

CS 5600/6600: F24: Intelligent Systems

Coding Lab for Assignment 4

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 21, 2024

Learning Objectives

1. Building, Training, Validating Nets with PyTorch
2. Synthetic Data Generation

Introduction

This is a coding lab for CS5600/6600: F24: Assignment 4. The programming component of this assignment (Problems 2) will give you an opportunity to build, train, validate, and test multi-layer ANNs (aka multi-layer perceptrons or MLPs) with PyTorch on synthetic data. In Problem 3, we will experiment with LeNet, a famous convolutional network, that we will build later in this lab with PyTorch. We will train and test it on an image dataset of handwritten Hiragana characters, also known as the Kuzushiji-MNIST (KMNIST). I think this dataset, while still simple, is a bit more interesting and challenging than the standard MNIST. My objective is simple – to make sure that you have a working PyTorch version on your computer and know how to construct ANNs and CNNs with it for the upcoming Project 1.

My second objective is to give you some exposure to generating and processing synthetic data. Working with synthetic data is invaluable if you want to test your ideas on datasets whose properties you can control through various statistical distributions and to compare different models on them.

This lab will require some installations. Don't be shy to share your installation experiences, recommendations, shortcuts with each other. Help each other. This class is not a rat race. Let's dive in.

Installing PyTorch

PyTorch is an open source deep learning (DL) library. It allows you to define ANNs and ConvNets (and other machine learning tools) and train them. PyTorch is one of the two state of the art DL libraries, the other being Keras/TensorFlow. I won't get into the debate about which library is better. In my opinion, PyTorch and TensorFlow APIs implement similar features in different ways. If you are interested in data-driven AI, you should know the basics of both. So, the fundamentals you'll learn with one will transfer into the other if you have to switch in the future.

I installed torch and torchvision with `pip install` with the following command for Python 3.10.12 on Ubuntu 22.04 LTS.

```
pip install torch torchvision
```

To check the correctness of the installation, we can import torch and check its version like so.

```
~/teaching/AI/F24/hw/hw04$ python
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
>>> import torch
>>> torch.__version__
'2.4.1+cu121'
>>>
```

We should also install `scikit-learn`. Here's the pip command I used.

```
pip install -U scikit-learn
```

PyTorch represents data as multi-dimensional, numpy-like arrays called tensors. Tensors store inputs and outputs of network layers. A scalar like 1 is a tensor of the 0D dimensionality. a vector, e.g., [1, 2], is a 1D tensor; a matrix, e.g. [[1, 2], [3, 4]], is a 2D tensor, and so on. Below are 2 examples in numpy and PyTorch side by side. They are very similar, except for float precision.

```
>>> import numpy as np
>>> np.array([[0.0, 1.3], [2.8, 3.3], [4.1, 5.2], [6.9, 7.0]])
array([[0. , 1.3],
       [2.8, 3.3],
       [4.1, 5.2],
       [6.9, 7. ]])
>>> import torch
>>> torch.tensor([[0.0, 1.3], [2.8, 3.3], [4.1, 5.2], [6.9, 7.0]])
tensor([[0.0000, 1.3000],
        [2.8000, 3.3000],
        [4.1000, 5.2000],
        [6.9000, 7.0000]])
```

ANN Construction

Open up `mlp.py` file in the zip. I called it `mlp.py`, because in PyTorch, ANNs are frequently referred to as multi-layer perceptrons (MLPs). The file starts with the following imports and function.

```
from collections import OrderedDict
import torch.nn as nn

def build_4_8_3_mlp_model(inFeatures=4,
                           hiddenDim1=8,
                           hiddenActFun1=nn.ReLU(),
                           nbClasses=3):
    mlp_model = nn.Sequential(OrderedDict([
        ("hidden_layer_1", nn.Linear(inFeatures, hiddenDim1)),
        ("activation_1",   hiddenActFun1),
        ("output_layer",   nn.Linear(hiddenDim1, nbClasses))
    ]))
    return mlp_model
```

`OrderedDict` is a dictionary that we'll use to provide human-readable names to each component of the network. It's up to us what those names are. I use self-evident names like `hidden_layer_1`, `activation_1`, and `output_layer`. The package `torch.nn` contains the PyTorch NN implementations.

