

Assignment III

Odunayo Adekoya

July 2024

1 Universal Approximation Theorem

1.1: You have a neural net with one hidden layer and no activation function (i.e., linear). Your features are X_1 and X_2 (the Euclidean coordinates of each point); explain how augmenting these features with new features could potentially make each dataset separable. For example, consider the following features

$$X_1, X_2, X_2^1, X_2^2, X_1X_2, \sin(X_1), \sin(X_2)$$

Does adding any subset of these features aid in linear separability of the datasets above?

Considering the features X_1 and X_2 on the datasets, it looks like Dataset 1, an XOR shape and Dataset 2, a spiral pattern both look non-linearly separable. On the other hand, Dataset 3 and Dataset 4 look to be linearly separable.

The product of X_1 and X_2 , can help to augment the features and convert the XOR dataset 1 pattern to a linear one. In addition, Using $\sin(X_1), \sin(X_2)$ could capture the spiral pattern of dataset 2, making it more linear.

1.2: Which one of the data distributions cannot be separable without an activation function, no matter what features are used? Add a nonlinear activation function and see how your results change.

Dataset 1 and Dataset 2 are not linearly separable without a nonlinear activation function. Figure 1 shows the dataset 1 with a linear activation and Figure 2 shows with relu activation function. We can observe that it is now separable and the test and training losses are significantly lower.

1.3: Using the Neural Network Playground Demo, modify the training parameters in order to train for the dataset(s) from the previous part . Report the parameter settings that allow for a good training and test loss on that model and dataset (by good, we mean that your test loss with a 50-50 data split is

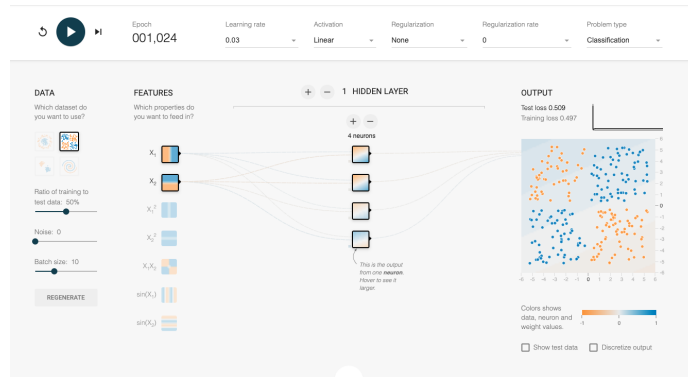


Figure 1: Dataset 1 without a nonlinear activation function

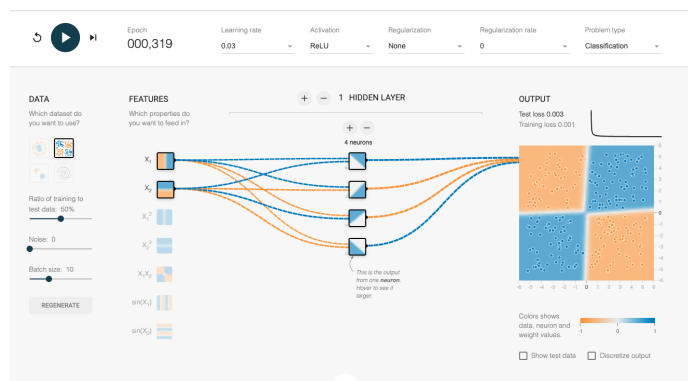


Figure 2: Dataset 1 with a relu activation function

S/N	Dataset	Activation Function	No of Neurons	Number of Hidden Layers	Test Loss	Training L
1	1	Relu	4	1	0.002	0.001
2	1	Tanh	4	1	0.003	0.001
3	2	Sigmoid	8	2	0.018	0.005
4	2	Tanh	6	2	0.027	0.005

Table 1: Caption

below 0.1). Describe any patterns you observe in terms of how this is related to the universal approximation theorem. Show your results for different activation functions and parameters, like the number of neurons, number of hidden layers and the features used.

2 Gradient Descent for Softmax Regression

2.1: Load the Iris data as provided in the notebook of the assignment. Take only petal length and petal width as your features (follow the instructions in the notebook). Add the bias term for every instance ($x_0 = 1$).

```
✓ [3] X = iris.data[['petal length (cm)', 'petal width (cm)']].values ;  
0s y = iris.target.values
```

add the bias term for every instance ($x_0 = 1$).

```
✓ [4] X_with_bias = np.c_[np.ones(X.shape[0]), X] #----- fill here  
0s
```

split the dataset into a training set, a validation set and a test set manually:

2.2: Write a function to convert the vector of class indices to a matrix of one-hot vector for each instance.

```
✓ [10] def to_one_hot(y):  
0s     y = np.array(y).reshape(-1)  
     ⚡ one_hot = np.eye(3)[y]  
     return one_hot
```

