEL

3) TRAINING THE MODEL

4) RESULT ON TESTING DATA

(3) T

- **Defin**
  - put our
  - iterate o
  - place th
  - clear the
  - pass our
  - calculat
  - calculat
  - calculat
  - update t
  - update c
  -

```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)

- Def

The evaluati

- we pu
- we wr
- we do
- we do
- we do

```
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)
```

(3) TRAINING THE M

- Start Training !!

Total epoch : 20

- Each Epoch :

1) loss is decreasing
2) accuracy is increasing

```
100% 10/10 [05:29<00:00, 31.95s/it]
Epoch: 01 | Epoch Time: 0m 33s
        Train Loss: 0.549 | Train Acc: 82.63%
         Val. Loss: 0.237 |  Val. Acc: 92.74%
Epoch: 02 | Epoch Time: 0m 31s
        Train Loss: 0.294 | Train Acc: 90.79%
         Val. Loss: 0.174 |  Val. Acc: 94.65%
Epoch: 03 | Epoch Time: 0m 41s
        Train Loss: 0.240 | Train Acc: 92.47%
         Val. Loss: 0.179 |  Val. Acc: 94.18%
Epoch: 04 | Epoch Time: 0m 32s
        Train Loss: 0.211 | Train Acc: 93.35%
         Val. Loss: 0.146 |  Val. Acc: 95.76%
Epoch: 05 | Epoch Time: 0m 32s
        Train Loss: 0.192 | Train Acc: 93.93%
         Val. Loss: 0.138 |  Val. Acc: 95.76%
Epoch: 06 | Epoch Time: 0m 31s
        Train Loss: 0.174 | Train Acc: 94.56%
         Val. Loss: 0.137 |  Val. Acc: 95.94%
Epoch: 07 | Epoch Time: 0m 32s
        Train Loss: 0.166 | Train Acc: 94.82%
         Val. Loss: 0.120 |  Val. Acc: 96.17%
Epoch: 08 | Epoch Time: 0m 32s
        Train Loss: 0.156 | Train Acc: 95.02%
         Val. Loss: 0.124 |  Val. Acc: 96.20%
Epoch: 09 | Epoch Time: 0m 31s
        Train Loss: 0.147 | Train Acc: 95.41%
         Val. Loss: 0.118 |  Val. Acc: 96.28%
Epoch: 10 | Epoch Time: 0m 31s
        Train Loss: 0.140 | Train Acc: 95.59%
         Val. Loss: 0.118 |  Val. Acc: 96.55%
```

(4) RESULT ON TESTING DATA

1) DATA PROCESSING → 2) DEFINING THE MODEL → 3) TRAINING THE MODEL → 4) RESULT ON TESTING DATA

(4) RESULT ON TESTING DATA

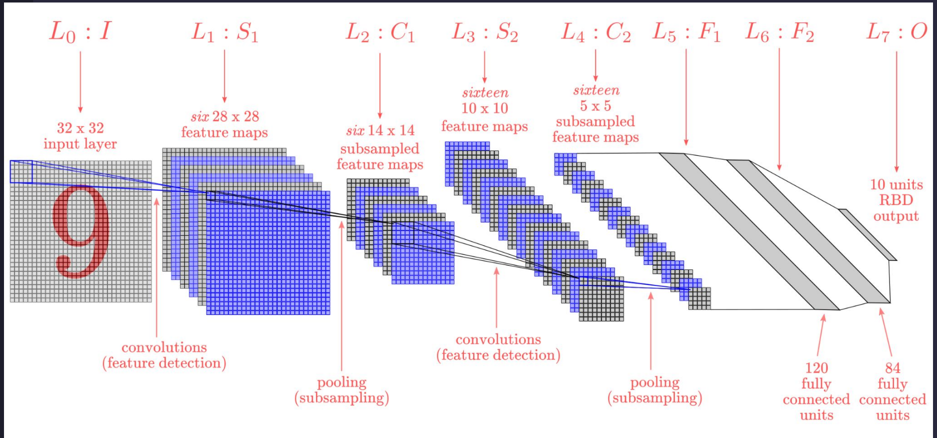
1) Load Model with best parameters on testing data

