



Header

## 3. Convolutional Neural Networks

In the previous section, we built and trained a simple model to classify ASL images. The model was able to learn how to correctly classify the training dataset with very high accuracy, but, it did not perform nearly as well on validation dataset. This behavior of not generalizing well to non-training data is called [overfitting](#), and in this section, we will introduce a popular kind of model called a [convolutional neural network](#) that is especially good for reading images and classifying them.

### 3.1 Objectives

- Prep data specifically for a CNN
- Create a more sophisticated CNN model, understanding a greater variety of model layers
- Train a CNN model and observe its performance

```
In [1]: import torch.nn as nn
import pandas as pd
import torch
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader

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

Out[1]: True

## 3.2 Loading and Preparing the Data

### 3.2.1 Preparing Images

Let's begin by loading our DataFrames like we did in the previous lab:

```
In [2]: train_df = pd.read_csv("data/asl_data/sign_mnist_train.csv")
valid_df = pd.read_csv("data/asl_data/sign_mnist_valid.csv")
```

This ASL data is already flattened.

```
In [3]: sample_df = train_df.head().copy() # Grab the top 5 rows
sample_df.pop('label')
sample_x = sample_df.values
sample_x
```

```
Out[3]: array([[107, 118, 127, ..., 204, 203, 202],
               [155, 157, 156, ..., 103, 135, 149],
               [187, 188, 188, ..., 195, 194, 195],
               [211, 211, 212, ..., 222, 229, 163],
               [164, 167, 170, ..., 163, 164, 179]])
```

```
In [4]: sample_x.shape
```

```
Out[4]: (5, 784)
```

In this format, we don't have all the information about which pixels are near each other. Because of this, we can't apply convolutions that will detect features. Let's [reshape](#) our dataset so that they are in a 28x28 pixel format. This will allow our convolutions to associate groups of pixels and detect important features.

Note that for the first convolutional layer of our model, we need to have not only the height and width of the image, but also the number of [color channels](#). Our images are grayscale, so we'll just have 1 channel.

That means that we need to convert the current shape `(5, 784)` to `(5, 1, 28, 28)`. With [NumPy](#) arrays, we can pass a `-1` for any dimension we wish to remain the same.

```
In [5]: IMG_HEIGHT = 28
        IMG_WIDTH = 28
        IMG_CHS = 1

        sample_x = sample_x.reshape(-1, IMG_CHS, IMG_HEIGHT, IMG_WIDTH)
        sample_x.shape
```

```
Out[5]: (5, 1, 28, 28)
```

### 3.2.2 Create a Dataset

Let's add the steps above into our `MyDataset` class.

#### Exercise

There are 4 `FIXME`s in the class definition below. Can you replace them with the correct values?

```
In [6]: class MyDataset(Dataset):
        def __init__(self, base_df):
            x_df = base_df.copy() # Some operations below are in-place
            y_df = x_df.pop(FIXME)
            x_df = x_df.values / 255 # Normalize values from 0 to 1
            x_df = x_df.reshape(-1, FIXME, FIXME, FIXME)
            self.xs = torch.tensor(x_df).float().to(device)
            self.ys = torch.tensor(y_df).to(device)

        def __getitem__(self, idx):
            x = self.xs[idx]
            y = self.ys[idx]
            return x, y
```

```
def __len__(self):
    return len(self.xs)
```

## Solution

Click the `...` below for the solution.

```
In [7]: # SOLUTION
class MyDataset(Dataset):
    def __init__(self, base_df):
        x_df = base_df.copy() # Some operations below are in-place
        y_df = x_df.pop('label')
        x_df = x_df.values / 255 # Normalize values from 0 to 1
        x_df = x_df.reshape(-1, IMG_CHS, IMG_WIDTH, IMG_HEIGHT)
        self.xs = torch.tensor(x_df).float().to(device)
        self.ys = torch.tensor(y_df).to(device)

    def __getitem__(self, idx):
        x = self.xs[idx]
        y = self.ys[idx]
        return x, y

    def __len__(self):
        return len(self.xs)
```

## 3.2.3 Create a DataLoader

Next, let's create the DataLoader from the Dataset

### Exercise

One of these function calls is missing the `shuffle=True` argument. Can you remember which one it is and add it back in?