The function constructs a 4x8x3 ANN. To define it, we specify the number of input features (i.e., the input layer of 4 neurons), the number of neurons in the hidden layer (8 neurons), and the number of neurons in the output layer. You may recall, in passing, that we did the same things, when we were building our own ANNs in HW02. The number of input neurons is defined with the keyword `inFeatures`, the number of hidden neurons is defined with the keyword `hiddenDim1`, and the number of output neurons is defined with the keyword `nbClasses`.

The activation function for the hidden layer neurons is defined with the keyword `hiddenActFun1` and is set to the PyTorch implementation of ReLU. The constructor `nn.Linear()` defines the fully connected 4x8 weight tensor (i.e., a 2D weight matrix). In other words, this construction

```
nn.Linear(inFeatures, hiddenDim1))
```

ensures that the net feeds the input neuron values to the hidden layer neurons via the synaptic weights from the input layer to the hidden layer. This weight tensor is stored in the dictionary under the string key "hidden_layer_1", i.e., the dictionary contains the following key-value pair:

```
("hidden_layer_1", nn.Linear(inFeatures, hiddenDim1)).
```

After the input neurons feedforward their values through the first weight tensor to the hidden neurons, they will be activated with the ReLU function, which is ensured with the second key-value pair.

```
("activation_1", hiddenActFun1)
```

Since we are building a 4x8x3 net, in order to feedforward the activation layer to the output layer, we need a 8x3 weight tensor, which we construct as follows.

```
nn.Linear(hiddenDim1, nbClasses)
```

and then place in the dictionary under the key "output_layer", i.e., as the third key-value pair

```
("output_layer", nn.Linear(hiddenDim1, nbClasses)).
```

The ANN is finally glued together by constructing a `nn.Sequential` object with the `OrderedDict` object we just constructed. The only thing we need to remember throughout this construction is that the output dimensions of the previous layer must match the input dimensions of the next layer. Otherwise, the matrix algebra won't go through. If we make a mistake, there'll be a run-time error.

The `mlp.py` file also furnishes an example of how to define 4x8x8x3 ANN with 2 hidden layers. Here it is.

```
def build_4_8_8_3_mlp_model(inFeatures=4,
                             hiddenDim1=8,
                             hiddenDim2=8,
                             hiddenActFun1=nn.ReLU(),
                             hiddenActFun2=nn.Sigmoid(),
                             nbClasses=3):
    mlp_model = nn.Sequential(OrderedDict([
        ("hidden_layer_1", nn.Linear(inFeatures, hiddenDim1)),
        ("activation_1", hiddenActFun1),
        ("hidden_layer_2", nn.Linear(hiddenDim1, hiddenDim2)),
        ("activation_2", hiddenActFun2),
        ("output_layer", nn.Linear(hiddenDim2, nbClasses))
    ]))
    return mlp_model
```

Take a look at it and make sure you understand how we construct 4x8, 8x8, and 8x3 tensor matrices sequentially. This function shows you how to connect two hidden layers to each other. Once you know how to do this on 2 layers, you can generalize this construction chain to 3 or more. Someone asked me in class if it is possible to define different activation functions for different layers. As this function shows, the answer is yes. The first hidden layer is activated with ReLU; the second one – with Sigmoid. You can read about other PyTorch activation functions, e.g., `nn.Hardtanh`, [here](#).

Training ANNs

Let's open up `train_eval_mlp.py`. This file starts with the following imports.

```
import mlp
from torch.optim import SGD
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs
import torch.nn as nn
import torch
```

The first import, `import mlp`, imports our ANN constructors defined above. SGD is the stochastic gradient descent we discussed a few lectures ago. The `make_blobs` is a very useful function that we can use to generate synthetic data. More on it below. The `train_test_split` is a function that allows us to randomly split datasets into the training and testing subsets.