```
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%
```

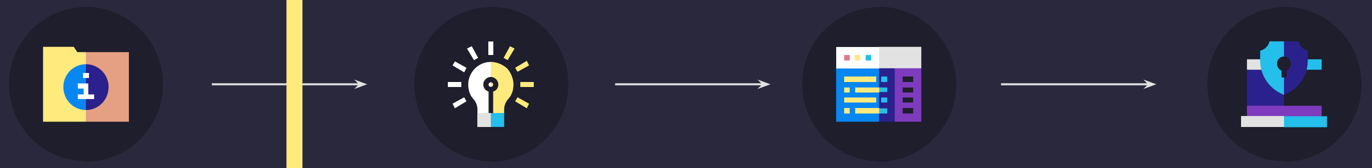
Test Accuracy : 92.24% !!



PPT will just show the brief idea, detailed codes & steps can refer to report

/Implementation On LeNet (CNN)

(1) DATA PROCESSING



1) DATA PROCESSING

2) DEFINING THE MODEL

3) TRAINING THE MODEL

4) RESULT ON TESTING DATA

(1) DATA PROCESSING

- Import all needed **modules** we need
- Find **Mean & Std** for normalization
- Define **transformation** :

1) Random Rotation
2) Random Crop
3) ToTensor()
4) Normalize

- **Load** Training & Testing dataset with transformation

(1) DATA PROCESSING

```
import torch
import torch.r
import torch.r
import torch.c
import torch.u

import torchvi
import torchvi

from sklearn
from sklearn
from sklearn.r
from sklearn.r
from tqdm.note
import matplot
import numpy

import copy
import random
import time
```

```
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])
                                       ])
```

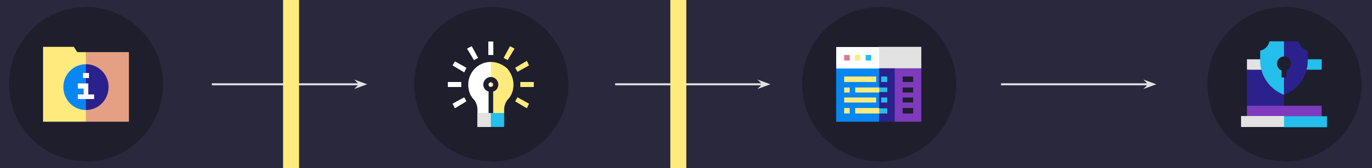
```
train_data = datasets.KMNIST(root=ROOT,
                           train=True,
                           download=True,
                           transform=train_transforms)

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

(1) DATA PROCESSING

- Create **validation** dataset from 10% of training data
- Define **DataLoader** as batch of 64 for each dataset
- The number of each datasets are below :

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

(2) DEFINING THE MODEL



1) DATA PROCESSING

2) DEFINING THE MODEL

3) TRAINING THE MODEL

4) RESULT ON TESTING DATA

(2) DEFINING THE MODEL

- Define a standard linear layer, which includes a **convolutional layer** and a **pooling layer.**
- Create an instance of model and give **correct output**

```
OUTPUT_DIM = 10

model = LeNet(OUTPUT_DIM)
```

(2) DEFINING THE MODEL

```
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)
```

