Deep Learning Project

ASL Alphabet Classification

1. About the dataset

The goal of the project is to correctly identify the letter from American Sign Language based on a picture of the said sign. The dataset consists of 3000 photos per letter/sign, and 29 classes (26 for the letters A-Z, a class for "Space", one for "Delete" and one for "Nothing".

2. Data processing

The images don't seem to hold any important information that would differentiate the classes they belong to from each other, except for the shape of the hand (Figure 1), so I decided to use the edges only as features (Figure 2).

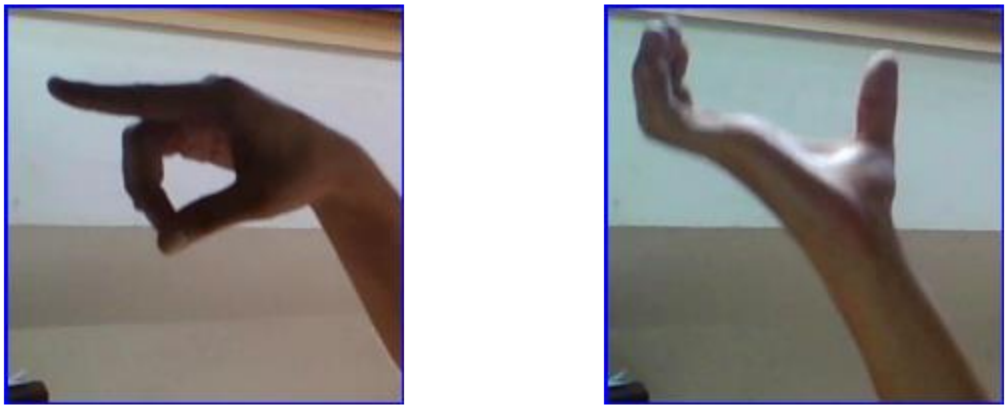


Figure 1: Example of images for letter P and "Space"

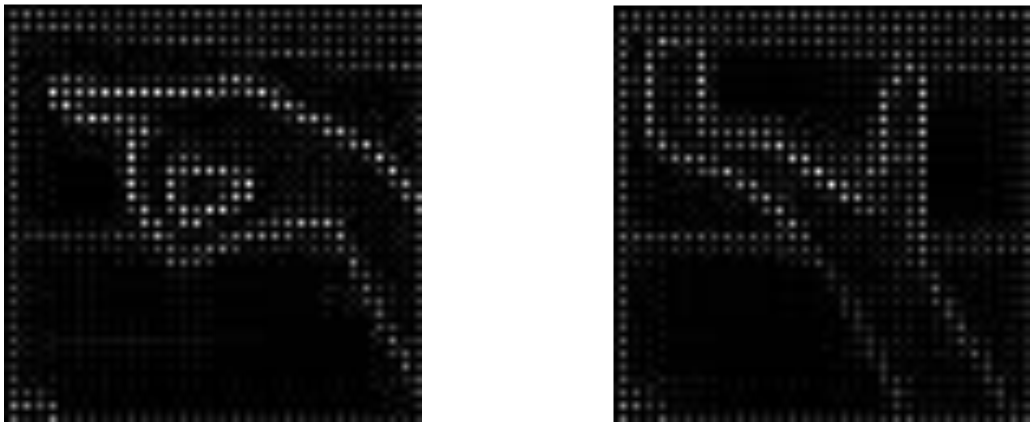


Figure 2: Histograms of Oriented Gradients of Images from Figure 1

The Histograms of Oriented Gradients (HOG) were used for training instead, applied on resized images (from 200x200 to 64x64 pixels).

The group of images was split into train (60%), test (28%) and validation (12%).