```
In [8]: BATCH_SIZE = 32

train_data = MyDataset(train_df)
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE)
train_N = len(train_loader.dataset)

valid_data = MyDataset(valid_df)
valid_loader = DataLoader(valid_data, batch_size=BATCH_SIZE)
valid_N = len(valid_loader.dataset)
```

## Solution

Click the `...` below for the solution.

```
In [9]: # SOLUTION
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True)
```

Let's grab a batch from the DataLoader to make sure it works.

```
In [10]: batch = next(iter(train_loader))  
batch
```

```

Out[10]: [tensor([[[[0.7569, 0.7647, 0.7686, ..., 0.6980, 0.6941, 0.6902],
[0.7608, 0.7647, 0.7686, ..., 0.7059, 0.7020, 0.6941],
[0.7686, 0.7725, 0.7725, ..., 0.7098, 0.7059, 0.7020],
...,
[0.8627, 0.8588, 0.8941, ..., 0.7961, 0.7922, 0.7765],
[0.8627, 0.8627, 0.9294, ..., 0.8039, 0.7961, 0.7804],
[0.8627, 0.8706, 0.9137, ..., 0.8039, 0.7922, 0.7843]]]],

[[[0.8000, 0.8196, 0.8235, ..., 0.8235, 0.8196, 0.8157],
[0.8078, 0.8235, 0.8353, ..., 0.8314, 0.8275, 0.8196],
[0.8196, 0.8275, 0.8353, ..., 0.8392, 0.8353, 0.8275],
...,
[0.8980, 0.9686, 0.7647, ..., 0.9294, 0.9255, 0.9216],
[0.9451, 0.8392, 0.3922, ..., 0.9373, 0.9294, 0.9216],
[0.9059, 0.4941, 0.3529, ..., 0.9412, 0.9333, 0.9216]]]],

[[[0.7333, 0.7373, 0.7412, ..., 0.7412, 0.7412, 0.7333],
[0.7412, 0.7451, 0.7490, ..., 0.7529, 0.7529, 0.7451],
[0.7529, 0.7529, 0.7569, ..., 0.7608, 0.7608, 0.7647],
...,
[0.8745, 0.8824, 0.8863, ..., 0.8235, 0.8000, 0.8353],
[0.7608, 0.7725, 0.7843, ..., 0.7059, 0.8314, 0.7765],
[0.3843, 0.3922, 0.4118, ..., 0.6392, 0.8471, 0.8235]]]],

...,

[[[0.7804, 0.7843, 0.7922, ..., 0.7216, 0.7137, 0.7059],
[0.7882, 0.7882, 0.7922, ..., 0.7255, 0.7216, 0.7176],
[0.7922, 0.7961, 0.8000, ..., 0.7373, 0.7294, 0.7216],
...,
[0.8706, 0.8745, 0.8706, ..., 0.8157, 0.8118, 0.8078],
[0.8588, 0.8667, 0.8706, ..., 0.8078, 0.8039, 0.7961],
[0.8549, 0.8627, 0.8706, ..., 0.8118, 0.8039, 0.8000]]]],

[[[0.8824, 0.8824, 0.8863, ..., 0.8118, 0.8118, 0.8039],
[0.8941, 0.8941, 0.8980, ..., 0.8314, 0.8235, 0.8196],
[0.9020, 0.9098, 0.9059, ..., 0.8431, 0.8392, 0.8314],
...,
[1.0000, 1.0000, 1.0000, ..., 0.9765, 0.9725, 0.9569],
[1.0000, 1.0000, 1.0000, ..., 0.9804, 0.9765, 0.9725],
[1.0000, 1.0000, 1.0000, ..., 0.9843, 0.9804, 0.9765]]]],

