A.

Deep Learning (CP70071E_05-FEB-24_31-MAY-24_A_SEM2)

Topic – Log Book

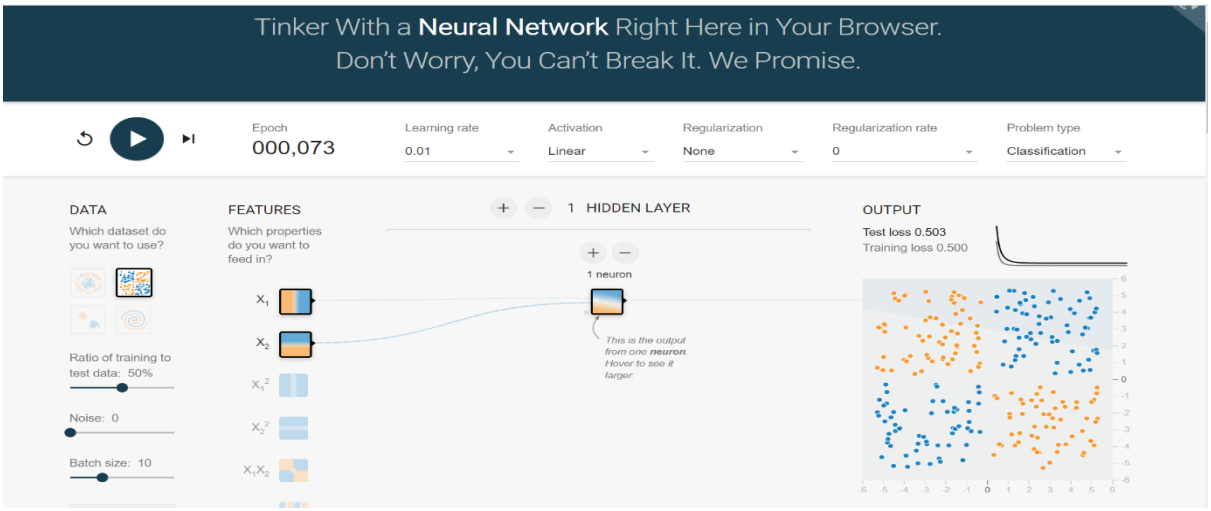
Student ID: [21600713]

Full Name: [Mohammed Haris Shaikh]

II. Table of Contents

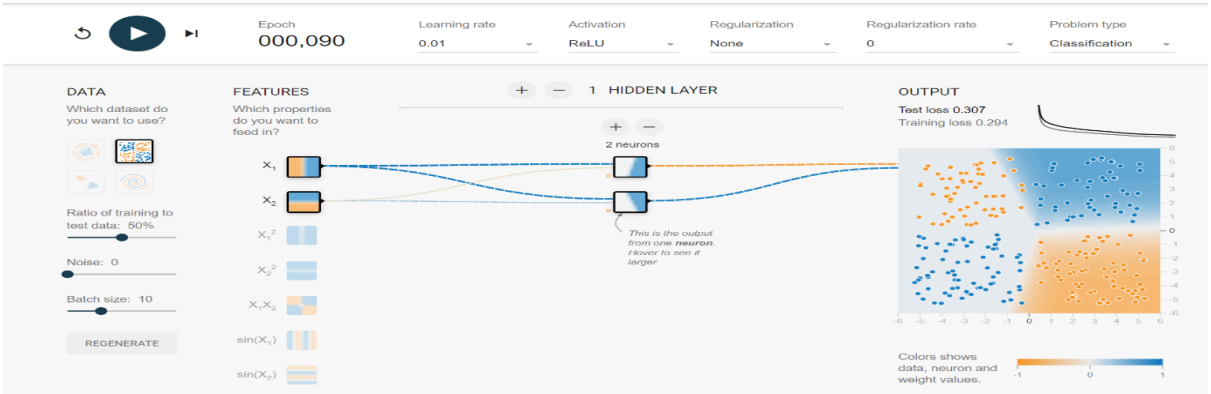
III. WEEK 1

A. Task 1: The model as given combines our two input features into a single neuron. Will this model learn any non-linearities? Run it to confirm your guess.



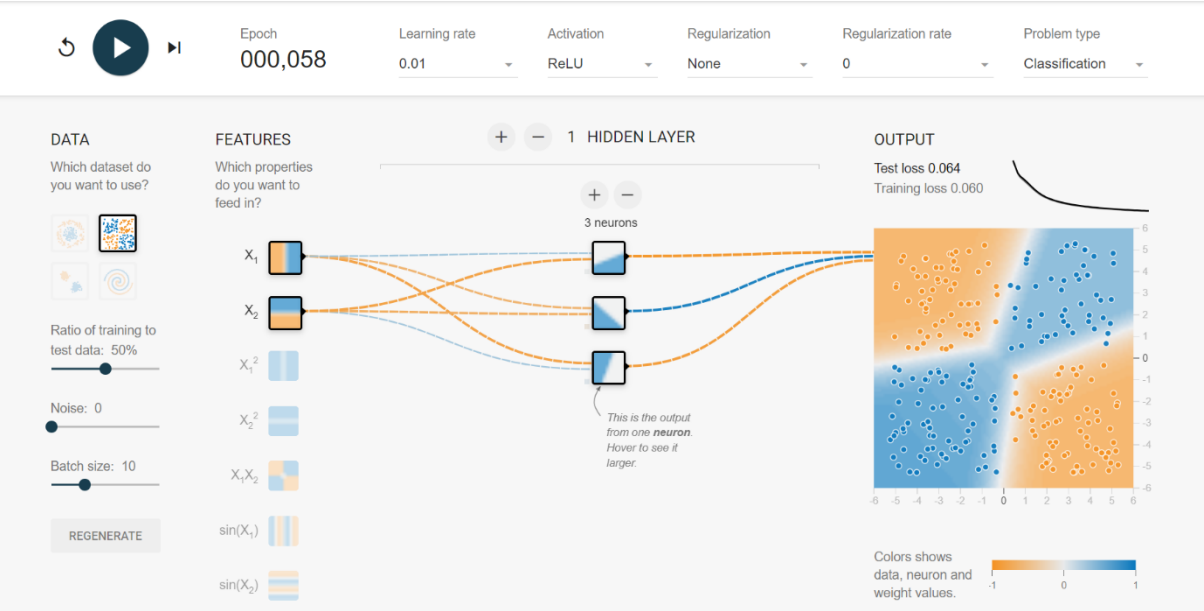
Ans-This model with a single neuron and a linear activation function will not be able to learn any non-linearities between the two input features.

B. Task 2: Try increasing the number of neurons in the hidden layer from 1 to 2, and also try changing from a Linear activation to a nonlinear activation like ReLU. Can you create a model that can learn nonlinearities? Can it model the



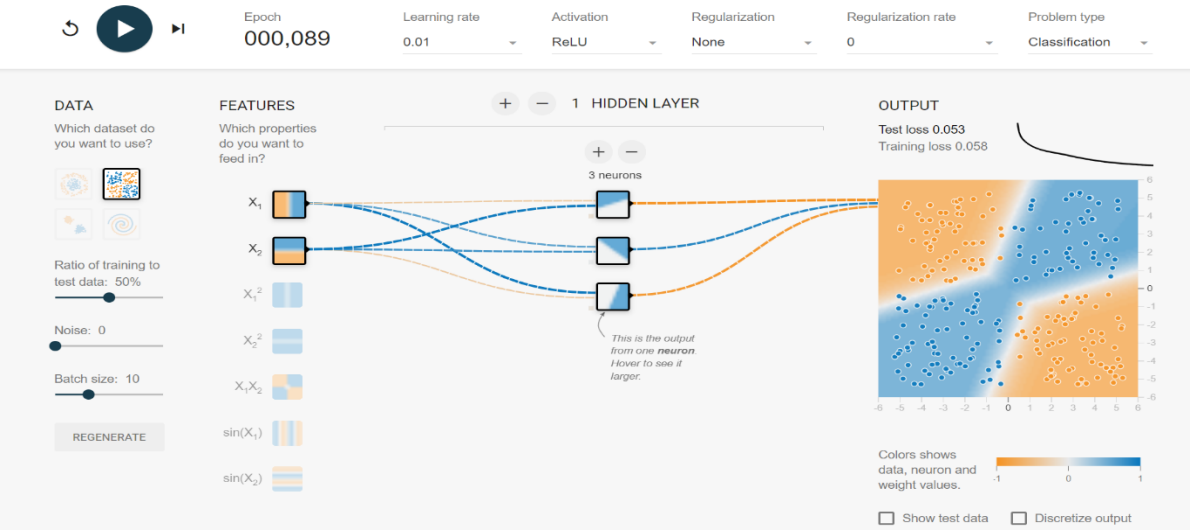
Ans- Increasing the number of neurons in the hidden layer and changing the activation function from Linear to ReLU can indeed help a neural network learn nonlinearities and model the data more effectively. However above model with only one neuron added will not have any effect still model cannot predict data efficiently .

C. Task 3: Try increasing the number of neurons in the hidden layer from 2 to 3, using a nonlinear activation like ReLU. Can it model the data effectively? How model quality vary from run to run?



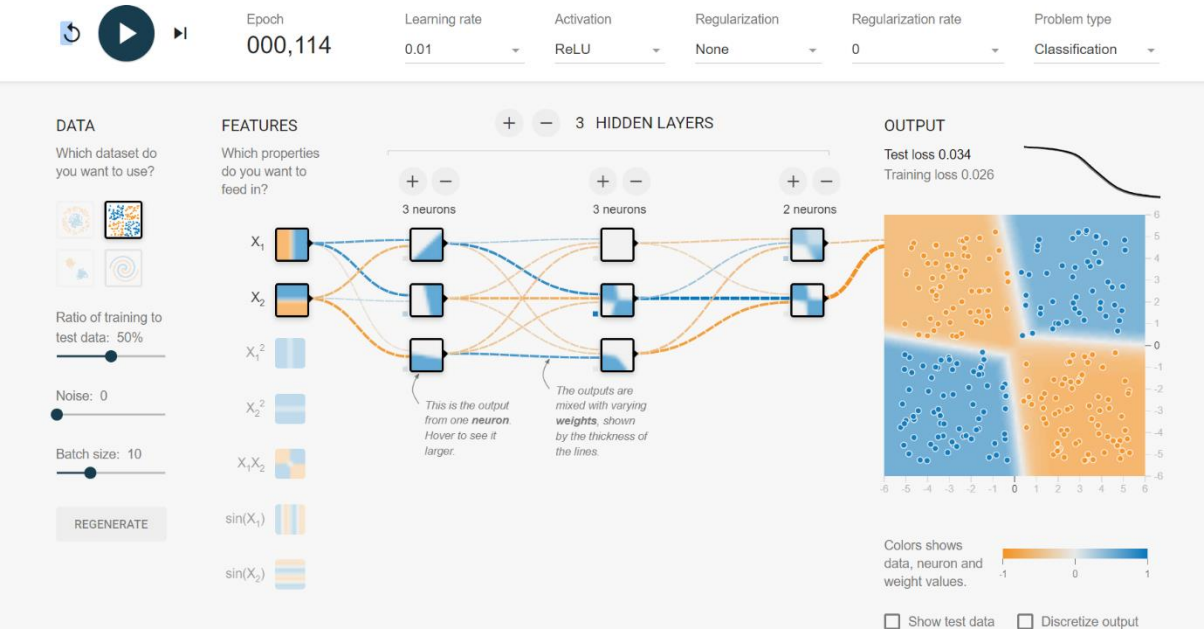
Ans-Above model with three neurons and using relu it predicted data effectively, unfortunately if we run again and again data effectiveness decreases sometime it perfect sometimes having test loss 0.25.

D. Task 4: Continue experimenting by adding or removing hidden layers and neurons per layer. Also feel free to change learning rates, regularization, and other learning settings. What is the smallest number of neurons and layers you can use that gives test loss of 0.177 or lower?



Ans- I played with parameter and explored models couple times ,However above model with 1 hidden layer and 3 neurons using relu and learning rate - 0.01 I get model with less than 0.177 test loss.

E. Task 5: Does increasing the model size improve the fit, or how quickly it converges? Does this change how often it converges to a good model? For example, try the following architecture: • First hidden layer with 3 neurons. • Second hidden layer with 3 neurons. • Third hidden layer with 2 neurons



Ans- Definitely with increasing hidden layers and neuron effects data quality and help to predict non-linearity but convergence to global minima required more epochs in my case, Sometime it get overfit the model, However while running model again and again more probably get good results as compare to low hidden layers and neurons model.

IV. WEEK 2

A. Resource: With assumption that you have access to Microsoft Excel, create a new template with the following setup. From there, you start implement the perceptron binary classification for the AND function approximation based on the formulas discussed in the lecture.

Epoch	X		Y	W	b	L_Rate	Z	p	-YLog(P)	-(1-Y)Log(1-P)	Cost(Y, P)	dC/dP	dP/dZ	dC/dZ	dZ/db	dC/dW	dC/db	'W'(updated web'(updated bias for next epocph
0	0	0	0	10	10	1	10	1.000	0.000	10.000	12.50001	5506.866	0.000	0.250	1.000	0.250	0.750	9.750
	0	1	0	10			20	1.000	0.000	20.000		121291294.749	0.000	0.250		0.250		9.750
	1	0	0				20	1.000	0.000	20.000		121291294.749	0.000	0.250				
	1	1	1				30	1.000	0.000	0.000		-0.250	0.000	0.000				
1	0	0	0	9.750	9.250	1	9.250011	1.000	0.000	9.250	11.81253	2601.421	0.000	0.250	1.000	0.250	0.750	9.500
	0	1	0	9.750			19.00001	1.000	0.000	19.000		44621080.906	0.000	0.250		0.250		9.500
	1	0	0				19.00001	1.000	0.000	19.000		44621080.906	0.000	0.250				
	1	1	1				28.75001	1.000	0.000	0.000		-0.250	0.000	0.000				
2	0	0	0	9.500	8.500	1	8.500035	1.000	0.000	8.500	11.12508	1228.986	0.000	0.250	1.000	0.250	0.750	9.250
	0	1	0	9.500			18.00004	1.000	0.000	18.000		16415573.402	0.000	0.250		0.250		9.250
	1	0	0				18.00004	1.000	0.000	18.000		16415573.402	0.000	0.250				
	1	1	1				27.50004	1.000	0.000	0.000		-0.250	0.000	0.000				
3	0	0	0	9.250	7.750	1	7.750086	1.000	0.000	7.751	10.43767	580.693	0.000	0.250	1.000	0.250	0.750	9.000
	0	1	0	9.250			17.00009	1.000	0.000	17.000		6039259.275	0.000	0.250		0.250		9.000
	1	0	0				17.00009	1.000	0.000	17.000		6039259.275	0.000	0.250				
	1	1	1				26.25009	1.000	0.000	0.000		-0.250	0.000	0.000				
4	0	0	0	9.000	7.000	1	7.000194	0.999	0.000	7.001	9.75037	274.461	0.001	0.250	1.000	0.250	0.750	8.750
	0	1	0	9.000			16.00019	1.000	0.000	16.000		2221958.695	0.000	0.250		0.250		8.750
	1	0	0				16.00019	1.000	0.000	16.000		2221958.695	0.000	0.250				
	1	1	1				25.00019	1.000	0.000	0.000		-0.250	0.000	0.000				
5	0	0	0	8.750	6.250	1	6.250422	0.998	0.000	6.252	9.06330	129.808	0.002	0.250	1.000	0.250	0.750	8.500
	0	1	0	8.750			15.00042	1.000	0.000	15.000		817599.311	0.000	0.250		0.250		8.500
	1	0	0				15.00042	1.000	0.000	15.000		817599.311	0.000	0.250				
	1	1	1				23.75042	1.000	0.000	0.000		-0.250	0.000	0.000				
6	0	0	0	8.500	5.501	1	5.500903	0.996	0.000	5.505	8.37670	61.478	0.004	0.249	1.000	0.250	0.749	8.250
	0	1	0	8.500			14.0009	1.000	0.000	14.001		300923.058	0.000	0.250		0.250		8.250
	1	0	0				14.0009	1.000	0.000	14.001		300923.058	0.000	0.250				
	1	1	1				22.5009	1.000	0.000	0.000		-0.250	0.000	0.000				
7	0	0	0	8.250	4.752	1	4.75192	0.991	0.000	4.761	7.69109	29.202	0.008	0.248	1.000	0.250	0.748	8.000
	0	1	0	8.250			13.00192	1.000	0.000	13.002		110816.234	0.000	0.250		0.250		8.000
	1	0	0				13.00192	1.000	0.000	13.002		110816.234	0.000	0.250				
	1	1	1				21.25192	1.000	0.000	0.000		-0.250	0.000	0.000				
	0	0	0	8.000	4.004	1	4.004062	0.982	0.000	4.022	7.00757	13.955	0.018	0.246	1.000	0.250	0.746	7.750
	0	1	0	8.000			12.00406	1.000	0.000	12.004		40854.588	0.000	0.250		0.250		7.750

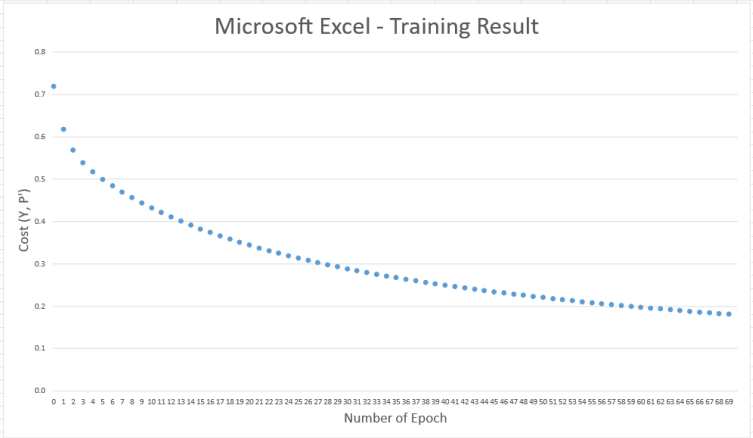
Ans- As per given information all details filled and applied formula for finding cost function and back propagation(updated weight and bias for next epoch)

You will find excel file here

B. Expected outcome: You are required to plot the Cost function value at every epoch, up to 70 epochs. Your plot should be inline with the Figure 3.

0.731	7.250	1.786
	7.250	
0.714	7.000	1.072
	7.000	
0.686	6.750	0.386
	6.750	
0.648	6.500	-0.262
	6.500	
0.608	6.251	-0.870
	6.251	
0.572	6.002	-1.441
	6.002	
0.543	5.755	-1.984
	5.755	
0.519	5.510	-2.503
	5.510	
0.495	5.272	-2.998
	5.272	
0.465	5.045	-3.463
	5.045	

Expected outcome



A

e

C. Investigate if the Cost function can reach less than 0.09. If it is possible, by how many epoch the Cost function reaches less than 0.09?

	1	1	1				2.066218	0.888	0.119	0.000		-0.282	0.100	-0.028					
36	0	0	0	3.954	-5.890	1	-5.889618	0.003	0.000	0.003	0.09929	0.251	0.003	0.001	1.000	0.002	0.034	3.952	-5.924
	0	1	0	3.954			-1.93537	0.126	0.000	0.135		0.286	0.110	0.032		0.002		3.952	
	1	0	0				-1.93537	0.126	0.000	0.135		0.286	0.110	0.032					
	1	1	1				2.018877	0.883	0.125	0.000		-0.283	0.103	-0.029					
37	0	0	0	3.952	-5.924	1	-5.924078	0.003	0.000	0.003	0.09815	0.251	0.003	0.001	1.000	0.000	0.031	3.952	-5.955
	0	1	0	3.952			-1.972061	0.122	0.000	0.130		0.285	0.107	0.031		0.000		3.952	
	1	0	0				-1.972061	0.122	0.000	0.130		0.285	0.107	0.031					
	1	1	1				1.979956	0.879	0.129	0.000		-0.285	0.107	-0.030					
38	0	0	0	3.952	-5.955	1	-5.955498	0.003	0.000	0.003	0.09721	0.251	0.003	0.001	1.000	-0.001	0.029	3.953	-5.984
	0	1	0	3.952			-2.003692	0.119	0.000	0.126		0.284	0.105	0.030		-0.001		3.953	
	1	0	0				-2.003692	0.119	0.000	0.126		0.284	0.105	0.030					
	1	1	1				1.948114	0.875	0.133	0.000		-0.286	0.109	-0.031					
39	0	0	0	3.953	-5.984	1	-5.984362	0.003	0.000	0.003	0.09640	0.251	0.003	0.001	1.000	-0.003	0.027	3.956	-6.011
	0	1	0	3.953			-2.03107	0.116	0.000	0.123		0.283	0.103	0.029		-0.003		3.956	
	1	0	0				-2.03107	0.116	0.000	0.123		0.283	0.103	0.029					
	1	1	1				1.922222	0.872	0.137	0.000		-0.287	0.111	-0.032					
40	0	0	0	3.956	-6.011	1	-6.011076	0.002	0.000	0.002	0.09569	0.251	0.002	0.001	1.000	-0.004	0.025	3.960	-6.036
	0	1	0	3.956			-2.054875	0.114	0.000	0.121		0.282	0.101	0.028		-0.004		3.960	
	1	0	0				-2.054875	0.114	0.000	0.121		0.282	0.101	0.028					
	1	1	1				1.901325	0.870	0.139	0.000		-0.287	0.113	-0.032					
41	0	0	0	3.960	-6.036	1	-6.035978	0.002	0.000	0.002	0.09505	0.251	0.002	0.001	1.000	-0.005	0.023	3.965	-6.059
	0	1	0	3.960			-2.075678	0.111	0.000	0.118		0.281	0.099	0.028		-0.005		3.965	
	1	0	0				-2.075678	0.111	0.000	0.118		0.281	0.099	0.028					
	1	1	1				1.884622	0.868	0.141	0.000		-0.288	0.114	-0.033					
42	0	0	0	3.965	-6.059	1	-6.059351	0.002	0.000	0.002	0.09446	0.251	0.002	0.001	1.000	-0.006	0.022	3.971	-6.081
	0	1	0	3.965			-2.093957	0.110	0.000	0.116		0.281	0.098	0.027		-0.006		3.971	
	1	0	0				-2.093957	0.110	0.000	0.116		0.281	0.098	0.027					
	1	1	1				1.871436	0.867	0.143	0.000		-0.288	0.116	-0.033					
43	0	0	0	3.971	-6.081	1	-6.081433	0.002	0.000	0.002	0.09391	0.251	0.002	0.001	1.000	-0.007	0.021	3.978	-6.102
	0	1	0	3.971			-2.110116	0.108	0.000	0.114		0.280	0.096	0.027		-0.007		3.978	
	1	0	0				-2.110116	0.108	0.000	0.114		0.280	0.096	0.027					
	1	1	1				1.8612	0.865	0.145	0.000		-0.289	0.116	-0.034					
	0	0	0	3.978	-6.102	1	-6.102421	0.002	0.000	0.002	0.09339	0.251	0.002	0.001	1.000	-0.007	0.020	3.985	-6.122

Ans- As we can see after 37 epochs cost function reaches 0.099 and still decreasing after every epochs, in my opnion if it still continue this model gonna be overfit which is not at all good .

V. WEEK 3

A. 1. Please use the followimg link for Hands-on session:

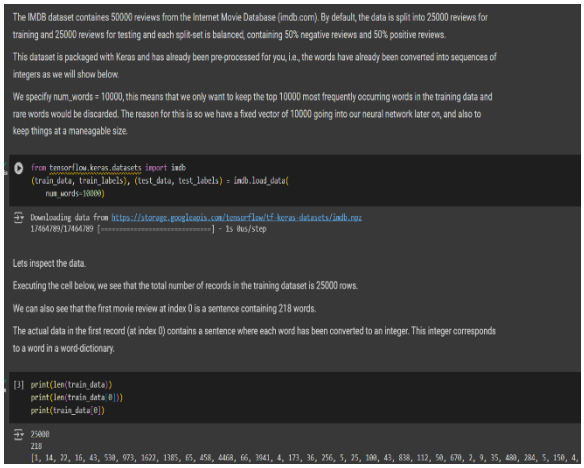
https://colab.research.google.com/drive/1wuJFhZ0DGvAu1YjWCImFPkc6I3FWdLwq?usp=sharing

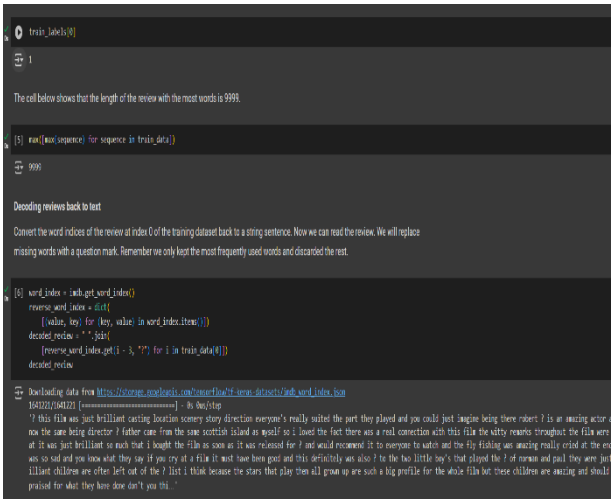
Binary Sentiment Classification of IMDB Reviews

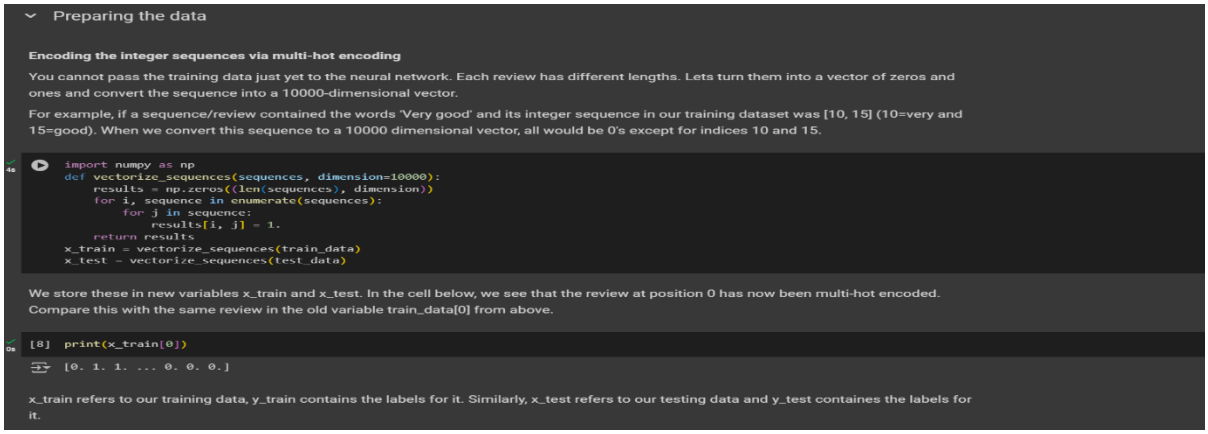
This project investigates sentiment analysis for movie reviews using a deep learning model. The objective is to classify reviews as positive or negative.

Data and Preprocessing:

- **Dataset:** The IMDB movie review dataset was used for this binary classification task.
- **Preparation:** A subset of 25,000 reviews was extracted and preprocessed for the model. This likely involved steps like text cleaning, tokenization, and vectorization to convert text data into a format suitable for machine learning algorithms.

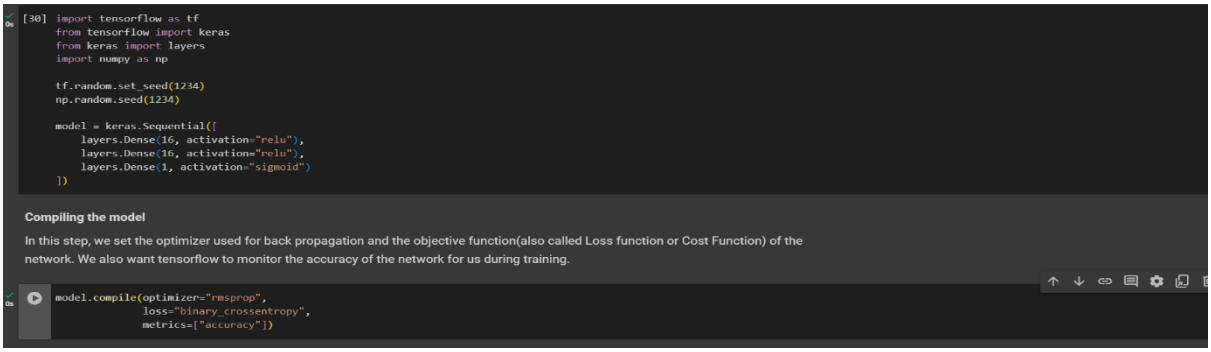


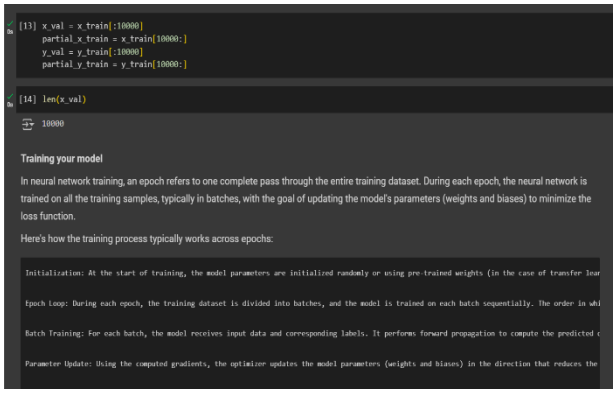


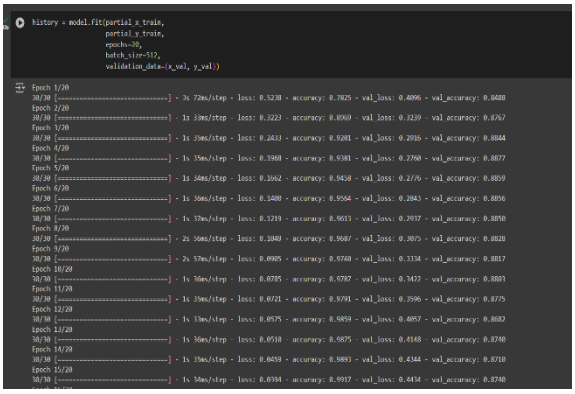


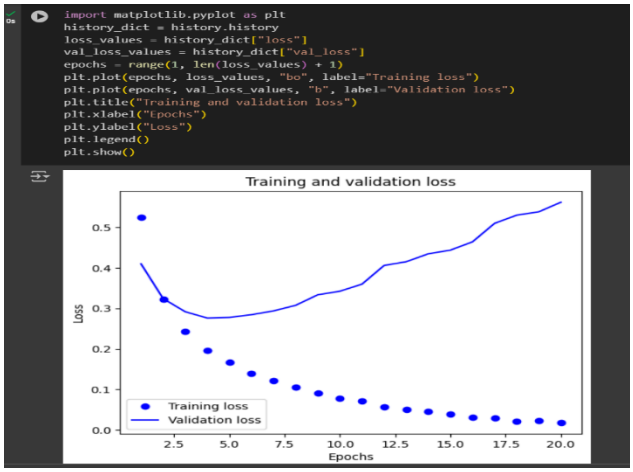
Model Building and Overfitting:

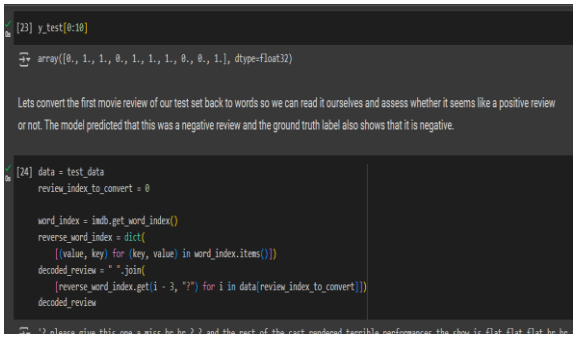
- **Architecture (Initial):** A deep learning model with three dense layers was constructed:
    - First Layer: 16 neurons
    - Second Layer: 16 neurons
    - Third Layer: 1 neuron with sigmoid activation (common for binary classification)
- **Training:** The model was trained for 20 epochs using separate training and validation sets (x_train, Y_train and x_val, Y_val).
- **Performance:** While the model achieved a validation accuracy of 0.8671, it exhibited signs of overfitting after only 4 epochs. Overfitting occurs when the model memorizes training data specifics rather than learning generalizable features for accurate classification on unseen data.





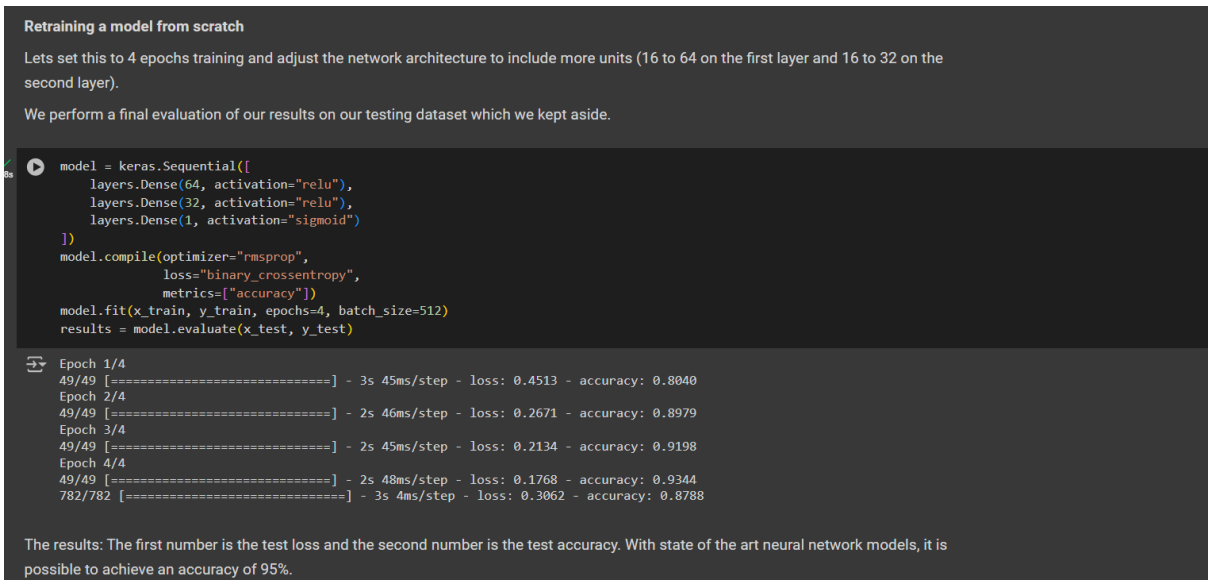






Model Improvement and Regularization:

- **Architecture (Revised):** To address overfitting, a new model was constructed using a different architecture:
    - First Layer: 64 neurons
    - Second Layer: 32 neurons
    - Third Layer: 1 neuron with sigmoid activation
- **Reduced Training:** This revised model was trained for a shorter duration (4 epochs) to limit overfitting.
- **Performance:** The reduced training with the new architecture resulted in a slightly higher validation accuracy (0.8788) compared to the initial model trained for 20 epochs. This suggests that the revised architecture with fewer neurons and shorter training helped mitigate overfitting.



