Front Matter

I want the Notebook to be as informative as possible, but model creating and training process follows some standard procedure that I do not want to repeat. Therefore, if you can, spend time reading the `PROLOGUE/Routine.ipynb` Notebook first.

Convolutional Neural Network - Tiny (TinyCNN)

CNN is a great class of model architecture inspired by the human biological eyes and particularly suited for computer vision task. The essential difference of CNN model is the introduction of two new operation: “convolution” and “max-pooling”, respectively `nn.Conv2d()` and `nn.MaxPool2d()` in PyTorch. Let’s explore them first.

To begin, let’s finish installing and importing all the necessary modules as we need some example data.

```
!pip install torchmetrics
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public
/simple/
Requirement already satisfied: torchmetrics in /usr/local/lib/python3.8/dist-packages
(0.11.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.8/dist-packages
(from torchmetrics) (4.4.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.8/dist-packages (from
torchmetrics) (21.3)
Requirement already satisfied: torch>=1.8.1 in /usr/local/lib/python3.8/dist-packages (from
torchmetrics) (1.13.0+cu116)
Requirement already satisfied: numpy>=1.17.2 in /usr/local/lib/python3.8/dist-packages
(from torchmetrics) (1.21.6)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.8/dist-
packages (from packaging->torchmetrics) (3.0.9)
```

```
!pip install --upgrade mlxtend
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public
/simple/
Requirement already satisfied: mlxtend in /usr/local/lib/python3.8/dist-packages (0.14.0)
Collecting mlxtend
  Downloading mlxtend-0.21.0-py2.py3-none-any.whl (1.3 MB)
     |████████████████████████████████| 1.3 MB 36.4 MB/s
Requirement already satisfied: setuptools in /usr/local/lib/python3.8/dist-packages (from
mlxtend) (57.4.0)
Requirement already satisfied: matplotlib>=3.0.0 in /usr/local/lib/python3.8/dist-packages
(from mlxtend) (3.2.2)
Requirement already satisfied: numpy>=1.16.2 in /usr/local/lib/python3.8/dist-packages
(from mlxtend) (1.21.6)
Requirement already satisfied: scipy>=1.2.1 in /usr/local/lib/python3.8/dist-packages (from
Requirement already satisfied: scipy>=1.2.1 in /usr/local/lib/python3.8/dist-packages (from
mlxtend) (1.7.3)
Requirement already satisfied: joblib>=0.13.2 in /usr/local/lib/python3.8/dist-packages
(from mlxtend) (1.2.0)
Requirement already satisfied: pandas>=0.24.2 in /usr/local/lib/python3.8/dist-packages
(from mlxtend) (1.3.5)
Requirement already satisfied: scikit-learn>=1.0.2 in /usr/local/lib/python3.8/dist-
packages (from mlxtend) (1.0.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.8/dist-packages (from
matplotlib>=3.0.0->mlxtend) (0.11.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local
/lib/python3.8/dist-packages (from matplotlib>=3.0.0->mlxtend) (3.0.9)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.8/dist-
packages (from matplotlib>=3.0.0->mlxtend) (2.8.2)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.8/dist-packages
(from matplotlib>=3.0.0->mlxtend) (1.4.4)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-packages (from
pandas>=0.24.2->mlxtend) (2022.6)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-packages (from
python-dateutil>=2.1->matplotlib>=3.0.0->mlxtend) (1.15.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.8/dist-
packages (from scikit-learn>=1.0.2->mlxtend) (3.1.0)
Installing collected packages: mlxtend
  Attempting uninstall: mlxtend
    Found existing installation: mlxtend 0.14.0
    Uninstalling mlxtend-0.14.0:
      Successfully uninstalled mlxtend-0.14.0
Successfully installed mlxtend-0.21.0
```