Check with the expected out put to make sure your code is doing the right thing:

```
✓ [11] y_train[:10]  
0s  
⇒ array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1])
```

```
✓ [12] to_one_hot(y_train[:10])  
0s  
⇒ array([[0., 1., 0.],  
         [1., 0., 0.],  
         [0., 0., 1.],  
         [0., 1., 0.],  
         [0., 1., 0.],  
         [0., 1., 0.],  
         [1., 0., 0.],  
         [0., 1., 0.],  
         [0., 0., 1.],  
         [0., 1., 0.]])
```

```
✓ [13] Y_train_one_hot = to_one_hot(y_train)  
0s     Y_valid_one_hot = to_one_hot(y_valid)  
     Y_test_one_hot = to_one_hot(y_test)
```

2.3: Normalize the data using Z-Score Normalization and define the softmax function to be used later.

```

0s [ ] # fill the following lines
      mean = np.mean(X_train[:, 1:], axis=0)
      std = np.std(X_train[:, 1:], axis=0)
      X_train[:, 1:] = (X_train[:, 1:] - mean) / std
      X_valid[:, 1:] = (X_valid[:, 1:] - mean) / std
      X_test[:, 1:] = (X_test[:, 1:] - mean) / std

0s [ ] def softmax(logits):
      exps = np.exp(logits)
      exp_sums = np.sum(exps, axis=1, keepdims=True)
      return exps / exp_sums

0s [ ] n_inputs = X_train.shape[1] # == 3 (2 features plus the bias term)
      n_outputs = len(np.unique(y_train)) # == 3 (there are 3 iris classes)

0s [ ] n_inputs
      3

```

2.4: Implement the gradient step using numpy. Make sure about the dimensions and the correctness of your calculations.

```

0s [ ] eta = 0.5
      n_epochs = 5001
      m = len(X_train)
      epsilon = 1e-5

      np.random.seed(42)
      Theta = np.random.randn(n_inputs, n_outputs)

      for epoch in range(n_epochs):
          logits = np.dot(X_train, Theta) # Compute logits
          Y_proba = softmax(logits) # Compute predicted probabilities

          if epoch % 1000 == 0:
              logits_valid = np.dot(X_valid, Theta)
              Y_proba_valid = softmax(logits_valid)
              xentropy_losses = -np.sum(Y_valid_one_hot * np.log(np.clip(Y_proba_valid, epsilon, 1 - epsilon)), axis=1)
              print(epoch, xentropy_losses.mean())

          error = Y_proba - Y_train_one_hot # Compute error
          gradients = np.dot(X_train.T, Y_proba - Y_train_one_hot) / len(X)
          Theta = Theta - eta * gradients

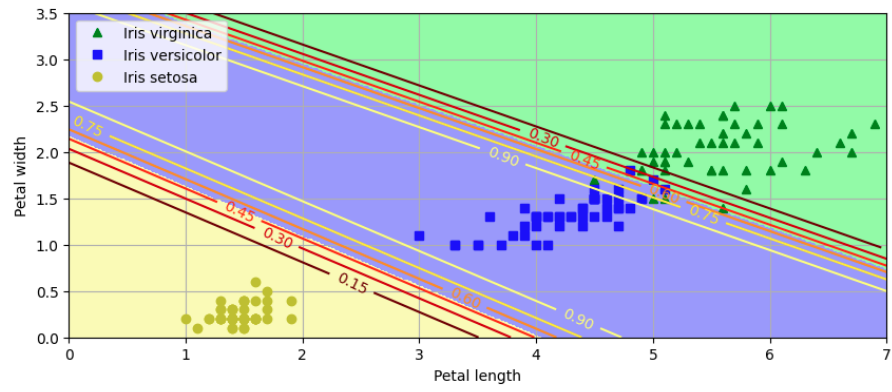
0s [ ] Theta
      array([[ 0.27563915,  5.40425127, -4.67375204],
             [-5.76092429, -0.04051629,  6.85618011],
             [-4.67397359, -0.08718461,  6.63833136]])

0s [ ] logits = X_valid @ Theta
      Y_proba = softmax(logits)
      y_predict = Y_proba.argmax(axis=1)

      accuracy_score = (y_predict == y_valid).mean()
      accuracy_score
      0.9333333333333333

```

2.5: Using the plotting function provided in the notebook document and plot the results of your model, showing a scatter plot of the decision boundary and the data points with respect to the petal length and petal width.



3 Backpropagation

3.1: Identify all the trainable parameters within this network.