Next we define a generator that will give us batches during training and testing.

```
def gen_next_batch(inputs, targets, batchSize):
    for i in range(0, inputs.shape[0], batchSize):
        yield (inputs[i:i + batchSize], targets[i:i + batchSize])
```

Then we need to specify the batch size, the number of training epochs, and the learning rate. Feel free to experiment with these hyperparameters. The `DEVICE` line determines if we have a GPU or not. If we do, it'll speed things up. If we don't, everything will still work, but run more slowly. I completed this lab without a GPU. I use the prefix "`jDBGj`:" to print debugging messages.

```
BATCH_SIZE = 64
EPOCHS = 10
LR = 1e-2
# determine the device we will be using for training
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print("<DBG>: training using {}".format(DEVICE))
```

Let's generate some synthetic data with `make_blobs`. You can read more on this great tool [here](#).

```
(X, y) = make_blobs(n_samples=1000, n_features=4, centers=3,
                    cluster_std=2.5, random_state=13)
```

X gives a numpy array of samples, and y – the corresponding values. The print statements run on the first 5 elements of X and y will print something like this.

```
>>> print('X = {}'.format(X[:5]))
X = [[ 3.52062157  6.48836026 -6.45394141 -2.12471496]
      [ 7.39719107  6.02489915  3.08007225  6.49844623]
      [ 4.17054167 -5.85293888  8.39114771  8.68368999]
      [13.51340502 -0.61491582  4.88949518  0.16335579]
      [ 5.29600164 -7.94670734  8.12522448  8.75328625]]
>>> print('y = {}'.format(y[:5]))
y = [2 1 0 1 0]
```

In other words, we've just generated 1,000 samples, each of which consists of 4 features (remember we'll be running nets with 4 input neurons on this dataset), and is classified as one of the three classes labeled as 0, 1, 2 (again, remember that our nets have 3 output neurons).

The standard deviation in the feature values in each of the three classes (aka blobs or clusters), is 2.5. We can change this as we see fit. We should also note that the function `make_blobs` generates Gaussian blobs of clustered data points in 4D. We can also generate 2D, 3D, etc. data with `make_blobs` and use different probability distributions, which is really helpful when we need to train different learners but don't have enough real data.

We then split our dataset randomly with 80% for training and 20% for testing (i.e., we do an 80/20 train/test split) and convert the training and testing arrays into floats like so.

```
(trainX, testX, trainY, testY) = train_test_split(X, y,
                                                    test_size=0.20,
                                                    random_state=13)

trainX = torch.from_numpy(trainX).float()
testX  = torch.from_numpy(testX).float()
trainY = torch.from_numpy(trainY).float()
testY  = torch.from_numpy(testY).float()
```

Here's what the individual elements of trainX and testX look like.

```
>>> trainX[:10]
tensor([[ 8.5425,  0.1411,  5.0938, 11.3992],
        [ 2.8952, -3.3455,  6.0776, 11.4984],
        [ 6.1584, -3.5732,  1.6438,  6.6643],
        [ 2.2238,  7.6743, -10.9174, -6.7050],
        [ 3.5202, -7.5891,  8.8931,  4.4953],
        [ 2.0726,  7.1669, -11.7967, -0.6002],
        [10.0953, -0.5128,  3.8685,  5.1703],
        [ 0.0369, -6.2570,  8.2689,  6.9484],
        [ 5.2960, -7.9467,  8.1252,  8.7533],
        [10.3928, -5.0403,  4.9330,  4.3938]])
>>> testX[:10]
tensor([[10.3986, -4.1033,  8.2987, 10.7670],
        [ 5.9477,  9.0744, -7.8463, -5.1389],
        [10.5173, -2.1252,  4.0314,  8.1264],
        [ 4.5929, -6.5750,  9.8613, 14.3530],
        [10.7349,  0.7290,  0.3921,  6.2299],
        [ 5.3782, -5.6553, -2.2619,  8.4273],
        [10.4786, -0.2532, -5.7827,  4.8975],
        [ 6.3362, -4.9072,  6.1063,  6.2229],
        [-1.3881, -1.1420, -2.3416, -6.4439],
        [ 8.6669, -6.5355,  4.6408,  8.1345]])
>>> trainY[:10]
tensor([1., 0., 1., 2., 0., 2., 1., 0., 0., 1.])
>>> testY[:10]
tensor([0., 2., 1., 0., 1., 1., 1., 0., 2., 1.])
```

We now initialize our ANNs (or MLPs) as follows.

```
mlp_4_8_3 = mlp.build_4_8_3_mlp_model().to(DEVICE)
mlp_4_8_8_3 = mlp.build_4_8_8_3_mlp_model().to(DEVICE)
```