3. Model training and hyperparameter tuning

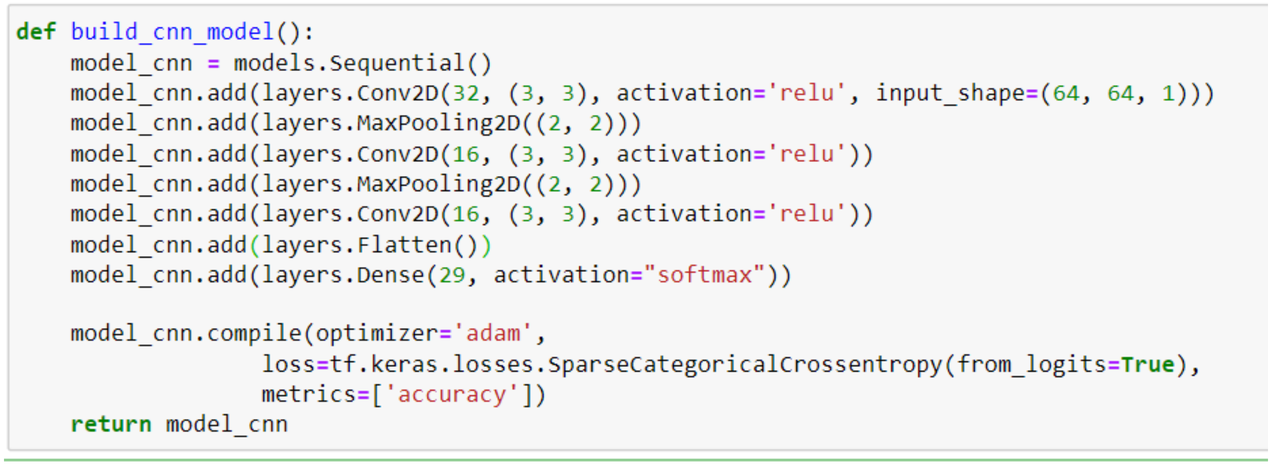
I trained 4 different models: a Convolutional Neural Network (CNN) model, a Recurrent Neural Network (RNN) model, and a Multi-Layer Perceptron Classifier.

In order to find the best hyperparameters combinations, I used the “validation_curve” function from the “sklearn” library with 5-fold cross-validation. The scoring metric I chose is **accuracy**

a) Convolutional Neural Network (CNN) Model

Complexity (Figure 3):

- 1 convolutional layer with 32 filters, 3x3 kernel size followed by a max pooling layer of 2 by 2, input shape 64x64x1
- 1 convolutional layer with 16 filters, 3x3 kernel size followed by a max pooling layer of 2 by 2
- 1 convolutional layer with 16 filters, 3x3 kernel size
- 1 dense layer with 29 outputs (for each class)



```python
def build_cnn_model():
    model_cnn = models.Sequential()
    model_cnn.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 1)))
    model_cnn.add(layers.MaxPooling2D((2, 2)))
    model_cnn.add(layers.Conv2D(16, (3, 3), activation='relu'))
    model_cnn.add(layers.MaxPooling2D((2, 2)))
    model_cnn.add(layers.Conv2D(16, (3, 3), activation='relu'))
    model_cnn.add(layers.Flatten())
    model_cnn.add(layers.Dense(29, activation="softmax"))

    model_cnn.compile(optimizer='adam',
                      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                      metrics=['accuracy'])
    return model_cnn
```

Figure 3: CNN Architecture (1st)

“ReLU” is the go-to activation function for most Neural Networks. It has been proven to work best (according to towardsdatascience.com). It has a good influence on the results if it is being used throughout as many layers as possible, not just one, as explained in the link.

“Softmax” was the output layer activation function. It is the best choice for classification tasks, because it simplifies the predicting process by returning the probabilities of each image belonging to a certain class (the sum of probabilities equals to 1).