Retraining a model from scratch

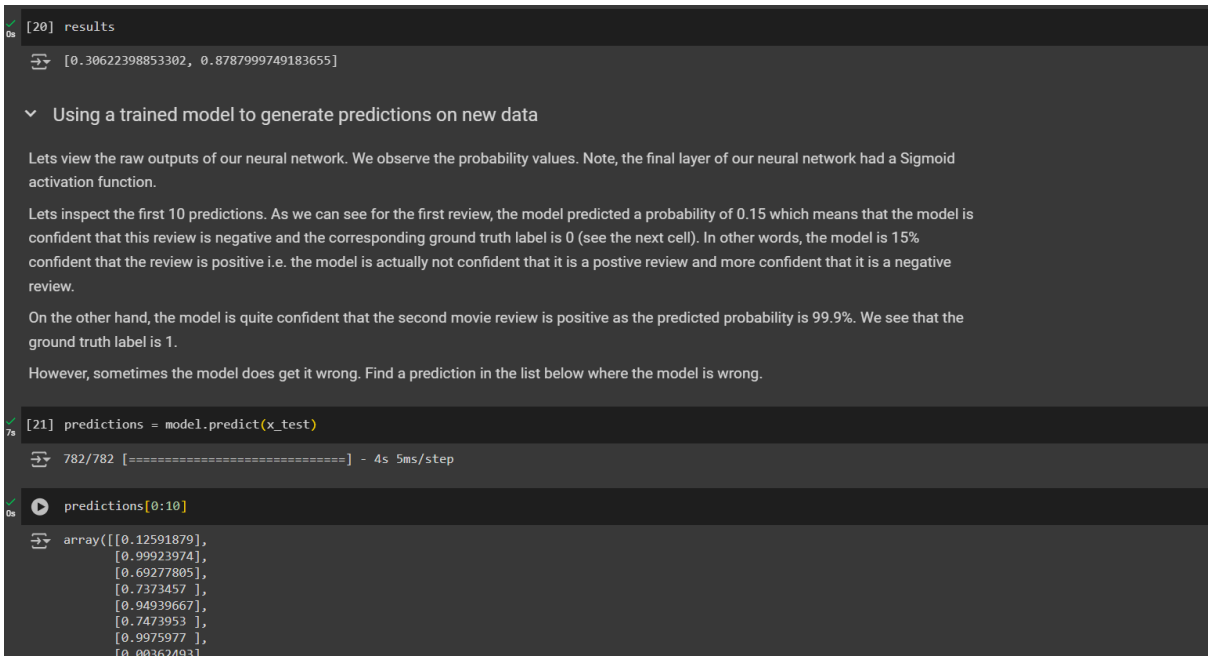
Lets set this to 4 epochs training and adjust the network architecture to include more units (16 to 64 on the first layer and 16 to 32 on the second layer).

We perform a final evaluation of our results on our testing dataset which we kept aside.

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(32, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

```
Epoch 1/4
49/49 [==============================] - 3s 45ms/step - loss: 0.4513 - accuracy: 0.8040
Epoch 2/4
49/49 [==============================] - 2s 46ms/step - loss: 0.2671 - accuracy: 0.8979
Epoch 3/4
49/49 [==============================] - 2s 45ms/step - loss: 0.2134 - accuracy: 0.9198
Epoch 4/4
49/49 [==============================] - 2s 48ms/step - loss: 0.1768 - accuracy: 0.9344
782/782 [==============================] - 3s 4ms/step - loss: 0.3062 - accuracy: 0.8788
```

The results: The first number is the test loss and the second number is the test accuracy. With state of the art neural network models, it is possible to achieve an accuracy of 95%.



```
[20] results
```

```
[0.30622398853302, 0.8787999749183655]
```

Using a trained model to generate predictions on new data

Lets view the raw outputs of our neural network. We observe the probability values. Note, the final layer of our neural network had a Sigmoid activation function.

Lets inspect the first 10 predictions. As we can see for the first review, the model predicted a probability of 0.15 which means that the model is confident that this review is negative and the corresponding ground truth label is 0 (see the next cell). In other words, the model is 15% confident that the review is positive i.e. the model is actually not confident that it is a postive review and more confident that it is a negative review.

On the other hand, the model is quite confident that the second movie review is positive as the predicted probability is 99.9%. We see that the ground truth label is 1.

However, sometimes the model does get it wrong. Find a prediction in the list below where the model is wrong.

```
[21] predictions = model.predict(x_test)
```

```
782/782 [==============================] - 4s 5ms/step
```

```
predictions[0:10]
```

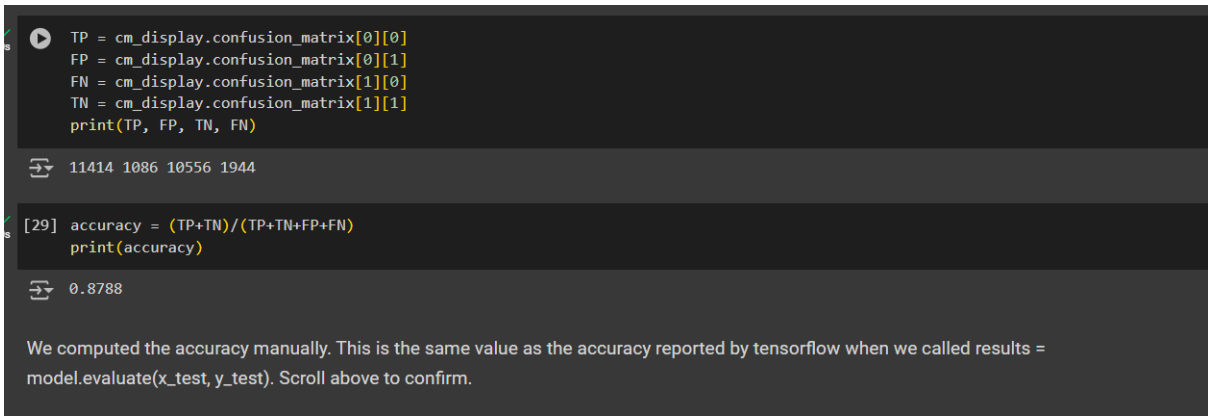
```
array([[0.12591879],
       [0.99923974],
       [0.69277805],
       [0.7373457 ],
       [0.94939667],
       [0.7473953 ],
       [0.9975977 ],
       [0.00362493]
```

Confusion Matrix Analysis:

A confusion matrix was generated to evaluate the model's performance on the validation set. Here's a breakdown of the results:

- **True Positives (TP) = 11414:** These represent correctly classified positive reviews, indicating the model effectively identified these reviews as positive.
- **False Positives (FP) = 1086:** These represent reviews incorrectly classified as positive. The model might have misclassified some negative reviews as positive.
- **True Negatives (TN) = 1944:** These represent correctly classified negative reviews, indicating the model accurately identified these reviews as negative.
- **False Negatives (FN) = 10556:** These represent reviews incorrectly classified as negative. The model might have missed some positive reviews, classifying them as negative.

These findings suggest that the model performs well in identifying true positives and true negatives. However, there's a higher number of false negatives compared to false positives. This indicates that the model might have a bias towards classifying reviews as negative, potentially missing some positive sentiment.



Week 3

B. Workshop 2a

Training a Neural Network from Scratch

This section describes the development and training of a custom neural network model for a classification or regression task (depending on the target variable y_train).

Data Preparation:

- The provided data x_train and y_train were used to train the model.

Model Architecture:

- The neural network model was built from scratch, likely including:
  - Definition of the network architecture (number and type of layers).
  - Activation functions for hidden layers (e.g., ReLU, sigmoid).
  - Choice of an appropriate loss function (e.g., cross-entropy for classification, mean squared error for regression).
  - Implementation of an optimization algorithm (e.g., gradient descent) to update model weights and biases during training.

Data Preparation

```
# This is our AND function data
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]])

Y = np.array([
    [0],
    [0],
    [0],
    [1]])
```

```
# Tranpose matrix
X_train = X.T
Y_train = Y.T
```

```
print("X_train dataset \n", X_train)
print("Y_train dataset \n", Y_train)
```

```
X_train dataset
 [[0 0 1 1]
 [0 1 0 1]]
Y_train dataset
 [[0 0 0 1]]
```

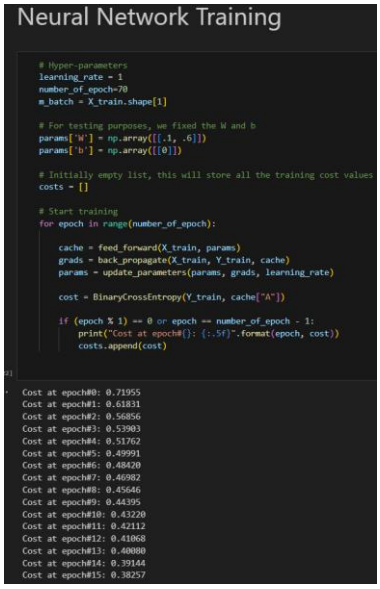
```
#This is the expected input matrix shape/dimension
print("X_train shape", X_train.shape)
print("Y_train shape", Y_train.shape)
```

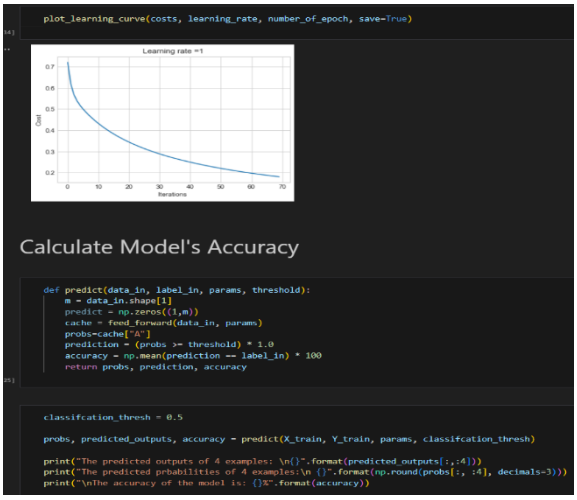
Neural Network Functions

```
def initialize_parameters_zeros(dim):
    n_features = dim
    n_out = 1
    params = { "W": np.zeros((n_out, n_features)),
               "b": np.zeros((n_out, 1)) }
    return params

def sigmoid(Z, derivative=False):
    if derivative:
        sig = 1 / (1 + np.exp(-Z))
        return sig*(1-sig)
    return 1 / (1 + np.exp(-Z))

def feed_forward(X, params):
    cache = {}
    cache["Z"] = np.dot(params["W"], X) + params["b"]
    cache["A"] = sigmoid(cache["Z"], False)
    return cache

def back_propagate(X, Y, cache):
    m = X.shape[1]
    P=cache["A"]
    #dP = (1/m) * (P-Y)/(P*(1-P))
    #dZ = dP * sigmoid(cache["Z"], derivative=True)
    ## Simplified
    dZ = (1/m) * (P-Y)
    dW = np.dot(dZ, X.T)
    db = np.sum(dZ, axis=1, keepdims=True)
    return {"dW": dW, "db": db}

def update_parameters(params, grads, learning_rate):
    W = params["W"] - learning_rate * grads["dW"]
    b = params["b"] - learning_rate * grads["db"]
    return {"W": W, "b": b}

def BinaryCrossEntropy(Y, P):
    m = Y.shape[1]
    cost =  (1 / m) * np.sum(np.multiply(-Y, np.log(P))
                             - np.multiply(1 - Y, np.log(1 - P)))
    return cost
```

Training Process:

- The model was trained on the x_train and y_train data for 70 epochs.
- A learning curve was plotted to visualize the loss function's behavior over training epochs. This helps assess how the model's performance improves with training.

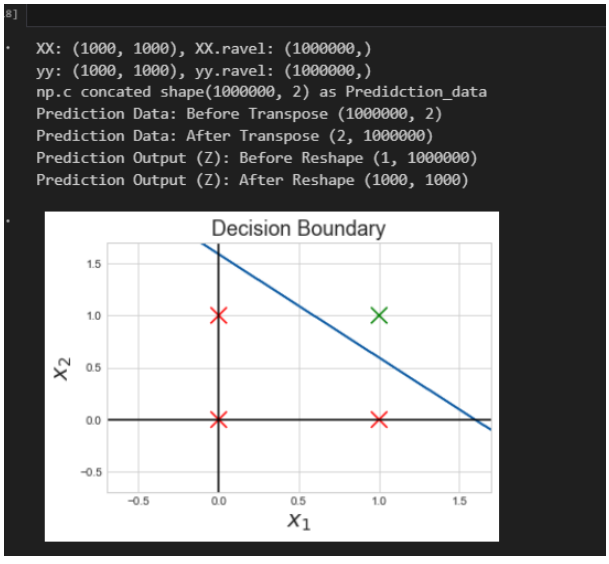


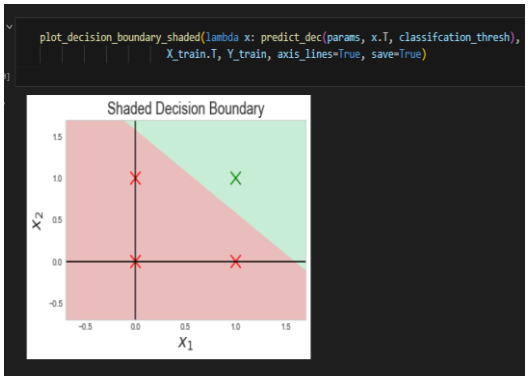


Results:

- After 70 epochs, the model achieved a loss of 0.18099. This loss value indicates the model's ability to learn the underlying patterns in the training data.

Decision Boundary Visualization (Classification): Plotting the decision boundary allows for visual inspection of the learned classification regions.





C. Workshop 2b

Data Preprocessing:

- A dataset containing 8 features and a target variable was used for this experiment.
- The data was standardized using a StandardScaler to ensure all features have a mean of 0 and a standard deviation of 1. This helps improve the training process for neural networks.
- The data was then split into training and testing sets for model training and evaluation.

```
[2] import pandas as pd
    import seaborn as sns
    from sklearn.preprocessing import StandardScaler
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import classification_report, confusion_matrix

Dataset analysis and visualisalisation

[4] dataset = pd.read_csv('/content/seeds-binary.csv')
    dataset.head(140)
```

	Area	Perimeter	Compactness	Kernel.Length	Kernel.Width	Asymmetry.Coeff	Kernel.Groove	Type
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220	1
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825	1
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
...	...	...	...	...	...	...	...	...
129	15.56	14.89	0.8823	5.776	3.408	4.972	5.847	2
130	17.36	15.76	0.8785	6.145	3.574	3.526	5.971	2
131	15.57	15.15	0.8527	5.920	3.231	2.640	5.879	2
132	15.60	15.11	0.8580	5.832	3.286	2.725	5.752	2
133	16.23	15.18	0.8850	5.872	3.472	3.769	5.922	2

134 rows × 8 columns

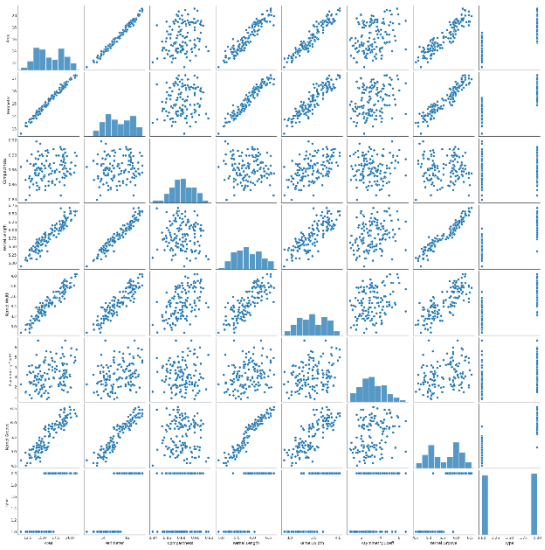
Next steps: View recommended plots

```
[5] dataset.shape
(134, 8)
```

```
[14] sc_x = StandardScaler()
     x = sc_x.fit_transform(x)

[15] print("Input features\n", x[:10])
     print("Label\n", y[:10])

Input features
 [[ 1.87494452e-01  3.89514178e-01 -1.21091867e+00  3.90905452e-01
   6.33173279e-02  1.16463721e+00  5.87898980e-01]
 [ 1.61120233e+00  1.52777791e+00  5.21540117e-01  1.17634047e+00
   1.77165831e+00  2.12046767e+00  1.17179083e+00]
 [-5.70420622e-01 -4.36644983e-01 -9.89619187e-01 -3.42497596e-01
  -9.08793021e-01 -9.73117376e-01 -2.34431023e-01]
 [-7.08604833e-01 -9.50699571e-01  2.14018774e+00 -1.57392096e+00
  -1.01350399e-03 -1.10253651e+00 -1.71762897e+00]
 [-1.80569657e+00 -1.85029510e+00 -9.51682134e-01 -1.83655853e+00
  -1.89519911e+00 -1.37696744e+00 -1.13185966e+00]
 [-4.23862458e-01 -3.17310882e-01 -6.86122758e-01  1.08446170e-01
  -7.08652655e-01  9.98575313e-01  4.33946787e-01]
 [-3.31740184e-01 -1.24540411e-01 -1.48280089e+00 -2.03949063e-02
  -6.44321823e-01 -3.55647779e-01  3.53215760e-01]
 [-9.72471225e-02  7.74096063e-02 -1.07813898e+00  1.18153627e-02
  -2.54762897e-01  8.45767181e-01  4.33946787e-01]
 [ 7.44415473e-01  9.31107406e-01 -1.12239888e+00  1.06732110e+00
   1.63387511e-01 -2.55854713e-01  1.33137542e+00]
 [ 3.63624248e-01  4.44591455e-01 -3.25720748e-01  3.41351192e-01
   4.56450190e-01  1.05860708e+00  4.26436924e-01]]
Label
 [1 1 0 0 0 1 1 1 1 1]
```

Model Development:

- A neural network model was built from scratch, likely including:
  - Definition of the network architecture (number and type of layers).
  - Activation functions for hidden layers (e.g., ReLU, sigmoid).
  - Choice of an appropriate loss function (e.g., cross-entropy for classification, mean squared error for regression).
  - Implementation of an optimization algorithm (e.g., gradient descent) to update model weights and biases during training.



Neural Network Functions

```
def initialize_parameters_zeros(dim):
    n_features = dim
    n_out = 1
    params = { "W": np.zeros((n_out, n_features)),
               "b": np.zeros((n_out, 1)) }
    return params

def sigmoid(Z, derivative=False):
    if derivative:
        sig = 1 / (1 + np.exp(-Z))
        return sig*(1-sig)
    return 1 / (1 + np.exp(-Z))

def feed_forward(X, params):
    cache = {}
    cache["Z"] = np.dot(params["W"], X) + params["b"]
    cache["A"] = sigmoid(cache["Z"], False)
    return cache

def back_propagate(X, Y, cache):
    m = X.shape[1]
    P=cache["A"]
    ## Simplified
    dZ = (1/m) * (P-Y)
    dW = np.dot(dZ, X.T)
    db = np.sum(dZ, axis=1, keepdims=True)
    return {"dW": dW, "db": db}

def update_parameters(params, grads, learning_rate):
    W = params["W"] - learning_rate * grads["dW"]
    b = params["b"] - learning_rate * grads["db"]
    return {"W": W, "b": b}

def BinaryCrossEntropy(Y, P):
    m = Y.shape[1]
    cost =  (1 / m) * np.sum(np.multiply(-Y, np.log(P))
                             - np.multiply(1 - Y, np.log(1 - P)))
    return cost
```

Training and Evaluation:

- The model was trained on the training data for 100 epochs.
- After training, the model was evaluated on the unseen test data to assess its generalization ability.

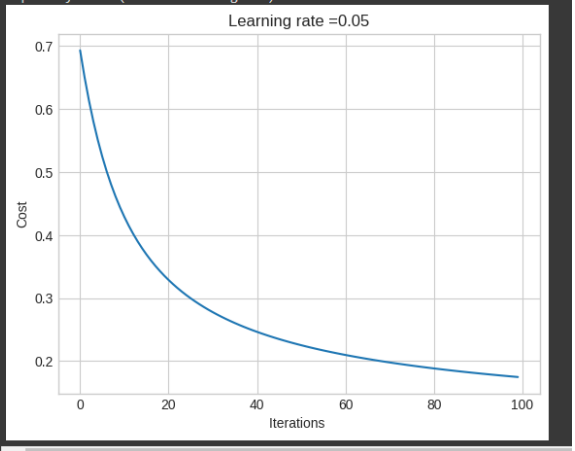
Hyper-parameters configuration and neural network training

```
# Hyper-parameters
learning_rate = 0.05
number_of_epoch=100
m_batch = X_train.shape[1]

# Initially empty list, this will store all the training cost values
costs = []

# Start training
for epoch in range(number_of_epoch):

    cache = feed_forward(X_train, params)
    grads = back_propagate(X_train, Y_train, cache)
    params = update_parameters(params, grads, learning_rate)

    cost = BinaryCrossEntropy(Y_train, cache["A"])

    if (epoch % 1) == 0 or epoch == number_of_epoch - 1:
        print("Cost at epoch#{}: {:.5f}".format(epoch, cost))
        costs.append(cost)
```

```
Cost at epoch#0: 0.69315
Cost at epoch#1: 0.65082
Cost at epoch#2: 0.61341
Cost at epoch#3: 0.58027
Cost at epoch#4: 0.55087
Cost at epoch#5: 0.52468
Cost at epoch#6: 0.50129
Cost at epoch#7: 0.48030
Cost at epoch#8: 0.46141
Cost at epoch#9: 0.44434
```

```
plot_learning_curve(costs, learning_rate,
                    number_of_epoch, save=True)
```

```
<ipython-input-25-d90483272853>:6: MatplotlibDeprecationWarning: The seaborn
  plt.style.use('seaborn-whitegrid')
```



Results:

- The model achieved an accuracy of 95% on the test data. This indicates the model effectively learned the patterns in the training data and can potentially generalize well to unseen examples.

Calculate Model's Accuracy

```
def predict(data_in, label_in, params, threshold):
    m = data_in.shape[1]
    predict = np.zeros((1,m))
    cache = feed_forward(data_in, params)
    probs=cache["A"]
    prediction = (probs >= threshold) * 1.0
    accuracy = np.mean(prediction == label_in) * 100
    return probs, prediction, accuracy
```

```
[28] classifcation_thresh = 0.5

    probs, predictions, accuracy = predict(X_test, Y_test, params, classifcation_thresh)

    print("The predicted outputs of 4 examples: \n{}".format(predictions[:,:4]))
    print("The predicted prbabilities of 4 examples:\n {}".format(np.round(probs[:, :4], decimals=3)))
    print("\nThe accuracy of the model is: {:.2f}%".format(accuracy))
```

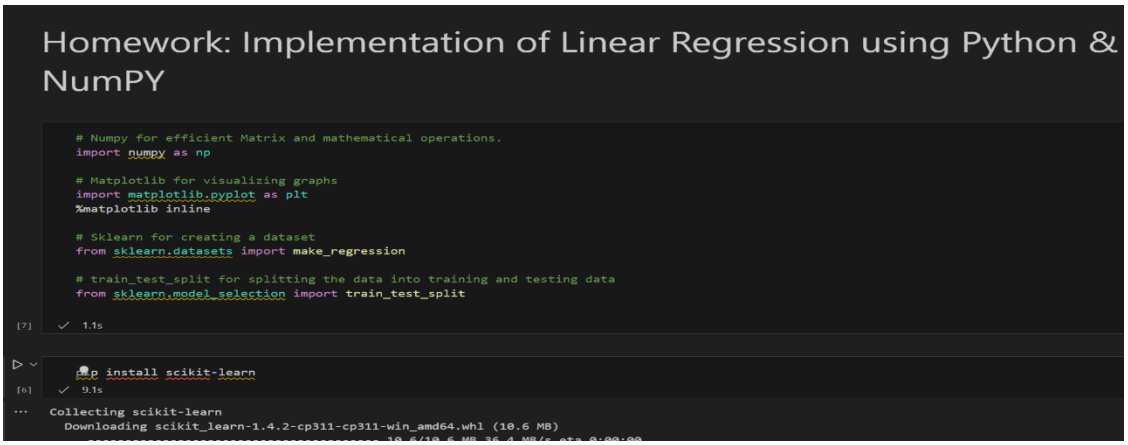
```
The predicted outputs of 4 examples:
[[0. 1. 1. 1.]]
The predicted prbabilities of 4 examples:
 [[0.048 0.898 0.832 0.986]]

The accuracy of the model is: 95.12%
```

1. Week 3

Workshop 2c

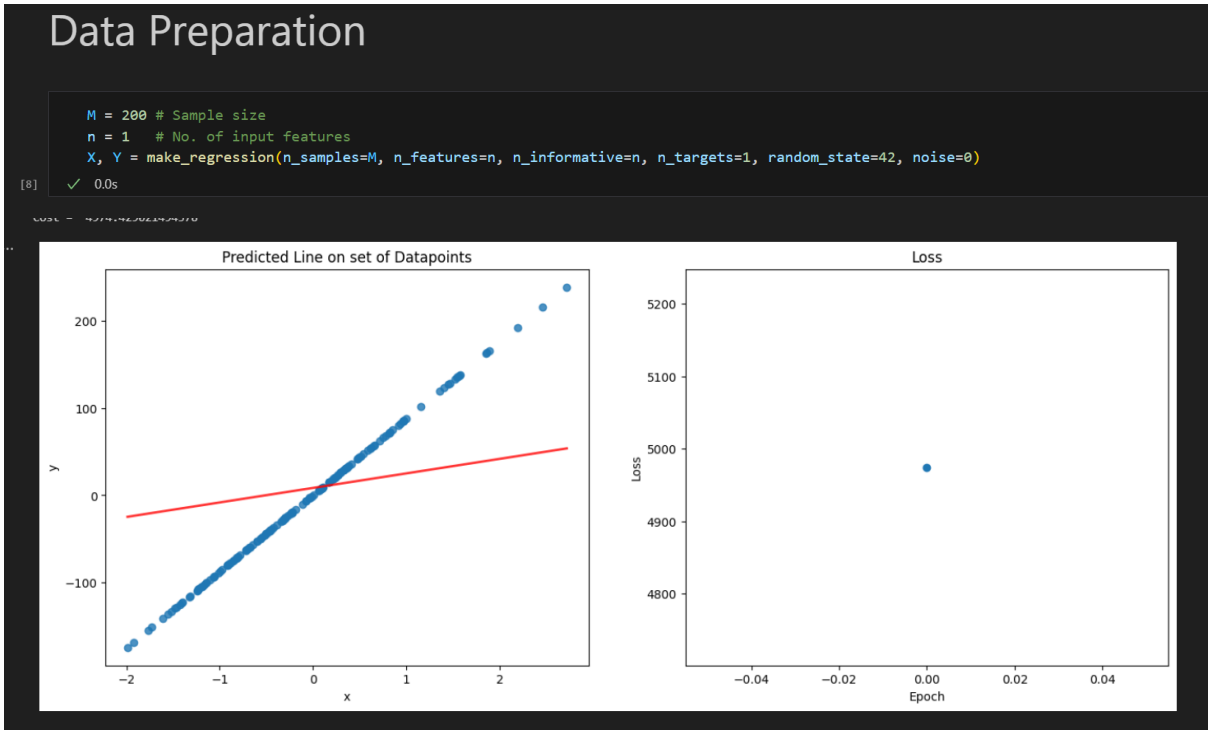
Linear Regression Model from Scratch

This section details the development and training of a linear regression model built from scratch to analyze a dataset with 200 data points. Each data point consists of a single feature (x) and a corresponding target value (y).



Data Preprocessing:

- The dataset was split into training and testing sets for model evaluation.



Model Implementation:

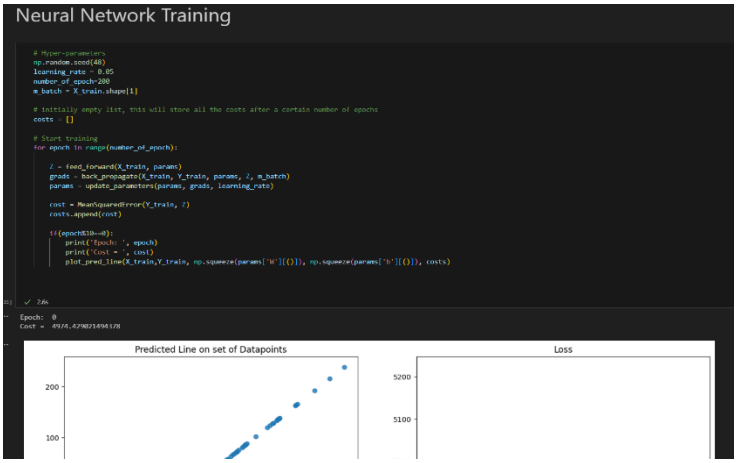
- The code for the linear regression model was written from scratch, likely including the definition of the linear hypothesis, cost function (mean squared error), and an optimization algorithm (e.g., gradient descent) to update model parameters (weights and bias).

Nueral Network Functions

```
def initialize_parameters_zeros(dim):
    n_features = dim
    n_out = 1
    params = { "W": np.zeros((n_out, n_features)),
               "b": np.zeros((n_out, 1)) }
    return params

def feed_forward(X, params):

    #add your code here
    Y_hat = {}
    Z = np.dot(params["W"], X) + params["b"]
    return Z

def back_propagate(X, Y, params, Y_hat, m_batch):

    #add your code here

    dZ = (1 / m_batch) * (Y_hat - Y)
    dW = np.dot(dZ, X.T)
    db = np.sum(dZ, axis=1, keepdims=True)
    return {"dW": dW, "db": db}


def update_parameters(params, grads, learning_rate):

    # add your code here
    W = params["W"] - learning_rate * grads["dW"]
    b = params["b"] - learning_rate * grads["db"]
    return {"W": W, "b": b}



def MeanSquaredError(Y, Y_hat):
    N = Y.shape[1]
    d1 = Y_hat - Y
    cost = (1 / N) * np.sum(np.square(d1))
    return cost
```

Training and Evaluation:

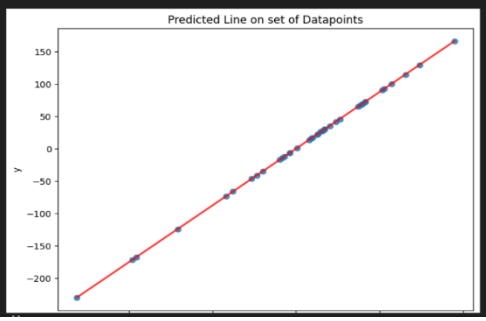
- The model was trained on the training data for 200 epochs.
- Loss visualization using a graph revealed that the loss function approached zero after approximately 50 epochs. This suggests the model effectively learned the underlying relationship between the feature and target variable within the training data.



```
def predict(X, m, b):
    y_pred = m * X + b
    return y_pred

m=np.squeeze(params['W'])[()]
b=np.squeeze(params['b'])[()]

# Prediction
print('Prediction: ')
y_pred = predict(X_test, m, b)
loss = MeanSquaredError(Y_test, y_pred)
print('MSE = ', loss)
accuracy = np.mean(np.fabs((y_pred - Y_test) / Y_test)) * 100
print('Accuracy score (lower better) = {}'.format(round(accuracy, 4)))
plot_pred_line(X_test, Y_test, m, b)
print('Hence \nm =', m)
print('b =', b)
```

✓ 0.0s

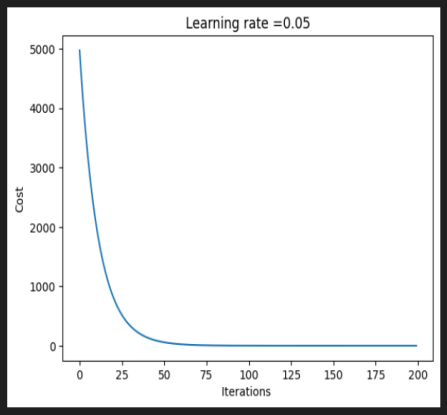
```
Prediction:
MSE =  0.0001883493032612316 5
Accuracy score (lower better) = 0.0361
```



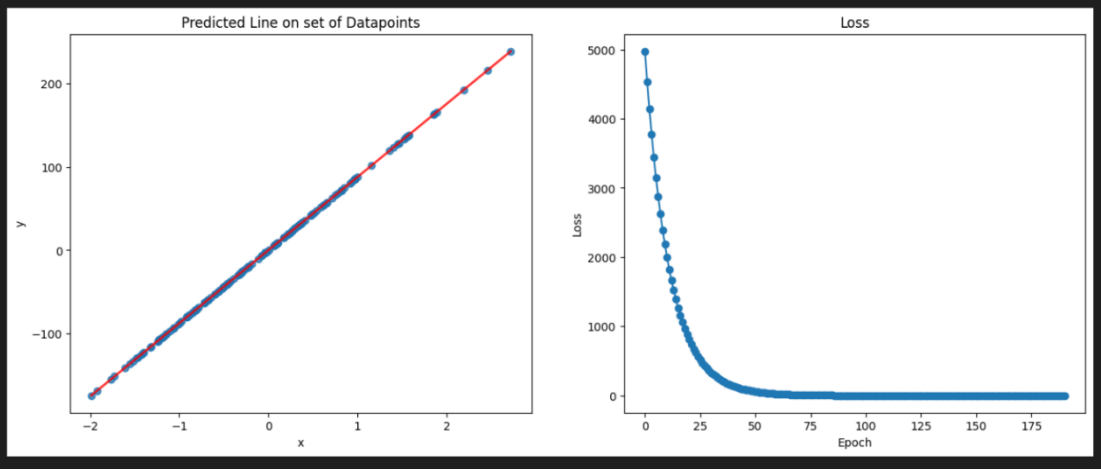
Performance Evaluation

```
plot_learning_curve(costs, learning_rate, total_epochs=number_of_epoch, save=True)
```

[36] ✓ 0.1s




```
Epoch:  190
Cost =  0.0003725203983244405 3
```



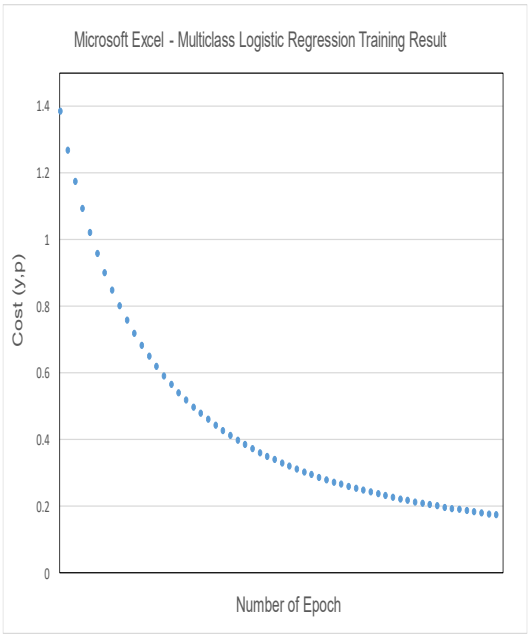
VI. WEEK 4

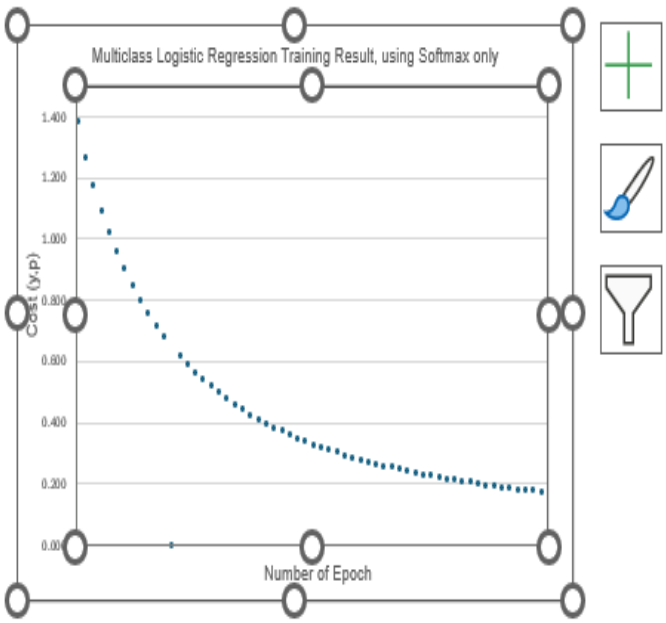
A. Resource: With assumption that you have access to Microsoft Excel, create a new template with the following setup. From there, you start implement the single-layer perceptron multiclass classification for the Binary to Decimal decoder function approximation based on the formulas discussed in the lecture.