We need to specify the backpropagation optimizer as the stochastic gradient descent (SGD) and our loss (i.e., cost) function as cross entropy. A good summary of PyTorch loss functions is [here](#).

```
# initialize optimizer and loss function
opt = SGD(ANN.parameters(), lr=LR)
lossFunc = nn.CrossEntropyLoss()
```

We're ready to train. Here's a standard PyTorch training loop. We go through each epoch and update the training statistics. This for-loop in the file has more comments.

```
print('----- TRAINING ANN -----')
for epoch in range(0, EPOCHS):
    print("<DBG>: epoch: {}".format(epoch + 1))
    trainLoss, trainAcc, num_samples = 0, 0, 0
    mlp_4_8_3.train()
    for (batchX, batchY) in gen_next_batch(trainX, trainY, BATCH_SIZE):
        (batchX, batchY) = (batchX.to(DEVICE), batchY.to(DEVICE))
        predictions = mlp_4_8_3(batchX)
        loss = lossFunc(predictions, batchY.long())
        opt.zero_grad()
        loss.backward()
        opt.step()
        trainLoss += loss.item() * batchY.size(0)
        trainAcc += (predictions.max(1)[1] == batchY).sum().item()
        num_samples += batchY.size(0)
```

```

trainTemplate = "epoch: {} train loss: {:.3f} train accuracy: {:.3f}"
print(trainTemplate.format(epoch + 1, (trainLoss / num_samples),
                           (trainAcc / num_samples)))

```

One last thing I'd like to comment on in the above for-loop is the following three steps:

1. `opt.zero_grad();`
2. `loss.backward();`
3. `opt.step().`

In PyTorch, these steps must be performed in this exact order. Always! The call `opt.zero_grad()` zeros the gradients accumulated from the model's previous batch training step. The call `loss.backward()` does backpropagation. The call `opt.step()` updates the weights in the trained NN based on the errors computed with the backpropagation. The most common mistake in PyTorch (been there, done that!) is not to zero the gradient. If we don't do that, we accumulate gradients across multiple batches and over multiple epochs, which will mess up the backpropagation and lead to bad weights.

Here's my output for training this ANN for 10 epochs. The ideal loss is, of course, 0 and the ideal accuracy is 1.0. But if that happens during training, your NN is most likely overfit.

```

~/teaching/AI/F24/hw/hw04$ python train_eval_mlp.py
----- TRAINING ANN -----
<DBG>: epoch: 1...
epoch: 1 train loss: 1.356 train accuracy: 0.412
<DBG>: epoch: 2...
epoch: 2 train loss: 0.825 train accuracy: 0.519
<DBG>: epoch: 3...
epoch: 3 train loss: 0.623 train accuracy: 0.666
<DBG>: epoch: 4...
epoch: 4 train loss: 0.513 train accuracy: 0.774
<DBG>: epoch: 5...
epoch: 5 train loss: 0.443 train accuracy: 0.820
<DBG>: epoch: 6...
epoch: 6 train loss: 0.392 train accuracy: 0.856
<DBG>: epoch: 7...
epoch: 7 train loss: 0.353 train accuracy: 0.879
<DBG>: epoch: 8...
epoch: 8 train loss: 0.321 train accuracy: 0.895
<DBG>: epoch: 9...
epoch: 9 train loss: 0.295 train accuracy: 0.899
<DBG>: epoch: 10...
epoch: 10 train loss: 0.273 train accuracy: 0.909

```

Testing ANNs

Now we can test the trained ANN. Below is a standard PyTorch evaluation loop. I put it in the same file (`train_eval_mlp.py`) for brevity, but it's also frequently placed in a separate script.

```

testLoss = 0
testAcc = 0
samples = 0
mlp_4_8_8_3.eval()
print('----- TESTING ANN -----')
with torch.no_grad():
    for epoch in range(0, EPOCHS):
        # loop over the current batch of test data
        for (batchX, batchY) in gen_next_batch(testX, testY, BATCH_SIZE):
            # flash the data to the current device
            (batchX, batchY) = (batchX.to(DEVICE), batchY.to(DEVICE))