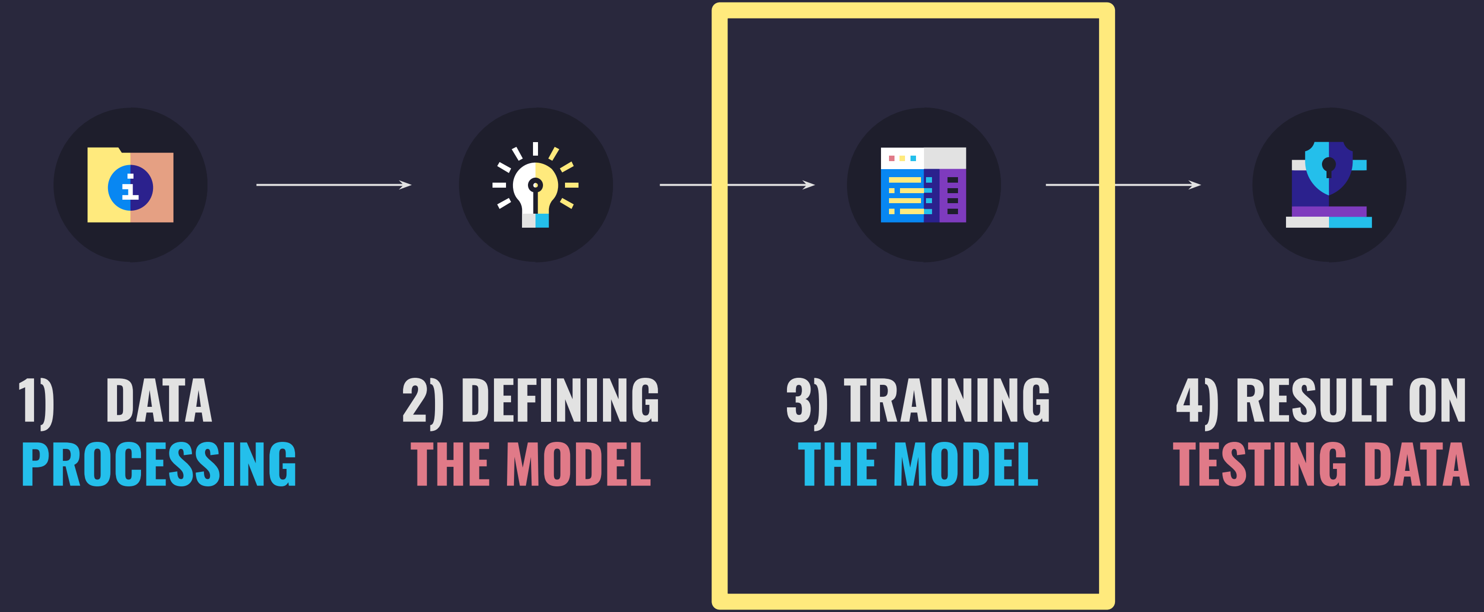
```
# x = [batch size, 16, 8, 8]

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

# x = [batch size, 16, 4, 4]

x = F.relu(x)

x = x.view(x.shape[0], -1)

# x = [batch size, 16*4*4 = 256]

h = x

x = self.fc_1(x)
```

(2) DEFINING THE MODEL

- Define Optimizer : Adam Optimizer with default parameters
- Define Criterion : CrossEntropyLoss
- Define Device : place on GPU if have one
- 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) TRAINING THE MODEL



(3) TRAININ

- **Define Training**
  - put our model into train
  - iterate over our dataload
  - place the batch on to our
  - clear the gradients calcu
  - pass our batch of images
  - calculate the loss betwee
  - calculate the accuracy b
  - calculate the gradients o
  - update the parameters b
  - 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) TRAINING THE MODEL

- Define Evaluatin

The evaluation loop is similar to t

- we put our model into eva
- we wrap the iterations ins
- we do not zero gradients a
- we do not calculate gradie
- we do not take an optimiz