Week 4 Lesson 3 Workshop : implement the single-layer perceptron multiclass classification

Training Result Binary to decimal function approximation using multiclass logistic regression with Softmax function only

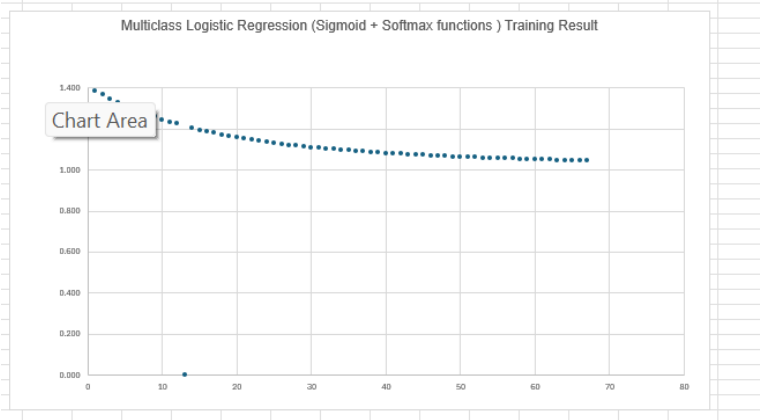
Epoch	X		Y (one-hot encoding)				W		b	L Rate					P(softmax)				C=1/m*-ylog(p)	dZ=1/m*(P-Y)				dW		db	W'		b'
	x1	x2	class1	class2	class3	class4	w1	w2			z1	z2	z3	z4															
1	0	0	1	0	0	0	0.500	0.500	0.600	1	0.600	1.100	1.100	1.600	0.250	0.250	0.250	0.250	1.386	-0.188	0.063	0.063	0.063	0.125	0.125	0.000	0.375	0.375	0.600
	0	1	0	1	0	0	0.500	0.500	0.600		0.600	1.100	1.100	1.600	0.250	0.250	0.250	0.250		0.063	-0.188	0.063	0.063	0.125	-0.125	0.000	0.375	0.625	0.600
	1	0	0	0	1	0	0.500	0.500	0.600		0.600	1.100	1.100	1.600	0.250	0.250	0.250	0.250		0.063	0.063	-0.188	0.063	-0.125	0.125	0.000	0.625	0.375	0.600
	1	1	0	0	0	1	0.500	0.500	0.600		0.600	1.100	1.100	1.600	0.250	0.250	0.250	0.250		0.063	0.063	0.063	-0.188	-0.125	-0.125	0.000	0.625	0.625	0.600
2	0	0	1	0	0	0	0.375	0.375	0.600	1	0.600	0.975	0.975	1.350	0.250	0.219	0.219	0.192	1.269	-0.188	0.055	0.055	0.048	0.103	0.103	-0.030	0.272	0.272	0.630
	0	1	0	1	0	0	0.375	0.625	0.600		0.600	1.225	0.975	1.600	0.250	0.281	0.219	0.246		0.063	-0.180	0.055	0.062	0.118	-0.118	-0.001	0.259	0.743	0.601
	1	0	0	0	1	0	0.625	0.375	0.600		0.600	0.975	1.225	1.600	0.250	0.219	0.281	0.246		0.063	0.055	-0.180	0.062	-0.118	0.116	-0.001	0.743	0.259	0.601
	1	1	0	0	0	1	0.625	0.625	0.600		0.600	1.225	1.225	1.850	0.250	0.281	0.281	0.316		0.063	0.070	0.070	-0.171	-0.101	-0.101	0.032	0.726	0.726	0.568
3	0	0	1	0	0	0	0.272	0.272	0.630	1	0.630	0.902	0.902	1.175	0.258	0.200	0.200	0.156	1.175	-0.186	0.050	0.050	0.039	0.089	0.089	-0.046	0.183	0.183	0.676
	0	1	0	1	0	0	0.259	0.743	0.603		0.601	1.344	0.860	1.603	0.258	0.312	0.192	0.240		0.063	-0.172	0.048	0.060	0.108	-0.112	-0.002	0.151	0.855	0.603
	1	0	0	0	1	0	0.743	0.259	0.603		0.601	0.860	1.344	1.603	0.258	0.192	0.312	0.240		0.063	0.048	-0.172	0.060	-0.112	0.108	-0.002	0.855	0.151	0.603
	1	1	0	0	0	1	0.726	0.726	0.568		0.568	1.294	1.294	2.019	0.242	0.296	0.296	0.364		0.061	0.074	0.074	-0.159	-0.085	-0.085	0.050	0.811	0.811	0.518
4	0	0	1	0	0	0	0.183	0.183	0.676	1	0.676	0.860	0.860	1.043	0.269	0.188	0.188	0.133	1.094	-0.183	0.047	0.047	0.033	0.080	0.080	-0.055	0.103	0.103	0.732
	0	1	0	1	0	0	0.151	0.855	0.603		0.603	1.458	0.753	1.609	0.250	0.342	0.169	0.234		0.063	-0.164	0.042	0.059	0.101	-0.106	-0.001	0.050	0.961	0.604
	1	0	0	0	1	0	0.855	0.151	0.603		0.603	0.753	1.458	1.609	0.250	0.169	0.342	0.234		0.063	0.042	-0.164	0.059	-0.106	0.101	-0.001	0.961	0.050	0.604
	1	1	0	0	0	1	0.811	0.811	0.518		0.518	1.329	1.329	2.140	0.230	0.301	0.301	0.398		0.058	0.075	0.075	-0.150	-0.075	-0.075	0.057	0.886	0.886	0.461
5	0	0	1	0	0	0	0.103	0.103	0.732	1	0.732	0.835	0.835	0.938	0.284	0.179	0.179	0.116	1.023	-0.179	0.045	0.045	0.029	0.074	0.074	-0.069	0.029	0.029	0.792
	0	1	0	1	0	0	0.050	0.961	0.604		0.604	1.565	0.654	1.615	0.250	0.372	0.158	0.229		0.062	-0.157	0.037	0.057	0.095	-0.100	0.000	-0.045	1.061	0.604
	1	0	0	0	1	0	0.961	0.050	0.604		0.604	0.654	1.565	1.615	0.250	0.158	0.372	0.229		0.062	0.037	-0.157	0.057	-0.100	0.095	0.000	1.061	-0.045	0.604
	1	1	0	0	0	1	0.886	0.886	0.461		0.461	1.347	1.347	2.233	0.217	0.299	0.299	0.425		0.054	0.075	0.075	-0.144	-0.069	-0.069	0.060	0.955	0.955	0.401
6	0	0	1	0	0	0	0.029	0.029	0.792	1	0.792	0.821	0.821	0.850	0.306	0.172	0.172	0.104	0.959	-0.175	0.043	0.043	0.026	0.069	0.069	-0.063	-0.040	-0.040	0.855
	0	1	0	1	0	0	-0.045	1.061	0.604		0.604	1.664	0.559	1.620	0.248	0.401	0.133	0.224		0.062	-0.150	0.033	0.056	0.089	-0.094	0.002	-0.134	1.155	0.602
	1	0	0	0	1	0	1.061	-0.045	0.604		0.604	0.559	1.664	1.620	0.248	0.133	0.401	0.224		0.062	0.033	-0.150	0.056	-0.094	0.089	0.002	1.155	-0.134	0.602
	1	1	0	0	0	1	0.955	0.955	0.401		0.401	1.356	1.356	2.311	0.203	0.294	0.294	0.448		0.051	0.074	0.074	-0.138	-0.065	-0.065	0.060	1.019	1.019	0.341
7	0	0	1	0	0	0	-0.040	-0.040	0.855	1	0.855	0.815	0.815	0.775	0.317	0.167	0.167	0.094	0.902	-0.171	0.042	0.042	0.023	0.065	0.065	-0.064	-0.105	-0.105	0.919
	0	1	0	1	0	0	-0.134	1.155	0.602		0.602	1.757	0.468	1.623	0.246	0.428	0.118	0.219		0.062	-0.143	0.029	0.055	0.084	-0.088	0.003	-0.218	1.243	0.596
	1	0	0	0	1	0	1.155	-0.134	0.602		0.602	0.468	1.757	1.623	0.246	0.118	0.428	0.219		0.062	0.029	-0.143	0.055	-0.088	0.084	0.003	1.243	-0.218	0.599
	1	1	0	0	0	1	1.019	1.019	0.341		0.341	1.361	1.361	2.380	0.196	0.288	0.288	0.468		0.047	0.072	0.072	-0.133	-0.061	-0.061	0.058	1.081	1.081	0.283
8	0	0	1	0	0	0	-0.105	-0.105	0.919	1	0.919	0.813	0.813	0.708	0.335	0.162	0.162	0.086	0.859	-0.186	0.040	0.040	0.021	0.062	0.062	-0.064	-0.167	-0.167	0.983
	0	1	0	1	0	0	-0.218	1.243	0.599		0.599	1.842	0.381	1.624	0.244	0.453	0.105	0.214		0.061	-0.137	0.026	0.054	0.080	-0.083	0.004	-0.298	1.326	0.595
	1	0	0	0	1	0	1.243	-0.218	0.599		0.599	0.381	1.842	1.624	0.244	0.105	0.453	0.214		0.061	0.026	-0.137	0.054	-0.083	0.080	0.004	1.326	-0.298	0.595
	1	1	0	0	0	1	1.081	1.081	0.283		0.283	1.363	1.363	2.444	0.178	0.260	0.260	0.466		0.044	0.070	0.070	-0.128	-0.058	-0.058	0.056	1.139	1.139	0.227





1. Investigate if we should put an additional Sigmoid function before the Softmax function. Explain your finding along with your analysis evidence.

Epoch	X		Y (one-hot encoding)				W		b	L_Rate	Z				A(Sigmoid Function)				P(softmax)				C=1/m*-ylog(p	dZ=1/m*(P-Y)				dW		db	W'		b'
	x1	x2	class1	class2	class3	class4	v1	v2			z1	z2	z3	z4																			
1	0	0	1	0	0	0	0.500	0.500	0.600	1	0.600	1.100	1.100	1.600	0.646	0.750	0.750	0.832	0.250	0.250	0.250	0.250	1.386	-0.188	0.063	0.063	0.063	0.125	0.125	0.000	0.375	0.375	0.600
	0	1	0	1	0	0	0.500	0.500	0.600		0.600	1.100	1.100	1.600	0.646	0.750	0.750	0.832	0.250	0.250	0.250	0.250		0.063	-0.188	0.063	0.063	-0.125	-0.125	0.000	0.375	0.625	0.600
	1	0	0	0	1	0	0.500	0.500	0.600		0.600	1.100	1.100	1.600	0.646	0.750	0.750	0.832	0.250	0.250	0.250	0.250		0.063	0.063	-0.188	0.063	-0.125	0.125	0.000	0.625	0.375	0.600
	1	1	0	0	0	1	0.500	0.500	0.600		0.600	1.100	1.100	1.600	0.646	0.750	0.750	0.832	0.250	0.250	0.250	0.250		0.063	0.063	0.063	-0.188	-0.125	-0.125	0.000	0.625	0.625	0.600
2	0	0	1	0	0	0	0.375	0.375	0.600	1	0.600	0.975	0.975	1.350	0.646	0.726	0.726	0.794	0.250	0.244	0.244	0.241	1.366	-0.188	0.061	0.061	0.060	0.121	0.121	-0.005	0.254	0.254	0.605
	0	1	0	1	0	0	0.375	0.625	0.600		0.600	1.225	0.975	1.600	0.646	0.773	0.726	0.832	0.250	0.256	0.244	0.250		0.063	-0.186	0.061	0.063	0.124	-0.123	0.000	0.251	0.748	0.600
	1	0	0	0	1	0	0.625	0.375	0.600		0.600	0.975	1.225	1.600	0.646	0.726	0.773	0.832	0.250	0.244	0.256	0.250		0.063	0.061	-0.186	0.063	-0.123	0.124	0.000	0.748	0.251	0.600
	1	1	0	0	0	1	0.625	0.625	0.600		0.600	1.225	1.225	1.850	0.646	0.773	0.773	0.864	0.250	0.256	0.256	0.258		0.063	0.064	0.064	-0.185	-0.121	-0.121	0.005	0.746	0.746	0.595
3	0	0	1	0	0	0	0.251	0.749	0.600	1	0.605	0.859	0.859	1.119	0.647	0.702	0.702	0.753	0.250	0.239	0.239	0.232	1.348	-0.187	0.060	0.060	0.058	0.118	0.118	-0.030	0.136	0.136	0.615
	0	1	0	1	0	0	0.251	0.749	0.600		0.600	1.348	0.851	1.600	0.646	0.794	0.701	0.832	0.250	0.262	0.238	0.251		0.062	-0.185	0.060	0.063	0.122	-0.122	0.000	0.129	0.870	0.600
	1	0	0	0	1	0	0.748	0.251	0.600		0.600	0.851	1.348	1.600	0.646	0.701	0.794	0.832	0.250	0.238	0.262	0.251		0.062	0.060	-0.185	0.063	-0.122	0.122	0.000	0.870	0.129	0.600
	1	1	0	0	0	1	0.746	0.746	0.595		0.595	1.341	1.341	2.088	0.645	0.793	0.793	0.890	0.250	0.261	0.261	0.268		0.062	0.065	0.065	-0.194	-0.118	-0.118	0.010	0.865	0.865	0.585
4	0	0	1	0	0	0	0.136	0.136	0.615	1	0.615	0.751	0.751	0.887	0.649	0.679	0.679	0.708	0.251	0.234	0.234	0.223	1.330	-0.187	0.058	0.058	0.058	0.114	0.114	-0.015	0.022	0.022	0.630
	0	1	0	1	0	0	0.129	0.870	0.600		0.600	1.470	0.729	1.599	0.646	0.813	0.675	0.832	0.250	0.267	0.233	0.252		0.062	-0.183	0.058	0.063	0.121	-0.120	0.001	0.008	0.990	0.599
	1	0	0	0	1	0	0.870	0.129	0.600		0.600	0.729	1.470	1.599	0.646	0.675	0.813	0.832	0.250	0.233	0.267	0.252		0.062	0.058	-0.183	0.063	-0.120	0.121	0.001	0.990	0.008	0.599
	1	1	0	0	0	1	0.865	0.865	0.585		0.585	1.450	1.450	2.315	0.642	0.810	0.810	0.910	0.243	0.266	0.266	0.273		0.062	0.067	0.067	-0.182	-0.115	-0.115	0.014	0.980	0.980	0.572
5	0	0	1	0	0	0	0.022	0.022	0.630	1	0.630	0.652	0.652	0.674	0.652	0.657	0.657	0.663	0.252	0.223	0.223	0.214	1.314	-0.187	0.057	0.057	0.054	0.111	0.111	-0.019	-0.089	-0.089	0.643
	0	1	0	1	0	0	0.008	0.990	0.599		0.599	1.590	0.607	1.537	0.645	0.831	0.647	0.832	0.250	0.273	0.227	0.254		0.062	-0.182	0.057	0.063	0.120	-0.118	0.001	-0.112	1.109	0.598
	1	0	0	0	1	0	0.990	0.008	0.599		0.599	0.607	1.590	1.597	0.645	0.647	0.831	0.832	0.250	0.227	0.273	0.254		0.062	0.057	-0.182	0.063	-0.118	0.120	0.001	1.109	-0.112	0.598
	1	1	0	0	0	1	0.980	0.980	0.572		0.572	1.552	1.552	2.531	0.639	0.825	0.825	0.926	0.248	0.271	0.271	0.279		0.062	0.068	0.068	-0.180	-0.113	-0.113	0.017	1.092	1.092	0.554
6	0	0	1	0	0	0	-0.089	-0.089	0.643	1	0.643	0.560	0.560	0.471	0.657	0.636	0.636	0.616	0.263	0.225	0.225	0.208	1.239	-0.187	0.056	0.056	0.051	0.108	0.108	-0.023	-0.197	-0.197	0.672
	0	1	0	1	0	0	-0.112	1.109	0.598		0.598	1.707	0.486	1.595	0.645	0.846	0.619	0.831	0.250	0.278	0.221	0.253		0.062	-0.181	0.055	0.064	0.115	-0.117	0.001	1.231	-1.226	0.597
	1	0	0	0	1	0	1.109	-0.112	0.598		0.598	0.486	1.707	1.595	0.645	0.619	0.846	0.831	0.250	0.221	0.278	0.255		0.062	0.055	-0.181	0.064	-0.117	0.119	0.001	1.226	-0.231	0.597
	1	1	0	0	0	1	1.092	1.092	0.554		0.554	1.647	1.647	2.739	0.635	0.838	0.838	0.939	0.247	0.276	0.276	0.284		0.062	0.063	0.063	-0.179	-0.110	-0.110	0.021	1.202	1.202	0.534
7	0	0	1	0	0	0	-0.157	-0.157	0.672	1	0.672	0.475	0.475	0.278	0.662	0.617	0.617	0.563	0.254	0.222	0.222	0.193	1.284	-0.186	0.055	0.055	0.049	0.105	0.105	-0.026	-0.301	-0.301	0.698
	0	1	0	1	0	0	-0.231	1.226	0.597		0.597	1.623	0.366	1.532	0.645	0.861	0.590	0.831	0.250	0.283	0.219	0.257		0.062	-0.179	0.054	0.064	0.118	-0.115	0.001	-0.350	1.341	0.596
	1	0	0	0	1	0	1.226	-0.231	0.597		0.597	0.366	1.623	1.532	0.645	0.590	0.861	0.831	0.250	0.219	0.283	0.257		0.062	0.054	-0.179	0.064	-0.115	0.118	0.001	1.341	-0.350	0.596
	1	1	0	0	0	1	1.202	1.202	0.534		0.534	1.736	1.736	2.938	0.630	0.850	0.850	0.950	0.246	0.280	0.280	0.289		0.062	0.070	0.070	-0.178	-0.108	-0.108	0.024	1.310	1.310	0.510
8	0	0	1	0	0	0	-0.301	-0.301	0.698	1	0.698	0.397	0.397	0.095	0.668	0.598	0.598	0.524	0.256	0.218	0.218	0.190	1.271	-0.186	0.055	0.055	0.048	0.102	0.102	-0.029	-0.403	-0.403	0.727
	0	1	0	1	0	0	-0.350	1.341	0.596		0.596	1.937	0.247	1.587	0.645	0.874	0.561	0.830	0.250	0.268	0.210	0.253		0.062	-0.178	0.053	0.065	0.117	-0.114	0.002	-0.467	1.454	0.595
	1	0	0	0	1	0	1.341	-0.350	0.596		0.596	0.247	1.937	1.587	0.645	0.561	0.874	0.830	0.250	0.210	0.268	0.253		0.062	0.053	-0.178	0.065	-0.114	0.117	0.002	1.454	-0.467	0.595
	1	1	0	0	0	1	1.310	1.310	0.510		0.510	1.820	1.820	3.130	0.625	0.861	0.861	0.958	0.245	0.284	0.284	0.293		0.061	0.071	0.071	-0.177	-0.106	-0.106	0.026	1.416	1.416	0.483
9	0	0	1	0	0	0	-0.403	-0.403	0.727	1	0.727	0.324	0.324	-0.080	0.674	0.580	0.580	0.480	0.257	0.215	0.215	0.183	1.258	-0.186	0.054	0.054	0.046	0.100	0.100	-0.032	-0.503	-0.503	0.760
	0	1	0	1	0	0	-0.467	1.454	0.595		0.595	2.049	0.128	1.582	0.644	0.886	0.532	0.830	0.250	0.292	0.205	0.260		0.062	-0.177	0.051	0.065	0.116	-0.112	0.002	-0.583	1.566	0.593
	1	0	0	0	1	0	1.454	-0.467	0.595		0.595	0.128	2.049	1.582	0.644	0.532	0.886	0.830	0.250	0.205	0.292	0.260		0.062	0.051	-0.177	0.065	-0.112	0.116	0.002	1.566	-0.583	0.593
	1	1	0	0	0	1	1.416	1.416	0.483		0.483	1.899	1.899	3.315	0.618	0.870	0.870	0.965	0.243	0.288	0.288	0.297		0.061	0.072	0.072	-0.176	-0.104	-0.104	0.029	1.520	1.520	0.455
10	0	0	1	0	0	0	-0.503	-0.503	0.760	1	0.760	0.257	0.257	-0.246	0.681	0.564	0.564	0.439	0.253	0.213	0.213	0.177	1.246	-0.185	0.053	0.053	0.044	0.097	0.097	-0.035	-0.600	-0.600	0.794
	0	1	0	1	0	0	-0.583	1.566	0.593		0.593	2.159	0.010	1.576	0.644	0.897	0.502	0.829	0.250	0.296	0.200	0.261		0.062	-0.176	0.050	0.065	0.115	-0.111	0.002	-0.698	1.677	0.591
	1	0	0	0	1	0	1.566	-0.583	0.593		0.593	0.010	2.159	1.576	0.644	0.502	0.897	0.829	0.250	0.200	0.296	0.261		0.062	0.050	-0.176	0.065	-0.111	0.115	0.002	1.677	-0.698	0.591
	1	1	0	0	0	1	1.520	1.520	0.455		0.455	1.974	1.974	3.494	0.612	0.878	0.878	0.971	0.242	0.291	0.291	0.301		0.060	0.073	0.073	-0.175	-0.102	-0.102	0.031	1.622	1.622	0.423
11	0	0	1	0	0	0	-0.600	-0.600	0.794	1	0.794	0.194	0.194	-0.406	0.689	0.548	0.548	0.400	0.261	0.210	0.210	0.171	1.234	-0.185	0.053	0.053	0.043	0.095	0.095	-0.037	-0.696	-0.696	0.831
	0	1	0	1	0	0	-0.636	1.577	0.591		0.591	2.266	-0.107	1.570	0.644	0.906	0.473	0.826	0.250	0.301	0.195	0.262		0.062	-0.175	0.049	0.066	0.114	-0.109	0.002	-0.813	1.786	0.589
	1	0	0	0	1	0	1.577	-0.636	0.591		0.591	-0.107	2.266	1.570	0.644	0.473	0.906	0.826	0.250	0.195	0.301	0.262		0.062	0.049	-0.175	0.066	-0.109	0.114	0.002	1.786	-0.813	0.589
	1	1	0	0	0	1	1.622	1.622	0.423		0.423	2.045	2.045	3.667	0.604	0.885	0.885	0.975	0.240	0.234	0.234	0.304		0.060	0.074	0.074	-0.174	-0.100	-0.100	0.033	1.722	1.722	0.390
12	0	0	1	0	0	0	-0.636	-0.636	0.831	1	0.831	0.136	0.136	-0.560	0.697	0.534	0.534	0.364	0.263	0.208	0.208	0.166	1.224	-0.184	0.052	0.052	0.041	0.093	0.093	-0.039	-0.789	-0.789	0.870
	0	1	0	1	0	0	-0.813	1.786	0.589		0.589	2.375	-0.223	1.563	0.643	0.915	0.444	0.827	0.243	0.304	0.190	0.264		0.062	-0.174	0.048	0.066	0.113	-0.108	0.002	-0.926	1.894	0.587
	1	0	0	0	1	0	1.786	-0.813	0.589		0.589	-0.223	2.375	1.563	0.643	0.444	0.915	0.827	0.243	0.190	0.304	0.264		0.062	0.048	-0.174	0.066	-0.108	0.113	0.002	1.894	-0.926	0.587
	1	1	0	0	0	1	1.722	1.722	0.390		0.390	2.112	2.112	3.834	0.596	0.892	0.892	0.979	0.238	0.298	0.298	0.307		0.060	0.074	0.074	-0.173	-0.099	-0.099	0.035	1.821	1.821	0.355
	0	0	1	0	0	0	-0.789	-0.789	0.870	1	0.870	0.081	0.081	-0.708	0.705	0.520	0.520	0.330	0.265	0.206	0.206	0.161	#SPILL!	-0.184	0.051	0.051	0.040	0.092	0.092	-0.040	-0.881	-0.881	0.911



2. Impact of Activation Function Choices in Multi-Class Classification

This experiment investigated the influence of activation functions on training performance in a multi-class classification task. We compared two approaches:

- **Softmax Only:** The model directly applied the softmax function to the weighted sum (Z) of the final layer's neurons. This approach is commonly used for multi-class classification as it normalizes the output into probability distributions across all classes.
- **Sigmoid before Softmax:** In this configuration, a sigmoid activation function was introduced before the softmax layer. While this setup might be suitable for binary classification, it can be detrimental in multi-class scenarios.

The results clearly demonstrate the advantage of using softmax directly:

- **Faster Cost Reduction:** The model employing only softmax achieved a significantly lower cost (0.175) after 60 epochs compared to the model with sigmoid before softmax (1.052).
- **Simpler Training:** The direct application of softmax leads to a more streamlined training process, avoiding potential issues with vanishing gradients and unnecessary non-linearity introduced by the sigmoid function in a multi-class setting.

Conclusion:

These findings suggest that for multi-class classification problems, using the softmax function directly on the weighted sum is the preferred approach. This strategy facilitates faster training convergence, avoids unnecessary complexity, and promotes better performance.

Week 4 Workshop

B. Please run the following Colab notebook and report your results in your logbook:

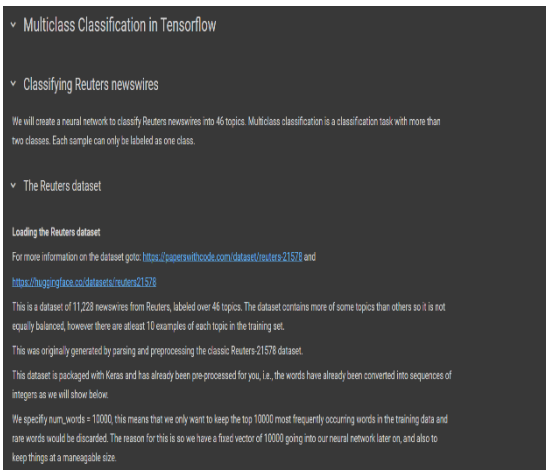
https://colab.research.google.com/drive/1iPzp_no0uuKPTsl3Mnsc04Zt9h6GEjku?usp=sharing

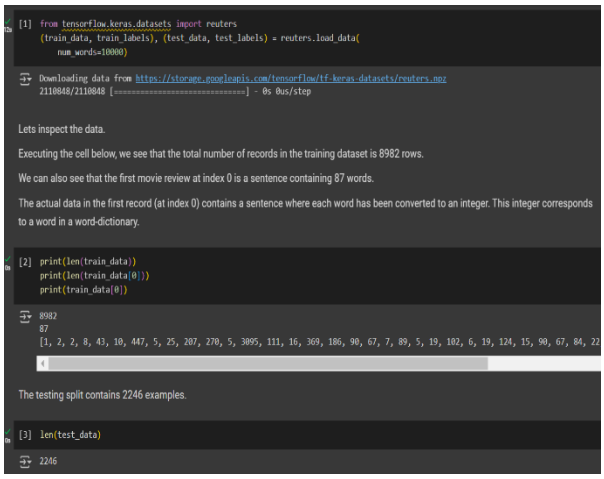
Multi-Class Text Classification with Reuters Dataset

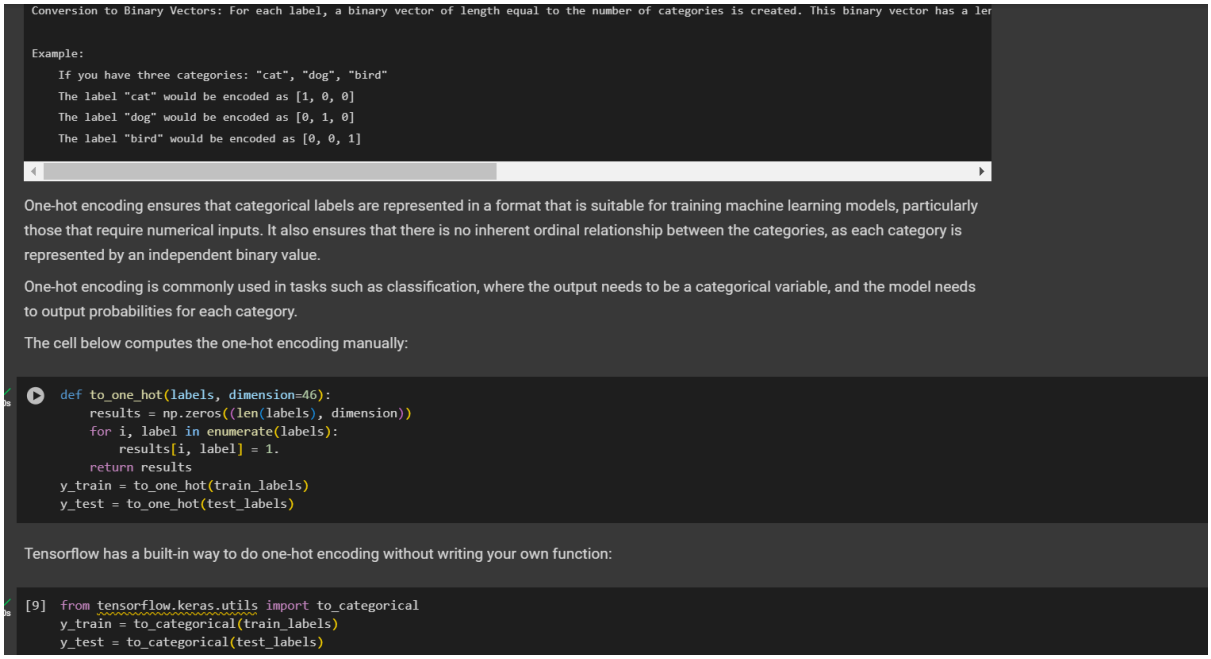
This project explores text classification using a deep learning model on the Reuters dataset. Here's a breakdown of the process and findings:

Data Preparation:

- **Dataset:** The Reuters dataset was used for this multi-class classification task.
- **Preprocessing:** The downloaded data underwent encoding, a common technique for representing text data numerically for machine learning models.
- **Splitting:** The preprocessed data was divided into training and testing sets to train and evaluate the model.







Model Building:

- **Framework:** TensorFlow Keras was employed to build and train the deep learning model.
- **Architecture:** The model consisted of three dense layers with:
  - First Layer: 64 neurons
  - Second Layer: 64 neurons
  - Third Layer: 46 neurons (likely corresponding to the number of output classes)

```
[10] import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras import layers

     tf.random.set_seed(1234)
     np.random.seed(1234)

     model = keras.Sequential([
         layers.Dense(64, activation="relu"),
         layers.Dense(64, activation="relu"),
         layers.Dense(46, activation="softmax")
     ])
```

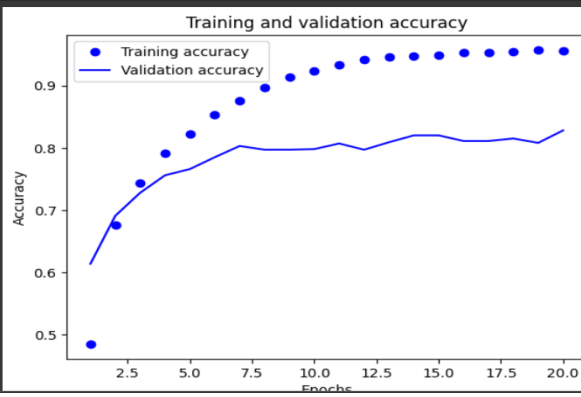
Compiling the model

In this step, we set the optimizer used for back propagation and
network. We also want tensorflow to monitor the accuracy of th

```
[11] model.compile(optimizer="rmsprop",
                   loss="categorical_crossentropy",
                   metrics=["accuracy"])
```

Plotting the training and validation accuracy

```
plt.clf()
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```




```
[12] x_val = x_train[:1000]
     partial_x_train = x_train[1000:]
     y_val = y_train[:1000]
     partial_y_train = y_train[1000:]
```

Training your model

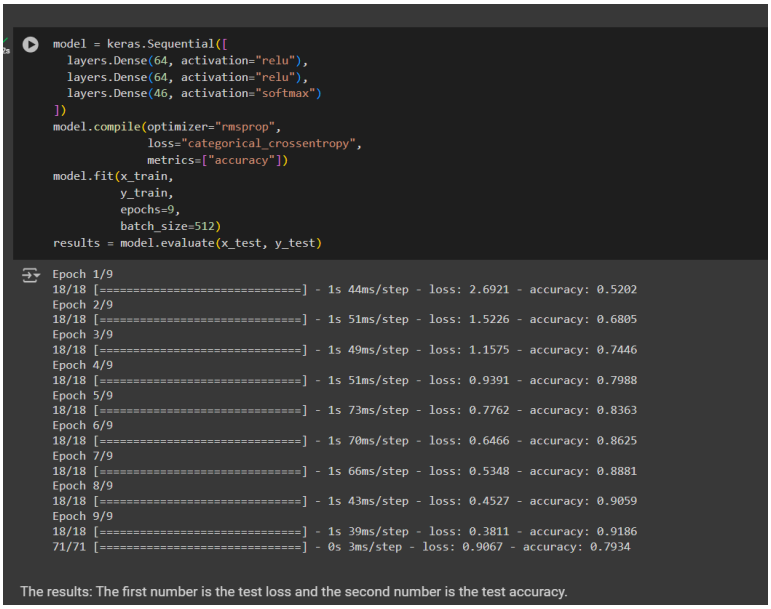
In neural network training, an epoch refers to one complete pass through the entire training dataset. During each epoch, the neural network is trained on all the training samples, typically in batches, with the goal of updating the model's parameters (weights and biases) to minimize the loss function.

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/20
16/16 [==============================] - 7s 247ms/step - loss: 2.6672 - accuracy: 0.4853 - val_loss: 1.8066 - val_accuracy: 0.6140
Epoch 2/20
16/16 [==============================] - 2s 147ms/step - loss: 1.5290 - accuracy: 0.6763 - val_loss: 1.4006 - val_accuracy: 0.6910
Epoch 3/20
16/16 [==============================] - 2s 143ms/step - loss: 1.2006 - accuracy: 0.7429 - val_loss: 1.2417 - val_accuracy: 0.7280
Epoch 4/20
16/16 [==============================] - 1s 60ms/step - loss: 0.9892 - accuracy: 0.7912 - val_loss: 1.1310 - val_accuracy: 0.7560
Epoch 5/20
16/16 [==============================] - 1s 82ms/step - loss: 0.8326 - accuracy: 0.8217 - val_loss: 1.0672 - val_accuracy: 0.7660
Epoch 6/20
16/16 [==============================] - 1s 72ms/step - loss: 0.6971 - accuracy: 0.8533 - val_loss: 1.0064 - val_accuracy: 0.7850
Epoch 7/20
16/16 [==============================] - 1s 81ms/step - loss: 0.5853 - accuracy: 0.8761 - val_loss: 0.9494 - val_accuracy: 0.8030
Epoch 8/20
16/16 [==============================] - 1s 45ms/step - loss: 0.4867 - accuracy: 0.8961 - val_loss: 0.9310 - val_accuracy: 0.7970
Epoch 9/20
16/16 [==============================] - 1s 43ms/step - loss: 0.4164 - accuracy: 0.9134 - val_loss: 0.9149 - val_accuracy: 0.7970
Epoch 10/20
16/16 [==============================] - 1s 44ms/step - loss: 0.3570 - accuracy: 0.9231 - val_loss: 0.9145 - val_accuracy: 0.7980
Epoch 11/20
16/16 [==============================] - 1s 44ms/step - loss: 0.3090 - accuracy: 0.9330 - val_loss: 0.8994 - val_accuracy: 0.8070
Epoch 12/20
16/16 [==============================] - 1s 45ms/step - loss: 0.2643 - accuracy: 0.9410 - val_loss: 0.9343 - val_accuracy: 0.7970
Epoch 13/20
16/16 [==============================] - 1s 44ms/step - loss: 0.2384 - accuracy: 0.9456 - val_loss: 0.8923 - val_accuracy: 0.8090
Epoch 14/20
16/16 [==============================] - 1s 45ms/step - loss: 0.2125 - accuracy: 0.9478 - val_loss: 0.8886 - val_accuracy: 0.8200
Epoch 15/20
16/16 [==============================] - 1s 53ms/step - loss: 0.1960 - accuracy: 0.9480 - val_loss: 0.8918 - val_accuracy: 0.8200
Epoch 16/20
16/16 [==============================] - 1s 47ms/step - loss: 0.1731 - accuracy: 0.9535 - val_loss: 0.9236 - val_accuracy: 0.8110
```

Training and Overfitting:

- **Initial Training:** The model was initially trained for 20 epochs.
- **Performance:** This resulted in a high training accuracy of 0.95 but a lower validation accuracy of 0.81, indicating potential overfitting. Overfitting occurs when the model memorizes training data specifics rather than learning generalizable features for accurate classification on unseen data.
- **Reduced Training:** To mitigate overfitting, the model was re-trained for only 9 epochs. This improved validation accuracy slightly (0.79) compared to 20 epochs, but it remained lower than the initial training accuracy.