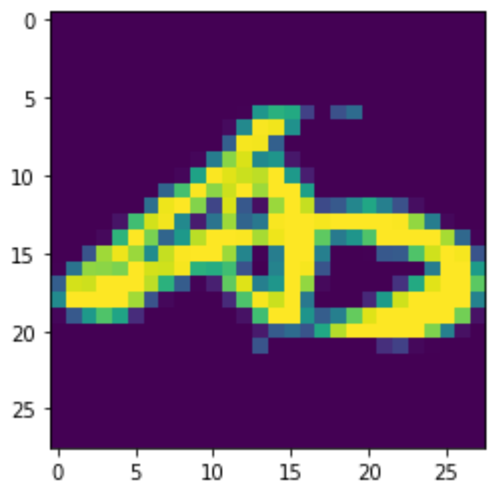
```python
from torch import nn
import torch
from torch.utils.data import DataLoader

import torchvision
from torchvision import datasets
from torchvision.transforms import ToTensor

from torchmetrics import ConfusionMatrix, Accuracy
from mlxtend.plotting import plot_confusion_matrix



# Import tqdm for progress bar
from tqdm.auto import tqdm
import matplotlib.pyplot as plt
```

```python
train_data = datasets.KMNIST(root='data',
                                   transform=ToTensor(),
                                   download=True)
test_data = datasets.KMNIST(root='data',
                                  transform=ToTensor(),
                                  download=True,
                                  train=False)
```

```
image, label = train_data[1]
image, label
```

```
(tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0039, 0.5255, 0.6353, 0.6078,
           0.1922, 0.0000, 0.2471, 0.3490, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0275, 0.4706, 0.9922, 0.9804, 0.5059,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.3020, 0.9882, 0.9843, 0.3059, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0157, 0.4196, 0.8235, 0.9451, 0.4745, 0.5216, 0.0627,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0157, 0.5765, 0.9961, 0.9569, 0.9216, 0.9020, 1.0000, 0.5412,
           0.0824, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3216,
           0.7059, 0.9961, 0.8275, 0.9255, 1.0000, 0.8549, 1.0000, 1.0000,
           0.5647, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3647, 0.9765,
           1.0000, 0.8157, 0.2078, 0.0451, 0.2745, 0.0392, 0.8510, 1.0000,
           1.0000, 0.8157, 0.2078, 0.9451, 0.2745, 0.0392, 0.8510, 1.0000,
           0.9137, 0.2235, 0.2784, 0.4588, 0.5922, 0.4314, 0.2471, 0.0353,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0510, 0.6941, 0.9333, 0.9843,
           0.4706, 0.0745, 0.3843, 0.8549, 0.3216, 0.4000, 0.9922, 1.0000,
           1.0000, 0.9882, 1.0000, 1.0000, 1.0000, 1.0000, 0.9961, 0.8392,
           0.4235, 0.0196, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0235, 0.6510, 1.0000, 0.8000, 0.3255,
           0.5373, 0.8980, 1.0000, 1.0000, 0.9961, 0.8706, 0.9765, 1.0000,
           1.0000, 0.7529, 0.4078, 0.5216, 0.7255, 1.0000, 1.0000, 1.0000,
           1.0000, 0.8706, 0.3373, 0.0078],
          [0.0000, 0.0000, 0.2588, 0.8588, 0.9765, 0.8510, 0.4863, 0.9137,
           0.9961, 1.0000, 1.0000, 0.8549, 0.4549, 0.0078, 0.4235, 1.0000,
           1.0000, 0.4314, 0.0000, 0.0000, 0.0039, 0.2745, 0.2863, 0.7255,
           0.9294, 1.0000, 0.9882, 0.2588],
          [0.0039, 0.3804, 0.8353, 0.8275, 0.8039, 1.0000, 0.9373, 0.9294,
           0.9882, 0.6392, 0.6196, 0.6941, 0.2510, 0.0000, 0.4471, 1.0000,
           0.9882, 0.2588, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.4196, 1.0000, 1.0000, 0.5490],
          [0.2980, 0.9373, 0.9255, 1.0000, 1.0000, 1.0000, 0.9020, 0.6392,
           0.2902, 0.0078, 0.0863, 0.7569, 0.9529, 0.4314, 0.5255, 1.0000,
           0.8392, 0.0235, 0.0000, 0.0000, 0.0000, 0.0000, 0.1451, 0.6078,
           0.9569, 1.0000, 1.0000, 0.7490],
          [0.4784, 1.0000, 1.0000, 1.0000, 1.0000, 0.8627, 0.2549, 0.0196,
           0.0000, 0.0000, 0.0000, 0.1176, 0.7216, 0.9882, 0.9216, 1.0000,
           0.4000, 0.0000, 0.0000, 0.0000, 0.1137, 0.5490, 0.9294, 1.0000,
           1.0000, 1.0000, 1.0000, 0.4431],
          [0.0157, 0.2784, 0.5373, 0.7255, 0.5843, 0.1255, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0471, 0.6941, 1.0000, 1.0000,
           0.5176, 0.2941, 0.4314, 0.7647, 0.9529, 1.0000, 1.0000, 1.0000,
           1.0000, 1.0000, 0.7882, 0.0471],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0706, 0.2863, 0.3490,
           0.2588, 0.5961, 0.8902, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
           0.8078, 0.4353, 0.0431, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2588, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1333, 0.1922, 0.0196,
           0.0039, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000],
          [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
           0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000]]]), 7)
```

```
plt.imshow(image.squeeze());
```