[[[0.7333, 0.7373, 0.7333, ..., 0.6745, 0.6706, 0.6667],
[0.7451, 0.7451, 0.7412, ..., 0.6863, 0.6824, 0.6784],
[0.7569, 0.7529, 0.7490, ..., 0.6980, 0.6902, 0.6863],
...,
[0.8471, 0.8471, 0.8510, ..., 0.8118, 0.8039, 0.7961],
[0.8510, 0.8510, 0.8471, ..., 0.8118, 0.8078, 0.7961],
[0.8471, 0.8510, 0.8510, ..., 0.8157, 0.7843, 0.6196]]]],
device='cuda:0'),
tensor([ 3, 23, 22, 13,  6, 13,  9, 10,  8, 20, 14,  6, 20, 12, 22, 15, 15, 1
7,
      10, 23, 14, 10, 19,  5,  5, 10, 19,  5, 22, 18,  8,  8],
device='cuda:0')]

```

It looks different, but let's check the `shape`s to be sure.

```
In [11]: batch[0].shape
```

```
Out[11]: torch.Size([32, 1, 28, 28])
```

```
In [12]: batch[1].shape
```

```
Out[12]: torch.Size([32])
```

## 3.3 Creating a Convolutional Model

These days, many data scientists start their projects by borrowing model properties from a similar project. Assuming the problem is not totally unique, there's a great chance that people have created models that will perform well which are posted in online repositories like [TensorFlow Hub](#) and the [NGC Catalog](#). Today, we'll provide a model that will work well for this problem.




No description has been provided for this image

We covered many of the different kinds of layers in the lecture, and we will go over them all here with links to their documentation. When in doubt, read the official documentation (or ask [Stack Overflow](#)).

```
In [13]: n_classes = 24
kernel_size = 3
flattened_img_size = 75 * 3 * 3

model = nn.Sequential(
    # First convolution
    nn.Conv2d(IMG_CHS, 25, kernel_size, stride=1, padding=1), # 25 x 28 x 28
    nn.BatchNorm2d(25),
    nn.ReLU(),
    nn.MaxPool2d(2, stride=2), # 25 x 14 x 14
    # Second convolution
    nn.Conv2d(25, 50, kernel_size, stride=1, padding=1), # 50 x 14 x 14
    nn.BatchNorm2d(50),
    nn.ReLU(),
    nn.Dropout(.2),
    nn.MaxPool2d(2, stride=2), # 50 x 7 x 7
    # Third convolution
    nn.Conv2d(50, 75, kernel_size, stride=1, padding=1), # 75 x 7 x 7
    nn.BatchNorm2d(75),
    nn.ReLU(),
    nn.MaxPool2d(2, stride=2), # 75 x 3 x 3
    # Flatten to Dense
    nn.Flatten(),
    nn.Linear(flattened_img_size, 512),
    nn.Dropout(.3),
    nn.ReLU(),
    nn.Linear(512, n_classes)
)
```

### 3.3.1 Conv2D

No description has been provided for this image

These are our 2D convolutional layers. Small kernels will go over the input image and detect features that are important for classification. Earlier convolutions in the model will detect simple features such as lines. Later convolutions will detect more complex features. Let's look at our first Conv2D layer:

```
nn.Conv2d(IMG_CHS, 25, kernel_size, stride=1, padding=1)
```


25 refers to the number of filters that will be learned. Even though `kernel_size = 3`, PyTorch will assume we want 3 x 3 filters. Stride refer to the step size that the filter will take as it passes over the image. Padding refers to whether the output image that's created from the filter will match the size of the input image.

### 3.3.2 BatchNormalization


Like normalizing our inputs, batch normalization scales the values in the hidden layers to improve training. [Read more about it in detail here.](#)

There is a debate on best where to put the batch normalization layer. [This Stack Overflow post](#) compiles many perspectives.

### 3.3.3 MaxPool2D

No description has been provided for this image Max pooling takes an image and essentially shrinks it to a lower resolution. It does this to help the model be robust to translation (objects moving side to side), and also makes our model faster.

### 3.3.4 Dropout

No description has been provided for this image Dropout is a technique for preventing overfitting. Dropout randomly selects a subset of neurons and turns them off, so that they do not participate in forward or backward propagation in that particular pass. This helps to make sure that the network is robust and redundant, and does not rely on any one area to come up with answers.

### 3.3.5 Flatten

Flatten takes the output of one layer which is multidimensional, and flattens it into a one-dimensional array. The output is called a feature vector and will be connected to the final classification layer.

### 3.3.6 Linear

We have seen dense linear layers before in our earlier models. Our first dense layer (512 units) takes the feature vector as input and learns which features will contribute to a particular classification. The second dense layer (24 units) is the final classification layer that outputs our prediction.

## 3.4 Summarizing the Model

This may feel like a lot of information, but don't worry. It's not critical that to understand everything right now in order to effectively train convolutional models. Most importantly we know that they can help with extracting useful information from images, and can be used in classification tasks.

```
In [14]: model = torch.compile(model.to(device))
         model
```

```
Out[14]: OptimizedModule(
  (_orig_mod): Sequential(
    (0): Conv2d(1, 25, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(25, 50, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(50, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): Dropout(p=0.2, inplace=False)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(50, 75, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(75, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (13): Flatten(start_dim=1, end_dim=-1)
    (14): Linear(in_features=675, out_features=512, bias=True)
    (15): Dropout(p=0.3, inplace=False)
    (16): ReLU()
    (17): Linear(in_features=512, out_features=24, bias=True)
  )
)
```

Since the problem we are trying to solve is still the same (classifying ASL images), we will continue to use the same `loss_function` and `accuracy` metric.

```
In [15]: loss_function = nn.CrossEntropyLoss()
         optimizer = Adam(model.parameters())
```

```
In [16]: def get_batch_accuracy(output, y, N):
         pred = output.argmax(dim=1, keepdim=True)
         correct = pred.eq(y.view_as(pred)).sum().item()
         return correct / N
```



## 3.5 Training the Model

Despite the very different model architecture, the training looks exactly the same.

### Exercise

These are the same `train` and `validate` functions as before, but they have been mixed up. Can you correctly name each function and replace the `FIXME` s?

One of them should have `model.train` and the other should have `model.eval` .

```
In [17]: def FIXME():
    loss = 0
    accuracy = 0

    model.FIXME()
    with torch.no_grad():
        for x, y in FIXME:
            output = model(x)

            loss += loss_function(output, y).item()
            accuracy += get_batch_accuracy(output, y, valid_N)
    print('FIXME - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

```
In [18]: def FIXME():
    loss = 0
    accuracy = 0

    model.FIXME()
    for x, y in FIXME:
        output = model(x)
        optimizer.zero_grad()
        batch_loss = loss_function(output, y)
        batch_loss.backward()
        optimizer.step()

        loss += batch_loss.item()
        accuracy += get_batch_accuracy(output, y, train_N)
    print('FIXME - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

### Solution

Click the two `...` s below for the solution.

```
In [19]: # SOLUTION
def validate():
    loss = 0
    accuracy = 0

    model.eval()
    with torch.no_grad():
        for x, y in valid_loader:
            output = model(x)

            loss += loss_function(output, y).item()
```

```
        accuracy += get_batch_accuracy(output, y, valid_N)
    print('Valid - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

```
In [20]: # SOLUTION
def train():
    loss = 0
    accuracy = 0

    model.train()
    for x, y in train_loader:
        output = model(x)
        optimizer.zero_grad()
        batch_loss = loss_function(output, y)
        batch_loss.backward()
        optimizer.step()

        loss += batch_loss.item()
        accuracy += get_batch_accuracy(output, y, train_N)
    print('Train - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

```
In [21]: epochs = 20

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    train()
    validate()
```