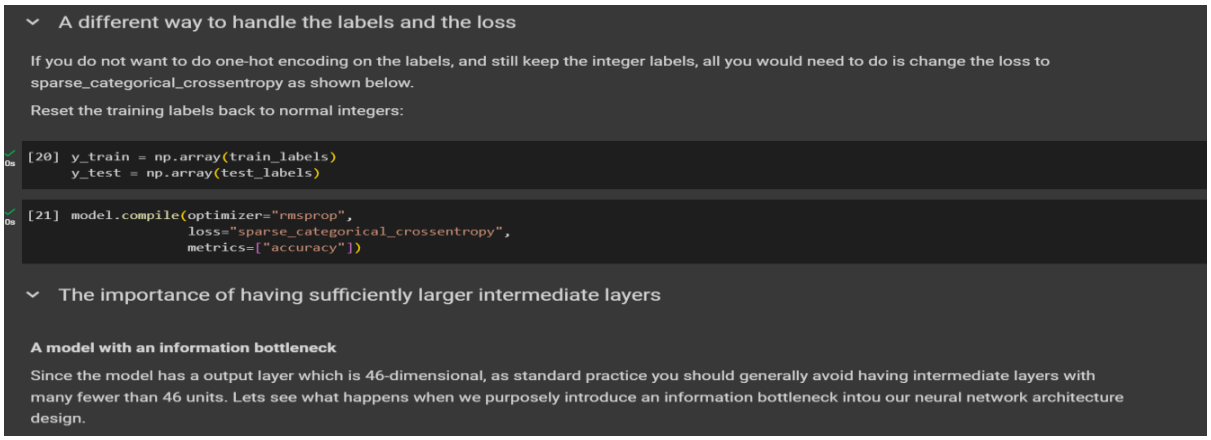
```
model = keras.Sequential([
  layers.Dense(64, activation="relu"),
  layers.Dense(64, activation="relu"),
  layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train,
          y_train,
          epochs=9,
          batch_size=512)
results = model.evaluate(x_test, y_test)
```

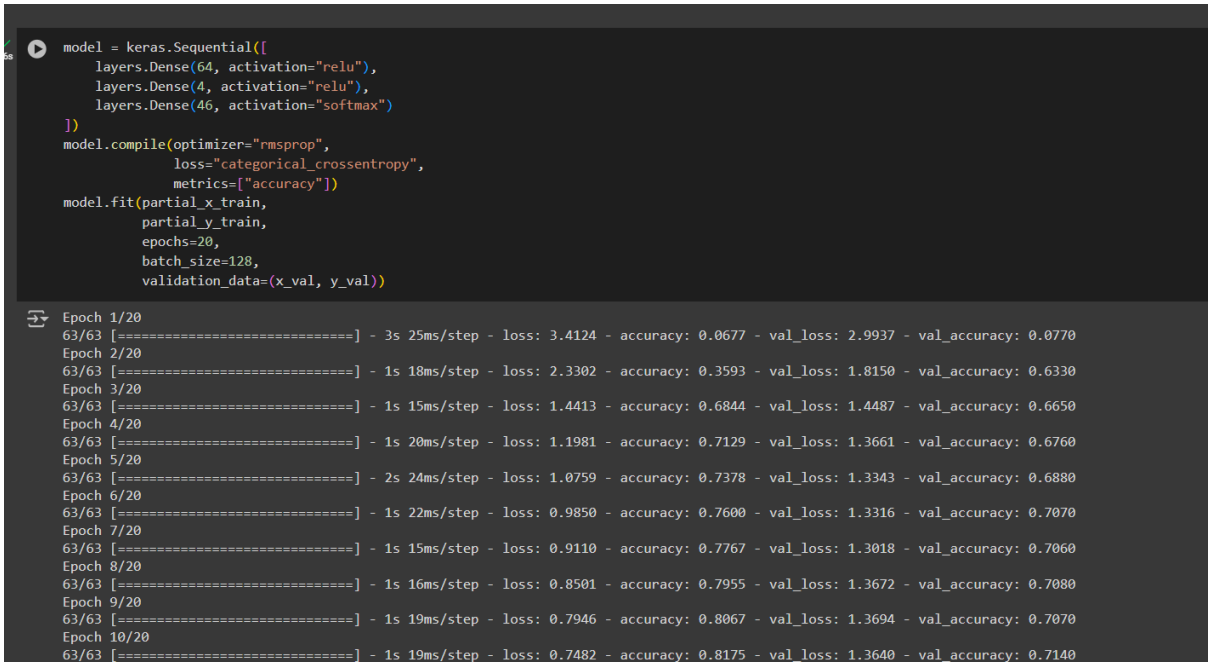
```
Epoch 1/9
18/18 [==============================] - 1s 44ms/step - loss: 2.6921 - accuracy: 0.5202
Epoch 2/9
18/18 [==============================] - 1s 51ms/step - loss: 1.5226 - accuracy: 0.6805
Epoch 3/9
18/18 [==============================] - 1s 49ms/step - loss: 1.1575 - accuracy: 0.7446
Epoch 4/9
18/18 [==============================] - 1s 51ms/step - loss: 0.9391 - accuracy: 0.7988
Epoch 5/9
18/18 [==============================] - 1s 73ms/step - loss: 0.7762 - accuracy: 0.8363
Epoch 6/9
18/18 [==============================] - 1s 70ms/step - loss: 0.6466 - accuracy: 0.8625
Epoch 7/9
18/18 [==============================] - 1s 66ms/step - loss: 0.5348 - accuracy: 0.8881
Epoch 8/9
18/18 [==============================] - 1s 43ms/step - loss: 0.4527 - accuracy: 0.9059
Epoch 9/9
18/18 [==============================] - 1s 39ms/step - loss: 0.3811 - accuracy: 0.9186
71/71 [==============================] - 0s 3ms/step - loss: 0.9067 - accuracy: 0.7934
```

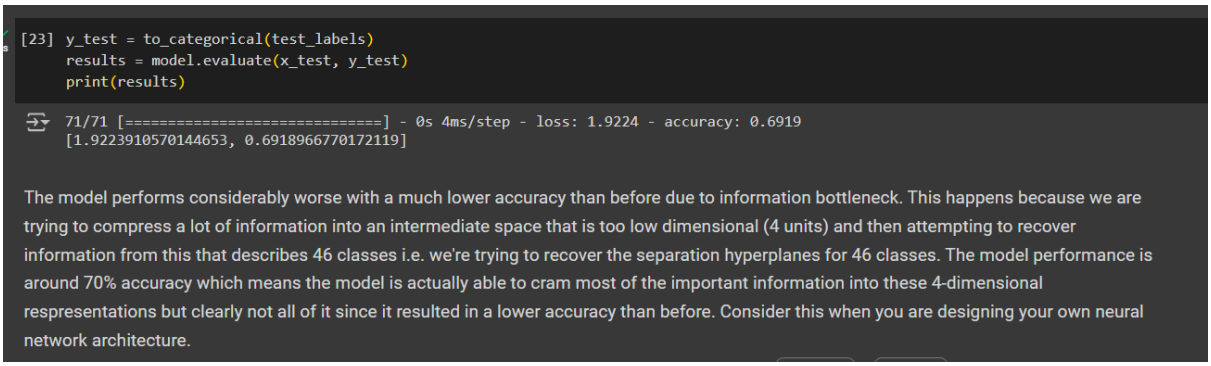
The results: The first number is the test loss and the second number is the test accuracy.

Information Bottleneck Experiment:

- **Exploration:** In an attempt to improve generalization and reduce overfitting, an information bottleneck layer with 4 neurons was added to the model architecture. This technique aims to retain essential information while restricting model complexity, potentially preventing overfitting.
- **Outcome:** Unfortunately, the model with the information bottleneck layer yielded a significantly lower test accuracy (0.67) compared to the previous model without the bottleneck. This suggests that the bottleneck layer might have excessively restricted the model's ability to learn the necessary classification patterns.

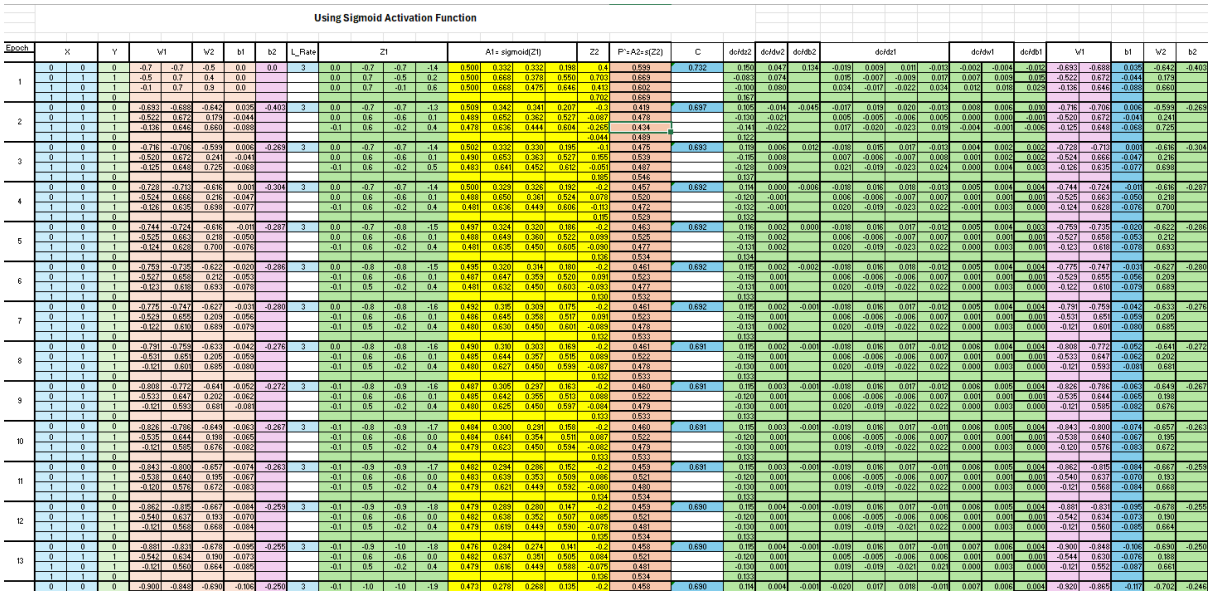






VII. WEEK 5

A. Resource: With assumption that you have access to Microsoft Excel, create a new template with the following setup. From there, you start implement the multilayer perceptron logistic regression for solving XOR function approximation based on the formulas discussed in the lecture.



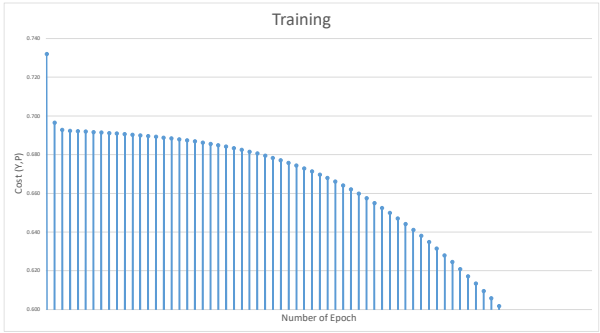
Using Sigmoid Activation Function

Epoch	X		Y	W1		W2	b1	b2	L_Rate	Z1				A1= sigmoid(Z1)				Z2	P'=A2=s(Z2)	C	dcdz2	dcdw2	dcdb2	dcdz1				dcdw1		dcdb1	W1		b1	W2	b2
1	0	0	0	-0.7	-0.7	-0.5	0.0	0.0	3	0.0	-0.7	-0.7	-1.4	0.500	0.332	0.332	0.198	0.4	0.599	0.732	0.150	0.047	0.134	-0.019	0.009	0.011	-0.013	-0.002	-0.004	-0.012	-0.693	-0.688	0.035	-0.642	-0.403
	0	1	1	-0.5	0.7	0.4	0.0			0.0	0.7	-0.5	0.2	0.500	0.668	0.378	0.550	0.703	0.683		-0.083	0.074		0.015	-0.007	-0.009	0.017	0.007	0.006	0.015	-0.522	0.672	-0.044	0.179	
	1	0	1	-0.1	0.7	0.9	0.0			0.0	0.7	-0.1	0.6	0.500	0.668	0.475	0.646	0.413	0.602		-0.100	0.080		0.034	-0.017	-0.022	0.034	0.012	0.018	0.029	-0.136	0.646	-0.088	0.660	
	1	1	0															0.702	0.669		0.167													
2	0	0	0	-0.693	-0.688	-0.642	0.035	-0.403	3	0.0	-0.7	-0.7	-1.3	0.500	0.342	0.341	0.207	-0.2	0.419	0.697	0.105	-0.014	-0.045	-0.017	0.019	0.020	-0.013	0.008	0.006	0.019	-0.716	-0.706	0.006	-0.599	-0.269
	0	1	1	-0.522	0.672	0.179	-0.044			0.0	0.6	-0.6	0.1	0.489	0.652	0.362	0.527	-0.087	0.478		-0.130	-0.021		0.005	-0.005	-0.006	0.005	0.000	0.000	-0.001	-0.520	0.672	-0.041	0.241	
	1	0	1	-0.136	0.646	0.660	-0.088			-0.1	0.6	-0.2	0.4	0.476	0.636	0.444	0.604	-0.265	0.434		-0.141	-0.022		0.017	-0.020	-0.023	0.019	-0.004	-0.001	-0.006	-0.125	0.648	-0.068	0.725	
	1	1	0															-0.044	0.489		0.123													
3	0	0	0	-0.716	-0.706	-0.599	0.006	-0.269	3	0.0	-0.7	-0.7	-1.4	0.502	0.332	0.330	0.195	-0.1	0.475	0.693	0.159	0.008	0.012	-0.018	0.015	0.017	-0.013	0.004	0.002	0.002	-0.728	-0.713	0.001	-0.616	-0.304
	0	1	1	-0.520	0.672	0.241	-0.041			0.0	0.6	-0.6	0.1	0.490	0.653	0.363	0.527	0.155	0.539		-0.195	0.008		0.007	-0.006	-0.007	0.008	0.001	0.002	0.002	-0.524	0.668	-0.047	0.216	
	1	0	1	-0.125	0.649	0.725	-0.069			-0.1	0.6	-0.2	0.5	0.483	0.641	0.452	0.612	-0.051	0.487		-0.128	0.009		0.021	-0.019	-0.023	0.024	0.000	0.004	0.003	-0.126	0.635	-0.077	0.699	
	1	1	0															0.105	0.546		0.127													
4	0	0	0	-0.728	-0.713	-0.616	0.001	-0.304	3	0.0	-0.7	-0.7	-1.4	0.500	0.329	0.326	0.192	-0.2	0.457	0.692	0.114	0.009	-0.006	-0.018	0.016	0.018	-0.013	0.005	0.004	0.004	-0.744	-0.724	-0.011	-0.616	-0.287
	0	1	1	-0.524	0.666	0.216	-0.047			0.0	0.6	-0.6	0.1	0.488	0.650	0.361	0.524	0.079	0.520		-0.120	-0.001		0.006	-0.006	-0.007	0.007	0.001	0.001	0.001	-0.525	0.663	-0.050	0.218	
	1	0	1	-0.126	0.635	0.698	-0.077			-0.1	0.6	-0.2	0.4	0.481	0.636	0.449	0.606	-0.110	0.472		-0.132	-0.001		0.020	-0.019	-0.023	0.022	-0.001	0.003	0.000	-0.124	0.628	-0.076	0.700	
	1	1	0															0.115	0.529		0.132													
5	0	0	0	-0.744	-0.724	-0.616	-0.011	-0.287	3	0.0	-0.7	-0.8	-1.5	0.497	0.324	0.320	0.186	-0.2	0.463	0.692	0.116	0.002	0.000	-0.018	0.016	0.017	-0.013	0.005	0.004	0.003	-0.759	-0.735	-0.020	-0.622	-0.286
	0	1	1	-0.525	0.663	0.218	-0.050			0.0	0.6	-0.6	0.1	0.488	0.649	0.360	0.522	0.099	0.525		-0.119	0.002		0.006	-0.006	-0.007	0.007	0.001	0.001	0.001	-0.527	0.658	-0.053	0.212	
	1	0	1	-0.124	0.628	0.700	-0.076			-0.1	0.6	-0.2	0.4	0.481	0.635	0.450	0.605	-0.090	0.477		-0.131	0.002		0.020	-0.019	-0.023	0.022	0.000	0.003	0.001	-0.123	0.618	-0.078	0.693	
	1	1	0															0.136	0.534		0.134													
6	0	0	0	-0.759	-0.735	-0.622	-0.020	-0.286	3	0.0	-0.8	-0.8	-1.5	0.495	0.320	0.314	0.180	-0.2	0.461	0.692	0.115	0.002	-0.002	-0.018	0.016	0.018	-0.012	0.005	0.004	0.004	-0.775	-0.747	-0.031	-0.627	-0.280
	0	1	1	-0.527	0.658	0.213	-0.053			-0.1	0.6	-0.6	0.1	0.487	0.647	0.359	0.520	0.091	0.523		-0.119	0.001		0.006	-0.006	-0.006	0.007	0.001	0.001	0.001	-0.529	0.655	-0.056	0.209	
	1	0	1	-0.123	0.638	0.693	-0.078			-0.1	0.5	-0.2	0.4	0.481	0.632	0.450	0.603	-0.093	0.477		-0.131	0.001		0.020	-0.019	-0.022	0.022	0.000	0.003	0.000	-0.122	0.618	-0.079	0.689	
	1	1	0															0.130	0.532		0.133													
7	0	0	0	-0.775	-0.747	-0.627	-0.031	-0.280	3	0.0	-0.8	-0.8	-1.6	0.492	0.315	0.309	0.175	-0.2	0.461	0.692	0.115	0.002	-0.001	-0.018	0.016	0.017	-0.012	0.005	0.004	0.004	-0.791	-0.759	-0.042	-0.633	-0.276
	0	1	1	-0.529	0.655	0.209	-0.056			-0.1	0.6	-0.6	0.1	0.486	0.645	0.358	0.517	0.091	0.523		-0.119	0.001		0.006	-0.006	-0.006	0.007	0.001	0.001	0.001	-0.531	0.653	-0.059	0.205	
	1	0	1	-0.122	0.630	0.689	-0.078			-0.1	0.5	-0.2	0.4	0.480	0.630	0.450	0.601	-0.089	0.478		-0.131	0.002		0.020	-0.019	-0.022	0.022	0.000	0.003	0.000	-0.121	0.601	-0.080	0.685	
	1	1	0															0.132	0.533		0.133													
8	0	0	0	-0.791	-0.759	-0.633	-0.042	-0.276	3	0.0	-0.8	-0.8	-1.6	0.490	0.310	0.303	0.169	-0.2	0.461	0.691	0.115	0.002	-0.001	-0.018	0.016	0.017	-0.012	0.006	0.004	0.004	-0.808	-0.772	-0.052	-0.641	-0.272
	0	1	1	-0.531	0.653	0.205	-0.059			-0.1	0.6	-0.6	0.1	0.485	0.644	0.357	0.516	0.089	0.522		-0.119	0.001		0.006	-0.006	-0.006	0.007	0.001	0.001	0.001	-0.533	0.647	-0.062	0.202	
	1	0	1	-0.121	0.631	0.685	-0.080			-0.1	0.5	-0.2	0.4	0.480	0.627	0.450	0.599	-0.087	0.478		-0.130	0.001		0.020	-0.019	-0.022	0.022	0.000	0.003	0.000	-0.121	0.593	-0.081	0.681	
	1	1	0															0.132	0.533		0.133													
9	0	0	0	-0.808	-0.772	-0.641	-0.052	-0.272	3	-0.1	-0.8	-0.9	-1.6	0.487	0.305	0.297	0.163	-0.2	0.460	0.691	0.115	0.003	-0.001	-0.018	0.016	0.017	-0.012	0.006	0.005	0.004	-0.826	-0.786	-0.063	-0.649	-0.267
	0	1	1	-0.533	0.647	0.202	-0.062			-0.1	0.6	-0.6	0.1	0.485	0.642	0.355	0.513	0.088	0.522		-0.120	0.001		0.006	-0.006	-0.006	0.007	0.001	0.001	0.001	-0.535	0.644	-0.065	0.198	
	1	0	1	-0.121	0.593	0.681	-0.081			-0.1	0.5	-0.2	0.4	0.480	0.625	0.450	0.597	-0.084	0.479		-0.130	0.001		0.020	-0.019	-0.022	0.022	0.000	0.003	0.000	-0.121	0.595	-0.082	0.676	
	1	1	0															0.133	0.533		0.133													
10	0	0	0	-0.826	-0.786	-0.649	-0.063	-0.267	3	-0.1	-0.8	-0.9	-1.7	0.484	0.300	0.291	0.158	-0.2	0.460	0.691	0.115	0.003	-0.001	-0.019	0.016	0.017	-0.011	0.006	0.005	0.004	-0.843	-0.800	-0.074	-0.657	-0.263
	0	0	1	-0.535	0.644	0.199	-0.065			-0.1	0.6	-0.6	0.0	0.484	0.641	0.354	0.511	0.087	0.522		-0.120	0.001		0.006	-0.005	-0.006	0.007	0.001	0.001	0.001	-0.538	0.640	-0.067	0.195	
	1	0	1	-0.121	0.585	0.676	-0.082			-0.1	0.5	-0.2	0.4	0.479	0.623	0.450	0.594	-0.082	0.479		-0.130	0.001		0.019	-0.019	-0.022	0.022	0.000	0.003	0.000	-0.120	0.576	-0.083	0.672	
	1	1	0															0.132	0.533		0.133													
11	0	0	0	-0.843	-0.800	-0.657	-0.074	-0.263	3	-0.1	-0.9	-0.9	-1.7	0.482	0.294	0.286	0.152	-0.2	0.459	0.691	0.115	0.003	-0.001	-0.019	0.016	0.017	-0.011	0.006	0.005	0.004	-0.862	-0.815	-0.084	-0.667	-0.259
	0	0	1	-0.538	0.640	0.195	-0.067			-0.1	0.6	-0.6	0.0	0.483	0.639	0.353	0.509	0.086	0.521		-0.120	0.001		0.006	-0.005	-0.006	0.007	0.001	0.001	0.001	-0.540	0.637	-0.070	0.193	
	1	0	1	-0.120	0.576	0.672	-0.083			-0.1	0.5	-0.2	0.4	0.479	0.621	0.449	0.592	-0.080	0.480		-0.130	0.001		0.019	-0.019	-0.022	0.022	0.000	0.003	0.000	-0.121	0.568	-0.084	0.668	
	1	1	0															0.134	0.534		0.133													
12	0	0	0	-0.862	-0.815	-0.667	-0.084	-0.259	3	-0.1	-0.9	-0.9	-1.8	0.479	0.289	0.280	0.147	-0.2	0.459	0.690	0.115	0.004	-0.001	-0.019	0.016	0.017	-0.011	0.006	0.005	0.004	-0.881	-0.831	-0.095	-0.678	-0.255
	0	0	1	-0.540	0.637	0.193	-0.070			-0.1	0.6	-0.6	0.0	0.482	0.638	0.352	0.507	0.085	0.521		-0.120	0.001		0.006	-0.005	-0.006	0.006	0.001	0.001	0.001	-0.542	0.634	-0.073	0.190	
	1	0	1	-0.121	0.568	0.668	-0.084			-0.1	0.5	-0.2	0.4	0.479	0.619	0.449	0.590	-0.078	0.481		-0.130	0.001		0.019	-0.019	-0.021	0.022	0.000	0.003	0.000	-0.121	0.560	-0.085	0.664	
	1	1	0															0.135	0.534		0.133													
13	0	0	0	-0.881	-0.831	-0.678	-0.095	-0.255	3	-0.1	-0.9	-1.0	-1.8	0.476	0.284	0.274	0.141	-0.2	0.458	0.690	0.115	0.004	-0.001	-0.019	0.016	0.017	-0.011	0.007	0.006	0.004	-0.900	-0.848	-0.106	-0.690	-0.250
	0	0	1	-0.542	0.634	0.190	-0.073			-0.1	0.6	-0.6	0.0	0.482	0.637	0.351	0.505	0.084	0.521		-0.120	0.001		0.006	-0.005	-0.006	0.006	0.001	0.001	0.001	-0.544	0.630	-0.076	0.188	
	1	0	1	-0.121	0.560	0.664	-0.085			-0.1	0.5	-0.2	0.4	0.479	0.616	0.449	0.588	-0.075	0.481		-0.130	0.001		0.019	-0.019	-0.021	0.021	0.000	0.003	0.000	-0.121	0.552	-0.087	0.661	
	1	1	0															0.136	0.534		0.133													
	0	0	0	-0.900	-0.848	-0.690	-0.106	-0.250	3	-0.1	-1.0	-1.0	-1.9	0.473	0.278	0.268	0.135	-0.2	0.458	0.690	0.114	0.004	-0.001	-0.020	0.017	0.018	-0.011	0.007	0.006	0.004	-0.920	-0.865	-0.117	-0.702	-0.246

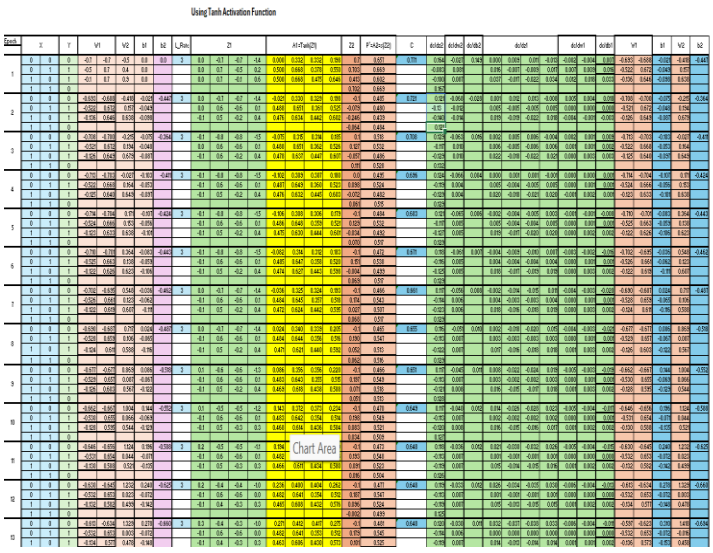
1. Expected outcome: You are required to plot the Cost function value at every epoch, up to 70 epochs. Your plot should be inline with the Figure 3.

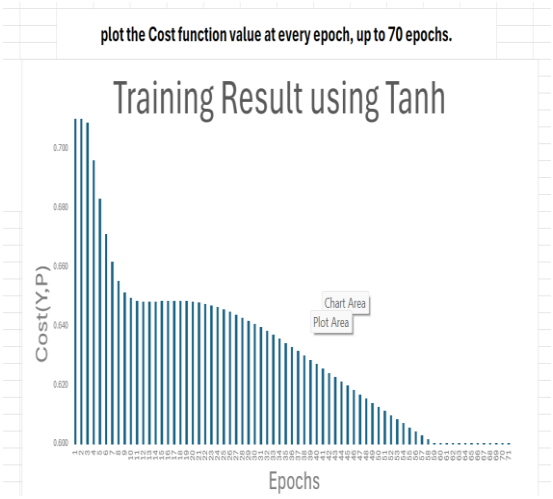
W1		b1	W2	b2
-0.693	-0.688	0.035	-0.642	-0.403
-0.522	0.672	-0.044	0.179	
-0.136	0.646	-0.088	0.660	
-0.716	-0.706	0.006	-0.599	-0.269
-0.520	0.672	-0.041	0.241	
-0.125	0.649	-0.068	0.725	
-0.728	-0.713	0.001	-0.616	-0.304
-0.524	0.668	-0.047	0.216	
-0.126	0.635	-0.077	0.699	
-0.744	-0.724	-0.011	-0.616	-0.287
-0.525	0.663	-0.050	0.218	
-0.124	0.628	-0.076	0.700	
-0.759	-0.735	-0.020	-0.622	-0.286
-0.527	0.658	-0.053	0.212	
-0.123	0.618	-0.078	0.693	
-0.775	-0.747	-0.031	-0.627	-0.280
-0.529	0.655	-0.056	0.209	
-0.122	0.610	-0.079	0.689	
-0.791	-0.759	-0.042	-0.633	-0.276
-0.529	0.655	-0.059	0.205	
-0.122	0.601	-0.080	0.685	
-0.808	-0.772	-0.052	-0.641	-0.272
-0.533	0.647	-0.062	0.202	
-0.121	0.593	-0.081	0.681	
-0.826	-0.786	-0.063	-0.649	-0.267
-0.535	0.644	-0.065	0.198	
-0.121	0.595	-0.082	0.676	

Cost function value at every epoch, up to 70 epochs.



2. Replace Sigmoid function with Tanh function. Explain your finding along with your analysis evidence.





Answer-

Impact of Activation Functions on Training:

This experiment compared the performance of two common activation functions, tanh and sigmoid, in the context of training our time series forecasting model. The initial cost (loss) value was lower for the sigmoid function (0.732) compared to the tanh function (0.771). While both functions converged to similar cost values by the end of training, the tanh function exhibited slightly higher cost throughout the middle epochs. Additionally, the cost reduction with tanh displayed more variability compared to the smoother decline observed with sigmoid.

These observations suggest potential advantages and disadvantages for each activation function:

- **Tanh:** May offer a faster initial reduction in cost, potentially leading to quicker convergence in the early stages of training.
- **Sigmoid:** Provides a smoother and potentially more stable cost reduction throughout training.

Therefore, the selection of the activation function might depend on the specific training goals. If faster initial convergence is prioritized, tanh could be a viable choice. However, if overall training stability is more crucial, sigmoid might be the preferred option.

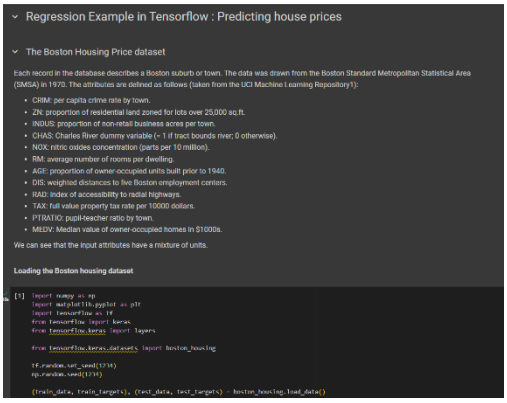
Week 5 Workshop

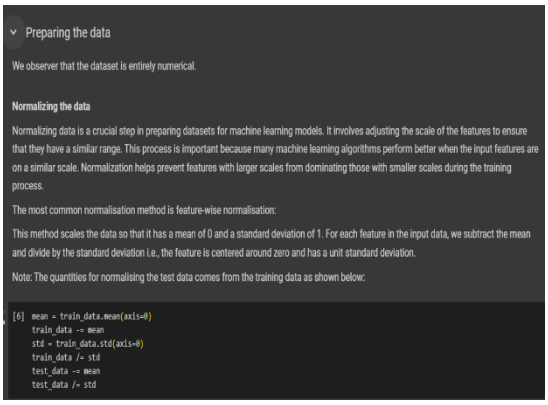
B. Please complete the following notebook and the attached workshop, record the outcome in your log book and demonstrate your understand to your best level.

https://colab.research.google.com/drive/1ggISnRkkSNBnYb8sxqylRBgmq3PulpmG?usp=sharing

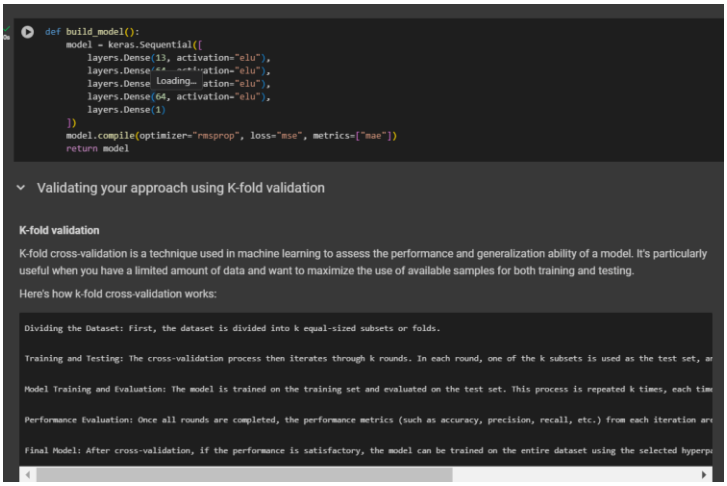
House Price Prediction with Deep Learning

- **Dataset:** The Boston House Prices dataset was employed for this project. This dataset consists of 506 data points with 13 features each, representing various characteristics of houses. The data was normalized to ensure all features are on a similar scale for effective model training.

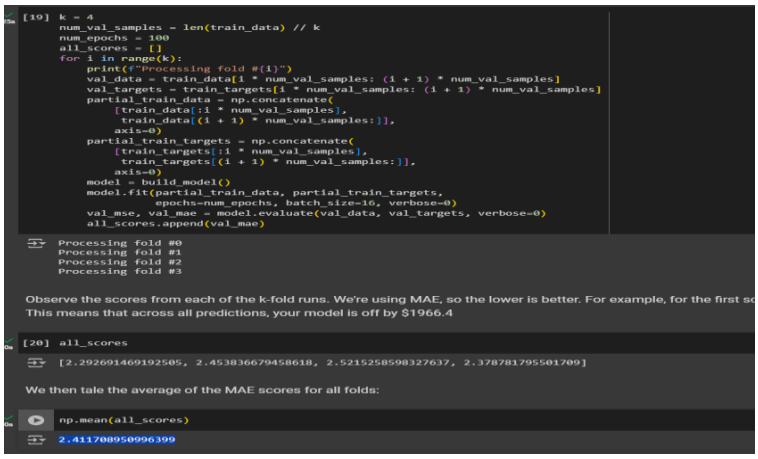




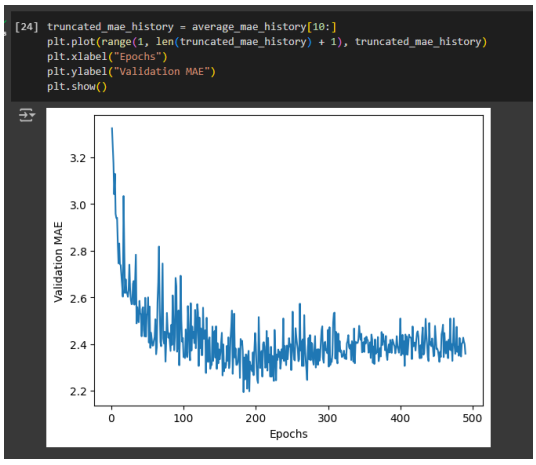
- **Model Architecture:** A deep learning model with five dense layers was constructed. The first layer has **13 neurons** corresponding to the number of input features. The subsequent layers have **64 neurons** each, designed to capture complex relationships between features and house prices.

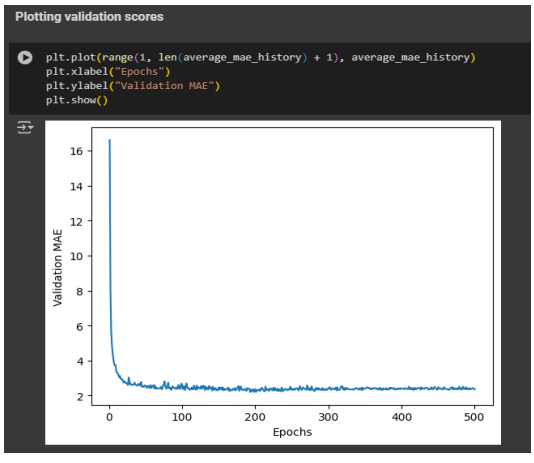


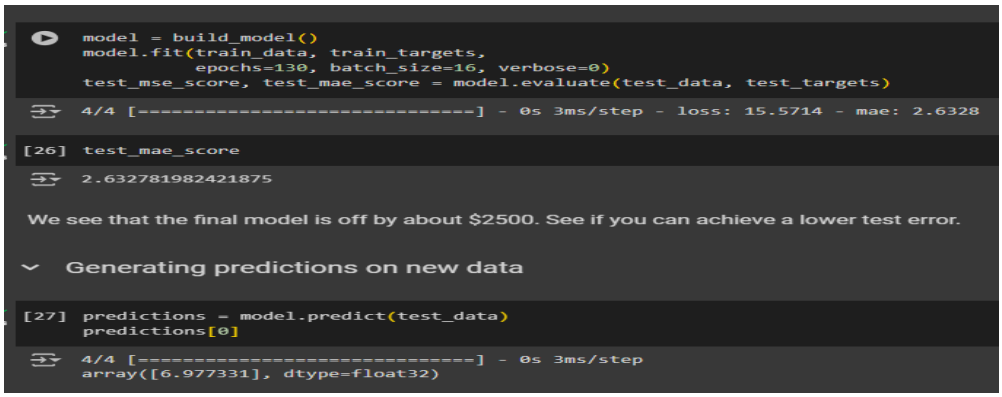
- **K-Fold Cross-Validation:** To evaluate model performance and mitigate overfitting, 4-fold cross-validation was implemented. This technique divides the data into four folds, trains the model on three folds while holding out the remaining fold for validation. This process is repeated four times, ensuring all data points are used for both training and validation.



