ARTIN

Artificial Intelligence

Lab work
CNN

Student:
Raphael Blanchard
So Onishi

December 18, 2023

Processing

The following program is for vectorize the images.

```
train_x = train_x.reshape(train_x.shape[0], -1).T
test_x = test_x.reshape(test_x.shape[0], -1).T
```

The train X shape before program execution is (209, 64, 64, 3)
The test X shape before program execution is (50, 64, 64, 3)

The train X shape after program execution is (12288, 209)
The test X shape after program execution is (12288, 50)

The following program is for normalization.

```
train_x = train_x/255.
test_x = test_x/255.
```

The reason for dividing by 255 is that in general digital images, the color components of each pixel are expressed in the range of 0 to 255.
The reason for normalization is to ensure that the data points are of the same scale and to help the machine learning model learn more efficiently.

1 Classification with a single neuron

a Create three function to define the single neuron model

The following program is the sigmoid activation function.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

The following program is the initialize_parameters function.

```
def initialize_parameters(dim):
    w = np.random.randn(dim, 1) * 0.01
    b = 0
    return w, b
```

- dim: This argument represents the number of weights the network should have.
- w = np.random.randn(dim, 1) * 0.01: Initialize weight vector W for the number of dimensions specified by dim. np.random.randn(dim, 1) generates random values from a standard normal distribution (mean 0, standard deviation 1) to create a vector with dim x 1 shape.

- * 0.01: The reason for multiplying by 0.01 is to reduce the scale of the weight values

The following program computes the forward pass of a single neuron.

```
def neuron(w, b, x):
    pred_y = np.dot(w.T, x) + b
    return sigmoid(pred_y)
```

b Forward Pass

Use the three functions above to compute a first forward pass for the input matrix X containing the loaded dataset, for some initialization of the weights and bias.

$$Y_{\text{pred}} = \sigma(w^\top X + b) = [y^{(1)}_{\text{pred}}, y^{(2)}_{\text{pred}}, \ldots, y^{(m)}_{\text{pred}}] \tag{1}$$

The following program computes a first forward pass.

```
dim = train_x.shape[0]
w, b = initialize_parameters(dim)
pred_y = neuron(w, b, train_x)
```

dim = train_x.shape[0] means that used to create a number of weights equal to the number of features.

c Cost estimation

We will use a binary cross-entropy loss, so that the empirical risk can be computed as:

$$E = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(y^{(i)}_{\text{pred}}) + (1 - y^{(i)}) \log(1 - y^{(i)}_{\text{pred}}) \tag{2}$$

Binary cross-entropy measures the discrepancy between the probability predicted by the model and the actual label. It is used to evaluate how far the model's output is from the truthful label, and minimizing it can improve the model's performance.

The following program is the binary cross-entropy function.

```
def crossentropy(train_y, pred_y, epsilon = 1e-8):
    m = train_y.shape[1]
    cost = -1/m * np.sum(train_y * np.log(pred_y + epsilon)
                + (1 - train_y) * np.log(1 - pred_y + epsilon))
    cost = np.squeeze(cost) # change result into scaler
    return cost
```

d Back propagation

After initializing the parameters and doing a forward pass, we need to backpropagate the cost by computing the gradient with respect to the model parameters to later update the weights

$$\frac{\partial E}{\partial w} = \frac{1}{m} X(Y_{\text{pred}} - Y)^T = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}(y_{\text{pred}}^{(i)} - y^{(i)}) \tag{3}$$

$$\frac{\partial E}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (y_{\text{pred}}^{(i)} - y^{(i)}) \tag{4}$$

The following program is the backpropagate function.

```
def backpropagate(X, Y, Ypred):
    m = X.shape[1]

    delta_o = Ypred - Y

    dw = np.dot(X, delta_o.T) / m
    db = np.sum(delta_o) / m
    grads = {"dw": dw,
             "db": db}

    return grads
```

δ_o is equal to (Ypred - Y) because of the following formula
y is the actual label (0 or 1), $\hat{y}$ is the prediction of the model (output of the sigmoid function)

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z}$$

$$\frac{\partial E}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y})$$

$$\frac{\partial E}{\partial z} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y})$$

$$\frac{\partial E}{\partial z} = -y(1-\hat{y}) + (1-y)\hat{y} = \hat{y} - y$$

The above results show that the error δ_o in the output layer is the difference between the predicted and actual labels.