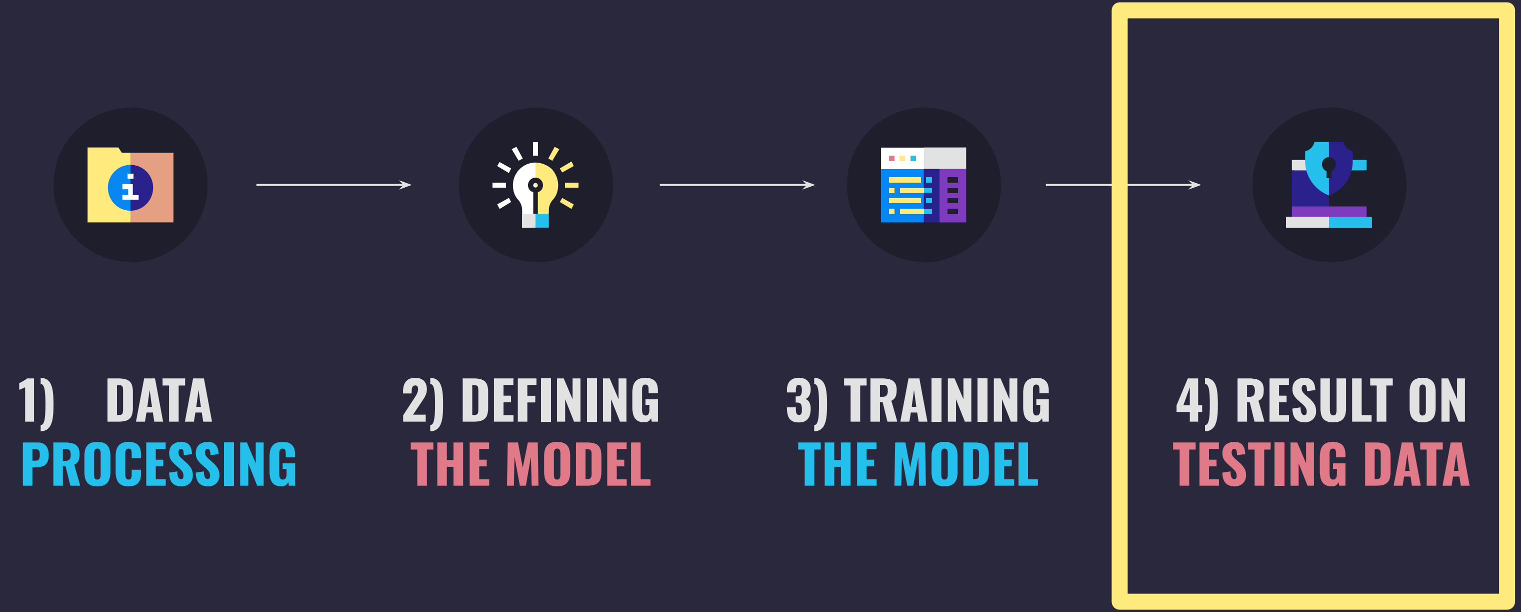
```
def evaluate(model, iterator, criterion, device):

    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)
```

```
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%
```

HE MODEL

```
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) RESULT ON TESTING DATA



(4) RESULT ON TESTING DATA

1) Load Model with best parameters on testing data

```
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%
```

Test Accuracy : 95.46% !!

/03

/COMPARISON

- Test error & accuracy
- Confusion Matrix
- Others

(1) Confusion Matrix