The trainable parameters are:

- k_1, k_2, k_3 (*kernel weights*)
- b (bias for the convolutional layer)
- w_1, w_2 (*weights for the fully connected layer*)
- a (bias for the fully connected layer)

3.2: Calculate the gradients of the loss function (L) with respect to the parameters w_1 , w_2 , and a .

With respect to w_1

$$\frac{dL}{d\hat{y}} = -(y - \hat{y})$$

$$\frac{d\hat{y}}{dw_1} = v_1$$

$$\frac{dL}{dw_1} = -(y - \hat{y}) \cdot v_1$$

With respect to w_2

$$\frac{d\hat{y}}{dw_2} = v_2$$

$$\frac{dL}{dw_2} = -(y - \hat{y}) \cdot v_2$$

With respect to a

$$\frac{dL}{da} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{da}$$

$$\frac{dL}{da} = -(y - \hat{y}) \cdot 1$$

$$\frac{dL}{da} = -(y - \hat{y})$$

3.3: Given the gradients of the loss L with respect to the second layer activations (v), compute the gradients of the loss with respect to the first layer activations (z).

With respect to z_1

1. $v_1 = z_1$:

when $z_1 \geq z_2$ and $z_1 \geq 0$

$$\frac{dv_1}{dz_1} = 1, \frac{dL}{dz_1} = \delta_1$$

2. Else:

$$\frac{dv_1}{dz_1} = 0, \frac{dL}{dz_1} = 0$$

With respect to z_2

1. $v_1 = z_2$:

when $z_2 \geq z_1$ and $z_2 \geq 0$

$$\frac{dv_1}{dz_2} = 1, \frac{dL}{dz_2} = \delta_1$$

2. $v_2 = z_2$:

when $z_2 \geq z_3$ and $z_2 \geq 0$

$$\frac{dv_2}{dz_2} = 1, \frac{dL}{dz_2} = \delta_2$$

Combining both gives :

$$\frac{dL}{dz_2} = \delta_1 + \delta_2$$

With respect to z_3

1. $v_2 = z_3$:

when $z_3 \geq z_2$ and $z_3 \geq 0$

$$\frac{dv_2}{dz_3} = 0, \frac{dL}{dz_3} = \delta_2$$

2. Else:

$$\frac{dv_2}{dz_3} = 0, \frac{dL}{dz_3} = 0$$

3.4: Given the gradients of the loss L with respect to the first layer activations (z), calculate the gradients of the loss with respect to the convolution filter (k) and bias (b).

With respect to k_1

$$\frac{dL}{dk_1} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dk_1} + \frac{dL}{dz_2} \cdot \frac{dz_2}{dk_1} + \frac{dL}{dz_3} \cdot \frac{dz_3}{dk_1}$$

$$\frac{dz_1}{dk_1} = x_1, \frac{dz_2}{dk_1} = x_2, \frac{dz_3}{dk_1} = x_3$$

$$\text{Hence, } \frac{dL}{dk_1} = \delta_1 x_1 + \delta_2 x_2 + \delta_3 x_3$$

With respect to k_2

$$\frac{dL}{dk_2} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dk_2} + \frac{dL}{dz_2} \cdot \frac{dz_2}{dk_2} + \frac{dL}{dz_3} \cdot \frac{dz_3}{dk_2}$$

$$\frac{dz_1}{dk_2} = x_2, \frac{dz_2}{dk_2} = x_3, \frac{dz_3}{dk_2} = x_4$$

$$\text{Hence, } \frac{dL}{dk_2} = \delta_1 x_2 + \delta_2 x_3 + \delta_3 x_4$$

With respect to k_3

$$\frac{dL}{dk_3} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dk_3} + \frac{dL}{dz_2} \cdot \frac{dz_2}{dk_3} + \frac{dL}{dz_3} \cdot \frac{dz_3}{dk_3}$$

$$\frac{dz_1}{dk_3} = x_3, \frac{dz_2}{dk_3} = x_4, \frac{dz_3}{dk_3} = x_5$$

$$\text{Hence, } \frac{dL}{dk_3} = \delta_1 x_3 + \delta_2 x_4 + \delta_3 x_5$$

With respect to b

$$\frac{dL}{db} = \frac{dL}{dz_1} \cdot \frac{dz_1}{db} + \frac{dL}{dz_2} \cdot \frac{dz_2}{db} + \frac{dL}{dz_3} \cdot \frac{dz_3}{db}$$

$$\frac{dz_1}{db} = 1, \frac{dz_2}{db} = 1, \frac{dz_3}{db} = 1$$

$$\text{Hence, } \frac{dL}{db} = \delta_1 + \delta_2 + \delta_3$$

4 Convolutional Neural Networks

4.1: Report the ‘Training Generator Info’ and ‘Validation Generator Info’ that you got when you executed `print_generator_info()` function.

```
Training Generator Info:
Number of images: 148
Batch size: 20
Class indices: {'ipynb_checkpoints': 0, 'not_polar': 1, 'polar': 2}
Number of classes: 3
Number of filenames loaded: 148

Validation Generator Info:
Number of images: 36
Batch size: 20
Class indices: {'ipynb_checkpoints': 0, 'not_polar': 1, 'polar': 2}
Number of classes: 3
Number of filenames loaded: 36
```