Calculation of weights and bias gradients:

- np.dot(X, delta_o.T: The sum of the errors for each feature is calculated.

- /m: It represents the number of samples, and the average slope is calculated by dividing the result of the matrix product by the number of samples.
- np.sum(delta_o): Compute the sum over all samples of the error δ_o in the output layer. Since the bias has a common value for each sample, the errors are summed.

e Optimization

The following program is the gradient_descent function that to update the parameters using gradient descent.

```
def gradient_descent(X, Y, iterations, learning_rate):
    costs = []
    w, b = initialize_parameters(X.shape[0])

    for i in range(iterations):
        Ypred = neuron(w, b, X)
        cost  = crossentropy(Y, Ypred)
        grads = backpropagate(X, Y, Ypred)

        #update parameters
        w -= learning_rate * grads["dw"]
        b -= learning_rate * grads["db"]
        costs.append(cost)

        if i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return w,b, costs
```

Use above program:

```
w, b, costs = gradient_descent(train_x,train_y,
                               iterations=2000,
                               learning_rate = 0.005)
```

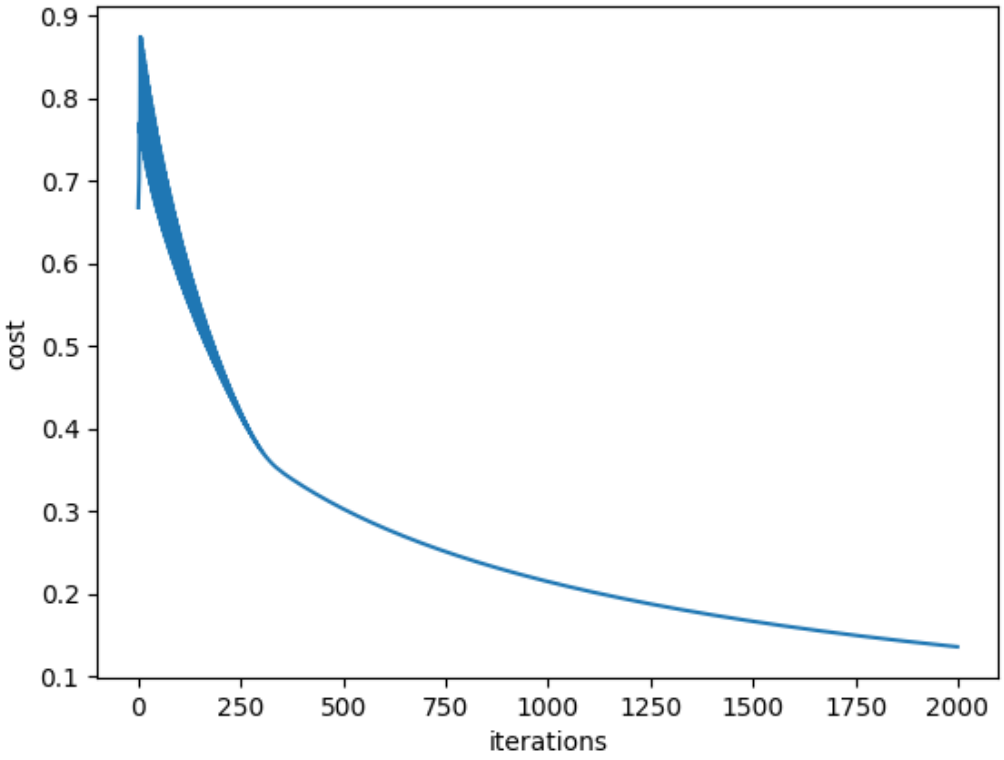
Results are below:

```
Cost after iteration 0: 0.667478
Cost after iteration 100: 0.581407
Cost after iteration 200: 0.463220
Cost after iteration 300: 0.373165
Cost after iteration 400: 0.330603
Cost after iteration 500: 0.302532
Cost after iteration 600: 0.279233
Cost after iteration 700: 0.259472
Cost after iteration 800: 0.242434
Cost after iteration 900: 0.227550
```

```
Cost after iteration 1000: 0.214410
Cost after iteration 1100: 0.202706
Cost after iteration 1200: 0.192204
Cost after iteration 1300: 0.182719
Cost after iteration 1400: 0.174108
Cost after iteration 1500: 0.166250
Cost after iteration 1600: 0.159050
Cost after iteration 1700: 0.152427
Cost after iteration 1800: 0.146314
Cost after iteration 1900: 0.140654
```