The CNN model was validated on different batch sizes (Figure 3) and number of epochs (Figure 4), and a different architecture/complexity (Figure 5):

```
plot_figure(train_scores_mean, val_scores_mean, [10, 15, 25], title=None, legend=['Train', 'Validation'],
            labels=['Batch Size', 'Accuracy'])
```

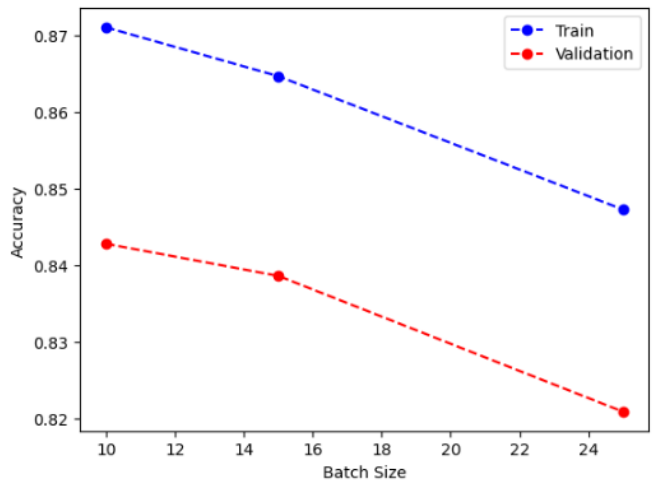


Figure 4: Train and Validation accuracy for a batch size of 10, 15, and 25

It works best with a batch size of 10.

```
plot_figure(train_scores_mean, val_scores_mean, [5, 7, 10, 15], title=None, legend=['Train', 'Validation'],
            labels=['Number of Epochs', 'Accuracy'])
```

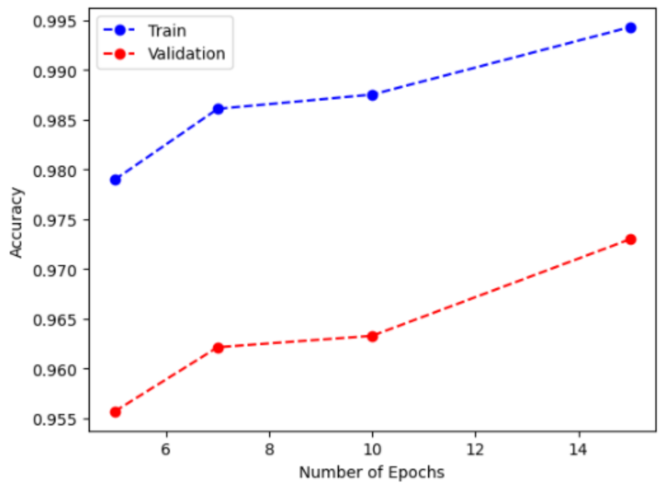


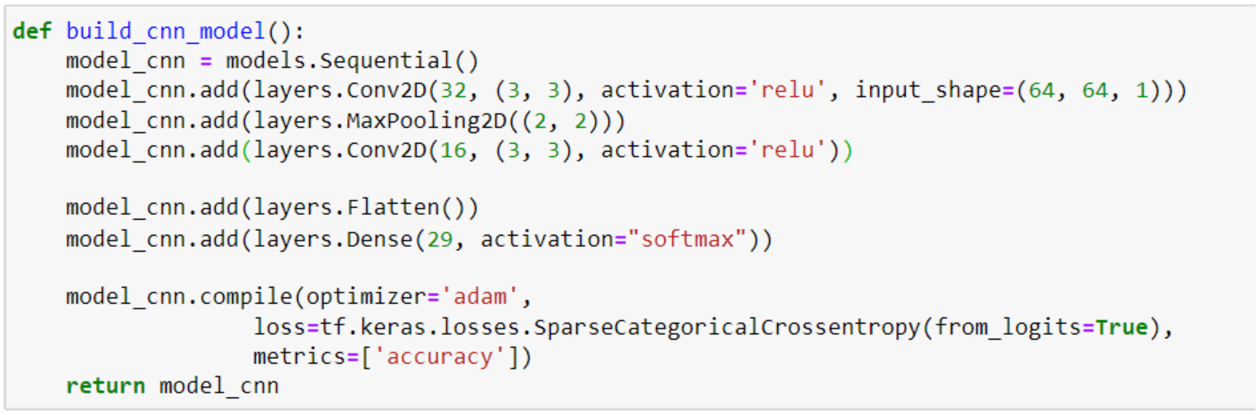
Figure 5: Train and Validation accuracy for a number of epochs of 5, 7, 10, and 15

The preferred number of epochs is 15.