```

```

# run data through our model and calculate loss
# predictions = mlp_4_8_3(batchX)
predictions = mlp_4_8_3(batchX)
loss = lossFunc(predictions, batchY.long())

# update test loss, accuracy, and the number of
# samples visited
testLoss += loss.item() * batchY.size(0)
testAcc += (predictions.max(1)[1] == batchY).sum().item()
samples += batchY.size(0)

# display model progress on the current test batch
testTemplate = "epoch: {} test loss: {:.3f} test accuracy: {:.3f}"
print(testTemplate.format(epoch + 1, (testLoss / samples),
    (testAcc / samples)))
print("")

```

In the above code segment, we initialize the statistics variables that we'll use to evaluate the NN's performance with, place our tested model in the the evaluation mode like so

```
mlp_4_8_8_3.eval()
```

and create a `no_grad` context, because there's no need to compute gradients in the absence of backprop during testing. For each batch, we make predictions and update the statistics variables, and display the model's progress. Finally, we should see something like this. Your stats will be different, of course, due to random weight initialization.

```
epoch: 10 test loss: 0.073 test accuracy: 0.900
```

In the ideal case, the test loss is 0 and the test accuracy is 1.0. But, with a test loss of 0.07 and a test accuracy of 0.9, we've done reasonably well. If you're new to PyTorch, let me congratulate you on training your first NN with PyTorch!

Let's go deeper and try training and testing the 4x8x8x3 net on the same randomly generated dataset. All we need to do is make a few changes in `train_eval_mlp.py` like so.

Change 1: comment out the 4x8x3 SGD optimizer and define the 4x8x8x3 SGD optimizer.

```
# opt = SGD(mlp_4_8_3.parameters(), lr=LR)
opt = SGD(mlp_4_8_8_3.parameters(), lr=LR)
```

Change 2: inside the training loop, specify that we will train the 4x8x8x3 model.

```
mlp_4_8_8_3.train()
```

Change 3: right before the test loop, set the 4x8x8x3 model into the evaluation mode.

```
mlp_4_8_8_3.eval()
```

Change 4: inside the test loop, make sure that the 4x8x8x3 model is called on the the current batch.

```
predictions = mlp_4_8_8_3(batchX)
```

We're ready to run train and test. Here' my output.

```

~python train_eval_mlp.py
<DBG>: training using cuda...
<DBG>: preparing data...
----- TRAINING ANN -----

```

```

<DBG>: epoch: 1...
epoch: 1 train loss: 1.277 train accuracy: 0.321
<DBG>: epoch: 2...
epoch: 2 train loss: 1.222 train accuracy: 0.321
<DBG>: epoch: 3...
epoch: 3 train loss: 1.179 train accuracy: 0.321
<DBG>: epoch: 4...
epoch: 4 train loss: 1.146 train accuracy: 0.321
<DBG>: epoch: 5...
epoch: 5 train loss: 1.117 train accuracy: 0.320
<DBG>: epoch: 6...
epoch: 6 train loss: 1.092 train accuracy: 0.316
<DBG>: epoch: 7...
epoch: 7 train loss: 1.069 train accuracy: 0.320
<DBG>: epoch: 8...
epoch: 8 train loss: 1.047 train accuracy: 0.422
<DBG>: epoch: 9...
epoch: 9 train loss: 1.026 train accuracy: 0.584
<DBG>: epoch: 10...
epoch: 10 train loss: 1.005 train accuracy: 0.714
----- TESTING ANN -----
epoch: 10 test loss: 0.997 test accuracy: 0.760

```

My deeper ANN did worse than my shallow ANN. So, staying shallow paid off this time.

Training and Testing a CNN

For this part of the lab, let's use an image dataset of handwritten Hiragana characters, also known as the Kuzushiji-MNIST (KMNIST) dataset. This dataset has 10 classes corresponding to 10 Hiragana characters. We will train a well-known CNN model to classify them. Another reason I chose KMNIST is that it is built into PyTorch. But, the steps we'll take to build this CNN can generalize to any image dataset so long as each image in it has a correct label, i.e., the dataset is curated.

Let's install a few more libraries. If we have them, pip will let us know what's installed.

```

pip install opencv-contrib-python
pip install matplotlib
pip install imutils

```

We'll work with the following Python scripts in the zip: `lenet.py`, `train_lenet.py`, and `predict_lenet.py`. The zip directory structure includes two subdirectories – `data` and `output`. The directory `data` contains KMNIST. The directory `output` is where we'll persist the trained models and accuracy plots (PNG images).