The above results show that the error between the prediction and the true value decreases as the number of iterations increases. This indicates that the implementation of backpropagation is successful. It can be seen that the change in error is larger in the beginning and hardly changes at all in the latter half of the period. An appropriate number of iterations should be selected to avoid over-training the model.

f Plot the training loss curve



(a) Training Loss Curve

Although many of the above considerations are covered by the previous discussion, let me summarize what can be seen from the above graphs:

- We can see a sharp decrease in cost in the first few iterations.

  - This indicates that the model is roughly learning at the beginning.

- As the number of iterations progresses, the rate of cost decrease slows gradually.

  - This indicates that parameter adjustments are becoming more subtle as the model approaches its optimal weights.

g Prediction

Use the optimized parameters to make predictions both for the train and test sets and compute the accuracy for each. What do you observe?
The results obtained using the weights and bias values obtained for the iteration number **2000** are as follows:

```
Train Acc: 88.10679712444778 %
Test Acc: 64.80977061116735 %
```

Results of when the iteration number is **1000**:

```
Train Acc: 82.11376242484754 %
Test Acc: 63.375881848659084 %
```

Results of when the iteration number is **3000**:

```
Train Acc: 91.08998211911403 %
Test Acc: 64.79063279485615 %
```

The above results show that increasing or decreasing the number of iterations only increases the percentage of correct responses to the train data, but not the percentage of correct responses to the test data.
From the above, we can see that the upper limit of classification performance with a single neuron on this dataset is around 64%.

h Choose learning rate on a validation set

The following program is that to make 10 percent of training data into validation data.

```
val_size = int(0.1 * train_x.shape[1])

train_x, val_x = train_x[:, :-val_size], train_x[:, -val_size:]
train_y, val_y = train_y[:, :-val_size], train_y[:, -val_size:]
```

The following program is that to calculate optimal learning rate.

```
lr_list = [0.0001, 0.001, 0.005, 0.01, 0.05, 0.1]

best_acc = 0

for lr in lr_list:
    w, b, costs = gradient_descent(val_x,val_y,iterations=2000,
                                                learning_rate = lr)

    val_pred_y = predict(w, b, val_x)
    print(f"W: {w}")
    print(f"b: {b}")
    print(f"Pred value: {val_pred_y}")

    accuracy = cal_acc(val_pred_y, val_y)
    print(f"Val Acc: {accuracy} with lr = {lr}")

    if best_acc < accuracy:
        best_acc = accuracy
        best_lr = lr
```

Result above:

```
Validation Accuracy: 99.99360922914458 with lr = 0.1
```

Results of using the above learning rates for training and test data.

```
Train Accuracy: 99.91717492378271
Test Accuracy: 70.76550256943786
```

Accuracy score is improved in both training and test data.

2 Introduction to Convolutional Neural Networks

a CNNs with Keras and TensorFlow

To adapt the example on the Keras website we first have to change the input shape as the example is for MNIST data which consists of $28x28$ images with only 1 channel as it is gray scale. However, we are here dealing with RGB (3 channels) images of size $64x64$, hence why we get an input shape of $(64, 64, 3)$.
The scaling of images is easily done for images by simply dividing each channel by 255 (maximum value for any RGB channel).
We later on convert our ground truth to One-Hot ground truth to make them usable with our networks.

a.1 Number of parameters

Using the .summary() function of keras, we can see that the total number of parameter is equal to 44482, with a description of where these weights are coming in the table (see Fig (2)). This number is coming from calculating how many weights and biases are learned in the network in the Conv layers and the FC layer at the end.