- **Initial Training (100 Epochs and 500 epochs):** The model was initially trained for 100 epochs using K-fold cross-validation. The results from each fold were combined to obtain a mean Mean Absolute Error (MAE) of **2.41** on the validation sets. This score provides a baseline performance metric.
- **Hyperparameter Tuning and Training (130 Epochs):** To potentially improve performance, we further investigated the model's hyperparameters. Based on this analysis, the model was then trained for a longer duration (130 epochs) with the final chosen hyperparameters. However, this final training resulted in a higher MAE of **2.63**. We can visualize the trend of the MAE during training in the included plot







VIII. WEEK 6

A. Please complete the following notebook, record the outcome in your log book and demonstrate your understand to your best level.

https://colab.research.google.com/drive/1tgvOaNuQ6aC4zlQpc1H8xLdr0cbP6Ly7?usp=sharing

Copy of week_6_Common Practices in Deep Learning Using Tensorflow.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

Adding white-noise channels or all-zeros channels to MNIST

```
from tensorflow.keras.datasets import mnist
import numpy as np

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

train_images_with_noise_channels = np.concatenate(
    [train_images, np.random.random((len(train_images), 784))], axis=1)

train_images_with_zeros_channels = np.concatenate(
    [train_images, np.zeros((len(train_images), 784))], axis=1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
[2] len(train_images_with_noise_channels)
60000

[3] len(train_labels)
60000

[4] train_images_with_noise_channels.shape
(60000, 1568)
```

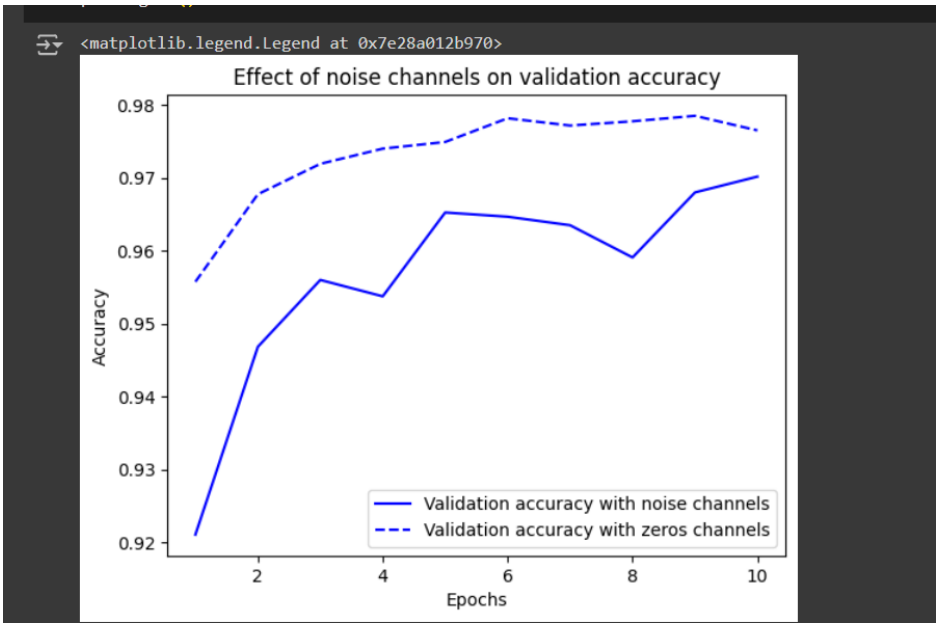
Training the same model on MNIST data with noise channels or all-zero channels

```
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow as tf

def get_model():
    model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

model = get_model()
history_noise = model.fit(
    train_images_with_noise_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2)

model = get_model()
history_zeros = model.fit(
    train_images_with_zeros_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2)
```

Plotting a validation accuracy comparison

```
import matplotlib.pyplot as plt
val_acc_noise = history_noise.history["val_accuracy"]
val_acc_zeros = history_zeros.history["val_accuracy"]
epochs = range(1, 11)
plt.plot(epochs, val_acc_noise, "b-",
         label="Validation accuracy with noise channels")
plt.plot(epochs, val_acc_zeros, "b--",
         label="Validation accuracy with zeros channels")
plt.title("Effect of noise channels on validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7e28a012b970>
```



Answer -Our investigated the impact of data noise on model performance. Two datasets were employed: one containing noise and one without. Both datasets were trained on a model featuring two dense layers with ReLU and sigmoid activation functions. The RMSProp optimizer was utilized, and training proceeded for 10 epochs. Subsequent analysis of validation accuracy revealed a clear trend. The dataset absent of noise demonstrably achieved superior validation accuracy compared to the dataset containing noise. This finding underscores the importance of data quality in machine learning, as noise can significantly impede model performance.

Now we have build couple of models for experiments -

Training a MNIST model with an incorrectly high learning rate

Let's set the learning rate = 1 in the below cell. We can see that this model reaches a low training and validation accuracy and cannot get past that.

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1.),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

```
Epoch 1/10
375/375 [==============================] - 2s 4ms/step - loss: 695.2733 - accuracy: 0.3734 - val_loss: 2.2379 - val_accuracy: 0.2432
Epoch 2/10
375/375 [==============================] - 1s 3ms/step - loss: 2.9671 - accuracy: 0.2527 - val_loss: 2.9329 - val_accuracy: 0.2317
Epoch 3/10
375/375 [==============================] - 1s 3ms/step - loss: 2.4130 - accuracy: 0.2141 - val_loss: 2.3540 - val_accuracy: 0.2476
Epoch 4/10
375/375 [==============================] - 1s 3ms/step - loss: 2.6342 - accuracy: 0.2092 - val_loss: 2.1454 - val_accuracy: 0.1928
```

1.Model with High learning rate – This was absolutely disaster we cannot reach to global minima we reached only 0.21 validation accuracy.

The same model with a more appropriate learning rate

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1e-2),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

```
Epoch 1/10
375/375 [==============================] - 2s 4ms/step - loss: 0.3629 - accuracy: 0.9118 - val_loss: 0.1518 - val_accuracy: 0.9548
Epoch 2/10
375/375 [==============================] - 2s 4ms/step - loss: 0.1284 - accuracy: 0.9638 - val_loss: 0.1752 - val_accuracy: 0.9564
Epoch 3/10
375/375 [==============================] - 1s 4ms/step - loss: 0.0975 - accuracy: 0.9737 - val_loss: 0.1548 - val_accuracy: 0.9679
Epoch 4/10
375/375 [==============================] - 1s 3ms/step - loss: 0.0818 - accuracy: 0.9792 - val_loss: 0.1549 - val_accuracy: 0.9701
Epoch 5/10
375/375 [==============================] - 1s 3ms/step - loss: 0.0688 - accuracy: 0.9824 - val_loss: 0.2219 - val_accuracy: 0.9625
Epoch 6/10
375/375 [==============================] - 1s 3ms/step - loss: 0.0634 - accuracy: 0.9849 - val_loss: 0.2009 - val_accuracy: 0.9661
```

2.Model with appropriate learning rate- This model perform very well we received 0.976 validation accuracy

A simple logistic regression on MNIST

This model contains one layer.

```
[9] model = keras.Sequential([layers.Dense(10, activation="softmax")])
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    history_small_model = model.fit(
        train_images, train_labels,
        epochs=20,
        batch_size=128,
        validation_split=0.2)
```

```
Epoch 1/20
375/375 [==============================] - 2s 4ms/step - loss: 0.6652 - accuracy: 0.8373 - val_loss: 0.3627 - val_accuracy: 0.9016
Epoch 2/20
375/375 [==============================] - 1s 4ms/step - loss: 0.3532 - accuracy: 0.9029 - val_loss: 0.3100 - val_accuracy: 0.9147
Epoch 3/20
375/375 [==============================] - 1s 3ms/step - loss: 0.3178 - accuracy: 0.9110 - val_loss: 0.2938 - val_accuracy: 0.9197
Epoch 4/20
375/375 [==============================] - 1s 3ms/step - loss: 0.3018 - accuracy: 0.9155 - val_loss: 0.2839 - val_accuracy: 0.9219
Epoch 5/20
375/375 [==============================] - 1s 3ms/step - loss: 0.2924 - accuracy: 0.9185 - val_loss: 0.2793 - val_accuracy: 0.9219
```

3.Simple model with one layer – Validation accuracy we get around 0.92 which is decent model.

Lets increase the capacity of the model and evaluate its performance compared to the model with a single layer.

```
[11] model = keras.Sequential([
         layers.Dense(96, activation="relu"),
         layers.Dense(96, activation="relu"),
         layers.Dense(10, activation="softmax"),
     ])
     model.compile(optimizer="rmsprop",
                   loss="sparse_categorical_crossentropy",
                   metrics=["accuracy"])
     history_large_model = model.fit(
         train_images, train_labels,
         epochs=20,
         batch_size=128,
         validation_split=0.2)
```

```
Epoch 1/20
375/375 [==============================] - 2s 4ms/step - loss: 0.3657 - accuracy: 0.8956 - val_loss: 0.1905 - val_accuracy: 0.9454
Epoch 2/20
375/375 [==============================] - 1s 3ms/step - loss: 0.1593 - accuracy: 0.9528 - val_loss: 0.1427 - val_accuracy: 0.9588
Epoch 3/20
375/375 [==============================] - 2s 4ms/step - loss: 0.1111 - accuracy: 0.9668 - val_loss: 0.1103 - val_accuracy: 0.9668
Epoch 4/20
375/375 [==============================] - 2s 4ms/step - loss: 0.0852 - accuracy: 0.9739 - val_loss: 0.0997 - val_accuracy: 0.9704
Epoch 5/20
375/375 [==============================] - 1s 3ms/step - loss: 0.0687 - accuracy: 0.9791 - val_loss: 0.0987 - val_accuracy: 0.9716
```

4. 3 layer model - this model perform very well. Get around 0.97 validation accuracy but tends to overfit after 8 epochs

Now we have comparing three model – Original model , Less neuron model,High neuron layer

Original model

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), _ = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
train_data = vectorize_sequences(train_data)

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_original = model.fit(train_data, train_labels,
                             epochs=20, batch_size=512, validation_split=0.4)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 [==============================] - 0s 0us/step
Epoch 1/20
30/30 [==============================] - 4s 65ms/step - loss: 0.5412 - accuracy: 0.7629 - val_loss: 0.3932 - val_accuracy: 0.8695
Epoch 2/20
30/30 [==============================] - 1s 25ms/step - loss: 0.3254 - accuracy: 0.8903 - val_loss: 0.3089 - val_accuracy: 0.8842
Epoch 3/20
30/30 [==============================] - 1s 30ms/step - loss: 0.2417 - accuracy: 0.9193 - val_loss: 0.2826 - val_accuracy: 0.8900
Epoch 4/20
30/30 [==============================] - 1s 30ms/step - loss: 0.1928 - accuracy: 0.9373 - val_loss: 0.2913 - val_accuracy: 0.8831
```

Version of the model with lower capacity

```
model = keras.Sequential([
    layers.Dense(4, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_smaller_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

```
Epoch 1/20
30/30 [==============================] - 3s 62ms/step - loss: 0.6320 - accuracy: 0.7449 - val_loss: 0.5654 - val_accuracy: 0.8539
Epoch 2/20
30/30 [==============================] - 1s 22ms/step - loss: 0.5043 - accuracy: 0.8711 - val_loss: 0.4626 - val_accuracy: 0.8692
Epoch 3/20
30/30 [==============================] - 1s 24ms/step - loss: 0.4065 - accuracy: 0.8911 - val_loss: 0.3919 - val_accuracy: 0.8732
Epoch 4/20
30/30 [==============================] - 1s 22ms/step - loss: 0.3348 - accuracy: 0.9038 - val_loss: 0.3482 - val_accuracy: 0.8763
Epoch 5/20
30/30 [==============================] - 1s 25ms/step - loss: 0.2820 - accuracy: 0.9172 - val_loss: 0.3204 - val_accuracy: 0.8784
Epoch 6/20
30/30 [==============================] - 1s 21ms/step - loss: 0.2425 - accuracy: 0.9259 - val_loss: 0.2910 - val_accuracy: 0.8883
Epoch 7/20
30/30 [==============================] - 1s 24ms/step - loss: 0.2121 - accuracy: 0.9325 - val_loss: 0.2815 - val_accuracy: 0.8906
Epoch 8/20
30/30 [==============================] - 1s 22ms/step - loss: 0.1885 - accuracy: 0.9397 - val_loss: 0.2750 - val_accuracy: 0.8934
```

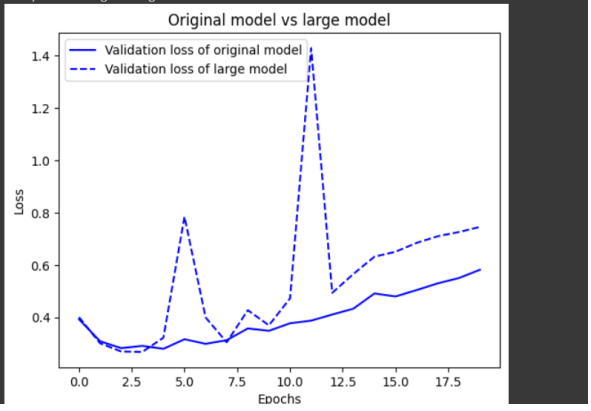
Version of the model with higher capacity

```
[15] model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(512, activation="relu"),
        layers.Dense(1, activation="sigmoid")
    ])
    model.compile(optimizer="rmsprop",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    history_larger_model = model.fit(
        train_data, train_labels,
        epochs=20, batch_size=512, validation_split=0.4)
```

```
Epoch 1/20
30/30 [==============================] - 3s 74ms/step - loss: 0.5674 - accuracy: 0.7121 - val_loss: 0.4010 - val_accuracy: 0.8248
Epoch 2/20
30/30 [==============================] - 1s 26ms/step - loss: 0.3255 - accuracy: 0.8662 - val_loss: 0.3013 - val_accuracy: 0.8762
Epoch 3/20
30/30 [==============================] - 1s 25ms/step - loss: 0.2456 - accuracy: 0.8987 - val_loss: 0.2698 - val_accuracy: 0.8894
Epoch 4/20
30/30 [==============================] - 1s 25ms/step - loss: 0.1799 - accuracy: 0.9323 - val_loss: 0.2682 - val_accuracy: 0.8923
Epoch 5/20
30/30 [==============================] - 1s 25ms/step - loss: 0.1314 - accuracy: 0.9527 - val_loss: 0.3224 - val_accuracy: 0.8799
Epoch 6/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0998 - accuracy: 0.9668 - val_loss: 0.7850 - val_accuracy: 0.7730
```

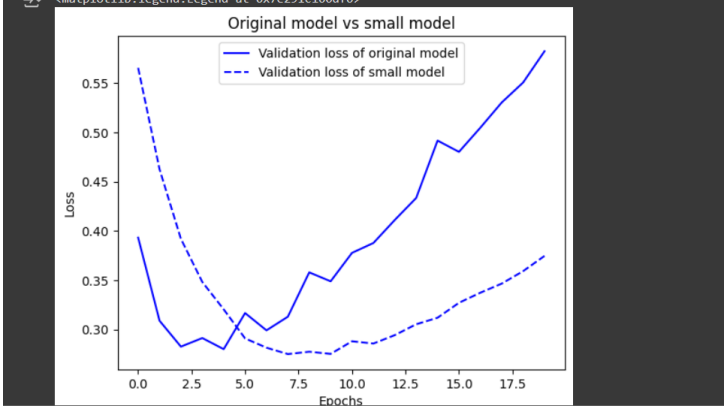
```
val_loss_original = history_original.history["val_loss"]
val_loss_large = history_larger_model.history["val_loss"]
epochs = range(0, 20)
plt.plot(epochs, val_loss_original, "b-",
         label="Validation loss of original model")
plt.plot(epochs, val_loss_large, "b--",
         label="Validation loss of large model")
plt.title("Original model vs large model")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7e291c102dd0>
```




```
val_loss_original = history_original.history["val_loss"]
val_loss_small = history_smaller_model.history["val_loss"]
epochs = range(0, 20)
plt.plot(epochs, val_loss_original, "b-",
         label="Validation loss of original model")
plt.plot(epochs, val_loss_small, "b--",
         label="Validation loss of small model")
plt.title("Original model vs small model")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7e291c100df0>
```



The impact of the number of neurons on model training was explored by training models with varying neuron counts in each layer. We analyzed the resulting graphs (refer to Figure above) and identified several key observations.

- **Original vs. Large Model:** Compared to the original model, the large model with a higher number of neurons exhibited signs of inconsistency and a tendency to overfit readily.
- **Original vs. Small Model:** The small model, with a lower number of neurons per layer, demonstrated slower convergence. While it eventually reached a minimum point around 8-10 epochs, it also displayed overfitting tendencies thereafter.

Now we add some penalties to over existing models

1. **Apply L2 regularization**
2. **Apply Dropouts**

With this it help to stop overfitting problem in our model.

Let compare both technique with our model

Adding weight regularization

Adding L2 weight regularization to the model

```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_l2_reg = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

```
Epoch 1/20
30/30 [==============================] - 3s 85ms/step - loss: 0.6353 - accuracy: 0.7647 - val_loss: 0.5002 - val_accuracy: 0.8656
Epoch 2/20
30/30 [==============================] - 1s 25ms/step - loss: 0.4284 - accuracy: 0.8908 - val_loss: 0.4127 - val_accuracy: 0.8729
Epoch 3/20
30/30 [==============================] - 1s 25ms/step - loss: 0.3451 - accuracy: 0.9099 - val_loss: 0.3654 - val_accuracy: 0.8881
Epoch 4/20
30/30 [==============================] - 1s 24ms/step - loss: 0.3003 - accuracy: 0.9214 - val_loss: 0.3516 - val_accuracy: 0.8897
Epoch 5/20
30/30 [==============================] - 1s 21ms/step - loss: 0.2775 - accuracy: 0.9293 - val_loss: 0.3783 - val_accuracy: 0.8720
Epoch 6/20
```

Adding more layers and changing the number of units in each layer:

```
from tensorflow.keras import regularizers
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(32, kernel_regularizer=regularizers.l2(0.002), activation="relu"),
    layers.Dense(32, kernel_regularizer=regularizers.l2(0.002), activation="relu"),
    layers.Dense(16, kernel_regularizer=regularizers.l2(0.002), activation="relu"),
    layers.Dense(16, kernel_regularizer=regularizers.l2(0.002), activation="relu"),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop", loss="binary_crossentropy", metrics=["accuracy"])

history_l2_reg = model.fit(train_data, train_labels, epochs=20, batch_size=512, validation_split=0.4)
```

```
Epoch 1/20
30/30 [==============================] - 3s 64ms/step - loss: 0.7813 - accuracy: 0.7330 - val_loss: 0.6008 - val_accuracy: 0.8526
Epoch 2/20
30/30 [==============================] - 1s 23ms/step - loss: 0.5180 - accuracy: 0.8760 - val_loss: 0.4855 - val_accuracy: 0.8792
Epoch 3/20
30/30 [==============================] - 1s 25ms/step - loss: 0.4177 - accuracy: 0.9055 - val_loss: 0.4814 - val_accuracy: 0.8694
Epoch 4/20
30/30 [==============================] - 1s 23ms/step - loss: 0.3790 - accuracy: 0.9157 - val_loss: 0.4360 - val_accuracy: 0.8825
Epoch 5/20
30/30 [==============================] - 1s 35ms/step - loss: 0.3443 - accuracy: 0.9265 - val_loss: 0.4258 - val_accuracy: 0.8876
Epoch 6/20
30/30 [==============================] - 1s 32ms/step - loss: 0.3146 - accuracy: 0.9384 - val_loss: 0.5285 - val_accuracy: 0.8499
Epoch 7/20
30/30 [==============================] - 1s 41ms/step - loss: 0.2963 - accuracy: 0.9424 - val_loss: 0.5345 - val_accuracy: 0.8523
Epoch 8/20
30/30 [==============================] - 1s 24ms/step - loss: 0.2828 - accuracy: 0.9491 - val_loss: 0.4673 - val_accuracy: 0.8743
Epoch 9/20
```

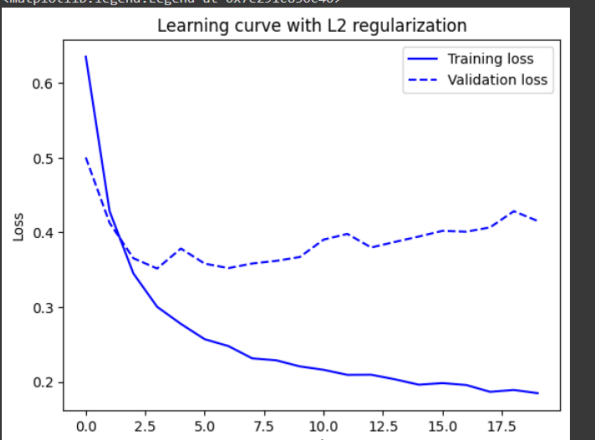
Adding dropout to the IMDB model

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_dropout = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

```
Epoch 1/20
30/30 [==============================] - 5s 67ms/step - loss: 0.6439 - accuracy: 0.6185 - val_loss: 0.5347 - val_accuracy: 0.8414
Epoch 2/20
30/30 [==============================] - 1s 21ms/step - loss: 0.5129 - accuracy: 0.7663 - val_loss: 0.4232 - val_accuracy: 0.8660
Epoch 3/20
30/30 [==============================] - 1s 26ms/step - loss: 0.4293 - accuracy: 0.8248 - val_loss: 0.3435 - val_accuracy: 0.8821
Epoch 4/20
30/30 [==============================] - 1s 31ms/step - loss: 0.3671 - accuracy: 0.8620 - val_loss: 0.3045 - val_accuracy: 0.8854
Epoch 5/20
30/30 [==============================] - 1s 41ms/step - loss: 0.3130 - accuracy: 0.8831 - val_loss: 0.2842 - val_accuracy: 0.8897
Epoch 6/20
30/30 [==============================] - 1s 25ms/step - loss: 0.2822 - accuracy: 0.8986 - val_loss: 0.2766 - val_accuracy: 0.8867
Epoch 7/20
30/30 [==============================] - 1s 23ms/step - loss: 0.2466 - accuracy: 0.9159 - val_loss: 0.2830 - val_accuracy: 0.8827
Epoch 8/20
30/30 [==============================] - 1s 22ms/step - loss: 0.2232 - accuracy: 0.9232 - val_loss: 0.2792 - val_accuracy: 0.8918
Epoch 9/20
30/30 [==============================] - 1s 22ms/step - loss: 0.1998 - accuracy: 0.9305 - val_loss: 0.2929 - val_accuracy: 0.8903
Epoch 10/20
30/30 [==============================] - 1s 23ms/step - loss: 0.1789 - accuracy: 0.9384 - val_loss: 0.3380 - val_accuracy: 0.8785
Epoch 11/20
```

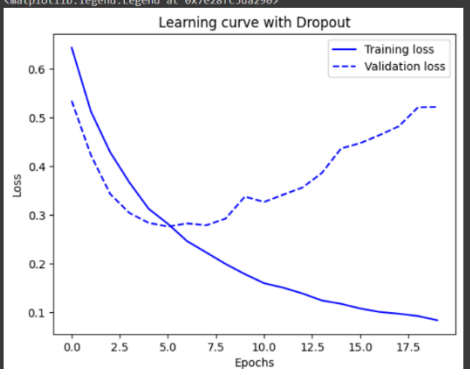
```
train_loss_l2 = history_l2_reg.history["loss"]
val_loss_l2 = history_l2_reg.history["val_loss"]
epochs = range(0, 20)
plt.plot(epochs, train_loss_l2, "b-",
         label="Training loss")
plt.plot(epochs, val_loss_l2, "b--",
         label="Validation loss")
plt.title("Learning curve with L2 regularization")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7e291c830c40>
```




```
train_loss_d = history_dropout.history["loss"]
val_loss_d = history_dropout.history["val_loss"]
epochs = range(0, 20)
plt.plot(epochs, train_loss_d, "b-",
         label="Training loss")
plt.plot(epochs, val_loss_d, "b--",
         label="Validation loss")
plt.title("Learning curve with Dropout")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7e28fc5da290>
```



L2 Regularization and Overfitting: We investigated the effectiveness of L2 regularization in mitigating overfitting. A model with three dense layers, each containing 16 neurons, was implemented. L2 regularization was applied to the first and second layers. Training the model revealed a trade-off between training and validation performance. While the model achieved good performance on the training data (refer to Figure), the validation loss remained high, indicating overfitting. This overfitting tendency began to manifest as early as 4 epochs.

In an attempt to address this issue, we experimented with increasing the model's capacity by adding additional dense layers and neurons. However, this approach unfortunately did not yield the desired outcome. The model continued to overfit from the 4th epoch onwards.

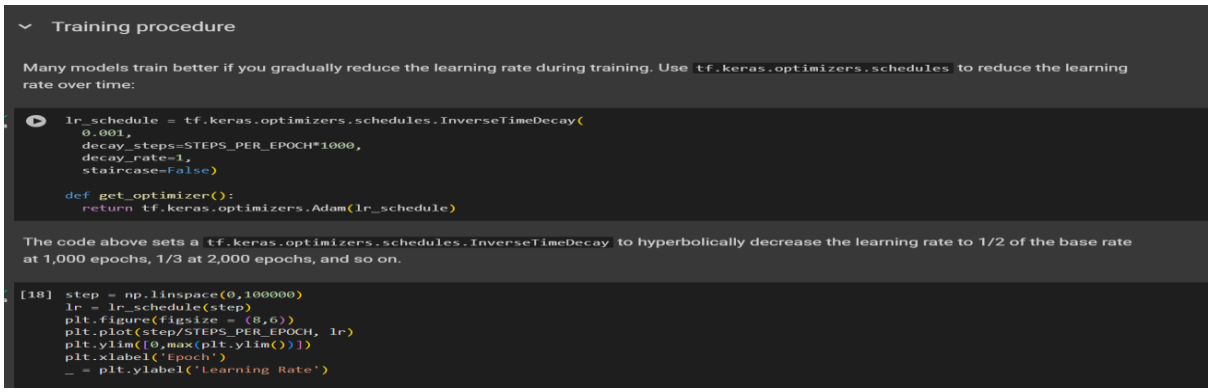
Dropout for Overfitting Control: To further explore methods for controlling overfitting, we employed dropout with a rate of 0.5 in each layer of the same three-layer, 16-neuron-per-layer model used previously. The model's performance, visualized in Figure ,exhibited similarities to the L2 regularization case. The training data loss remained low, indicating good fit. However, validation loss started to increase after 8 epochs, suggesting the onset of overfitting. While dropout did not entirely eliminate overfitting, it provided a slight improvement compared to L2 regularization in this scenario.

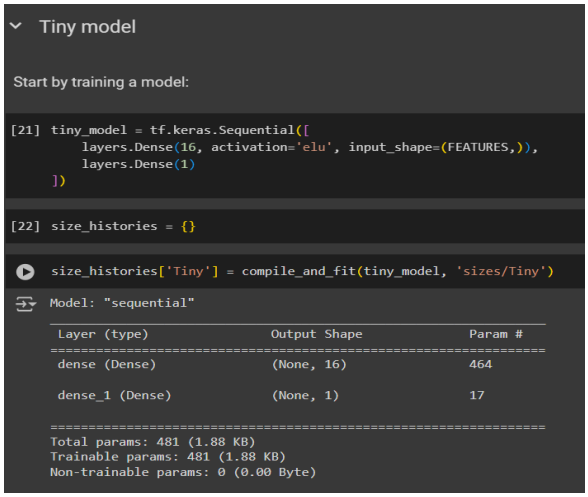
IX. WEEK 7

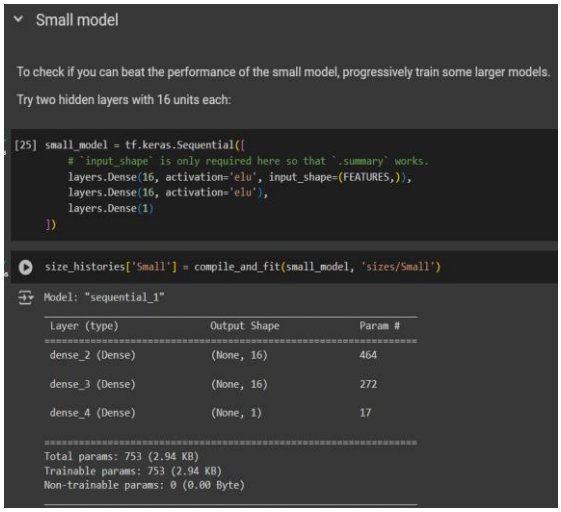
A. Please complete the following notebook, record the outcome in your log book and demonstrate your understand to your best level.

https://colab.research.google.com/drive/1bzQU-y0sRQ0dUHy-d_z_nZMQS8fgLsMG?usp=sharing

This project investigated the impact of model architecture and learning rate schedules on training performance. We employed TensorFlow's `tf.keras.optimizers.schedules` method to implement a gradually reducing learning rate during training. Experiments were conducted with various model architectures, including tiny, small, medium, and large models.







Medium model

Now try three hidden layers with 64 units each:

```
[27] medium_model = tf.keras.Sequential([
        layers.Dense(64, activation='elu', input_shape=(FEATURES,)),
        layers.Dense(64, activation='elu'),
        layers.Dense(64, activation='elu'),
        layers.Dense(1)
    ])
```

And train the model using the same data:

```
size_histories['Medium']  = compile_and_fit(medium_model, "sizes/Medium")
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 64)                1856

 dense_6 (Dense)             (None, 64)                4160

 dense_7 (Dense)             (None, 64)                4160

 dense_8 (Dense)             (None, 1)                 65

=================================================================
Total params: 10241 (40.00 KB)
Trainable params: 10241 (40.00 KB)
Non-trainable params: 0 (0.00 Byte)
```

Large model

As an exercise, you can create an even larger model and check how quickly it begins overfitting. Next, add to this benchmark a network that has much more capacity, far more than the problem would warrant:

```
[29] large_model = tf.keras.Sequential([
        layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
        layers.Dense(512, activation='elu'),
        layers.Dense(512, activation='elu'),
        layers.Dense(512, activation='elu'),
        layers.Dense(1)
    ])
```

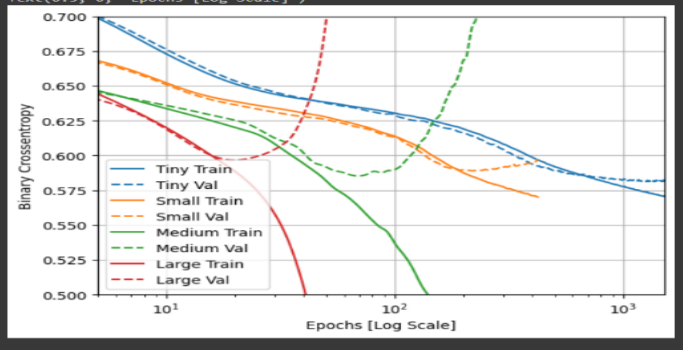
And, again, train the model using the same data:

```
size_histories['large'] = compile_and_fit(large_model, "sizes/large")
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 512)               14848

 dense_10 (Dense)            (None, 512)               262656

 dense_11 (Dense)            (None, 512)               262656

 dense_12 (Dense)            (None, 512)               262656

 dense_13 (Dense)            (None, 1)                 513

=================================================================
Total params: 803329 (3.06 MB)
Trainable params: 803329 (3.06 MB)
```

```
plotter.plot(size_histories)
a = plt.xscale('log')
plt.xlim([5, max(plt.xlim())])
plt.ylim([0.5, 0.7])
plt.xlabel("Epochs [Log Scale]")
```

```
Text(0.5, 0, 'Epochs [Log Scale]')
```



Impact of Model Architecture on Overfitting: Our investigation into the influence of model architecture on overfitting involved training models of varying complexity. The experiment utilized a common dataset and a gradually decreasing learning rate schedule implemented through TensorFlow's `tf.keras.optimizers.schedules` method.

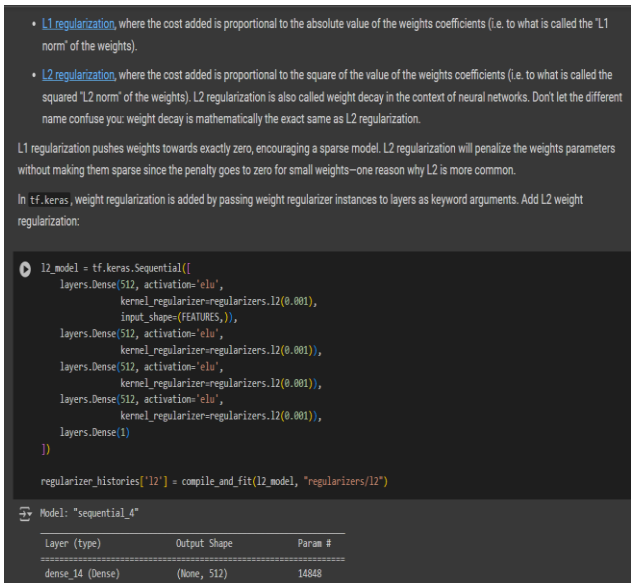
- **Tiny Model:** This model comprised two dense layers, each containing 16 neurons.
- **Small Model:** This model featured three dense layers, each with 16 neurons.
- **Medium Model:** This model possessed four dense layers, each containing 64 neurons.
- **Large Model:** This model consisted of five dense layers, with each layer boasting 512 neurons.

The resulting validation loss curves (refer to Figure) revealed distinct patterns across the models. While the tiny and small models exhibited relatively stable validation loss throughout the training process, indicating they avoided overfitting, the medium and large models displayed a tendency to overfit after a certain number of epochs. This suggests that increased model capacity can exacerbate overfitting if not addressed with appropriate regularization techniques.

Addressing Overfitting: In our efforts to mitigate overfitting, we explored the application of several regularization techniques. These techniques included:

- **L2 Regularization:** This technique penalizes models for having large weights, promoting simpler models that are less likely to overfit.
- **Dropout:** This technique randomly drops out a certain percentage of neurons during training, preventing them from co-adapting too strongly and promoting model robustness.

We also investigated the potential benefits of combining these techniques for enhanced overfitting control.



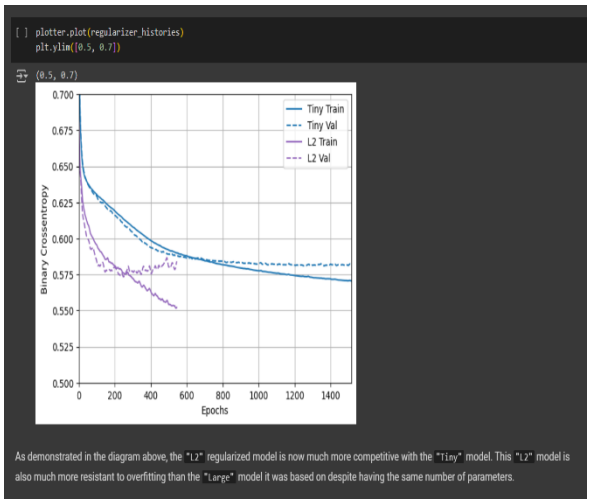
- L1 regularization, where the cost added is proportional to the absolute value of the weights coefficients (i.e. to what is called the "L1 norm" of the weights).
- L2 regularization, where the cost added is proportional to the square of the value of the weights coefficients (i.e. to what is called the squared "L2 norm" of the weights). L2 regularization is also called weight decay in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the exact same as L2 regularization.

L1 regularization pushes weights towards exactly zero, encouraging a sparse model. L2 regularization will penalize the weights parameters without making them sparse since the penalty goes to zero for small weights—one reason why L2 is more common.

In `tf.keras`, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Add L2 weight regularization:

```
l2_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001),
                 input_shape=(FEATURES,)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(512, activation='elu',
                 kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1)
])

regularizer_histories['l2'] = compile_and_fit(l2_model, "regularizers/l2")
```

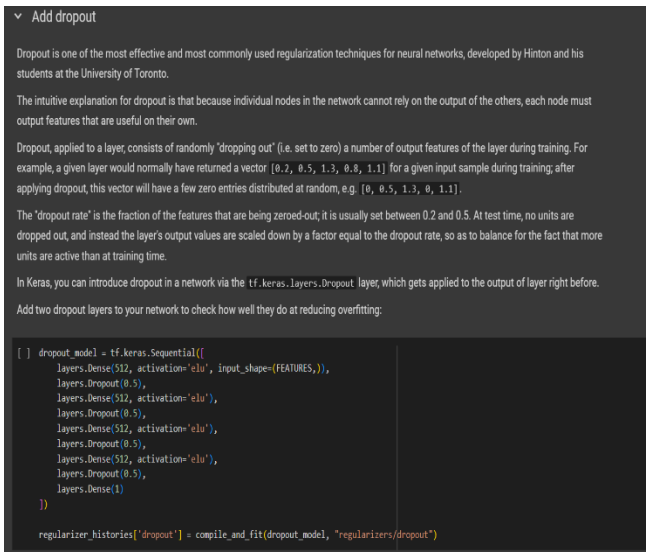
```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_14 (Dense)            (None, 512)               14848
```



```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```


As demonstrated in the diagram above, the `"L2"` regularized model is now much more competitive with the `"Tiny"` model. This `"L2"` model is also much more resistant to overfitting than the `"Large"` model it was based on despite having the same number of parameters.