The file `lenet.py` is a PyTorch implementation of LeNet, a legendary CNN originally developed for optical character recognition by Dr. Yann LeCun (hence, the name LeNet, by the way) in the mid 1990's. Figure 1 shows the LeNet architecture. You can read more about this architecture and its history on the [LeNet Wiki](#) page. Today LeNet would probably be considered by many researchers and practitioners as a shallower CNN.

The file `train_lenet.py` uses PyTorch to train LeNet on KMNIST and persist the trained model in the directory `output`. The file `predict_lenet.py` loads and runs persisted LeNet models.

You can read comments in `lenet.py` to understand how it is implemented. In implementing LeNet, we subclass `Module`, which is a common object-oriented technique that allows us to reuse variables, implement custom methods, etc. In the `__init__`, `numChannels` is 1 for grayscale images and 2 for RGB and the `classes` is the number of class labels in the dataset, e.g., 10.

Take a few minutes to read the comments in `train_lenet.py` to familiarize yourself with the structure of the code. The main training loop kicks in after all the initializations are done. The training is followed with an evaluation loop that should already be familiar to you from the previous lab activity. When the training and evaluation are done, we generate a plot and save it in the `output` directory as a PNG image.

Almost there! Let's run `train_net.py`. The abbreviation `.pth` is used to denote that a file is a persisted

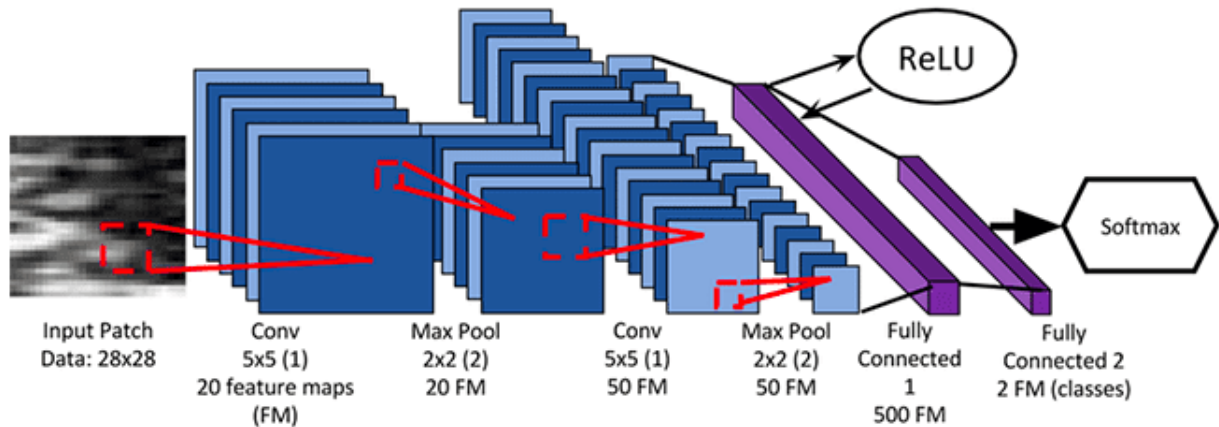


Figure 1: LeNet Architecture.

PyTorch model (it's really a zip under the hood). Here's my run in Python 3.10.12 on the Ubuntu 22.04 desktop in my office, which took me about 1 minute.

```
~/hw04$ python train_lenet.py --model output/kmnist_lenet.pth --plot output/kmnist_plot.png
<DBG> loading KMNIST train and test datasets...
<DBG> generating the train/validation split...
<DBG> initializing LeNet...
<DBG> training LeNet...
<DBG> EPOCH: 1/10
Train loss: 0.3627, Train accuracy: 0.8877
Val loss: 0.1445, Val accuracy: 0.9567
<DBG> EPOCH: 2/10
Train loss: 0.1056, Train accuracy: 0.9677
Val loss: 0.0974, Val accuracy: 0.9708
<DBG> EPOCH: 3/10
Train loss: 0.0603, Train accuracy: 0.9815
Val loss: 0.0951, Val accuracy: 0.9725
...
<DBG> EPOCH: 10/10
Train loss: 0.0107, Train accuracy: 0.9967
Val loss: 0.0867, Val accuracy: 0.9825
<DBG> total time taken to train the model: 55.30s
```