```python
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
```

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

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

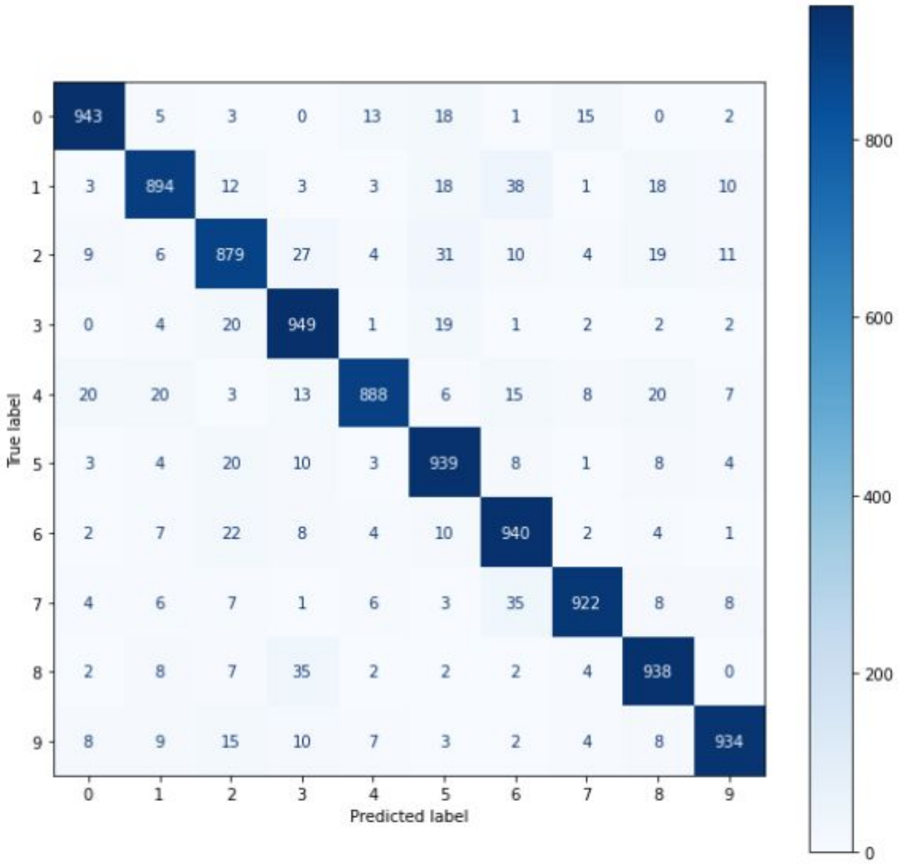
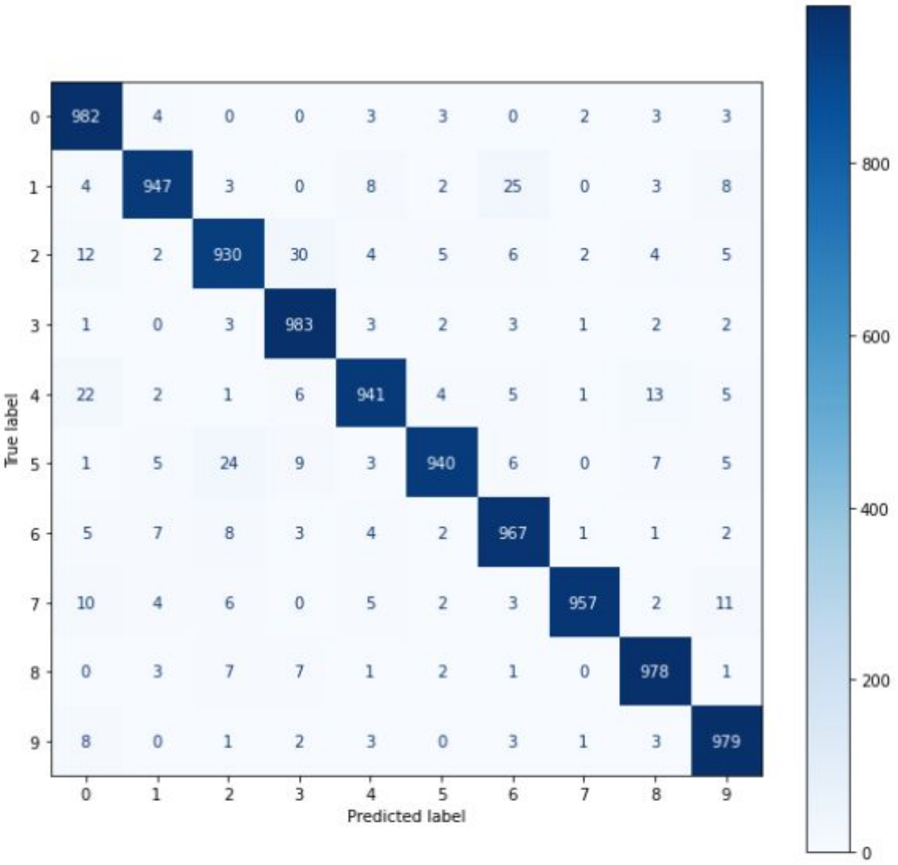
“model.eval”: change to evaluate model ()

"with torch.no_grad": The requirements_grad of the calculated tensor is automatically set to False.

“torch.cat”: Splicing multiple tensors