Calculate total parameters by hands

1. Conv2D: the number of parameters for the fist convolution layer can be calculated by this equation '(filter height * filter width * number of input channels + bias) * number of filters'. In this case:

$$(3 * 3 * 3 + 1) * 32 = 896$$

2. MaxPooling2D: It does not have any parameters.

3. Conv2D_1: Using the previous equation, in this case:

$$(3 * 3 * 32 + 1) * 64 = 18496$$

4. Flatten: It does not have any parameters.

5. Dropout: It does not have any parameters.

6. Dence: The number of parameter for this layer can calculated by this equation '(number of input features * number of unit(number of neuron) + number of unit)'. In this case:

$$12544 * 2 + 2 = 25090$$

7. Total Parameters: Adding up all the parameters obtained above:

$$896 + 18496 + 25090 = 44482$$

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 32)        896

 max_pooling2d (MaxPooling2D  (None, 31, 31, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 29, 29, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 flatten (Flatten)           (None, 12544)             0

 dropout (Dropout)           (None, 12544)             0

 dense (Dense)               (None, 2)                 25090

=================================================================
Total params: 44,482
Trainable params: 44,482
Non-trainable params: 0
_________________________________________________________________
```

Fig (2) Network 1 Summary table

b Comparing multiple networks

We decided to work on 4 different networks, to increment the complexity of the network bit by bit while explaining each pros obtained from these amelioration.

b.1 Baseline network

Firstly, we train the given network, and obtain reasonable training accuracy (0.86). However, the simplicity of the network makes it overfit the data the more it gets trained, which results in a slightly lower accuracy for the test set (0.72).
As we can see on Fig (3) and (4), training it for too many epochs (here, 200), we get interesting numbers for the training set with an accuracy of 0.97 and a loss of 0.16. However, we can see that it is not as great (maybe even worse) for the test set as we get a better accuracy (from 0.72 to 0.82) BUT we get a higher loss, meaning the network is less sure about it's decisions.

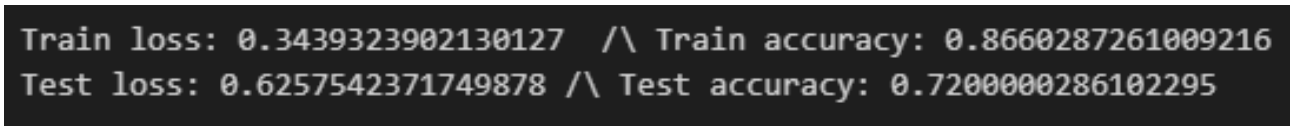


Fig (3) Baseline Network - 15 epochs

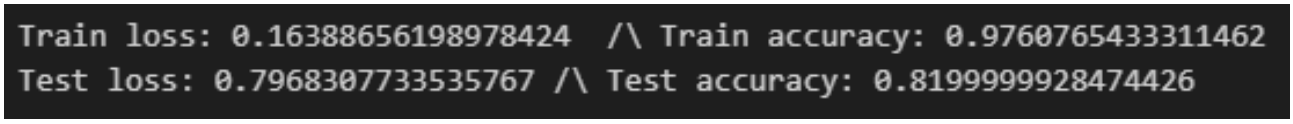


Fig (4) Baseline Network - 200 epochs

b.2 Tweaked Baseline network