Figure 7 shows the performance of a CNN of a different architecture (2 convolutional layers, 1 Max Pooling layer and 1 Dense).

2nd architecture, a less complex one (more prone to underfitting and less prone to overfitting):

- 1 convolutional layer with 32 filters, 3x3 kernel size followed by a max pooling layer of 2 by 2, input shape 64x64x1
- 1 convolutional layer with 16 filters, 3x3 kernel size
- 1 dense layer with 29 outputs (for each class)



```
def build_cnn_model():
    model_cnn = models.Sequential()
    model_cnn.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 1)))
    model_cnn.add(layers.MaxPooling2D((2, 2)))
    model_cnn.add(layers.Conv2D(16, (3, 3), activation='relu'))

    model_cnn.add(layers.Flatten())
    model_cnn.add(layers.Dense(29, activation="softmax"))

    model_cnn.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
    return model_cnn
```

Figure 6: 2nd CNN Architecture being validated

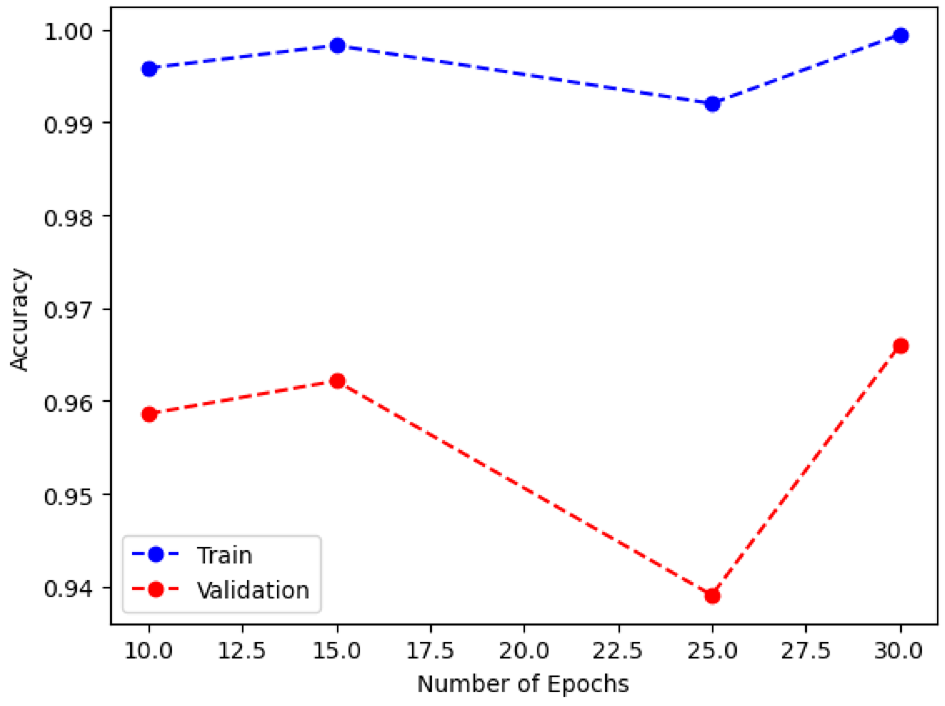
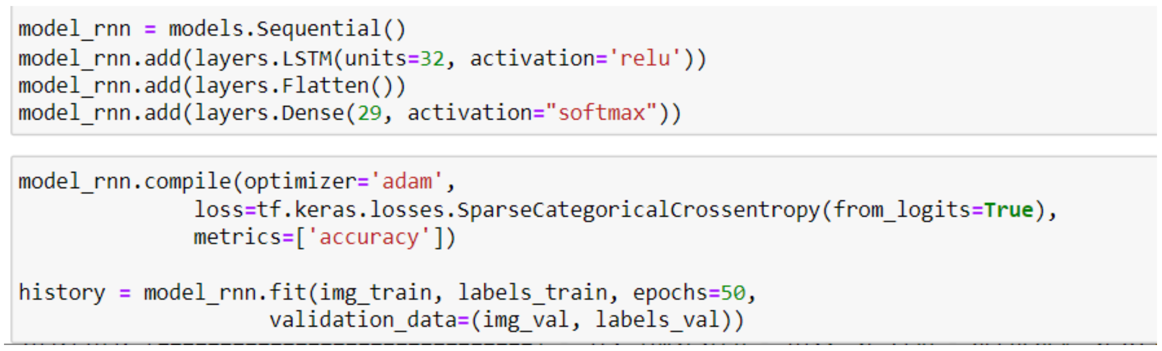