Convolution and Max-Pooling:

Okay, in the introduction, I say convolution is just `nn.Conv2d()`. It is partially incorrect. For the task of image classification in this notebook, yes, `nn.Con2d()` is what we need, because our image data have just 2 dimensions: height and width. For text, which is 1-dimensional, you will need `nn.Conv1d()`, and for 3D objects, of course there is `nn.Conv3d()`. The same is true for `nn.MaxPool2d()`.

Convolution

First, let's talk about convolution. The intuition is *a convolution layer will apply a filter on the data and try to extract a fundamental pattern out of the data*. Mathematically, it is represented by an example operation:

- We take an image such as above, which has $n * m$ pixel
- We have a *filter*, or a *kernel*, which is essentially a matrix of size $p * q$, and is smaller than the image. It is usually seen that that kernel is square a.k.a of size $p * p$. The value at each position of the kernel is a *weight* in the layer (Yup, we're gonna multiply them with something next).
- We pick a region in the image of size $p * p$, starting from the top-left-most pixel, and perform element-wise multiplication with the kernel, receiving an intermediate matrix of size $p * p$. Next, we will sum together all the elements of the intermediate matrix plus the *bias* term. We get a single value, which could be thought of as a single new pixel in this case.
- We consecutively pick the next region, moving right one pixel at a time, going down one pixel at

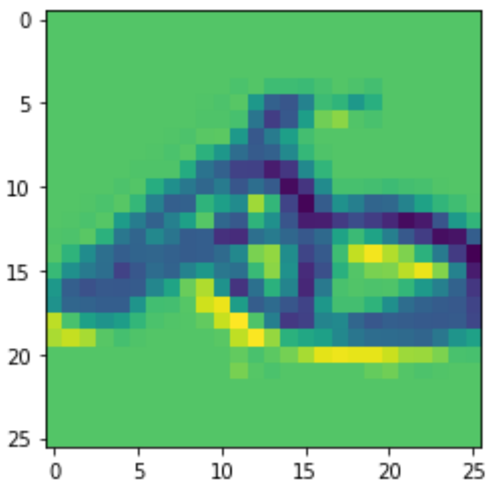
a time, until we hit the right-bottom-most pixel. The new pixel values are place side-by-side in the exact sequence of calculation.
- If you still have no idea what I am talking about, take a look at this website: CNN Explainer.

The operation described above is the most simple case, one can specify many things differently. A quick look at the attributes of `nn.Conv2d()` yields:

- `in_channels` (int) – Number of channels in the input image. Usually referred to number of color channels.
- `out_channels` (int) – Number of channels produced by the convolution.
- `kernel_size` (int or tuple) – Size of the convolving kernel. Passing just a number (such as 3) and we get a square kernel, passing a tuple of integers and we will get a non-square kernel.
- `stride` (int or tuple, optional) – Stride of the convolution. Default: 1. Essentially how many pixel we will move right to pick a new region. Pick a stride n bigger than 1 and we will skip $n-1$ pixel at a time.
- `padding` (int, tuple or str, optional) – Padding added to all four sides of the input. Default: 0. Padding is a little special. If you do the math, you will realize that convolution will reduce each dimension of the image by $p-1$ pixels for a kernel of size $p*p$. In the common case of kernel size 3, each dimension will reduce by 2, but by setting `padding=1`, the output image will have the same dimensions as the the input image.
- `padding_mode` (str, optional) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`

Now let's jump into the code. The `conv` instance we created takes in image with just 1 color channel, and output image with 1 color channel. `kernel_size` is $3*3$, or picking out 9 pixels at a time. `stride` is 1, which means we does not skip any pixel. There is no padding to the original image, which makes the new image have dimensions $26*26$.

```python
torch.manual_seed(17)

conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, stride=1, padding=0)
conv_image = conv(image)
plt.imshow(conv_image.squeeze().detach().numpy());
```



Honestly, this is not much to see, as the convolution weights are just randomly initialized and means next to nothing. However, after training, convolution can become very good at picking out pattern.