We decided here to simply add 2 dropout layers (0.25) which is a regularization technique that allows a better generalization from the network. As we can see from Fig (5), training this network for 50 epochs drastically improves the general performance of the model by obtaining a much lower difference between the training and testing accuracies/losses.

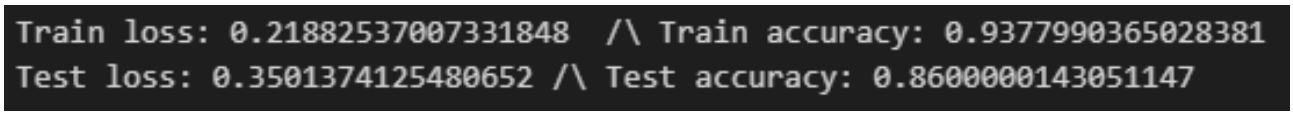


Fig (5) Tweaked baseline network - 50 epochs

b.3 Network 2: More Conv layers

We were then interested by adding more Conv layers using a general good practice which is to us 2 Conv blocks with the first block of input size X, and the second block of size $2 * X$. We decided to train this network for 30 epochs, which took much longer than the previous 2 networks as it has many more parameters (5.8 millions). However, the additional computation time was well worth it when looking at the obtained results in Fig (6). We can see that we get very good training accuracy while still maintaining a good testing accuracy (which is higher than training acc here). We also get a testing loss that is acceptable for such a high accuracy.

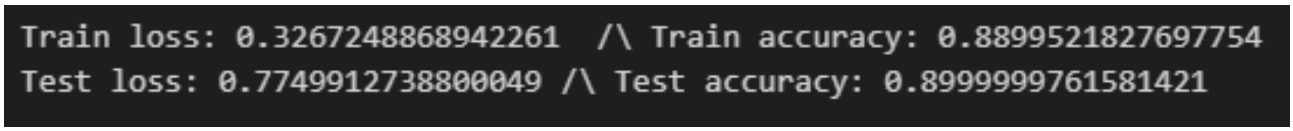


Fig (6) Network 2 - 30 epochs

b.4 Network 3: ResNet inspired

Lastly, we decided to inspire ourselves from the known ResNet network, specifically by implementing a resnet block, which consists of multiple Conv layers, Batch norm layers which are good for regularization. We decided to not use the Batch Norm layers because of lower performance. However, the most interesting part of the resnet block is it's skip connection, corresponding to the $+x$ in $F(x)+x$ in Fig (7), which allows the network to reduce the vanishing gradient problem in deep networks by skipping some Conv layers (that make the gradient small).

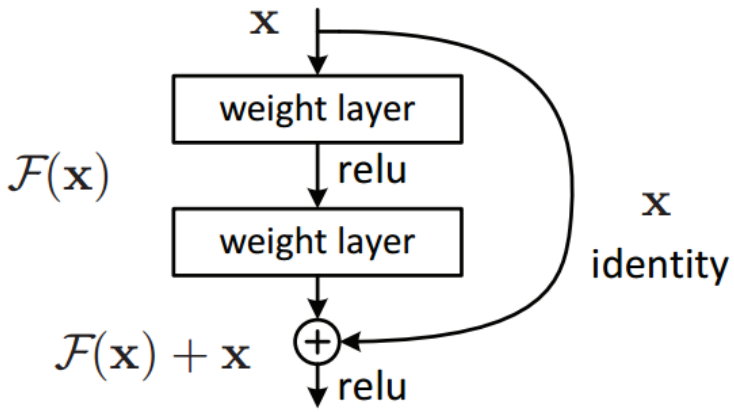


Fig (7) ResNet block

For the training of this model since it is struggling much more to converge, we firstly trained it using our own learning rate (0.01 instead of 0.001 by default) for 30 epochs and then switched to a learning rate equal to 0.001 for 10 epochs to fine tune our model. This resulted in a much better result than directly training it and we were able to obtain slightly lower accuracies (than Network 2) which could be explained by the fact that we don't have enough data at all for a deeper network like a ResNet. However, its robustness allowed it to get a lower test loss than our previous best model. See Fig (8) under for the exact results.