The precision, recall, and F1 score plot printed at the end should look as follows. The F1 score of 95%, which combines precision and recall, is pretty good for just 10 epochs.

```
<DBG> evaluating network...
      precision    recall  f1-score   support
o         0.93        0.97        0.95        1000
ki         0.96        0.95        0.95        1000
su         0.94        0.91        0.92        1000
tsu        0.97        0.97        0.97        1000
na         0.95        0.93        0.94        1000
ha         0.96        0.97        0.96        1000
ma         0.94        0.96        0.95        1000
ya         0.96        0.96        0.96        1000
re         0.96        0.97        0.97        1000
wo         0.97        0.94        0.95        1000
```



Figure 2: KMIST Accuracy Plot over 10 epochs.

accuracy			0.95	10000
macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

The characters being classified are: o, ki, su, tsu, na, ha, ma, ya, re, and wo. The classification report is a useful utility that allows us to see the performance of the net quickly.

My accuracy plot saved in `output/kmnist_plot.png` is shown in Figure 2. The accuracy goes up and plateaus around 95% while the loss is going down, which is what we generally would like to see.

Now we can now use the trained LeNet persisted in `output/kmnist_lenet.pth` for prediction. Here's my output.

```
python predict_lenet.py --model output/kmnist_lenet.pth
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: ki, predicted: ki
<DBG>: ground truth: ma, predicted: ma
<DBG>: ground truth: ha, predicted: o
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: na, predicted: na
<DBG>: ground truth: na, predicted: na
<DBG>: ground truth: ha, predicted: ha
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: ya, predicted: ya
<DBG>: ground truth: ki, predicted: ki
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: ma, predicted: ma
<DBG>: ground truth: tsu, predicted: tsu
```

Figures 3 and 4 show my screenshots with one correct and one incorrect prediction, respectively. When a prediction is correct, the character's name in the top left corner is green. When it's incorrect, the character's

```
(ai3) vladimir@OfficeGPU:~/teaching/AI/F24/hw/hw04_2$ python predict_lenet.py --model output/kmnist_lenet.pth
<DBG> loading KMNIST test dataset...
/home/vladimir/teaching/AI/F24/hw/hw04_2/predict_lenet.py:34: FutureWarning: You are using 'torch.load' with 'weights_only=False' (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for 'weights_only' will be flipped to 'True'. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via 'torch.serialization.add_safe_globals'. We recommend you start setting 'weights_only=True' for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model = torch.load(args['model']).to(device)
<DBG>: ground truth: tsu, predicted: tsu
█
```

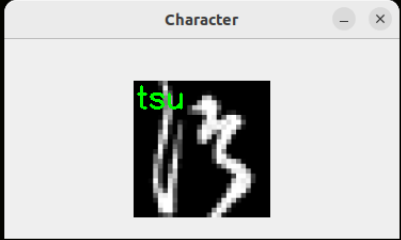


Figure 3: **Correct prediction.** A screenshot of my Linux terminal with a correct prediction.

name is red. As you see characters displayed on your screen, just hit ENTER to move to the image of next prediction.

Congratulations! We've trained and tested LeNet on KMNIST. Now we are ready to work on the homework problems.

```
(ai3) vladimir@OfficeGPU:~/teaching/AI/F24/hw/hw04_2$ python predict_lenet.py --model output/kmnist_lenet.pth
<DBG> loading KMNIST test dataset...
/home/vladimir/teaching/AI/F24/hw/hw04_2/predict_lenet.py:34: FutureWarning: You are using 'torch.load' with 'weights_only=False' (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a future release, the default value for 'weights_only' will be flipped to 'True'. This limits the functions that could be executed during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly allowlisted by the user via 'torch.serialization.add_safe_globals'. We recommend you start setting 'weights_only=True' for any use case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experimental feature.
  model = torch.load(args['model']).to(device)
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: tsu, predicted: tsu
<DBG>: ground truth: su, predicted: su
<DBG>: ground truth: kt, predicted: kt
<DBG>: ground truth: ma, predicted: ma
<DBG>: ground truth: ha, predicted: o
█
```




Figure 4: **Incorrect prediction.** A screenshot of my Linux terminal with an incorrect prediction.