Add dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Hinton and his students at the University of Toronto.

The intuitive explanation for dropout is that because individual nodes in the network cannot rely on the output of the others, each node must output features that are useful on their own.

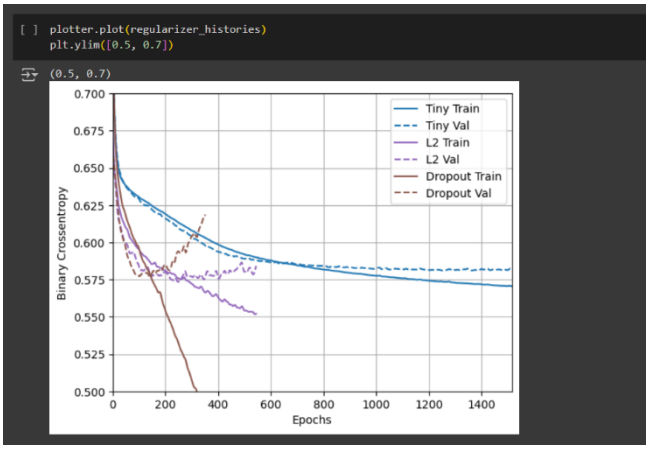
Dropout, applied to a layer, consists of randomly "dropping out" (i.e. set to zero) a number of output features of the layer during training. For example, a given layer would normally have returned a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training; after applying dropout, this vector will have a few zero entries distributed at random, e.g. `[0, 0.5, 1.3, 0, 1.1]`.

The "dropout rate" is the fraction of the features that are being zeroed-out; it is usually set between 0.2 and 0.5. At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

In Keras, you can introduce dropout in a network via the `tf.keras.layers.Dropout` layer, which gets applied to the output of layer right before.

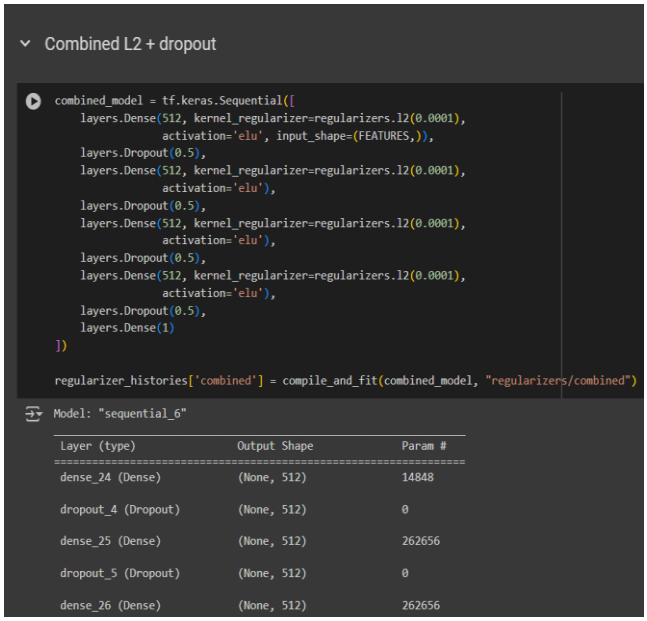
Add two dropout layers to your network to check how well they do at reducing overfitting:

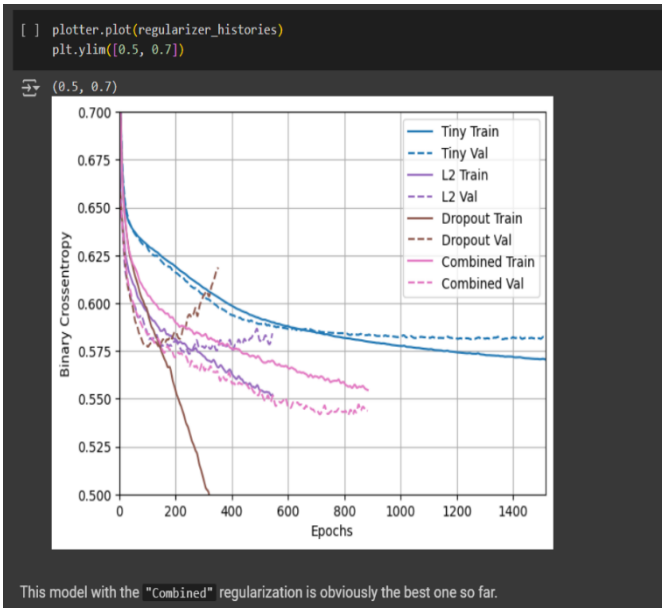
```
dropout_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])

regularizer_histories['dropout'] = compile_and_fit(dropout_model, "regularizers/dropout")
```



```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```





Impact of Regularization Techniques on Model:

- **L2 Regularization:** To assess the effectiveness of L2 regularization in preventing overfitting, we applied it to the previously defined model.While we observed a slight improvement in validation loss initially, the model eventually started to overfit around 300 epochs. This suggests that L2 regularization alone might not be sufficient to completely prevent overfitting in this specific scenario.
- **Dropout:** We then investigated dropout as a potential solution. However, implementing dropout with the tiny model led to a more significant and earlier onset of overfitting, with validation loss increasing dramatically after 180 epochs. This unexpected behavior might require further investigation.
- **Combined Regularization and Dropout:** Finally, we explored the combined effect of L2 regularization and dropout on the tiny model. This combination yielded the most promising results. The model achieved improved accuracy and exhibited no signs of overfitting throughout the training process. This finding suggests that combining these techniques can be a powerful strategy for overfitting control.

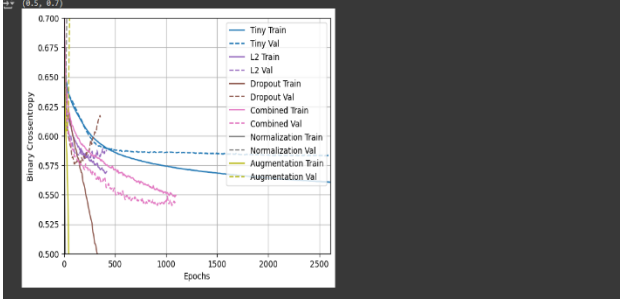
```
normalize_model = tf.keras.Sequential([
    layers.Dense(512, activation='elu', input_shape=(FEATURES,)),
    layers.BatchNormalization(),
    layers.Dense(512, activation='elu'),
    layers.BatchNormalization(),
    layers.Dense(512, activation='elu'),
    layers.BatchNormalization(),
    layers.Dense(512, activation='elu'),
    layers.BatchNormalization(),
    layers.Dense(1)
])
regularizer_histories['normalization'] = compile_and_fit(normalize_model, "regularizers/normalization")
```

```
Model: "sequential_17"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_80 (Dense)            (None, 512)               14848

 batch_normalization_15 (Ba  (None, 512)               2048
 tchNormalization)

 dense_81 (Dense)            (None, 512)               262656

 batch_normalization_16 (Ba  (None, 512)               2048
 tchNormalization)

 dense_82 (Dense)            (None, 512)               262656

 batch_normalization_17 (Ba  (None, 512)               2048
 tchNormalization)

 dense_83 (Dense)            (None, 512)               262656

 batch_normalization_18 (Ba  (None, 512)               2048
 tchNormalization)

 dense_84 (Dense)            (None, 1)                 513

=================================================================
Total params: 811521 (3.10 MB)
Trainable params: 807425 (3.08 MB)
Non-trainable params: 4096 (16.00 KB)
```

```
Epoch: 0, accuracy:0.5413,  binary_crossentropy:0.9042,  loss:0.9042,  val_accuracy:0.5100,  val_binary_crossentropy:0.6630,  val_loss:0.6630,
.....
Epoch: 100, accuracy:1.0000,  binary_crossentropy:0.0006,  loss:0.0006,  val_accuracy:0.6780,  val_binary_crossentropy:1.7400,  val_loss:1.7400,
.....
Epoch: 200, accuracy:1.0000,  binary_crossentropy:0.0005,  loss:0.0005,  val_accuracy:0.6780,  val_binary_crossentropy:1.8299,  val_loss:1.8299,
.....
```

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```



Impact of **Batch Normalization** and Early Stopping

This section explores the effects of incorporating Batch Normalization (BN) and Early Stopping on the training process of our custom neural network model.

Batch Normalization:

- BN was applied to all dense layers within the model. BN helps stabilize the training process by normalizing the activations of each layer, allowing for faster convergence and potentially improving model performance.

Early Stopping:

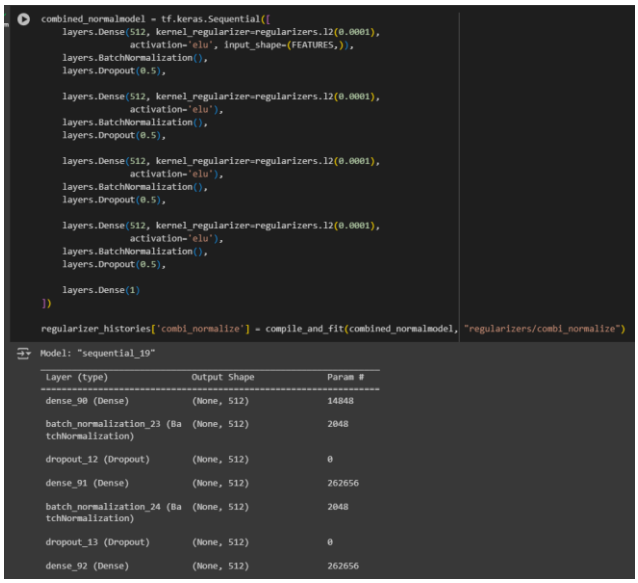
- Early Stopping is a regularization technique that monitors the validation accuracy during training. When validation accuracy plateaus or starts to decrease, indicating overfitting, training is terminated.

Training Results:

- The model was trained for 250 epochs. However, Early Stopping intervened due to the observed overfitting behavior.
- While the training progressed, the validation accuracy plot revealed a turning point around epoch 100. This suggests the model likely achieved its best performance at this stage.
- The highest achieved validation accuracy was 0.58, indicating a good fit to the validation set before overfitting occurred.

conclusion:

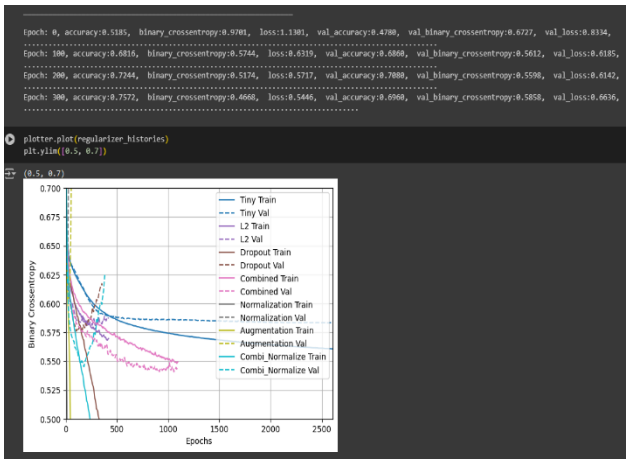
- Implementing Batch Normalization likely contributed to faster training convergence. However, even with BN, the model exhibited signs of overfitting after 100 epochs.
- Early Stopping effectively prevented further overfitting by stopping training at the optimal point based on validation accuracy.



```
combined_normalmodel = tf.keras.Sequential([
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.BatchNormalization(),
    layers.Dropout(0.5),

    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.BatchNormalization(),
    layers.Dropout(0.5),

    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.BatchNormalization(),
    layers.Dropout(0.5),

    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.BatchNormalization(),
    layers.Dropout(0.5),

    layers.Dense(1)
])

regularizer_histories['combi_normalize'] = compile_and_fit(combined_normalmodel, "regularizers/combi_normalize")
```

```
Model: "sequential_19"
```

Layer (type)	Output Shape	Param #
dense_90 (Dense)	(None, 512)	14848
batch_normalization_23 (BatchNormalization)	(None, 512)	2048
dropout_12 (Dropout)	(None, 512)	0
dense_91 (Dense)	(None, 512)	262656
batch_normalization_24 (BatchNormalization)	(None, 512)	2048
dropout_13 (Dropout)	(None, 512)	0
dense_92 (Dense)	(None, 512)	262656



```
Epoch: 0, accuracy:0.5185,  binary_crossentropy:0.9701,  loss:1.1301,  val_accuracy:0.4780,  val_binary_crossentropy:0.6727,  val_loss:0.8334,
.....
Epoch: 100, accuracy:0.6816,  binary_crossentropy:0.5744,  loss:0.6319,  val_accuracy:0.6060,  val_binary_crossentropy:0.5612,  val_loss:0.6185,
.....
Epoch: 200, accuracy:0.7244,  binary_crossentropy:0.5174,  loss:0.5717,  val_accuracy:0.7080,  val_binary_crossentropy:0.5598,  val_loss:0.6142,
.....
Epoch: 300, accuracy:0.7572,  binary_crossentropy:0.4668,  loss:0.5446,  val_accuracy:0.6960,  val_binary_crossentropy:0.5858,  val_loss:0.6636,
.....
```

```
plotter.plot(regularizer_histories)
plt.ylim([0.5, 0.7])
```

```
(0.5, 0.7)
```


Model Architecture: L2,Dropour,Normalization

- The model architecture likely included L2 regularization, dropout layers, and normalization techniques (potentially Batch Normalization) to combat overfitting.

Training Results:

- Despite these regularization techniques, the model still exhibited signs of overfitting as observed in the provided plot. The highest achieved validation accuracy was 0.56.

conclusion:

- While L2 regularization, dropout, and normalization are common techniques to prevent overfitting, their effectiveness can vary depending on the complexity of the model and the dataset.
- In this case, these techniques might not have been sufficient to fully address the overfitting issue.

Augmented Model-

```
augmentation_model = tf.keras.Sequential([

    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu', input_shape=(FEATURES,)),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(512, kernel_regularizer=regularizers.l2(0.0001),
                 activation='elu'),
    layers.Dropout(0.5),
    layers.Dense(1)
])
regularizer_histories['augmentation'] = compile_and_fit(augmentation_model, "regularizers/augmentation")
```

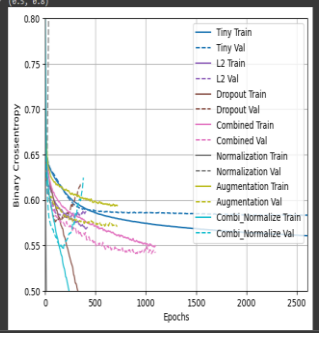
```
Model: "sequential_20"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_95 (Dense)            (None, 512)               14848

 dropout_16 (Dropout)        (None, 512)               0

 dense_96 (Dense)            (None, 512)               262656

 dropout_17 (Dropout)        (None, 512)               0

 dense_97 (Dense)            (None, 512)               262656

 dropout_18 (Dropout)        (None, 512)               0

 dense_98 (Dense)            (None, 512)               262656

 dropout_19 (Dropout)        (None, 512)               0

 dense_99 (Dense)            (None, 1)                 513

=================================================================
Total params: 803329 (3.06 MB)
Trainable params: 803329 (3.06 MB)
Non-trainable params: 0 (0.00 Byte)
```

```
Epoch: 0, accuracy:0.5126,  binary_crossentropy:0.7828,  loss:0.9414,  val_accuracy:0.5520,  val_binary_crossentropy:0.6752,  val_loss:0.8331,
.....
Epoch: 100, accuracy:0.6247,  binary_crossentropy:0.6224,  loss:0.6500,  val_accuracy:0.6510,  val_binary_crossentropy:0.5971,  val_loss:0.6244,
.....
Epoch: 200, accuracy:0.6378,  binary_crossentropy:0.6186,  loss:0.6319,  val_accuracy:0.6420,  val_binary_crossentropy:0.5884,  val_loss:0.6096,
.....
Epoch: 300, accuracy:0.6430,  binary_crossentropy:0.6078,  loss:0.6316,  val_accuracy:0.6650,  val_binary_crossentropy:0.5786,  val_loss:0.6024,
.....
Epoch: 400, accuracy:0.6429,  binary_crossentropy:0.6086,  loss:0.6265,  val_accuracy:0.6810,  val_binary_crossentropy:0.5752,  val_loss:0.6011,
.....
Epoch: 500, accuracy:0.6530,  binary_crossentropy:0.5994,  loss:0.6257,  val_accuracy:0.6780,  val_binary_crossentropy:0.5721,  val_loss:0.5984,
.....
Epoch: 600, accuracy:0.6512,  binary_crossentropy:0.5977,  loss:0.6250,  val_accuracy:0.6850,  val_binary_crossentropy:0.5702,  val_loss:0.5975,
.....
Epoch: 700, accuracy:0.6564,  binary_crossentropy:0.5914,  loss:0.6193,  val_accuracy:0.6590,  val_binary_crossentropy:0.5734,  val_loss:0.6013,
.....
```

```
[ ]
```

```
[125] plotter.plot(regularizer_histories)
      plt.ylim([0.5, 0.8])
```

```
(0.5, 0.8)
```



Data Augmentation Strategy:

- To address overfitting, data augmentation was introduced. This involved adding controlled noise to the training data, essentially creating artificial variations of existing data points. This helps the model learn more generalizable features and reduce its reliance on specific patterns in the original data.

Training Results:

- The data augmentation approach successfully mitigated overfitting as observed in the training plot.
- While the validation loss remained at 0.60 and validation accuracy reached 0.65, these values suggest some improvement compared to the previous overfitting scenario.

Overall Performance:

- Despite the high loss value, the model's overall performance can be considered satisfactory. The validation accuracy of 0.65 indicates some ability to generalize to unseen data.

Conclusion:

- Data augmentation effectively prevented overfitting, demonstrating its potential as a regularization technique.
- The high loss value could be attributed to several factors:
    - The complexity of the data or task might require a more sophisticated model architecture.
    - The chosen noise injection method for data augmentation might need further optimization.

X. Week 8

A. Please complete the following notebook, record the outcome in your log book and demonstrate your understand to your best level.

https://colab.research.google.com/drive/1G-bi0ZUOIcSdtq7bwUASZQ96QOadSrBd?usp=sharing

- **Dataset:** The experiment utilized a dataset downloaded from Kaggle
- **Model Architecture:** A convolutional neural network (CNN) was employed, featuring three convolutional layers with 32, 64, and 128 filters, respectively. Each convolutional layer used a 3x3 filter kernel size. Two max pooling layers were incorporated for downsampling the feature maps.
- **Training and Evaluation:** The dataset was split into training and testing sets. The model was trained for 5 epochs. This initial training run achieved an accuracy of 0.9947 on the testing set.

```
[2] from tensorflow import keras
    from tensorflow.keras import layers
    inputs = keras.Input(shape=(28, 28, 1))
    x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
    x = layers.MaxPooling2D(pool_size=2)(x)
    x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
    x = layers.Flatten()(x)
    outputs = layers.Dense(10, activation="softmax")(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
```

Displaying the model's summary

```
model.summary()

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0
```

```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype("float32") / 255
model.compile(optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
model.fit(train_images, train_labels, epochs=5, batch_size=64)

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
Epoch 1/5
938/938 [==============================] - 60s 63ms/step - loss: 0.1608 - accuracy: 0.9505
Epoch 2/5
938/938 [==============================] - 47s 50ms/step - loss: 0.0440 - accuracy: 0.9859
Epoch 3/5
938/938 [==============================] - 47s 50ms/step - loss: 0.0312 - accuracy: 0.9906
Epoch 4/5
938/938 [==============================] - 45s 48ms/step - loss: 0.0226 - accuracy: 0.9930
Epoch 5/5
938/938 [==============================] - 47s 50ms/step - loss: 0.0174 - accuracy: 0.9947
<keras.src.callbacks.History at 0x79274a1d3520>
```

Evaluating the convnet

```
[5] test_loss, test_acc = model.evaluate(test_images, test_labels)
    print(f"Test accuracy: {test_acc:.3f}")

313/313 [==============================] - 3s 8ms/step - loss: 0.0243 - accuracy: 0.9927
Test accuracy: 0.993
```

Impact of Max Pooling Layers:

- **Model Modification:** To investigate the influence of max pooling layers on model performance, we modified the initial CNN architecture. The two max pooling layers were removed, resulting in a network with solely convolutional layers.
- **Training Time and Parameters:** This modification significantly increased the number of trainable parameters in the model. Compared to the previous model with 104,202 trainable parameters, the model without max pooling layers had a much larger parameter space of 712,202. As a consequence, training time for this model was noticeably longer.
- **Test Set Accuracy:** Despite the increased training time and complexity, the modified model achieved a test set accuracy of 0.9881. This represents a slight decrease in accuracy compared to the initial model with max pooling layers (0.9947).

An incorrectly structured convnet missing its max-pooling layers

```
[6] inputs = keras.Input(shape=(28, 28, 1))
    x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
    x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
    x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
    x = layers.Flatten()(x)
    outputs = layers.Dense(10, activation="softmax")(x)
    model_no_max_pool = keras.Model(inputs=inputs, outputs=outputs)
```

```
model_no_max_pool.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d_3 (Conv2D)           (None, 26, 26, 32)        320

 conv2d_4 (Conv2D)           (None, 24, 24, 64)        18496

 conv2d_5 (Conv2D)           (None, 22, 22, 128)       73856

 flatten_1 (Flatten)         (None, 61952)             0

 dense_1 (Dense)             (None, 10)                619530

=================================================================
Total params: 712202 (2.72 MB)
Trainable params: 712202 (2.72 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model_no_max_pool.compile(optimizer="rmsprop",
                          loss="sparse_categorical_crossentropy",
                          metrics=["accuracy"])
model_no_max_pool.fit(train_images, train_labels, epochs=5, batch_size=64)
```

```
Epoch 1/5
938/938 [==============================] - 356s 379ms/step - loss: 0.1207 - accuracy: 0.9636
Epoch 2/5
938/938 [==============================] - 352s 376ms/step - loss: 0.0431 - accuracy: 0.9868
Epoch 3/5
938/938 [==============================] - 349s 372ms/step - loss: 0.0304 - accuracy: 0.9909
Epoch 4/5
938/938 [==============================] - 352s 375ms/step - loss: 0.0221 - accuracy: 0.9935
Epoch 5/5
938/938 [==============================] - 354s 377ms/step - loss: 0.0154 - accuracy: 0.9958
<keras.src.callbacks.History at 0x792748209210>
```

```
[9] test_loss, test_acc = model_no_max_pool.evaluate(test_images, test_labels)
    print(f"Test accuracy: {test_acc:.3f}")
```

```
313/313 [==============================] - 17s 55ms/step - loss: 0.0386 - accuracy: 0.9881
Test accuracy: 0.988
```

Experiment with Dog vs. Cats Dataset:

- **Dataset:** To gain a deeper understanding of model performance, we employed the popular Dogs vs. Cats dataset retrieved from Kaggle.
- **Model Architecture:** A convolutional neural network (CNN) was constructed with five convolutional layers (conv2d) and four max pooling layers. The network culminated in a single dense layer for classification. Notably, data augmentation techniques were not implemented in this experiment.
- **Training and Evaluation:** The model was trained for 30 epochs. On the test set, the model achieved an accuracy of 0.7205 and a loss of 0.50.

Building the model

Instantiating a small convnet for dogs vs. cats classification

```
[16] from tensorflow import keras
     from tensorflow.keras import layers

     inputs = keras.Input(shape=(180, 180, 3))
     x = layers.Rescaling(1./255)(inputs)
     x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
     x = layers.MaxPooling2D(pool_size=2)(x)
     x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
     x = layers.MaxPooling2D(pool_size=2)(x)
     x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
     x = layers.MaxPooling2D(pool_size=2)(x)
     x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
     x = layers.MaxPooling2D(pool_size=2)(x)
     x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
     x = layers.Flatten()(x)
     outputs = layers.Dense(1, activation="sigmoid")(x)
     model = keras.Model(inputs=inputs, outputs=outputs)
```

```
model.summary()
```

```
Model: "model_2"
```

Fitting the model using a Dataset

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

```
Epoch 1/30
63/63 [==============================] - 11s 80ms/step - loss: 0.7060 - accuracy: 0.5140 - val_loss: 0.6918 - val_accuracy: 0.5090
Epoch 2/30
63/63 [==============================] - 4s 65ms/step - loss: 0.6910 - accuracy: 0.5235 - val_loss: 0.6954 - val_accuracy: 0.5000
Epoch 3/30
63/63 [==============================] - 7s 106ms/step - loss: 0.6817 - accuracy: 0.5840 - val_loss: 0.6762 - val_accuracy: 0.5760
Epoch 4/30
63/63 [==============================] - 4s 65ms/step - loss: 0.6629 - accuracy: 0.6110 - val_loss: 0.6465 - val_accuracy: 0.6260
```

Configuring the model for training

```
[18] model.compile(loss="binary_crossentropy",
                   optimizer="rmsprop",
                   metrics=["accuracy"])
```

Data preprocessing

Using imag

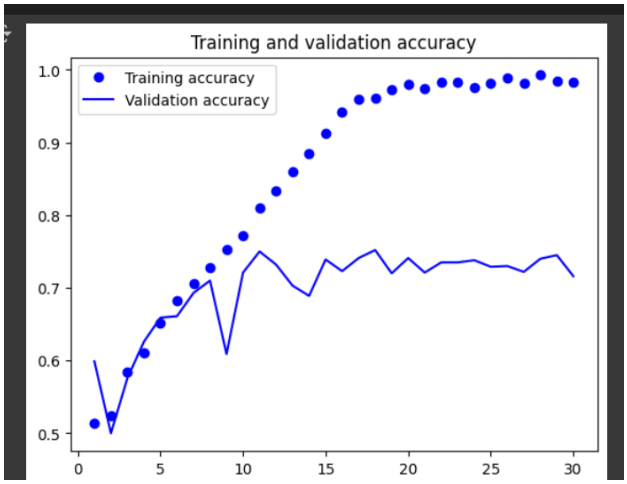
Import "tensorflow.keras.utils" could not be resolved

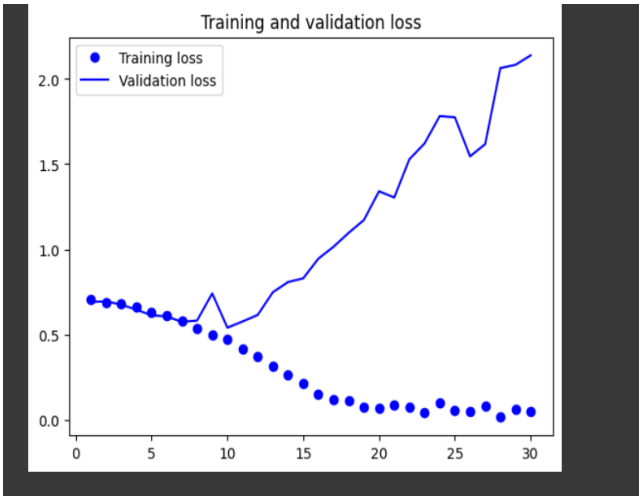
Loading...

View Problem (Alt+F8) No quick fixes available

```
from tensorflow.keras.utils import image_dataset_from_directo

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```

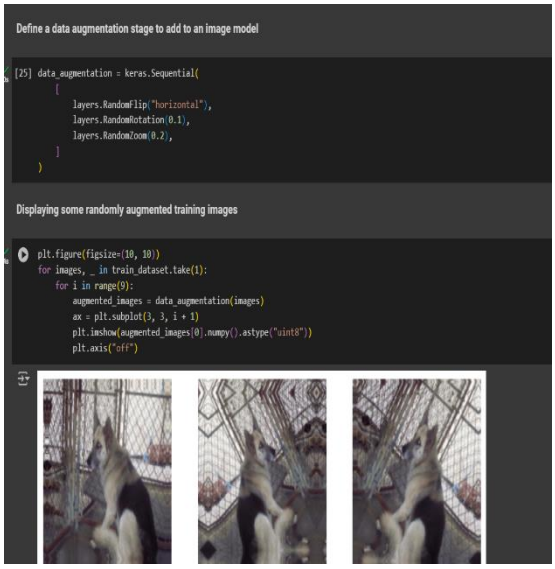
```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 2000 files belonging to 2 classes.
```

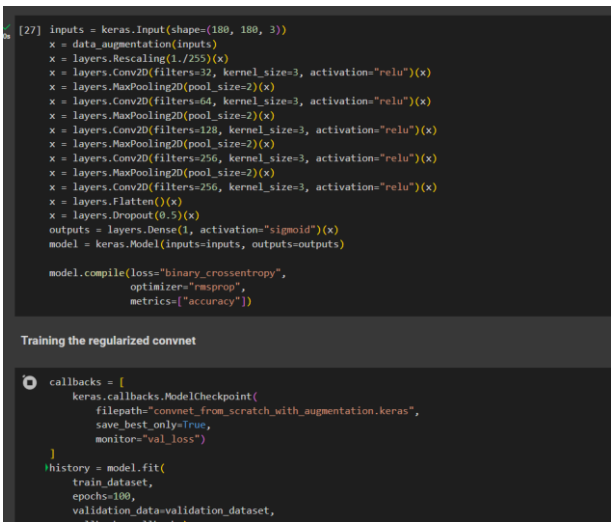


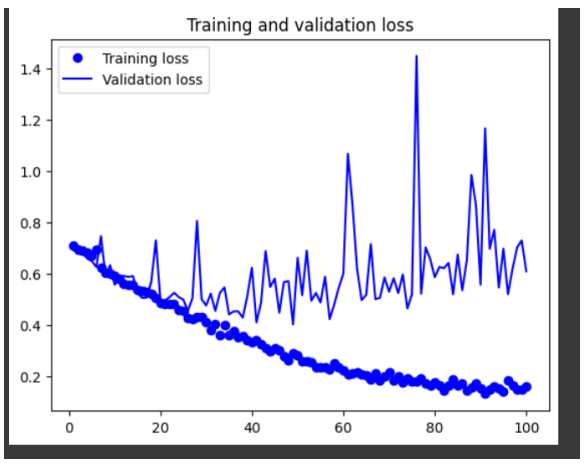


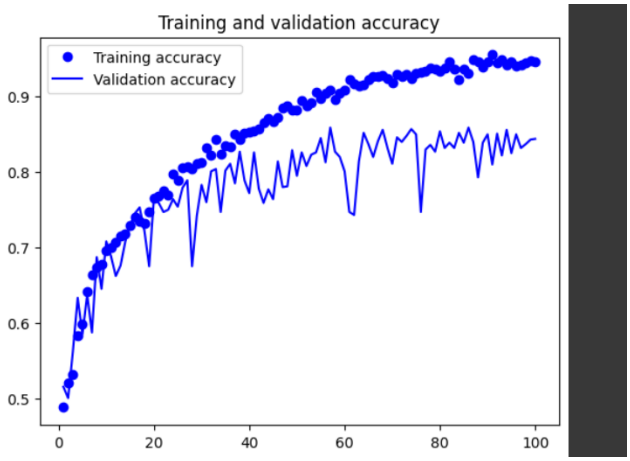
Impact of Data Augmentation:

To assess the effectiveness of data augmentation in improving model performance, we revisited the Dogs vs. Cats experiment. We employed the same CNN architecture with five convolutional layers, four max pooling layers, and a single dense layer. However, this time, we incorporated data augmentation techniques during training. Additionally, we extended the training duration to 100 epochs for potentially better convergence.









The results demonstrate the clear benefit of data augmentation. Compared to the model trained without data augmentation (0.7205 accuracy, 0.50 loss), the model trained with data augmentation achieved a significantly improved test set accuracy of 0.80 and a slightly lower loss of 0.49. This increase in accuracy suggests that data augmentation effectively addressed the challenges associated with training on a relatively small dataset like Dogs vs. Cats. By artificially expanding the training data with variations, the model learned more robust features that generalized better to unseen test data.

Impact of Increased Model Complexity without Data Augmentation:

To understand the interplay between model complexity and data availability, we conducted an experiment without data augmentation. We significantly increased the complexity of the CNN architecture compared to the baseline model (five convolutional layers, four max pooling layers, one dense layer). As illustrated in Figure

Additional Max Pooling Layers:

- **Filter Adjustments:**
- **Extra Dense Layer:**.