Figure 7: Accuracy of a less complex CNN model trained on 10, 15, 25 and 30 epochs

It gives a result (96.5%) about as good as the more complex CNN (~97.3%), but it needs a bigger number of epochs (*30 vs 15*). That is because more complex models are capable of learning better. However, the less complex model is still optimal enough for the current task.

b) Recurrent Neural Network (RNN) Model



```
model_rnn = models.Sequential()
model_rnn.add(layers.LSTM(units=32, activation='relu'))
model_rnn.add(layers.Flatten())
model_rnn.add(layers.Dense(29, activation="softmax"))
```

```
model_rnn.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model_rnn.fit(img_train, labels_train, epochs=50,
                    validation_data=(img_val, labels_val))
```

Figure 8: the RNN model architecture

While training an RNN model with the above architecture, I noticed that it takes a greater number of epochs (~50-60) in order to get a similar accuracy score to the CNN model (Figure 9). Also, it seems that after it gains some knowledge, it "forgets" it at some point during the training process (goes back to 0.5 accuracy, but it only happens certain times when the code is executed: Figure). Both problems might be caused by the fact that the architecture is quite simple and it has a single LSTM layer. It aso could be because RNNs were not designed for images (according to machinelearningmastery.com).

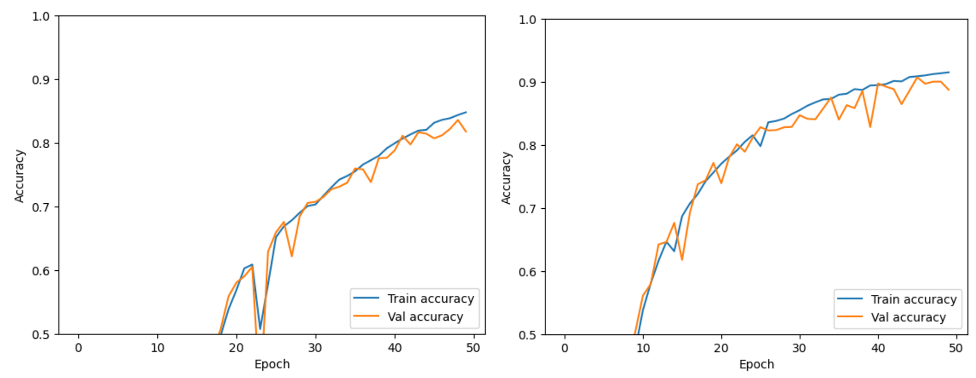


Figure 9: Two separate times when the same RNN architecture was trained

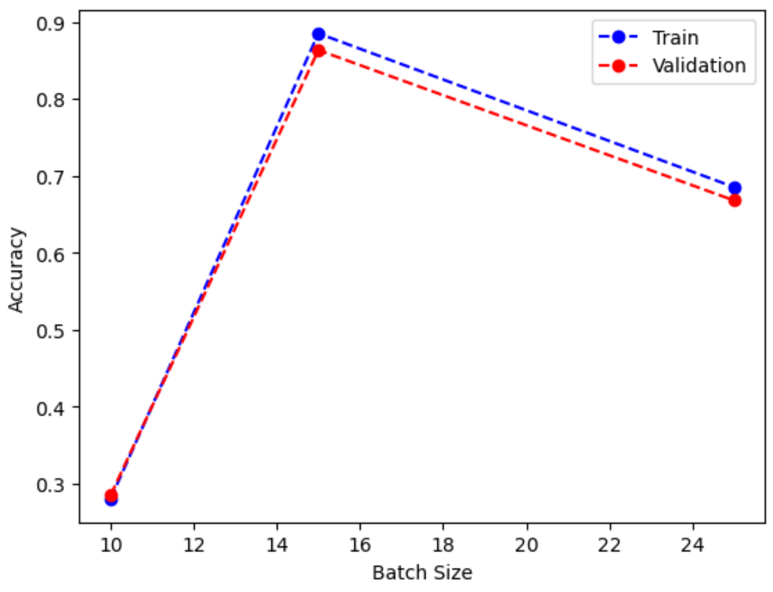


Figure 10: Picking the best batch size

It works best for a batch size of 15.