```
Epoch: 0
Train - Loss: 301.7515 Accuracy: 0.8940
Valid - Loss: 48.6050 Accuracy: 0.9327
Epoch: 1
Train - Loss: 15.1753 Accuracy: 0.9960
Valid - Loss: 18.6190 Accuracy: 0.9695
Epoch: 2
Train - Loss: 13.8267 Accuracy: 0.9952
Valid - Loss: 20.2582 Accuracy: 0.9596
Epoch: 3
Train - Loss: 3.0745 Accuracy: 0.9993
Valid - Loss: 230.9491 Accuracy: 0.7995
Epoch: 4
Train - Loss: 21.0102 Accuracy: 0.9926
Valid - Loss: 22.5936 Accuracy: 0.9700
Epoch: 5
Train - Loss: 7.2960 Accuracy: 0.9978
Valid - Loss: 54.7620 Accuracy: 0.9441
Epoch: 6
Train - Loss: 7.8509 Accuracy: 0.9974
Valid - Loss: 16.1648 Accuracy: 0.9774
Epoch: 7
Train - Loss: 5.8721 Accuracy: 0.9981
Valid - Loss: 10.0669 Accuracy: 0.9796
Epoch: 8
Train - Loss: 0.4390 Accuracy: 0.9999
Valid - Loss: 13.1425 Accuracy: 0.9816
Epoch: 9
Train - Loss: 8.7129 Accuracy: 0.9971
Valid - Loss: 12.6556 Accuracy: 0.9826
Epoch: 10
Train - Loss: 5.2926 Accuracy: 0.9980
Valid - Loss: 48.6905 Accuracy: 0.9395
Epoch: 11
Train - Loss: 5.5099 Accuracy: 0.9983
Valid - Loss: 17.2883 Accuracy: 0.9755
Epoch: 12
Train - Loss: 4.0667 Accuracy: 0.9985
Valid - Loss: 14.5109 Accuracy: 0.9774
Epoch: 13
Train - Loss: 4.0736 Accuracy: 0.9985
Valid - Loss: 9.0670 Accuracy: 0.9873
Epoch: 14
Train - Loss: 4.3141 Accuracy: 0.9991
Valid - Loss: 14.9435 Accuracy: 0.9822
Epoch: 15
Train - Loss: 0.4445 Accuracy: 0.9999
Valid - Loss: 22.8482 Accuracy: 0.9742
Epoch: 16
Train - Loss: 4.8388 Accuracy: 0.9984
Valid - Loss: 21.3626 Accuracy: 0.9738
Epoch: 17
Train - Loss: 5.1629 Accuracy: 0.9985
Valid - Loss: 15.3899 Accuracy: 0.9838
Epoch: 18
Train - Loss: 2.9120 Accuracy: 0.9992
Valid - Loss: 12.7541 Accuracy: 0.9827
Epoch: 19
Train - Loss: 1.7551 Accuracy: 0.9993
Valid - Loss: 29.7630 Accuracy: 0.9688
```

### 3.5.1 Discussion of Results

It looks like this model is significantly improved! The training accuracy is very high, and the validation accuracy has improved as well. This is a great result, as all we had to do was swap in a new model.

You may have noticed the validation accuracy jumping around. This is an indication that our model is still not generalizing perfectly. Fortunately, there's more that we can do. Let's talk about it in the next lecture.

## 3.6 Summary

In this section, we utilized several new kinds of layers to implement a CNN, which performed better than the more simple model used in the last section. Hopefully the overall process of creating and training a model with prepared data is starting to become even more familiar.

### 3.6.1 Clear the Memory

Before moving on, please execute the following cell to clear up the GPU memory. This is required to move on to the next notebook.

```
In [22]: import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

```
Out[22]: {'status': 'ok', 'restart': True}
```

### 3.6.2 Next

In the last several sections you have focused on the creation and training of models. In order to further improve performance, you will now turn your attention to *data augmentation*, a collection of techniques that will allow your models to train on more and better data than what you might have originally at your disposal.