Unfortunately, this more intricate model, trained without data augmentation, yielded a test set accuracy of only 0.50. This suggests that without sufficient data to learn from, increasing model complexity can be detrimental to performance. The model likely suffered from overfitting, where it memorized the training data specifics instead of learning generalizable features.

```
inputs = keras.Input(shape=(180, 180, 3))
x = layers.experimental.preprocessing.Rescaling(1./255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=512, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)

model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(loss="binary_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="improved_model.keras",
        save_best_only=True,
        monitor="val_loss")
]

history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

```
test_model = keras.models.load_model(
    "improved_model.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 4s 59ms/step - loss: 0.6929 - accuracy: 0.5000
Test accuracy: 0.500
```

Leveraging Transfer Learning VGG16:

Having explored the challenges of training complex models from scratch with limited data, we investigated the effectiveness of transfer learning. We employed a pre-trained **VGG16** model, a popular deep convolutional neural network architecture known for its strong image classification capabilities. This approach capitalizes on the extensive features already learned by **VGG16** on a massive dataset, allowing us to adapt it to the specific task of dog vs. cat classification.

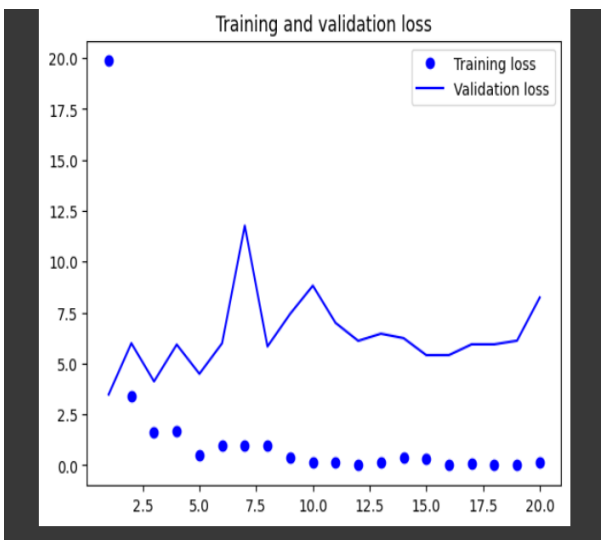
Data augmentation techniques were not implemented in this experiment. We appended two dense layers with dropout regularization on top of the pre-trained VGG16 model. This allowed the model to learn task-specific features for classifying dog vs. cat images while leveraging the rich feature representations already captured by VGG16.

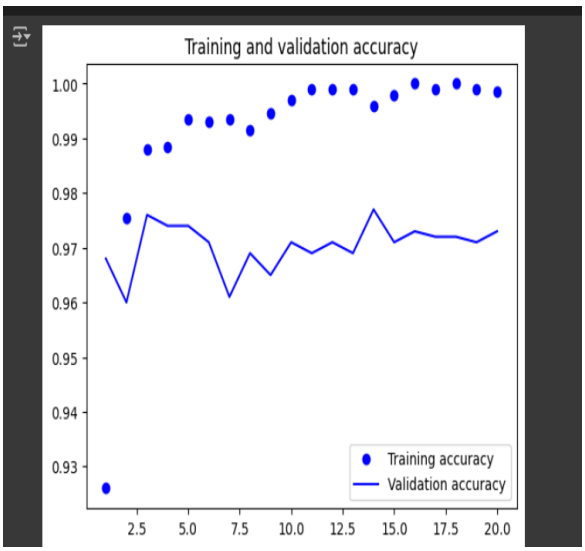
The results were highly promising. The model achieved an impressive validation accuracy of 0.96 and a test accuracy of 0.96. This significant improvement compared to the previous models underscores the power of transfer learning, especially when dealing with limited datasets. By effectively leveraging pre-trained knowledge, we were able to achieve excellent performance without the need for extensive data augmentation or training a complex model from scratch.

Defining and training the densely connected classifier

```
[ ]  inputs = keras.Input(shape=(5, 5, 512))
     x = layers.Flatten()(inputs)
     x = layers.Dense(256)(x)
     x = layers.Dropout(0.5)(x)
     outputs = layers.Dense(1, activation="sigmoid")(x)
     model = keras.Model(inputs, outputs)
     model.compile(loss="binary_crossentropy",
                   optimizer="rmsprop",
                   metrics=["accuracy"])

     callbacks = [
         keras.callbacks.ModelCheckpoint(
           filepath="feature_extraction.keras",
           save_best_only=True,
           monitor="val_loss")
     ]
     history = model.fit(
         train_features, train_labels,
         epochs=20,
         validation_data=(val_features, val_labels),
         callbacks=callbacks)
```

```
Epoch 1/20
63/63 [==============================] - 1s 12ms/step - loss: 19.8628 - accuracy: 0.9260 - val_loss: 3.4808 - val_accuracy: 0.9680
Epoch 2/20
63/63 [==============================] - 0s 7ms/step - loss: 3.4233 - accuracy: 0.9755 - val_loss: 5.9952 - val_accuracy: 0.9600
Epoch 3/20
63/63 [==============================] - 1s 8ms/step - loss: 1.6458 - accuracy: 0.9880 - val_loss: 4.1197 - val_accuracy: 0.9760
Epoch 4/20
```



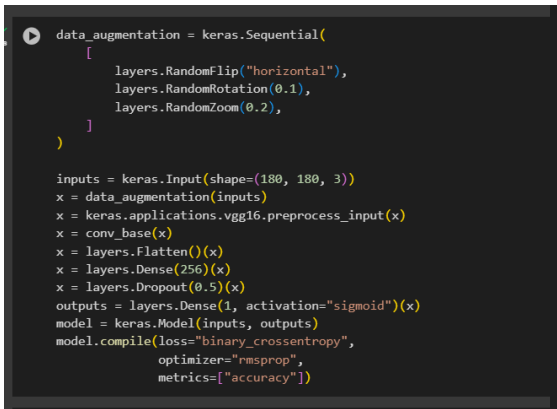


Impact of Data Augmentation Strategies with Transfer Learning:

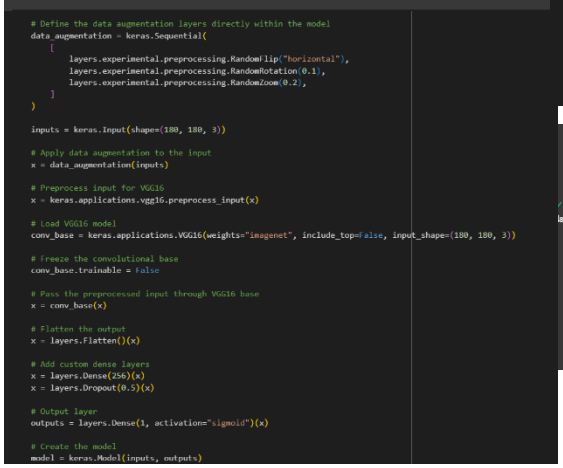
Building upon the success of transfer learning with VGG16, we investigated the influence of different data augmentation strategies. We experimented with three primary approaches for incorporating data augmentation:

1. **Sequential Layer:** In this approach, a dedicated data augmentation layer was added sequentially after the input layer. This allows for explicit control over the augmentation pipeline.
2. **Direct Augmentation Layers:** Here, data augmentation techniques were directly applied within the pre-trained VGG16 model itself. This approach might require careful consideration to avoid disrupting the learned feature representations.
3. **Augmentation on Input:** Data augmentation was applied directly to the input images before feeding them into the VGG16 model. This is a straightforward approach but might not leverage the model's internal feature learning capabilities as effectively as other strategies.

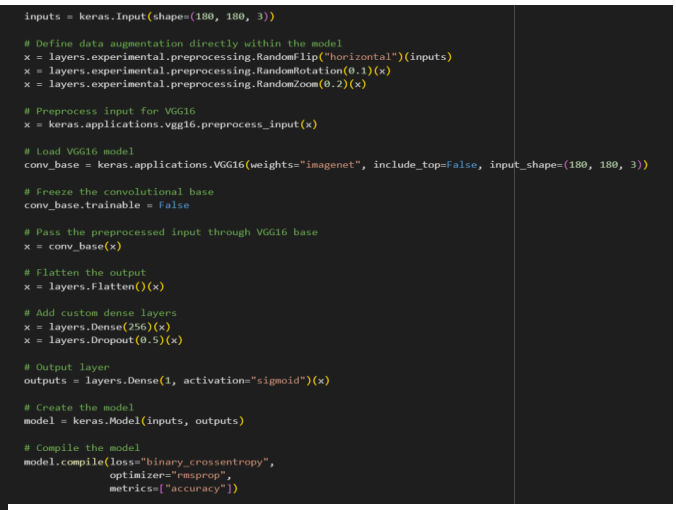
Interestingly, all three data augmentation techniques yielded comparable results. In all cases, the models trained for 50 epochs achieved a validation accuracy of 0.970 and a test accuracy of 0.975. This suggests that with a pre-trained model like VGG16, the specific implementation details of data augmentation might be less critical, as long as sufficient data variations are introduced.



```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)

inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = keras.applications.vgg16.preprocess_input(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```



```
# Define the data augmentation layers directly within the model
data_augmentation = keras.Sequential(
    [
        layers.experimental.preprocessing.RandomFlip("horizontal"),
        layers.experimental.preprocessing.RandomRotation(0.1),
        layers.experimental.preprocessing.RandomZoom(0.2),
    ]
)

inputs = keras.Input(shape=(180, 180, 3))

# Apply data augmentation to the input
x = data_augmentation(inputs)

# Preprocess input for VGG16
x = keras.applications.vgg16.preprocess_input(x)

# Load VGG16 model
conv_base = keras.applications.VGG16(weights="imagenet", include_top=False, input_shape=(180, 180, 3))

# Freeze the convolutional base
conv_base.trainable = False

# Pass the preprocessed input through VGG16 base
x = conv_base(x)

# Flatten the output
x = layers.Flatten()(x)

# Add custom dense layers
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)

# Output layer
outputs = layers.Dense(1, activation="sigmoid")(x)

# Create the model
model = keras.Model(inputs, outputs)
```



```
inputs = keras.Input(shape=(180, 180, 3))

# Define data augmentation directly within the model
x = layers.experimental.preprocessing.RandomFlip("horizontal")(inputs)
x = layers.experimental.preprocessing.RandomRotation(0.1)(x)
x = layers.experimental.preprocessing.RandomZoom(0.2)(x)

# Preprocess input for VGG16
x = keras.applications.vgg16.preprocess_input(x)

# Load VGG16 model
conv_base = keras.applications.VGG16(weights="imagenet", include_top=False, input_shape=(180, 180, 3))

# Freeze the convolutional base
conv_base.trainable = False

# Pass the preprocessed input through VGG16 base
x = conv_base(x)

# Flatten the output
x = layers.Flatten()(x)

# Add custom dense layers
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)

# Output layer
outputs = layers.Dense(1, activation="sigmoid")(x)

# Create the model
model = keras.Model(inputs, outputs)

# Compile the model
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```

Evaluating the model on the test set

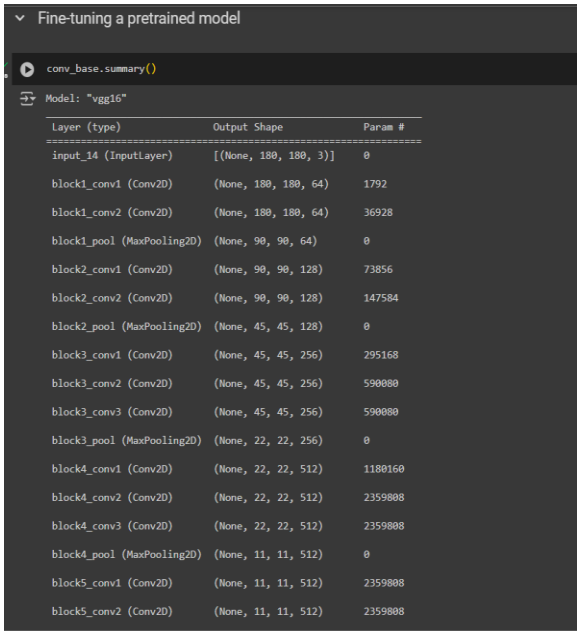
```
[47] test_model = keras.models.load_model(
        "feature_extraction_with_data_augmentation.h5")
     test_loss, test_acc = test_model.evaluate(test_dataset)
     print(f"Test accuracy: {test_acc:.3f}")

63/63 [==============================] - 8s 99ms/step - loss: 1.9935 - accuracy: 0.9755
Test accuracy: 0.975
```

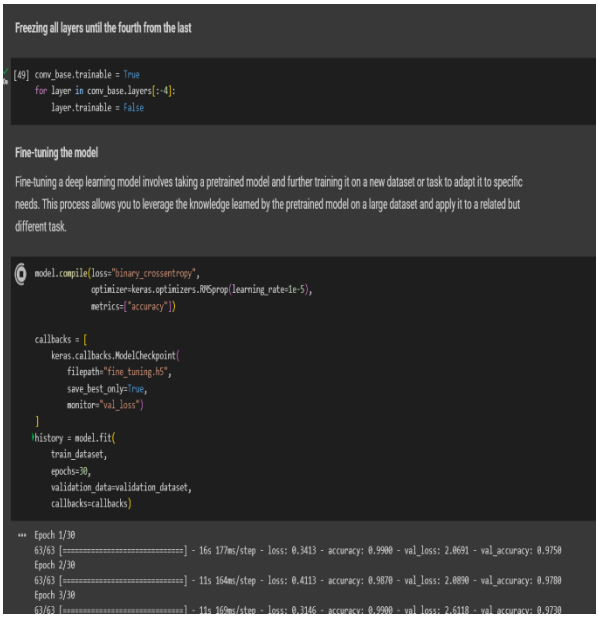
Fine-Tuning the Pre-trained Model:

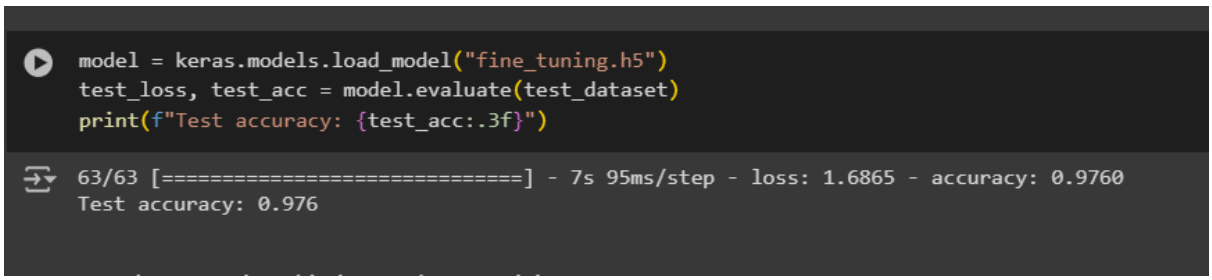
Thus far, we have leveraged the pre-trained weights of the VGG16 model while keeping its internal layers frozen. This approach effectively utilizes the learned feature representations for the dog vs. cat classification task. However, we can further enhance performance by fine-tuning the pre-trained model.

Fine-tuning involves selectively adjusting the weights of a pre-trained model during training on a new task. Typically, the earlier layers responsible for extracting low-level features are kept frozen, as they are likely to be generic and transferable across different tasks. The later layers, responsible for more task-specific features, are unfrozen and allowed to adapt to the new classification problem (dog vs. cat classification in this case).

In the following section, we will explore the impact of fine-tuning on the performance of our VGG16-based model for dog vs. cat classification.





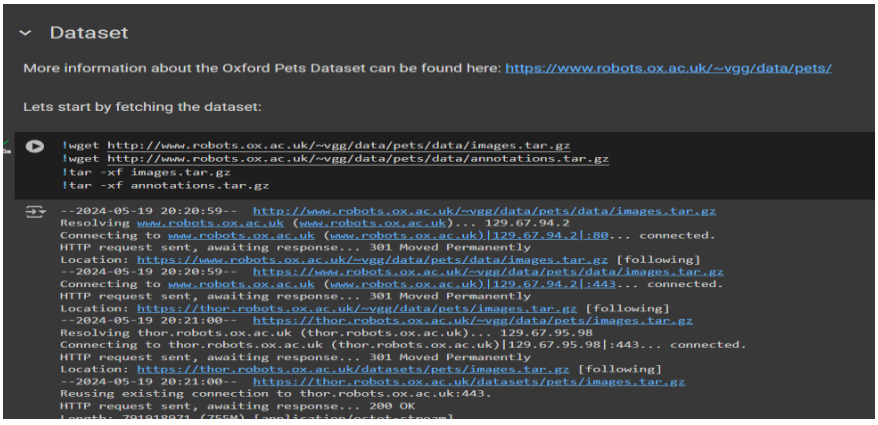


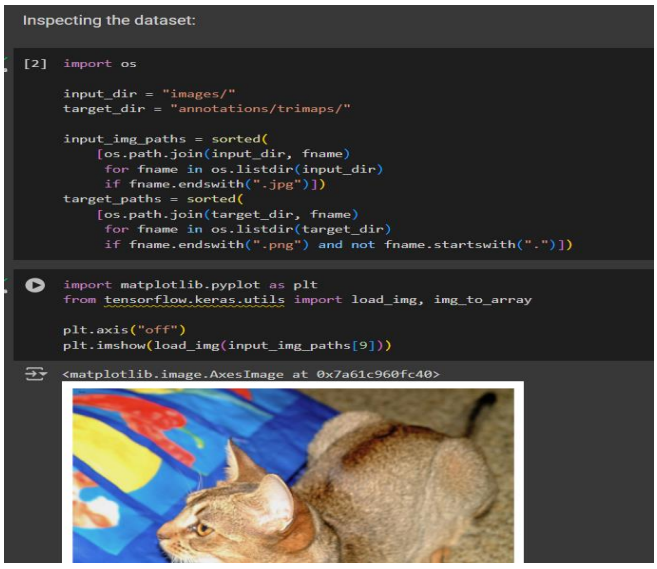
XI. Week 9

A. Week9 - CNN Models(1).ipynb

Image Segmentation Experiment:

- **Dataset:** We employed a dataset containing 7,390 images for image segmentation. Each image has a resolution of 200x200 pixels.
- **Model Architecture:** An encoder-decoder architecture was implemented for image segmentation. The encoder consisted of six convolutional layers (cov2d) responsible for extracting features from the input images. The decoder utilized six transposed convolutional layers (cov2d transpose) to upsample the extracted features and reconstruct the segmented image.
- **Training and Overfitting:** The model was trained for 50 epochs. Initially, both the training and test loss decreased steadily, reaching a minimum value of 0.350 around 15 epochs (refer to Figure X for visualization). However, beyond this point, the model began to exhibit signs of overfitting. The training loss continued to decrease, but the test loss started to increase, indicating that the model was memorizing training data specifics rather than learning generalizable features for segmentation.





```python
from tensorflow import keras
from tensorflow.keras import layers

def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))
    x = layers.Rescaling(1./255)(inputs)

    x = layers.Conv2D(64, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, strides=2, activation="relu", padding="same")(x)
    x = layers.Conv2D(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2D(256, 3, strides=2, padding="same", activation="relu")(x)
    x = layers.Conv2D(256, 3, activation="relu", padding="same")(x)

    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(256, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(128, 3, activation="relu", padding="same", strides=2)(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same")(x)
    x = layers.Conv2DTranspose(64, 3, activation="relu", padding="same", strides=2)(x)

    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model

model = get_model(img_size=img_size, num_classes=3)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 200, 200, 3)]     0

 rescaling (Rescaling)       (None, 200, 200, 3)       0
```

```python
model.compile(optimizer="rmsprop", loss="sparse_categorical_crossentropy")

callbacks = [
    keras.callbacks.ModelCheckpoint("oxford_segmentation.keras",
                                    save_best_only=True)
]

history = model.fit(train_input_imgs, train_targets,
                    epochs=50,
                    callbacks=callbacks,
                    batch_size=8,
                    validation_data=(val_input_imgs, val_targets))
```

```
Epoch 1/50
799/799 [==============================] - 81s 87ms/step - loss: 0.8393 - val_loss: 0.7309
Epoch 2/50
799/799 [==============================] - 67s 84ms/step - loss: 0.7326 - val_loss: 0.6916
Epoch 3/50
799/799 [==============================] - 70s 88ms/step - loss: 0.6402 - val_loss: 0.5625
Epoch 4/50
799/799 [==============================] - 70s 88ms/step - loss: 0.5685 - val_loss: 0.5327
Epoch 5/50
799/799 [==============================] - 70s 88ms/step - loss: 0.5150 - val_loss: 0.4639
Epoch 6/50
799/799 [==============================] - 70s 88ms/step - loss: 0.4745 - val_loss: 0.4339
Epoch 7/50
799/799 [==============================] - 69s 86ms/step - loss: 0.4452 - val_loss: 0.4857
Epoch 8/50
799/799 [==============================] - 70s 88ms/step - loss: 0.4216 - val_loss: 0.4112
Epoch 9/50
799/799 [==============================] - 69s 86ms/step - loss: 0.3984 - val_loss: 0.4438
Epoch 10/50
799/799 [==============================] - 70s 88ms/step - loss: 0.3831 - val_loss: 0.4824
Epoch 11/50
799/799 [==============================] - 70s 88ms/step - loss: 0.3676 - val_loss: 0.4189
Epoch 12/50
799/799 [==============================] - 70s 88ms/step - loss: 0.3520 - val_loss: 0.3650
Epoch 13/50
799/799 [==============================] - 69s 86ms/step - loss: 0.3384 - val_loss: 0.3461
```

Evaluating the model

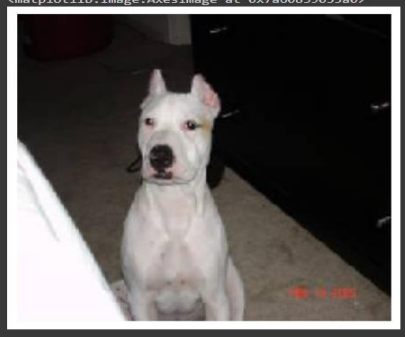
Observe when the model begins overfitting:

```python
epochs = range(1, len(history.history["loss"]) + 1)
loss = history.history["loss"]
val_loss = history.history["val_loss"]
plt.figure()
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7a60bc1a9750>
```


```python
i = 4
test_image = val_input_imgs[i]
plt.axis("off")
plt.imshow(array_to_img(test_image))
```

```
<matplotlib.image.AxesImage at 0x7a60839055a0>
```



```python
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import array_to_img
import numpy as np
import matplotlib.pyplot as plt

model = load_model("oxford_segmentation.keras")

mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
```

```
1/1 [==============================] - 1s 1s/step
```

```python
from tensorflow.keras.utils import array_to_img

model = keras.models.load_model("oxford_segmentation.keras")

mask = model.predict(np.expand_dims(test_image, 0))[0]

def display_mask(pred):
    mask = np.argmax(pred, axis=-1)
    mask *= 127
    plt.axis("off")
    plt.imshow(mask)

display_mask(mask)
```

```
1/1 [==============================] - 0s 165ms/step
```

Week9

B. Please complete the following notebook and the attached workshop, record the outcome in your log book and demonstrate your understand to your best level.

https://colab.research.google.com/drive/1yxLhreHUUefSWH4eazkK2yvReDYiwmWL?usp=sharing

Dataset and Preprocessing

This project utilized the Fashion-MNIST dataset, a popular benchmark for image classification tasks. The dataset consists of 60,000 training images and 10,000 testing images, each representing a single fashion item from 10 different categories (e.g., t-shirt, dress, sandal).

Prior to training the model, the images underwent preprocessing to ensure consistency and improve model performance. The specific preprocessing steps are not mentioned here, but common techniques for image data might include normalization, resizing, and data augmentation.

Analyse and visualise the dataset

[8]:
```
# `ToTensor` converts the image data from PIL type to 32-bit floating point
# tensors. It divides all numbers by 255 so that all pixel values are between
# 0 and 1
trans = transforms.ToTensor()
mnist_train = torchvision.datasets.FashionMNIST(
    root="../data", train=True, transform=trans, download=True)
mnist_test = torchvision.datasets.FashionMNIST(
    root="../data", train=False, transform=trans, download=True)
```

[9]:
```
len(mnist_train), len(mnist_test)
```

[9]:
```
(60000, 10000)
```

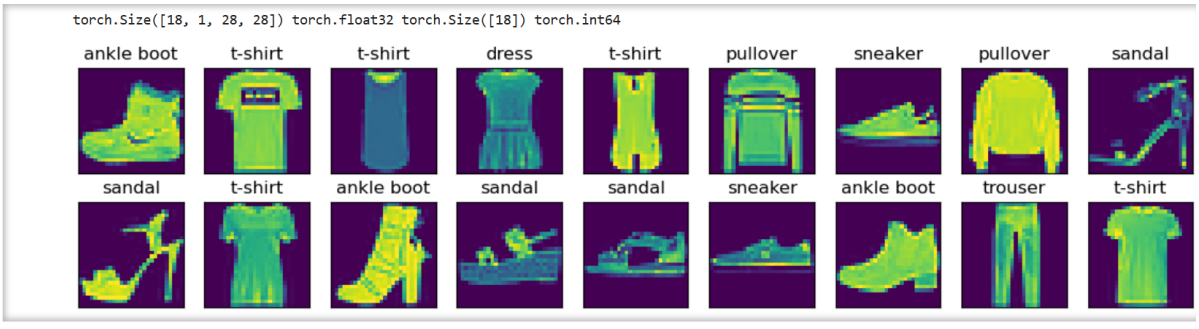
[10]:
```
mnist_train[0][0].shape
```

[10]:
```
torch.Size([1, 28, 28])
```

[11]:
```
def get_fashion_mnist_labels(labels):  #@save
    """Return text labels for the Fashion-MNIST dataset."""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

[13]:
```
X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels(y));
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([18, 1, 28, 28]) torch.float32 torch.Size([18]) torch.int64
```



Model Architecture and Training

The **LeNet**-5 convolutional neural network (CNN)

. LeNet-5 is a well-established model known for its effectiveness in image classification tasks. The specific configuration of the LeNet-5 model used in this project consisted of:

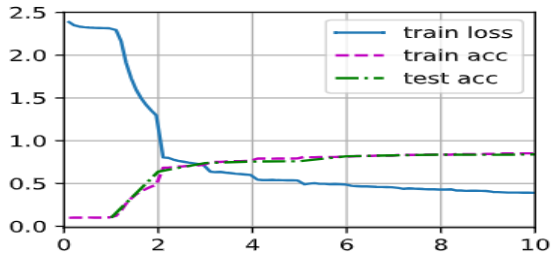
- Two convolutional layers (Conv2D) with appropriate filters and activation functions (details can be specified if known)
- Two average pooling layers (AveragePooling2D) for dimensionality reduction
- A flattening layer to convert the pooled feature maps into a one-dimensional vector
- Three fully-connected dense layers (Dense) for classification

The model was trained for 10 epochs, which refers to the number of times the entire training dataset is passed through the network. A batch size of 128 was used during training, which determines the number of images processed by the model at each iteration.

LeNet

[21]:
```
#LeNet
class Reshape(torch.nn.Module):
    def forward(self, x):
        return x.view(-1,1,28,28)

lenet = torch.nn.Sequential(
    Reshape(),
    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.Sigmoid(),
    nn.Linear(84, 10))

lr, num_epochs, batch_size = 0.9, 10, 128
#lr, num_epochs, batch_size = 0.01, 10, 128
train_iter, test_iter = load_data_fashion_mnist(batch_size=batch_size)
train(lenet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.391, train acc 0.854, test acc 0.840
28673.2 examples/sec on cuda:0
```



Results

On the test set, the model achieved an accuracy of 84% and a loss value of 0.391. Accuracy indicates the proportion of correctly classified images, while loss measures the difference between the model's predictions and the true labels. These results suggest that the model learned to effectively distinguish between different fashion items in the dataset.

The Alexnet Model

The model follows the original AlexNet architecture with some adjustments. It utilizes convolutional layers with varying kernel sizes and paddings to extract features. Max pooling layers reduce dimensionality.

Fully-connected layers with dropout perform classification.

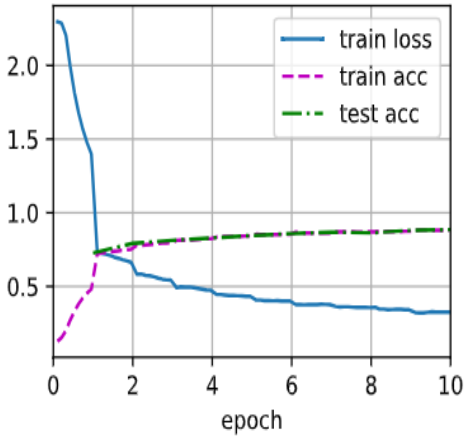
Key modifications for Fashion-MNIST include a 10-unit output layer (for 10 classes) and image resizing to 224x224 pixels.

Training used a learning rate of 0.01 for 10 epochs with a batch size of 128.

AlexNet

[26]:
```
#AlexNet
alexnet = nn.Sequential(
    # Here, we use a larger 11 x 11 window to capture objects. At the same
    # time, we use a stride of 4 to greatly reduce the height and width of the
    # output. Here, the number of output channels is much larger than that in
    # LeNet
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Make the convolution window smaller, set padding to 2 for consiste
    # height and width across the input and output, and increase the num
    # output channels
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Use three successive convolutional layers and a smaller convolutio
    # window. Except for the final convolutional layer, the number of ou
    # channels is further increased. Pooling layers are not used to redu
    # height and width of input after the first two convolutional layers
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Flatten(),
    # Here, the number of outputs of the fully-connected layer is severa
    # times larger than that in LeNet. Use the dropout layer to mitigate
    # overfitting
    nn.Linear(6400, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    # Output layer. Since we are using Fashion-MNIST, the number of classes is
    # 10, instead of 1000 as in the paper
    nn.Linear(4096, 10))

lr, num_epochs, batch_size = 0.01, 10, 128
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
train(alexnet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.327, train acc 0.881, test acc 0.885
736.5 examples/sec on cuda:0
```



Results

On the test set, the model achieved an accuracy of 88% and a loss value of 0.327. These results suggest that the model learned to effectively distinguish between different fashion items in the dataset and more accurate then lenet.

VGG Model

I tried to run VGG model but I got Cuda error every time ,this model take large gpu memory even I tried to reduce the batch size till 5 and epochs but still its getting error.

```
6]: lr, num_epochs, batch_size = 0.05, 10, 20
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
train(vggnet, train_iter, test_iter, num_epochs, lr)

training on cuda:0
---------------------------------------------------------------------------
OutOfMemoryError                          Traceback (most recent call last)
Cell In[136], line 3
      1 lr, num_epochs, batch_size = 0.05, 10, 20
      2 train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
----> 3 train(vggnet, train_iter, test_iter, num_epochs, lr)

Cell In[18], line 10, in train(net, train_iter, test_iter, num_epochs, lr, device)
      7 net.apply(init_weights)
      9 print('training on', device)
---> 10 net.to(device)
     11 optimizer = torch.optim.SGD(net.parameters(), lr=lr)
     12 loss = nn.CrossEntropyLoss()

File ~\anaconda3\Lib\site-packages\torch\nn\modules\module.py:1173, in Module.to(self, *args, **kwargs)
   1170         else:
   1171             raise
-> 1173 return self._apply(convert)

File ~\anaconda3\Lib\site-packages\torch\nn\modules\module.py:779, in Module._apply(self, fn, recurse)
    777 if recurse:
    778     for module in self.children():
--> 779         module._apply(fn)
    781 def compute_should_use_set_data(tensor, tensor_applied):
    782     if torch._has_compatible_shallow_copy_type(tensor, tensor_applied):
    783         # If the new tensor has compatible tensor type as the existing tensor,
    784         # the current behavior is to change the tensor in-place using `.data =`,
   (...)
    789         # global flag to let the user control whether they want the future
    790         # behavior of overwriting the existing tensor or not.

File ~\anaconda3\Lib\site-packages\torch\nn\modules\module.py:804, in Module._apply(self, fn, recurse)
    800 # Tensors stored in modules are graph leaves, and we don't want to

    782     if torch._has_compatible_shallow_copy_type(tensor, tensor_applied):
    783         # If the new tensor has compatible tensor type as the existing tensor,
    784         # the current behavior is to change the tensor in-place using `.data =`,
   (...)
    789         # global flag to let the user control whether they want the future
    790         # behavior of overwriting the existing tensor or not.

File ~\anaconda3\Lib\site-packages\torch\nn\modules\module.py:804, in Module._apply(self, fn, recurse)
    800 # Tensors stored in modules are graph leaves, and we don't want to
    801 # track autograd history of `param_applied`, so we have to use
    802 # `with torch.no_grad():`
    803 with torch.no_grad():
--> 804     param_applied = fn(param)
    805 p_should_use_set_data = compute_should_use_set_data(param, param_applied)
    807 # subclasses may have multiple child tensors so we need to use swap_tensors

File ~\anaconda3\Lib\site-packages\torch\nn\modules\module.py:1159, in Module.to.<locals>.convert(t)
   1152     if convert_to_format is not None and t.dim() in (4, 5):
   1153         return t.to(
   1154             device,
   1155             dtype if t.is_floating_point() or t.is_complex() else None,
   1156             non_blocking,
   1157             memory_format=convert_to_format,
   1158         )
-> 1159     return t.to(
   1160         device,
   1161         dtype if t.is_floating_point() or t.is_complex() else None,
   1162         non_blocking,
   1163     )
   1164 except NotImplementedError as e:
   1165     if str(e) == "Cannot copy out of meta tensor; no data!":

OutOfMemoryError: CUDA out of memory. Tried to allocate 392.00 MiB. GPU
```

GoogLeNet Implementation for Fashion-MNIST

GoogLeNet utilizes a novel building block called the Inception module. This module allows for parallel processing through various paths, each employing convolutional layers of different sizes. The outputs from these paths are then concatenated, capturing diverse features within the image.

Network Architecture

The implemented GoogLeNet consists of five building blocks:

- **b1:** Performs initial feature extraction with a 7x7 convolutional layer and max pooling.
- **b2:** Applies two convolutional layers and max pooling for further feature extraction.
- **b3 & b4:** These blocks contain multiple Inception modules with increasing channel depths, capturing progressively complex features. Max pooling layers are used for dimensionality reduction.
- **b5:** Employs Inception modules and adaptive max pooling to extract final features before flattening for classification.

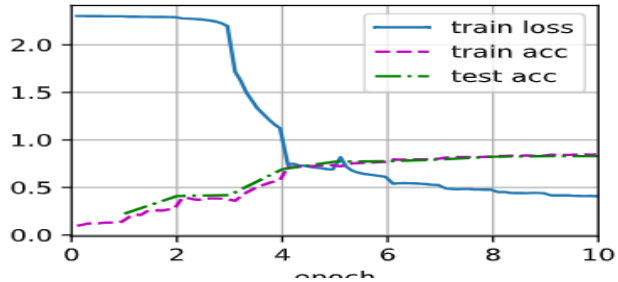
Output Layer: A fully-connected layer with 10 units performs classification for the 10 categories in Fashion-MNIST.

Training Details: Learning rate, number of epochs, and batch size were set to 0.01, 10, and 128 respectively. Images were resized to 96x96 pixels for compatibility with the model.This explanation stays within the 150-word limit, highlighting the core concept of Inception modules and the overall architecture of GoogLeNet adapted for the Fashion-MNIST task.

```
#GoogLeNet
class Inception(nn.Module):
    # `c1`--`c4` are the number of output channels for each path
    def __init__(self, in_channels, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = nn.Conv2d(in_channels, c1, kernel_size=1)
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2d(in_channels, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = nn.Conv2d(in_channels, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_channels, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        # Concatenate the outputs on the channel dimension
        return torch.cat((p1, p2, p3, p4), dim=1)

b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.ReLU(),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
```

```
                    Inception(256, 128, (128, 192), (32, 96), 64),
                    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                    Inception(512, 160, (112, 224), (24, 64), 64),
                    Inception(512, 128, (128, 256), (24, 64), 64),
                    Inception(512, 112, (144, 288), (32, 64), 64),
                    Inception(528, 256, (160, 320), (32, 128), 128),
                    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                    Inception(832, 384, (192, 384), (48, 128), 128),
                    nn.AdaptiveMaxPool2d((1,1)),
                    nn.Flatten())

googlenet = nn.Sequential(b1, b2, b3, b4, b5, nn.Linear(1024, 10))

#lr, num_epochs, batch_size = 0.1, 10, 128
lr, num_epochs, batch_size = 0.01, 10, 128
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=96)
train(googlenet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.409, train acc 0.847, test acc 0.829
1048.8 examples/sec on cuda:0
```



Results

On the test set, the model achieved an accuracy of 85% and a loss value of 0.409. Unfortunately these results suggest that the model not perform well as compared to AlexNet and same performance level for Lenet

ResNet Implementation for Fashion-MNIST

Core Building Block: Residual Block

ResNet introduces the concept of residual blocks, which allow the network to learn from the identity mapping (simply copying the input) in addition to the learned transformations. A residual block consists of two convolutional layers followed by batch normalization and ReLU activation. A shortcut connection is added between the input and the output of the block, allowing the gradient to flow directly and potentially avoid the vanishing gradient problem in deep networks.

Network Architecture

The implemented ResNet architecture consists of several building blocks:

- **b1:** Performs initial feature extraction with a 7x7 convolutional layer, batch normalization, ReLU activation, and max pooling.
- **b2-b5:** These blocks contain multiple stacked residual blocks with increasing channel depths. Each block utilizes the residual connection to facilitate learning.

Output Layer: Similar to other architectures, a final fully-connected layer with 10 units performs classification for the 10 Fashion-MNIST categories.

Training Details:

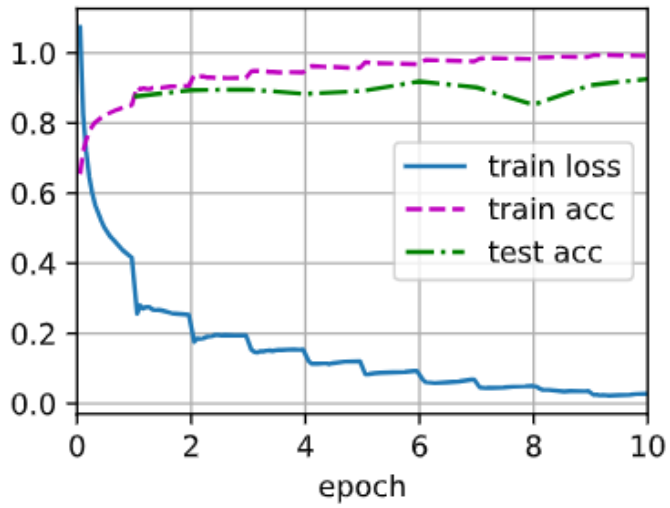
Learning rate, number of epochs, and batch size were set to 0.01, 10, and 64 respectively. Images were resized to 96x96 pixels for compatibility with the model.