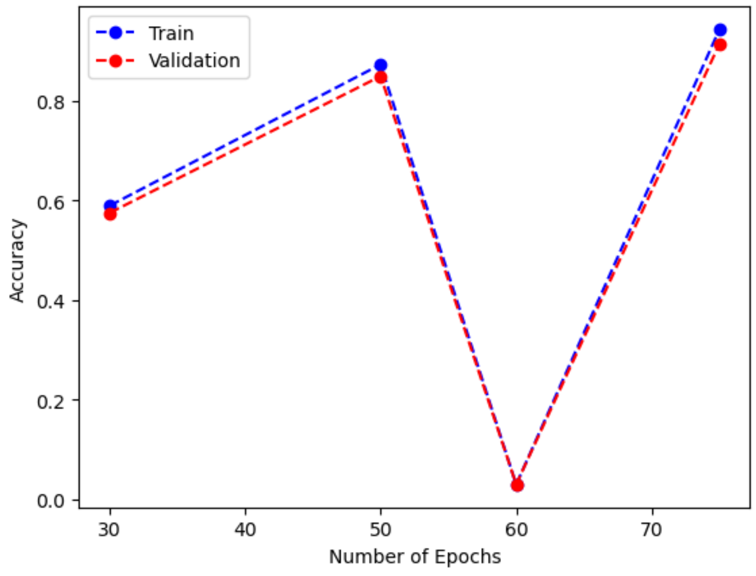


Figure 11: Picking the optimal number of epochs

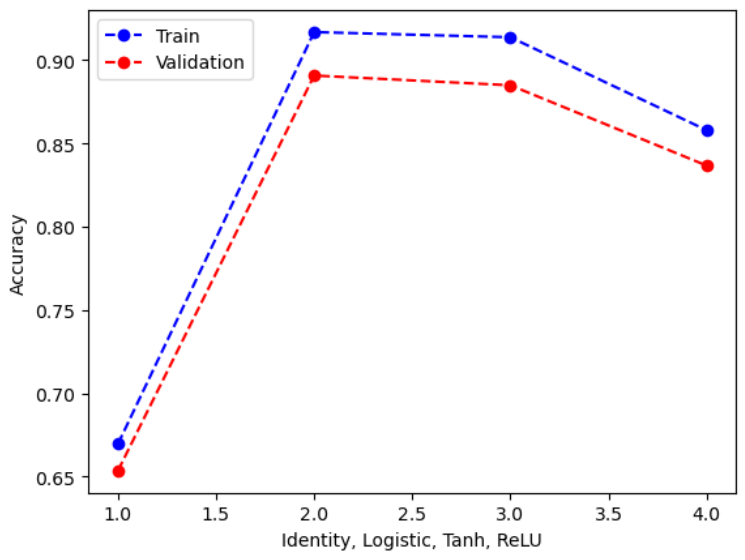


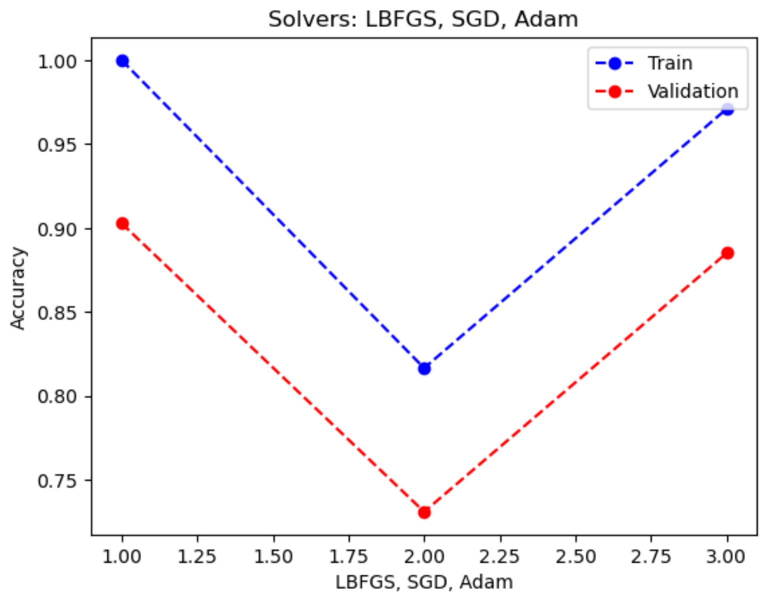
Figure 12: Picking the most optimal activation function

The Identity function is by far the worst contender. And there is a reason why:

This function makes no changes to the input (the input is equal to the output). In order for the model to be non-linear, there needs to be a mathematical operation applied to each layer's output.