Just checking out this demonstration by Jeremy, where he showed a convolution that pick out the horizontal edges. Do what the next part as well, where he trained the convolution neural network in Excel!

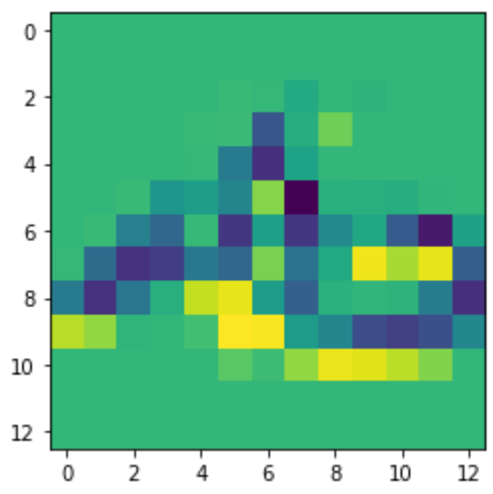
Next, let's jump into Max-Pooling

Max-Pooling

As the name suggests, we now still have a kernel of some size, let's say $p * p$. We will go through a square region of p^2 pixels at a time, and output the maximum value of the region. Then we will move to the next region, usually with a stride of $p - 1$ i.e. the next region has no common pixel with any previous region. Without padding, this means that the output image will have its dimensions divided by p.

The attributes of `nn.MaxPool2d()` is essentially similar to `nn.Conv2d()`. For the code, `max_pool` is a max-pooling layer with a kernel size of 2, and that's it.

```
max_pool = nn.MaxPool2d(kernel_size=2)
max_pool_image = max_pool(conv_image)
plt.imshow(max_pool_image.squeeze().detach().numpy());
```



The model I build will be aptly named TinyVGG, with the architecture adapted from CNN Explainer mentioned above.

```
BATCH_SIZE = 2048

TrainLoader = DataLoader(train_data, BATCH_SIZE, shuffle=True)
TestLoader = DataLoader(test_data, BATCH_SIZE, shuffle=False)
```

```
print(f"Dataloaders: {TrainLoader, TestLoader}")
print(f"Length of train dataloader: {len(TrainLoader)} batches of {BATCH_SIZE}")
print(f"Length of test dataloader: {len(TestLoader)} batches of {BATCH_SIZE}")
```

```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x7ff4903aa550>,
<torch.utils.data.dataloader.DataLoader object at 0x7ff4903aafa0>)
Length of train dataloader: 30 batches of 2048
Length of test dataloader: 5 batches of 2048
```

```
class_names = train_data.classes
class_names
```

```
['o', 'ki', 'su', 'tsu', 'na', 'ha', 'ma', 'ya', 're', 'wo']
```

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

```
class TinyConvNet(nn.Module):
    def __init__(self, input_shape, hidden_units, output_shape):
        super().__init__()
        self.block1 = nn.Sequential(
            nn.Conv2d(input_shape, hidden_units, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(hidden_units*7*7, output_shape)
        )

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.classifier(x)
        return x
```

```
torch.manual_seed(17)
model = TinyConvNet(input_shape=1,
    hidden_units=10,
    output_shape=len(class_names)).to(device)
model
```

```
TinyConvNet(
  (block1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (block2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=490, out_features=10, bias=True)
  )
)
```

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
```

```
accuracy = Accuracy(task='multiclass', num_classes=len(class_names)).to(device)
```

```
def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               criterion: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               metric: Accuracy,
               device: torch.device = device):
    train_loss, train_acc = 0, 0
    for batch, (X,y) in enumerate(data_loader):
        X, y = X.to(device), y.to(device)
        # 1. Forward pass
        y_pred = model(X)

        # 2. Calculate loss & accuracy
        loss = criterion(y_pred, y)
        train_loss += loss
        train_acc += metric(y_pred.argmax(dim=1), y)