```
Train loss: 0.5189478397369385  /\ Train accuracy: 0.7464115023612976
Test loss: 0.5789552330970764 /\ Test accuracy: 0.8399999737739563
```

Fig (8) Network 3 - 30 + 10 epochs

b.5 Conclusion

On Fig (9) we can see the ROC curves and their corresponding AUC. It might be surprising that the most performant (when looking at ROC curves and AUC) model is the most "simplest" one, but it is not. In fact, the main reason why this happens is that the 3 other models we made are too deep and too general for the simple task that is binary classification with only 209 images in the dataset. Hence why here the simpler network is able to capture the underlying patterns in the data (giving good performance) while still not overfitting too much compared to the 3 other models. It would be interesting to run the same tests but this time on a bigger dataset, which would show the robustness of the deeper networks.

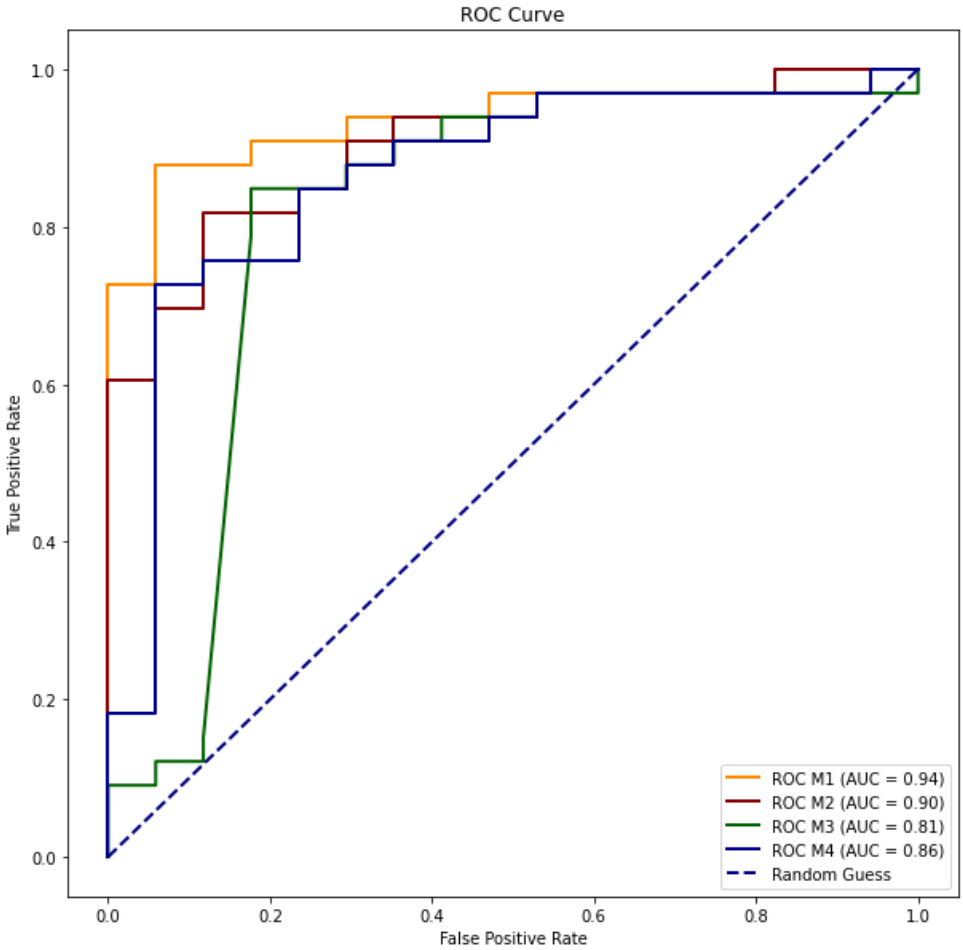


Fig (9) ROC curves and corresponding AUC

c Custom training loop

We defined our own custom training loop by transforming our dataset into tf.data.Dataset objects to make them usable. Later on we defined our own optimizer (Adam) and losses that we will use during the training. Then, before starting the training loop, we need to overwrite the tf function called train_step and test_step. It is in the train_step function that the GradientTape is needed as it is how we will update our weights and biases (gradients). After all this is done, we just need to loop for a certain amount of epochs and call these 2 functions. We also pretty print the results obtained while training like TensorFlow is doing in the fit function, and obtain Fig (10).