Let's say we have 2 layers, and no activation function. Let V and W be the weights for each layer, and the input – A. At the end of layer 1, we get A*V, and at the end of layer 2 – A*V*W. If there is no function f, so that at the end of layer 2 – f(A*V*W), then the whole equation can be rewritten as A*B, where B = V*W, which means the model is linear and there is no point in multiple layers.

c) Multi-Layer Perceptron Classifier



.

Figure 13: Picking the best solver

LBFGS overpowers Adam by a bit.

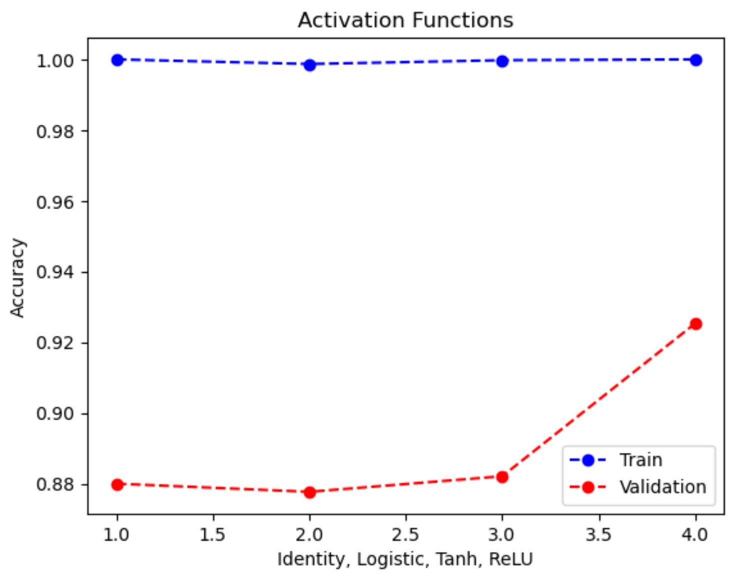


Figure 14: Picking the best activation function

ReLU works best for MLP.