        # 3. Empty out gradient
        optimizer.zero_grad()

        # 4. Backpropagation
        loss.backward()

        # 5. Optimize 1 step
        optimizer.step()

    train_loss /= len(data_loader)
    train_acc /= len(data_loader)
    print(f"Train loss: {train_loss:.5f} | Train accuracy: {train_acc:.2f}")
```

```
def test_step(model: torch.nn.Module,
              data_loader: torch.utils.data.DataLoader,
              criterion: torch.nn.Module,
              metric: Accuracy
              metric: Accuracy,
              device: torch.device = device):
    test_loss, acc = 0, 0
    model.eval()
    with torch.inference_mode():
        for (X,y) in data_loader:
            X, y = X.to(device), y.to(device)
        # 1. Forward pass
            y_pred = model(X)

        # 2. Calculate loss & accuracy
            test_loss += criterion(y_pred, y)
            acc += metric(y_pred.argmax(dim=1), y)

        test_loss /= len(data_loader)
        acc /= len(data_loader)
        print(f"Test loss: {test_loss:.5f} | Test accuracy: {acc:.2f}")
```

```
epochs = 20
for epoch in tqdm(range(epochs)):
    print(f"Epoch: {epoch}\n---------")
    train_step(data_loader=TrainLoader,
        model=model,
        criterion=criterion,
        optimizer=optimizer,
        metric=accuracy,
        device=device
    )
    test_step(data_loader=TestLoader,
        model=model,
        criterion=criterion,
        metric=accuracy,
        device=device
    )
```

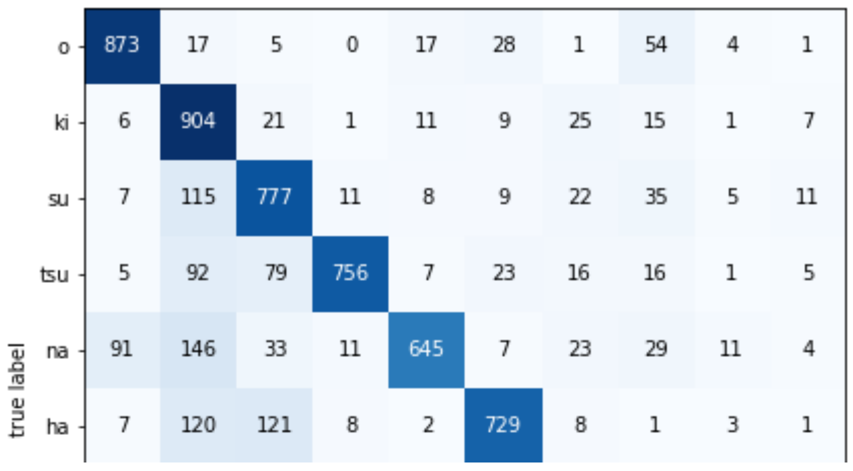
```
Epoch: 0
---------
Train loss: 2.30309 | Train accuracy: 0.10
Test loss: 2.30214 | Test accuracy: 0.12
Epoch: 1
---------
Train loss: 2.30146 | Train accuracy: 0.11
Test loss: 2.30087 | Test accuracy: 0.14
Epoch: 2
---------
Train loss: 2.29937 | Train accuracy: 0.12
Test loss: 2.29867 | Test accuracy: 0.14
Epoch: 3
---------
Train loss: 2.29444 | Train accuracy: 0.19
Test loss: 2.29213 | Test accuracy: 0.20
Epoch: 4
---------
```

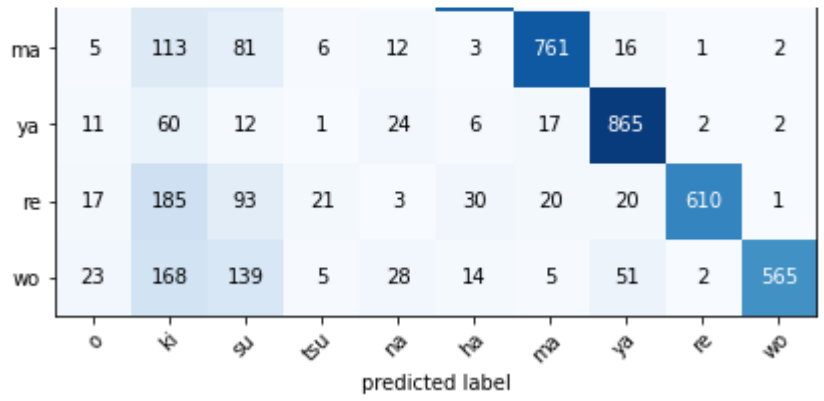
```
Train loss: 2.26622 | Train accuracy: 0.29
Test loss: 2.22187 | Test accuracy: 0.30
Epoch: 5
---------
Train loss: 1.77649 | Train accuracy: 0.49
Test loss: 1.63724 | Test accuracy: 0.50
Epoch: 6
---------
Train loss: 1.19701 | Train accuracy: 0.63
Test loss: 1.84510 | Test accuracy: 0.49
Epoch: 7
---------
Train loss: 1.01789 | Train accuracy: 0.68
Test loss: 1.51725 | Test accuracy: 0.50
Epoch: 8
---------
Train loss: 0.89062 | Train accuracy: 0.73
Test loss: 1.27252 | Test accuracy: 0.60
Epoch: 9
---------
Train loss: 0.80031 | Train accuracy: 0.75
Test loss: 1.29735 | Test accuracy: 0.57
Epoch: 10
---------
Train loss: 0.72534 | Train accuracy: 0.78
Test loss: 1.10146 | Test accuracy: 0.66
Epoch: 11
---------
Train loss: 0.64889 | Train accuracy: 0.80
Test loss: 1.09186 | Test accuracy: 0.64
Epoch: 12
---------
Train loss: 0.60812 | Train accuracy: 0.81
Test loss: 0.96741 | Test accuracy: 0.70
Epoch: 13
---------
Train loss: 0.54985 | Train accuracy: 0.83
Test loss: 0.91418 | Test accuracy: 0.71
Epoch: 14
---------
Train loss: 0.51208 | Train accuracy: 0.84
Test loss: 0.84423 | Test accuracy: 0.73
Epoch: 15
---------
Train loss: 0.45502 | Train accuracy: 0.86
Test loss: 0.74528 | Test accuracy: 0.77
Epoch: 16
---------
Train loss: 0.41647 | Train accuracy: 0.87
Test loss: 0.81173 | Test accuracy: 0.74
Epoch: 17
---------
Train loss: 0.39236 | Train accuracy: 0.88
Test loss: 0.79019 | Test accuracy: 0.75
Test loss: 0.79019 | Test accuracy: 0.75
Epoch: 18
---------
Train loss: 0.38087 | Train accuracy: 0.88
Test loss: 0.70741 | Test accuracy: 0.78
Epoch: 19
---------
Train loss: 0.34548 | Train accuracy: 0.89
Test loss: 0.78307 | Test accuracy: 0.75
```

Our model looks good! Let's make prediction on the test set.