```
Epoch 47, Loss: 0.032540742307901 38, Accuracy: 99.52153015136719, Test Loss: 0.6644739508628845, Test Accuracy: 88.0
Epoch 48, Loss: 0.02497062273323536, Accuracy: 100.0, Test Loss: 0.6371155977249146, Test Accuracy: 88.0
Epoch 49, Loss: 0.023703621700406075, Accuracy: 100.0, Test Loss: 0.480729877948761, Test Accuracy: 94.0
Epoch 50, Loss: 0.01915730908513069, Accuracy: 100.0, Test Loss: 0.8715122938156128, Test Accuracy: 80.0
```

Fig (10) Pretty print own training loop

d Additional bonus

We also decided to implement early stopping and the tensorboard callbacks to get more information on our models. However, the data is hardly readable from tensorboard as the accuracy of our models go up very quickly, making the whole plot having a small range and

very fluctuating values (see Fig (11)). Using the early stopping callback worked well and stopped the training of the model after seeing X time a validation loss (the metric we used to early stop).

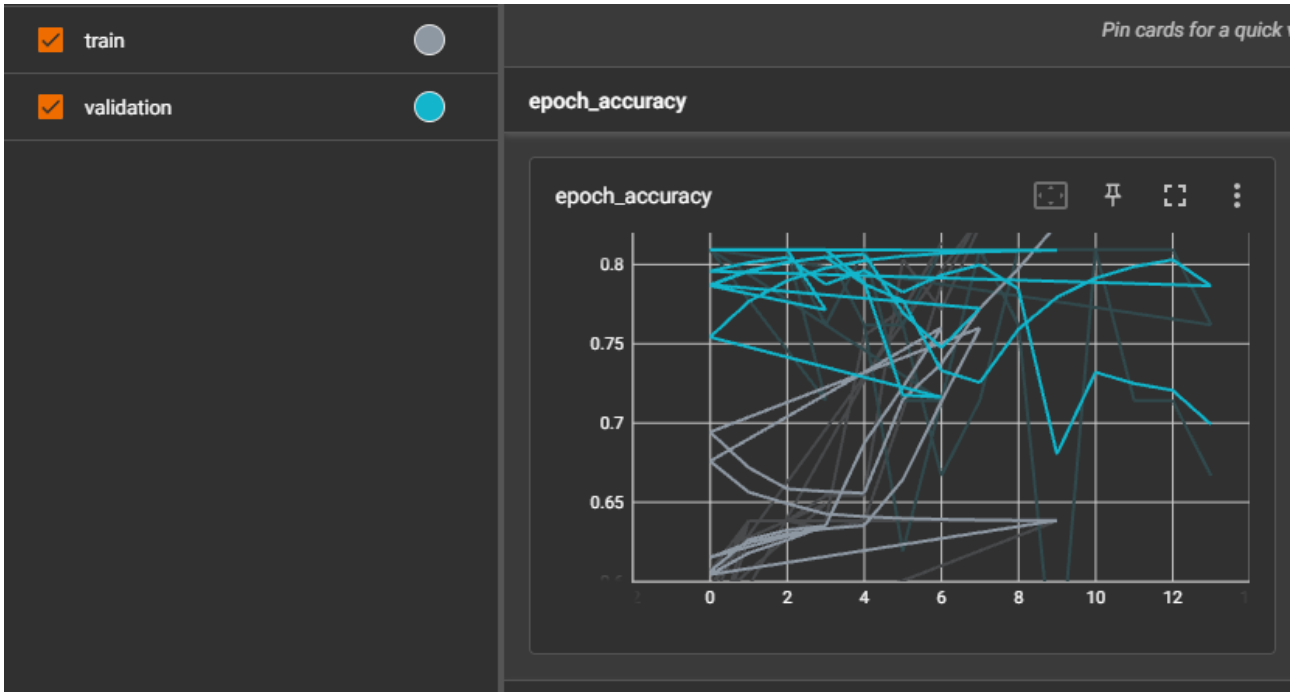


Fig (11) TensorBoard