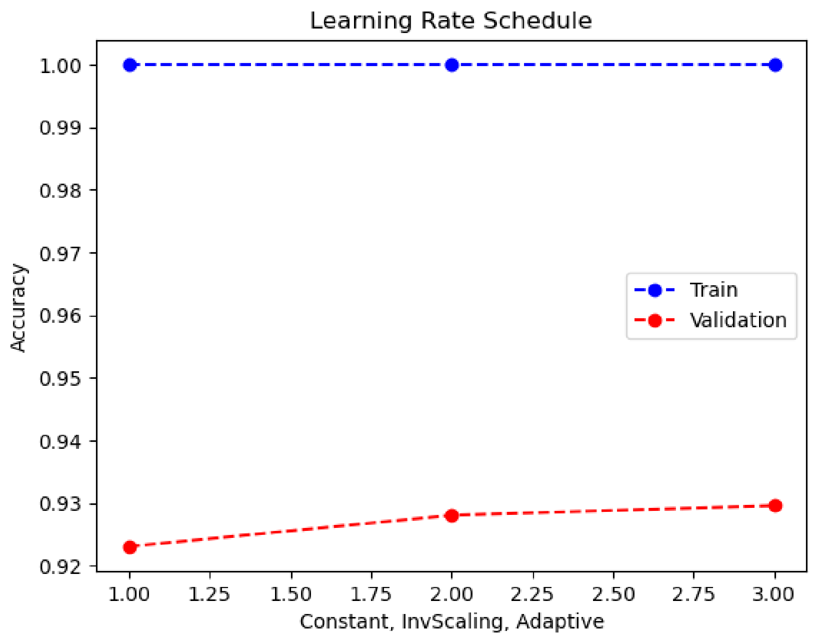


Figure 15: Picking the best learning rate schedule

The best learning rate schedule is “adaptive”, which means that the learning rate keeps decreasing as long as the training loss does not decrease by at least tol (tolerance).

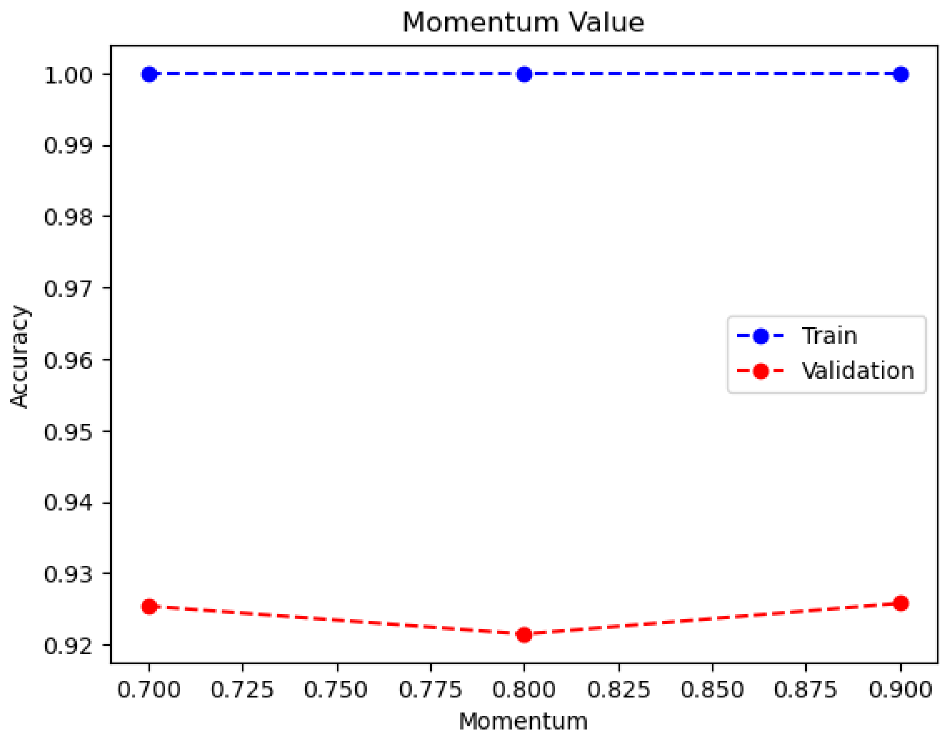


Figure 16: Trying different momentum values

The most optimal momentum value is 0.9.