```python
# 1. Make predictions with trained model
y_preds = []
model.eval()
with torch.inference_mode():
  for X, y in tqdm(TestLoader, desc="Making predictions"):
    # Send data and targets to target device
    X, y = X.to(device), y.to(device)
    # Do the forward pass
    y_logit = model(X)
    # Turn predictions from logits -> prediction probabilities -> predictions labels
    y_pred = torch.softmax(y_logit, dim=1).argmax(dim=1)
    # Put predictions on CPU for evaluation
    y_preds.append(y_pred.cpu())
# Concatenate list of predictions into a tensor
y_pred_tensor = torch.cat(y_preds)
```

```python
# 2. Setup confusion matrix instance and compare predictions to targets
confmat = ConfusionMatrix(num_classes=len(class_names), task='multiclass')
confmat_tensor = confmat(preds=y_pred_tensor,
                         target=test_data.targets)

# 3. Plot the confusion matrix
fig, ax = plot_confusion_matrix(
    conf_mat=confmat_tensor.numpy(), # matplotlib likes working with NumPy
    class_names = class_names, # turn the row and column labels into class names
    figsize=(10, 7)
);
```





	o	ki	su	tsu	na	ha	ma	ya	re	wo
ma	5	113	81	6	12	3	761	16	1	2
ya	11	60	12	1	24	6	17	865	2	2
re	17	185	93	21	3	30	20	20	610	1
wo	23	168	139	5	28	14	5	51	2	565

The convolution architecture seems to be better than baseline model in the Routine notebook. However, notice that the accuracy for the training set is much higher than the accuracy for the testing set, indicating overfitting.The confusion matrix also shows that the model seems to mistake other charaters for `'ki'` the most.