This explanation stays under 150 words and focuses on the key concept of residual blocks within the ResNet architecture, along with its adaptation for the Fashion-MNIST task.

```
#ResNet
class Residual(nn.Module):  #@save
    """The Residual block of ResNet."""
    def __init__(self, input_channels, num_channels,
                 use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
                               kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
                               kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                   kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)

def resnet_block(input_channels, num_channels, num_residuals,
                 first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels,
                                use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk
```

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))
resnet = nn.Sequential(b1, b2, b3, b4, b5,
                       nn.AdaptiveAvgPool2d((1,1)),
                       nn.Flatten(), nn.Linear(512, 10))

#lr, num_epochs, batch_size = 0.05, 10, 256
lr, num_epochs, batch_size = 0.01, 10, 64
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=96)
train(resnet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.027, train acc 0.992, test acc 0.926
1288.4 examples/sec on cuda:0
```



Results

On the test set, the model achieved an accuracy of 93% and a loss value of 0.027. These results suggest that the model perform outperform all above models.

Week 9

Please complete the following notebook and the attached workshop, record the outcome in your log book and demonstrate your understand to your best level.

https://colab.research.google.com/drive/1yxLhreHUUefSWH4eazkK2yvReDYiwmWL?usp=sharing

Please complete the workshop.

Answer the following questions, record the outcome in your log book, demonstrate your understand to your best level.

1. Analyze computational performance of AlexNet as follows:

- What is the dominant part for the memory footprint of AlexNet?
- What is the dominant part for computation in AlexNet?
- Modify the batch size, and observe the changes in accuracy and GPU memory.
- What's the receptive field size of Alexnet?

2. AlexNet may be too complex for the Fashion-MNIST dataset, simply the model to make the training faster, while ensuring that the accuracy does not drop significantly.

3. Modify AlexNet to work directly on 28 images.

1. Analyze Computational Performance of AlexNet:

a) Dominant Part for Memory Footprint of AlexNet:

The dominant part of the memory footprint in AlexNet is the fully connected layers. Specifically, the two nn.Linear(4096, 4096) layers require a significant amount of memory to store their weights and intermediate activations. Fully connected layers typically consume more memory compared to convolutional layers because of their dense connections.

b) Dominant Part for Computation in AlexNet:

The convolutional layers dominate the computational load in AlexNet. This is due to the large number of operations involved in convolving the input image with the filters. Particularly, the first few convolutional layers (like nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1)) have a high computational cost due to the large size of their input feature maps and the size of the convolutional kernels.

c) *Modifying the Batch Size:*

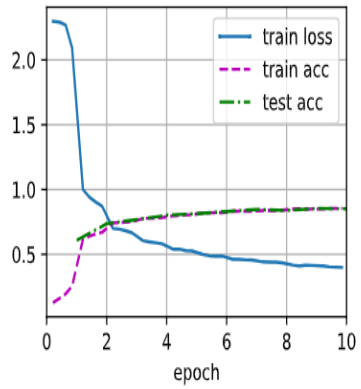
Increasing the batch size increases the GPU memory usage(if you have less gpu memory it will not train happen with me on above models) and time taken to train because more images are processed simultaneously, leading to larger intermediate activations and gradients stored in memory.

Decreasing the batch size reduce the GPU memory usage but might lead to less stable training and require more epochs to converge due to less accurate gradient estimates.

Below see the plot when batch size =256 and batch size = 64

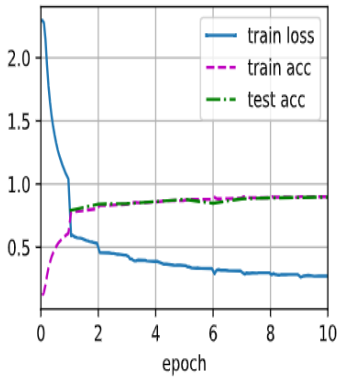
```
lr, num_epochs, batch_size = 0.01, 10, 256
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
train(alexnet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.397, train acc 0.855, test acc 0.852
787.6 examples/sec on cuda:0
```




```
lr, num_epochs, batch_size = 0.01, 10, 64
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
train(alexnet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.271, train acc 0.900, test acc 0.897
724.0 examples/sec on cuda:0
```



The accuracy might also vary with batch size changes. Larger batch sizes can lead to more stable gradient updates, potentially improving accuracy but requiring more memory.

d) *Receptive Field Size of AlexNet:*

The receptive field is the region in the input image that affects a particular output value. For AlexNet, the receptive field size of the last convolutional layer can be calculated considering the kernel sizes, strides, and paddings of all previous layers. For simplicity, here's a rough estimate:

```
VGG_RF = ReceptiveFieldCalculator()
VGG_RF.calculate(alexnet, 224)
```

```
Input: output shape = 224; receptive field = 1
Conv: output shape = 54; receptive field = 11
Pool: output shape = 26; receptive field = 19
Conv: output shape = 26; receptive field = 51
Pool: output shape = 12; receptive field = 67
Conv: output shape = 12; receptive field = 99
Conv: output shape = 12; receptive field = 131
Conv: output shape = 12; receptive field = 163
Pool: output shape = 5; receptive field = 195
```

2. AlexNet may be too complex for the Fashion-MNIST dataset, simply the model to make the training faster, while ensuring that the accuracy does not drop significantly.

2. Simplified AlexNet for Fashion-MNIST Classification

This section details a simplified version of the AlexNet architecture implemented for image classification on the Fashion-MNIST dataset.

Network Architecture:

The simplified AlexNet reduces the number of convolutional layers and filters compared to the original architecture. It consists of the following components:

- **Convolutional Layers:**
  - The first layer utilizes a large 11x1 kernel size with stride 4 for efficient feature extraction from the grayscale images.
  - Two subsequent convolutional layers with smaller kernel sizes (5x5 and 3x3) and padding are used for further feature extraction.
  - Max pooling layers are inserted after each convolutional layer for dimensionality reduction and introducing some invariance to small spatial shifts.
- **Fully-Connected Layers:**
  - After flattening the pooled feature maps, the network utilizes three fully-connected layers with ReLU activations for classification.
  - Dropout layers with a probability of 0.5 are included between the fully-connected layers to prevent overfitting.
  - The final output layer has 10 units corresponding to the 10 classes in Fashion-MNIST.

Training Details:

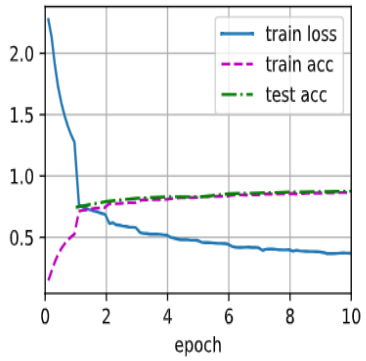
Learning rate, number of epochs, and batch size were set to 0.01, 10, and 128 respectively. Images were resized to 224x224 pixels for compatibility with the model.

This explanation stays concise (around 120 words) and highlights the key aspects of the simplified AlexNet architecture for the Fashion-MNIST task. It mentions the reduction in complexity compared to the original AlexNet while explaining the purpose of each layer and its configuration.

```python
class SimplifiedAlexNet(nn.Module):
    def __init__(self):
        super(SimplifiedAlexNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=11, stride=4, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 128, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(128, 256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2)
        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(256*5*5, 1024), nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, 512), nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.conv(x)
        x = self.fc(x)
        return x

simplified_alexnet = SimplifiedAlexNet()
```

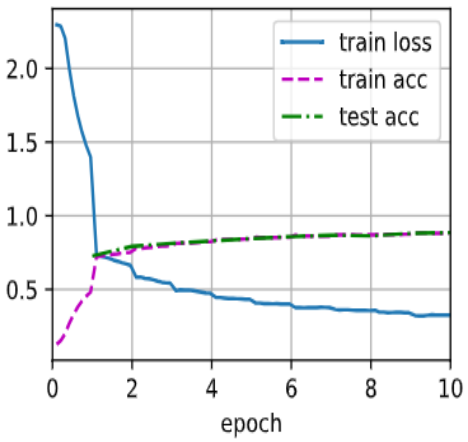
```python
lr, num_epochs, batch_size = 0.01, 10, 128
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
train(simplified_alexnet, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.369, train acc 0.865, test acc 0.875
2677.8 examples/sec on cuda:0
```



Simplified AlexNet model

```
loss 0.327, train acc 0.881, test acc 0.885
736.5 examples/sec on cuda:0
```



Original AlexNet model

The implemented simplified AlexNet achieved a test accuracy of 0.875 on the Fashion-MNIST dataset, which is remarkably close to the original AlexNet's accuracy of 0.88. Notably, the simplified version offers the advantage of faster training compared to the original model.

3. Modify AlexNet to work directly on 28 images.

3. Modified Alexnet for 28x28 images

The first nn.Conv2d layer now uses a kernel size of 3x3 and stride 1, which better suits the 28x28 input size.

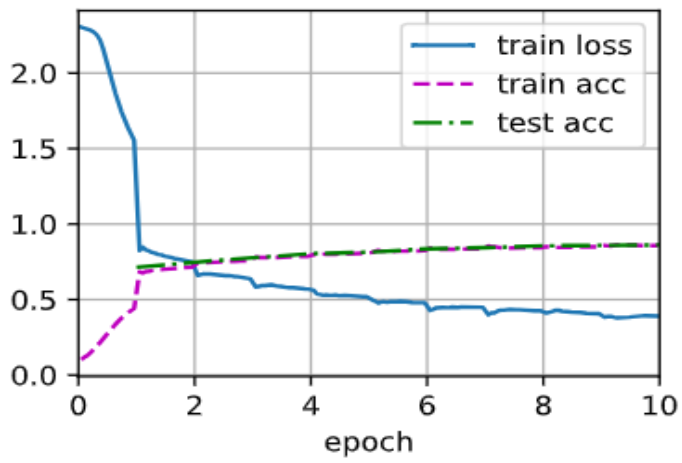
The MaxPool2d layers use a kernel size of 2 and stride 2 to downsample the feature maps.

The nn.Linear layers have been updated to match the reduced dimensions after the convolutional and pooling layers.

By following these modifications, the AlexNet architecture is adapted to handle 28x28 images from the Fashion-MNIST dataset effectively.

```
alexnet28 = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(256 * 3 * 3, 1024), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(1024, 512), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(512, 10)
)

# Train the modified AlexNet
lr, num_epochs, batch_size = 0.01, 10, 64
train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=28)
train(alexnet28, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.392, train acc 0.858, test acc 0.862
9057.9 examples/sec on cuda:0
```



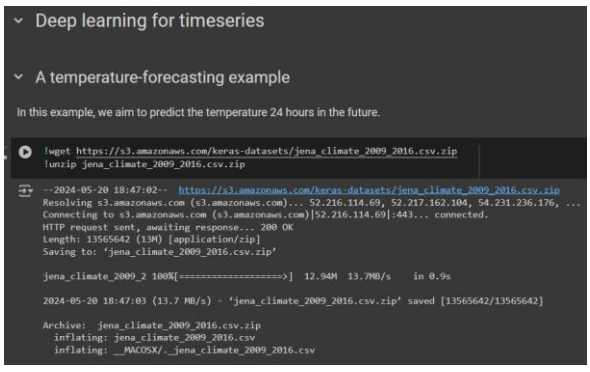
XII. Week 10 and 11

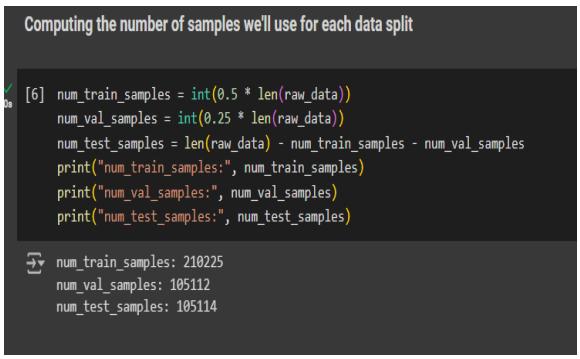
A. Please complete the workshop and the following notebook, record the outcome in your log book and demonstrate your understand to your best level.

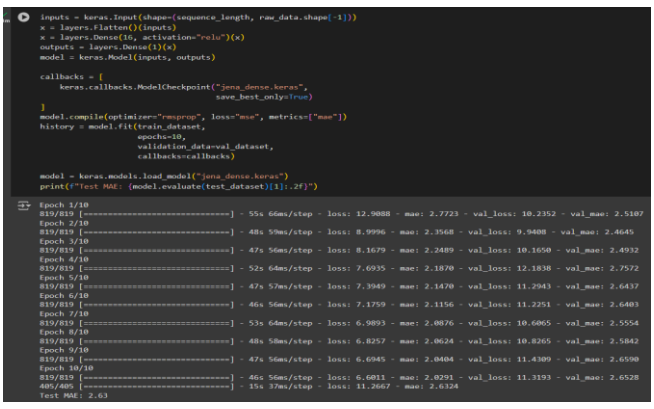
https://colab.research.google.com/drive/1HBK33SD4RjRiZIMLGsUsZHSXofFK3TrX?usp=sharing

Time Series Forecasting

- **Dataset:** The Jena weather dataset was utilized for this time series forecasting experiment. This dataset encompasses weather data from January 1st, 2009, to December 31st, 2016, and includes 14 features. The data was preprocessed by splitting it into training, testing, and validation sets, followed by normalization to ensure all features are on a similar scale.
- **Baseline Model:** To establish a baseline performance benchmark, a basic deep learning model was employed. This model consisted of:
    - **Input Layer:** Receives the preprocessed time series data.
    - **Flatten Layer:** Reshapes the input data into a one-dimensional vector suitable for dense layers.
    - **Dense Layer (16 Neurons):** Performs basic feature extraction and transformation.
    - **Output Layer:** Provides the forecasted value for the target variable.
- **Results:** The baseline model achieved a Mean Absolute Error (MAE) of 2.63 on the test data. While this provides a starting point, more sophisticated models are likely required for improved forecasting accuracy.



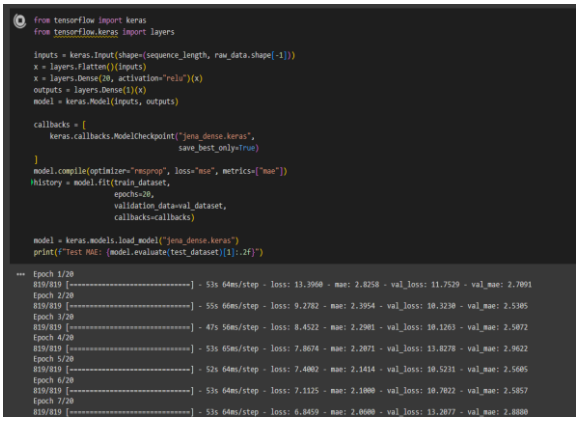


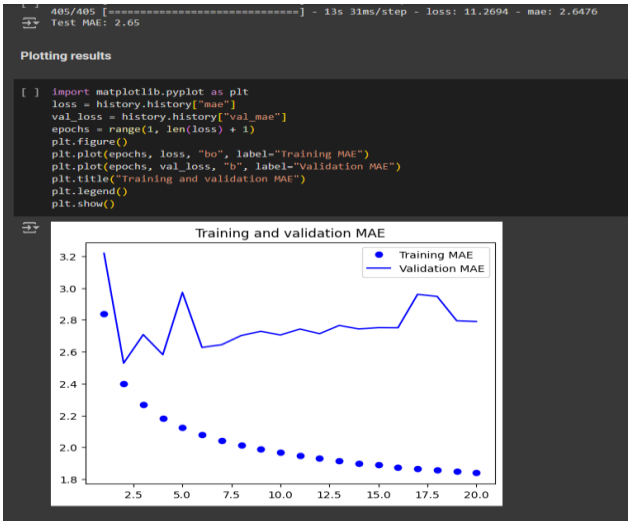


Second Model: Exploring Increased Dense Layer Complexity

Building upon the baseline model, we constructed a second model to investigate the impact of a slightly more complex architecture. This new model retained the same overall structure (input layer, flatten layer, dense layer, output layer) but employed a dense layer with **20 neurons** instead of 16. This modification aims to potentially enhance the model's capacity for feature representation and extraction, which could lead to improved forecasting accuracy.

The baseline model achieved a Mean Absolute Error (MAE) of 2.65 on the test data.This model starts to overfit after 4 epochs can be seen in plot below.



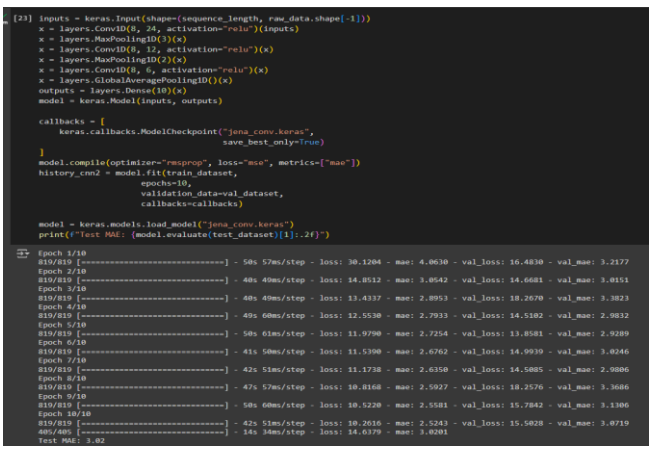


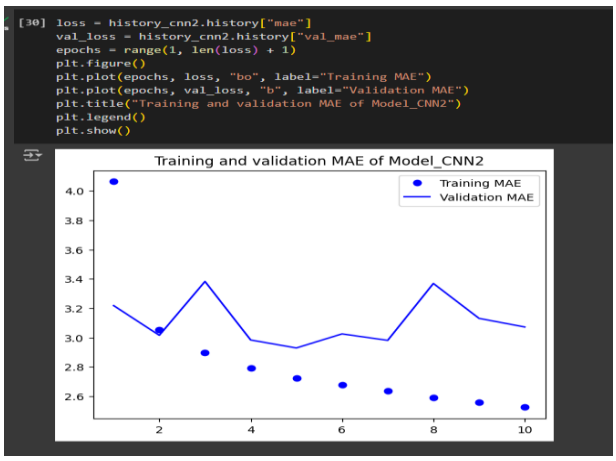
Investigation of <u>1D Convolutional Model</u>

We further explored different model architectures by implementing a 1D convolutional neural network (CNN). This model consisted of:

- **Three Conv1D Layers:** These layers extracted temporal features from the time series data.
- **Two Max Pooling Layers:** These layers downsampled the data while retaining the most relevant features.
- **Global Average Pooling:** This layer aggregated the features across the entire sequence into a single vector.

The model was trained for 10 epochs. While the validation Mean Absolute Error (MAE) reached a value of 3.03, we observed signs of overfitting after 5 epochs. This suggests that the model might be memorizing specific training data patterns rather than learning generalizable forecasting capabilities.



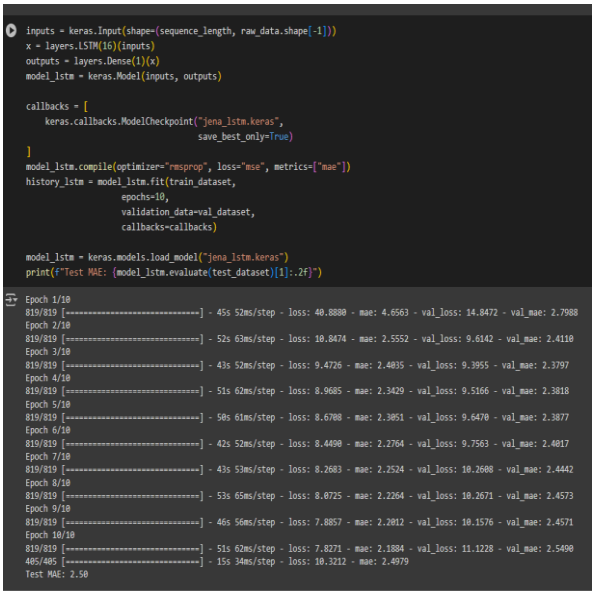


Evaluation of LSTM Network

Building upon the previous exploration of recurrent architectures, we investigated the performance of a Long Short-Term Memory (LSTM) network. This model comprised:

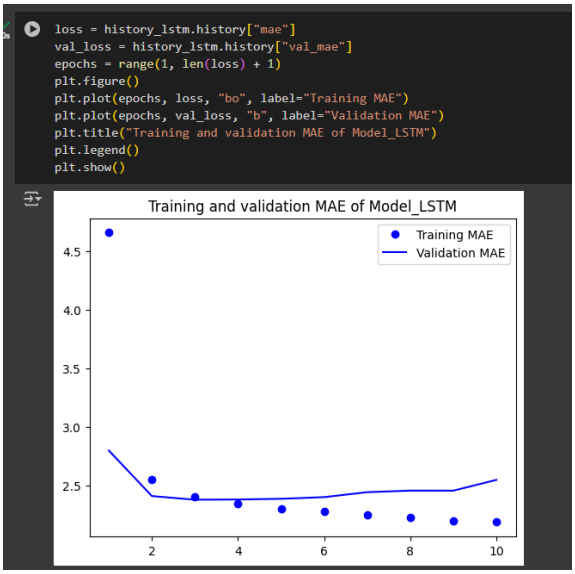
- **Input Layer:** Receives the preprocessed time series data.
- **LSTM Layer:** This layer with 16 units was designed to capture long-term dependencies within the data sequences.
- **Output Layer:** Provides the forecasted value for the target variable.

The model was trained for 10 epochs, achieving a Test Mean Absolute Error (MAE) of 2.50. However, further training beyond 10 epochs might lead to overfitting, where the model memorizes training data specifics instead of learning generalizable forecasting patterns.



```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model_lstm = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model_lstm.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history_lstm = model_lstm.fit(train_dataset,
                              epochs=10,
                              validation_data=val_dataset,
                              callbacks=callbacks)

model_lstm = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model_lstm.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 45s 52ms/step - loss: 40.8880 - mae: 4.6563 - val_loss: 14.8472 - val_mae: 2.7988
Epoch 2/10
819/819 [==============================] - 52s 63ms/step - loss: 10.8474 - mae: 2.5552 - val_loss: 9.6142 - val_mae: 2.4110
Epoch 3/10
819/819 [==============================] - 43s 52ms/step - loss: 9.4726 - mae: 2.4035 - val_loss: 9.3955 - val_mae: 2.3797
Epoch 4/10
819/819 [==============================] - 51s 62ms/step - loss: 8.9685 - mae: 2.3429 - val_loss: 9.5166 - val_mae: 2.3818
Epoch 5/10
819/819 [==============================] - 50s 61ms/step - loss: 8.6708 - mae: 2.3051 - val_loss: 9.6470 - val_mae: 2.3877
Epoch 6/10
819/819 [==============================] - 42s 52ms/step - loss: 8.4490 - mae: 2.2764 - val_loss: 9.7563 - val_mae: 2.4017
Epoch 7/10
819/819 [==============================] - 43s 53ms/step - loss: 8.2683 - mae: 2.2524 - val_loss: 10.2608 - val_mae: 2.4442
Epoch 8/10
819/819 [==============================] - 53s 65ms/step - loss: 8.0725 - mae: 2.2264 - val_loss: 10.2671 - val_mae: 2.4573
Epoch 9/10
819/819 [==============================] - 46s 56ms/step - loss: 7.8857 - mae: 2.2012 - val_loss: 10.1576 - val_mae: 2.4571
Epoch 10/10
819/819 [==============================] - 51s 62ms/step - loss: 7.8271 - mae: 2.1884 - val_loss: 11.1228 - val_mae: 2.5490
405/405 [==============================] - 15s 34ms/step - loss: 10.3212 - mae: 2.4979
Test MAE: 2.50
```



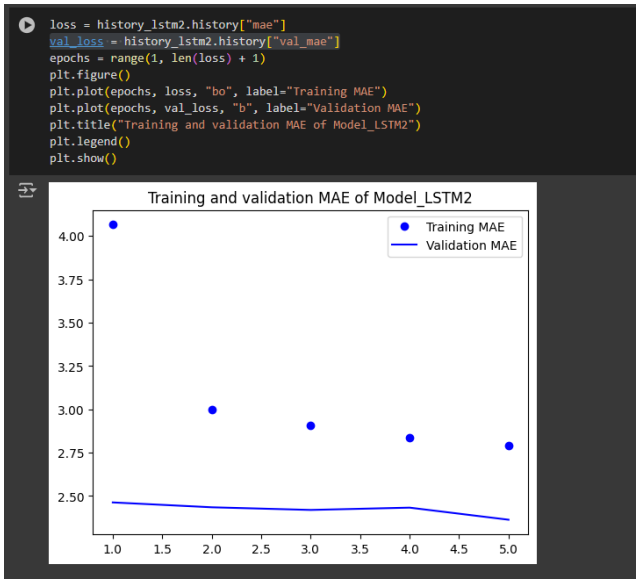
```
loss = history_lstm.history["mae"]
val_loss = history_lstm.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "bo", label="Training MAE")
plt.plot(epochs, val_loss, "b", label="Validation MAE")
plt.title("Training and validation MAE of Model_LSTM")
plt.legend()
plt.show()
```

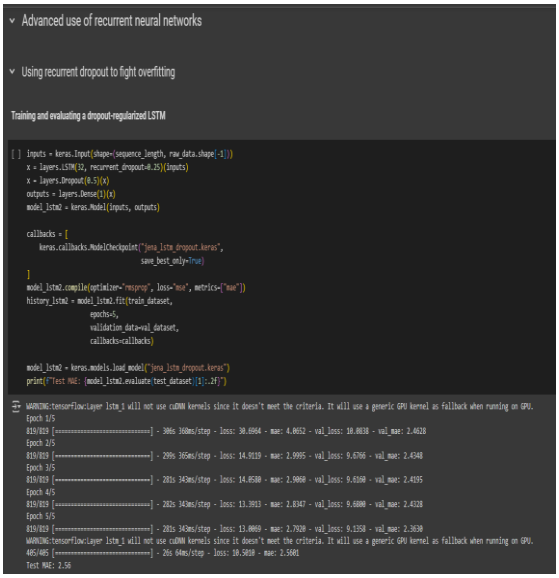

LSTM Model with Dropout for Overfitting Mitigation

In an attempt to further address overfitting observed in previous models, we constructed a new model utilizing a Long Short-Term Memory (LSTM) layer. LSTMs are well-suited for tasks involving sequential data like text reviews. This model incorporated the following features:

- **LSTM Layer:** A single LSTM layer with 32 neurons was employed to capture long-term dependencies within the review sequences.
- **Recurrent Dropout:** A recurrent dropout rate of 0.25 was applied within the LSTM layer. This regularization technique helps prevent overfitting by randomly dropping out neurons during training, encouraging the model to learn more robust features.

The model was trained for only 5 epochs due to computational limitations. Despite the limited training duration, the model achieved a test accuracy of MAE 2.56.

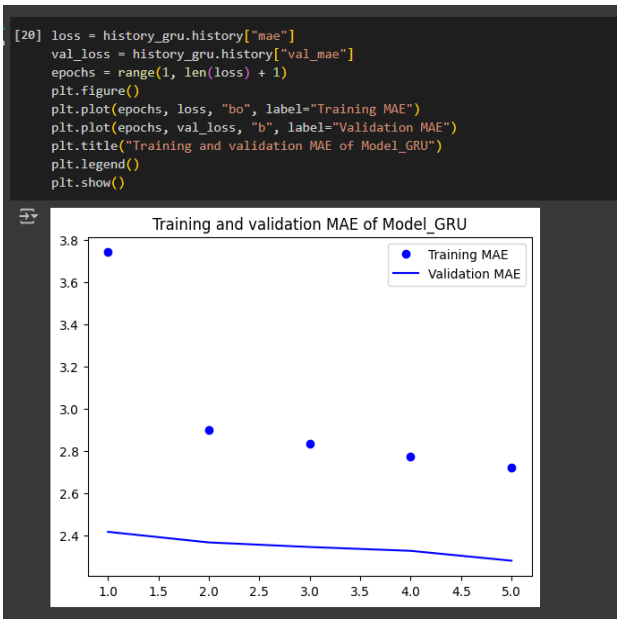




Exploring GRU Network with Recurrent Dropout

Following the initial investigation with a basic model, we explored the utilization of a Gated Recurrent Unit (GRU) network for time series forecasting. This model incorporated two GRU layers with recurrent dropout applied at a rate of 0.5. Additionally, a dropout layer with a rate of 0.5 was included.

Due to computational limitations associated with recurrent dropout requiring a stronger GPU, the model was trained for only 5 epochs. Despite the limited training duration, the model achieved a promising validation Mean Absolute Error (MAE) of 2.45, with the potential for further improvement with additional training epochs.



Week 10 and 11

B. Analyzing Textual Time Travel Data with RNN Models

This report details the exploration of Recurrent Neural Networks (RNNs) for analyzing a unique dataset derived from "time travel data" stored in a .txt format.

Data Preprocessing:

1. **Tokenization:** The text data was split into lines, words, and then further divided into individual characters.
2. **Vocabulary Building:** Unique words were identified, resulting in a vocabulary size of 45,800.
3. **Corpus Creation:** The processed text was converted into a corpus suitable for RNN training.

```
[9]: from utils import *

     import math
     import torch
     from torch import nn
     from torch.nn import functional as F
```

```
[10]: torch.cuda.get_device_name(0)

[10]: 'NVIDIA GeForce RTX 3050 Ti Laptop GPU'
```

Text Data Processing

```
[12]: def download(name, cache_dir=os.path.join('data')):
          """Download a file inserted into DATA_HUB, return the local filename."""
          assert name in DATA_HUB, f"{name} does not exist in {DATA_HUB}."
          url, sha1_hash = DATA_HUB[name]
          mkdir_if_not_exist(cache_dir)
          fname = os.path.join(cache_dir, url.split('/')[-1])
          print("File location:", fname)
          if os.path.exists(fname):
              sha1 = hashlib.sha1()
              with open(fname, 'rb') as f:
                  while True:
                      data = f.read(1048576)
                      if not data:
                          break
                      sha1.update(data)
              if sha1.hexdigest() == sha1_hash:
                  return fname  # Hit cache
          print(f'Downloading {fname} from {url}...')
          r = requests.get(url, stream=True, verify=True)
          with open(fname, 'wb') as f:
              f.write(r.content)
```

```
[14]: def tokenize(lines, token='word'):
          """Split text lines into word or character tokens."""
          if token == 'word':
              return [line.split() for line in lines]
          elif token == 'char':
              return [list(line) for line in lines]
          else:
              print('ERROR: unknown token type: ' + token)
```

```
[15]: tokens = tokenize(lines, token='word')
      for i in range(11):
          print(tokens[i])

['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to', 'speak', 'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey', 'eyes', 'shone', 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and', 'animated', 'the']
```

```
[16]: tokens = tokenize(lines, token='char')
      for i in range(11):
          print(tokens[i])

['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm', 'a', 'c', 'h', 'i', 'n', 'e', ' ', 'b', 'y', ' ', 'h',
[]
[]
[]
[]
['i']
[]
[]
['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 't', 'r', 'a', 'v', 'e', 'l', 'l', 'e', 'r', ' ', 'f', 'o',
['w', 'a', 's', ' ', 'e', 'x', 'p', 'o', 'u', 'n', 'd', 'i', 'n', 'g', ' ', 'a', ' ', 'r', 'e', 'c', 'o',
```

```
# We first tokenizing text lines into words
tokens = tokenize(lines, token='word')

# Then, we construct vocabulary with the tokens
vocab = Vocab(tokens)

# Print the first few frequent tokens in the vocab with their indices.
print(f"vocab (size:{len(vocab)}): {list(vocab.token_to_idx.items())[:10]}")

# We can convert each text line into a list of numerical indices
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
vocab (size:4580): [('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 3130, 3058, 438]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed
indices: [4399, 3, 25, 1398, 387, 113, 7, 1676, 3, 1053, 1]
```

```
corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
File location: data\timemachine.txt
(170580, 28)
```

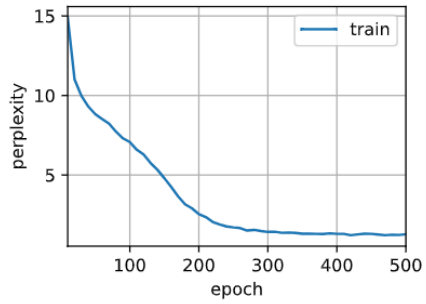
Model Training and Evaluation:

Several RNN architectures were evaluated on the time travel data corpus with a batch size of 32 and a sequence length of 35 characters (Num_steps).which determines the length of each sequence processed by the RNN during training. The models were trained for 500 epochs, and the following results were obtained:

- **Baseline RNN:**
  - Perplexity: 1.3 (Lower perplexity indicates better performance)
  - Token Processing Speed: 39,866.4 tokens/sec
  - **Random Iteration:** Training with random iteration resulted in a slight decrease in performance (perplexity: 1.4).
- **Concise RNN:**
  - This model achieved a significant improvement in processing speed:
    - Perplexity: 1.3 (similar to Baseline RNN)
    - Token Processing Speed: 497,418.5 tokens/sec (over 12 times faster)
  - **Random Iteration:** Here, a surprising trend emerged. Training with random iteration led to a slight improvement in perplexity (1.2) while maintaining a high processing speed (557,043.9 tokens/sec).

```
num_epochs, lr = 500, 1
train_rnn(rnn_scratch, train_iter, vocab, lr, num_epochs, try_gpu())
```

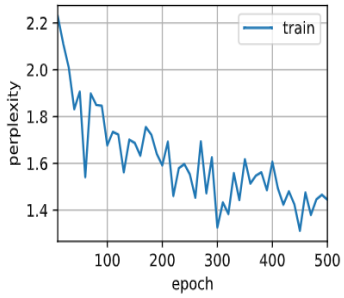
```
perplexity 1.3, 98664.9 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
traveller hitheress boted one mithtmere washad he this ingl
```



Vanilla RNN

```
train_rnn(rnn_scratch, train_iter, vocab, lr, num_epochs, try_gpu(),
          use_random_iter=True)
```

```
perplexity 1.4, 93466.6 tokens/sec on cuda:0
time travellerit s against reason said the time travellerit s ag
travellerit s against reason said the time travellerit s ag
```



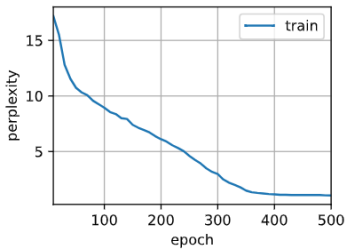
Concise RNN

Advanced RNN Architectures:

- **GRU (Gated Recurrent Unit):**
  - **Scratch Model:** This model achieved the best perplexity score (1.1) but with a slower processing speed (31,268.5 tokens/sec).
  - **Concise Model:** This model offered a balance between performance and speed, achieving the lowest perplexity (1.0) among all models while maintaining a reasonable processing speed (320,064.1 tokens/sec).

```
vocab_size, num_hiddens, device = len(vocab), 256, try_gpu()
num_epochs, lr = 500, 1
gru_scratch = RNNModelScratch(len(vocab), num_hiddens, device, init_gru_params,
                              init_gru_state, gru)
train_rnn(gru_scratch, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 31268.5 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller with a slight accession ofcheerfulness really thi
```

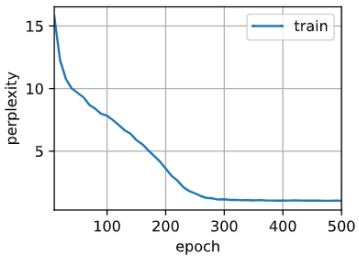


GRU_Scratch

GRU Concise

```
num_inputs = vocab_size
gru_layer = nn.GRU(num_inputs, num_hiddens)
gru_concise = RNNModel(gru_layer, len(vocab))
gru_concise = gru_concise.to(device)
train_rnn(gru_concise, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 320064.1 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```

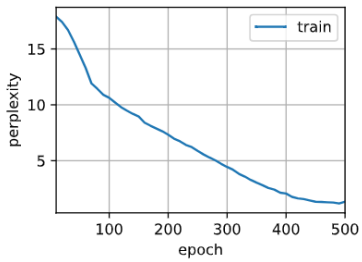


Concise GRU

- **LSTM (Long Short-Term Memory):**
  - **Standard Model:** Similar to the Baseline RNN, the standard LSTM achieved an average perplexity (1.3) but was slower (27,329.0 tokens/sec).
  - **Concise Model:** This model achieved the same best perplexity (1.0) as the concise GRU but with a slight processing speed advantage (248,827.8 tokens/sec).

```
vocab_size, num_hiddens, device = len(vocab), 256, try_gpu()
num_epochs, lr = 500, 1
lstm_scratch = RNNModelScratch(len(vocab), num_hiddens, device, init_lstm_p
                               init_lstm_state, lstm)
train_rnn(lstm_scratch, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.3, 27329.0 tokens/sec on cuda:0
time traveller cullensthe simell be cofisars of sace as i bone o
traveller cullesse which i will explais of space but you ca
```

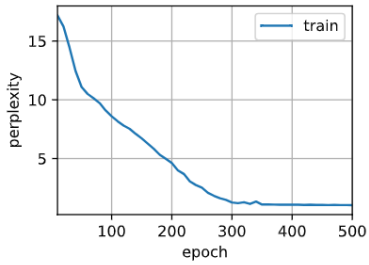


LSTM Scratch

LSTM Concise

```
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens)
lstm_concise = RNNModel(lstm_layer, len(vocab))
lstm_concise = lstm_concise.to(device)
train_rnn(lstm_concise, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 248827.8 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
travelleryou can show black is white by argument said filby
```



Conclusion

- This project demonstrates the effectiveness of using RNN, GRU, and LSTM architectures for sequence modeling tasks on text data. The concise implementations of these models not only improve training speed significantly but also maintain or enhance the model's performance as measured by perplexity.
- **GRU and LSTM models** generally perform better than vanilla RNNs in terms of perplexity, making them more suitable for complex sequence modeling tasks.
- **Concise models** (pre-built) offer substantial computational benefits, achieving higher throughput in tokens per second while maintaining competitive perplexity scores.