```python
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)
```

```python
plot_confusion_matrix(labels, pred_labels)
```

MLP model	CNN model
	

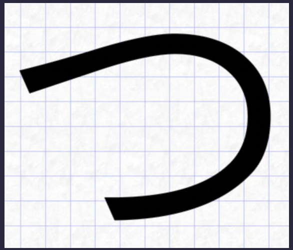
Most Confused Pair

/MLP Model

Class 1 & Class 6

/LeNet Model

Class 2 & Class 3



Plot incorrect label visualisation

"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".

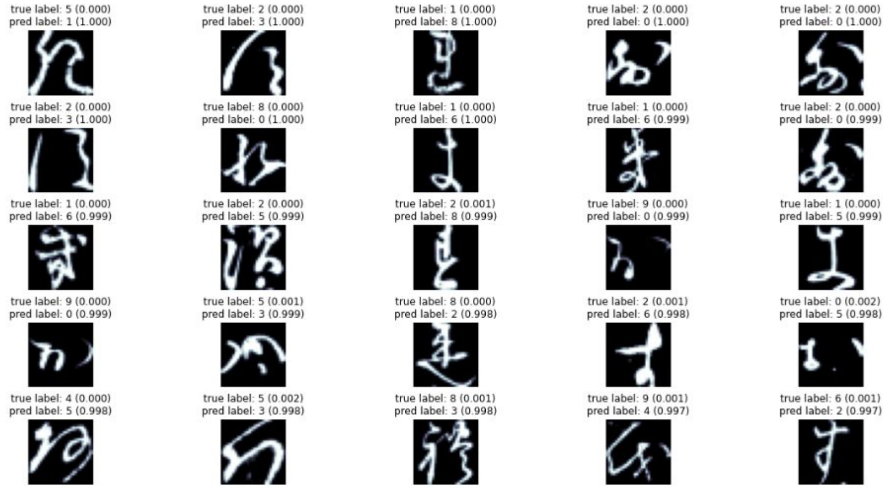
```
corrects = torch.eq(labels, pred_labels)
```

```
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)
```

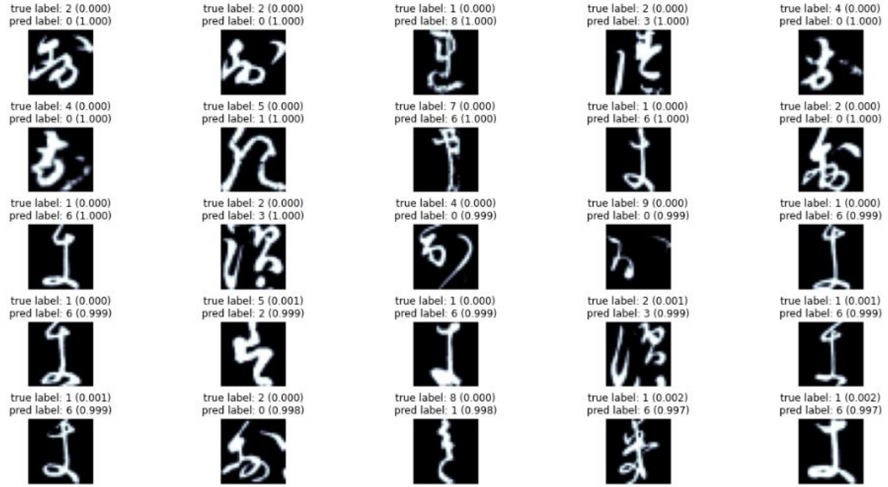
```
def plot_most_incorrect(incorrect, n_images):

    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)
```

```
N_IMAGES = 25

plot_most_incorrect(incorrect_examples, N_IMAGES)
```

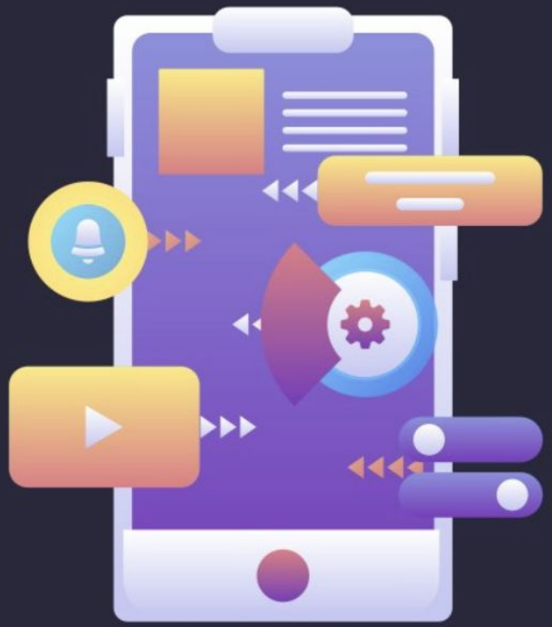
MLP model



CNN model



/04 /CONCLUSION



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).

- Discovered that CNN models normally perform better on image processing compared to older networks.

/THANKS!

/DO YOU HAVE ANY QUESTIONS?

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

> Please keep this slide for attribution