4.2: Report the model summary and total number of parameters in the model you just built.

```
model = create_cnn_model()

# Print the model summary
print(model.summary())
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_1 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 64)	0
flatten (Flatten)	(None, 82944)	0
dropout (Dropout)	(None, 82944)	0
dense (Dense)	(None, 512)	42467840
dense_1 (Dense)	(None, 1)	513

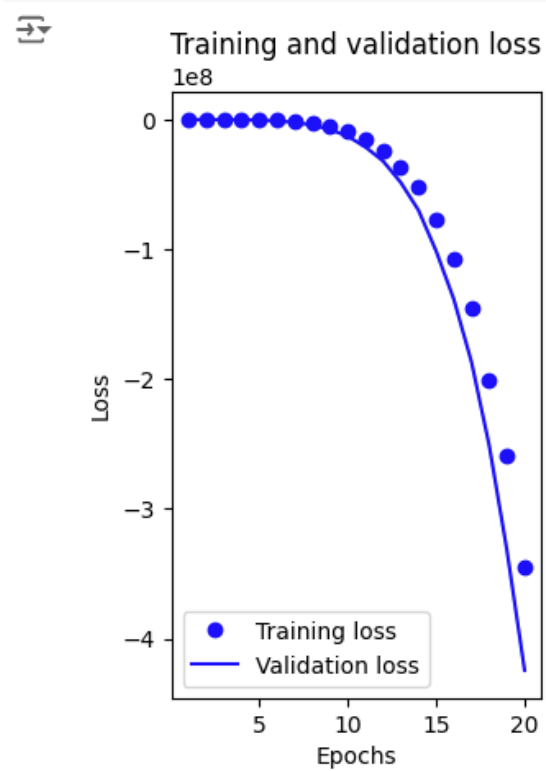
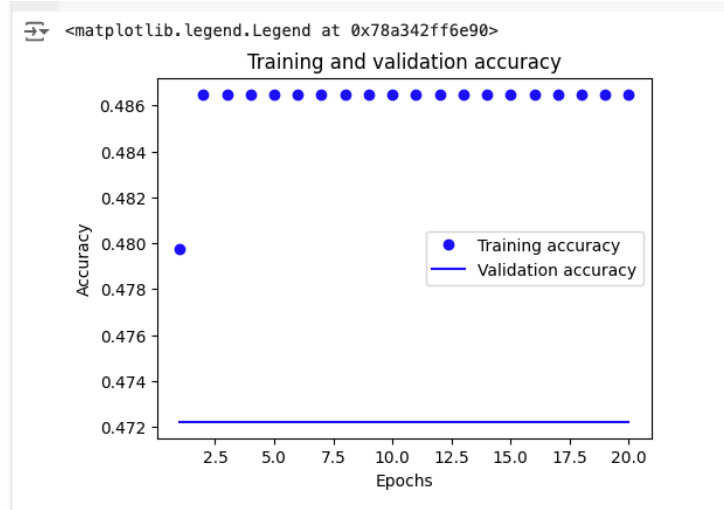
=====
Total params: 42487745 (162.08 MB)
Trainable params: 42487745 (162.08 MB)
Non-trainable params: 0 (0.00 Byte)
=====
None

```
[ ] total_params = model.count_params()
print(f"Total number of parameters: {total_params}")
```

Total number of parameters: 42487745

4.3: Plot the training and validation accuracies and losses. Does this model

appear to be efficient at this stage?



4.4: Report the predictions of the model. Which of the test images were classified wrong and why did that happen and how can we address that?

✓ Inference for test_1.jpg

```
img_path = '/content/test/test_1.jpg'
load_display_predict_image(img_path, model)
```



```
1/1 [=====] - 0s 159ms/step
[[1.]]
Polar Bear
```

Prediction is Correct

✓ Inference for test_2.jpg

```
[ ] img_path = '/content/test/test_2.jpg'
load_display_predict_image(img_path, model)
```



```
1/1 [=====] - 0s 132ms/step
[[1.]]
Polar Bear
```

Prediction is Correct

▼ Inference for test_3.jpg

```
img_path = '/content/test/test_3.jpg'  
load_display_predict_image(img_path, model)
```



```
1/1 [=====] - 0s 61ms/step  
[[1.]]  
Polar Bear
```

Prediction is Correct

▼ Inference for test_4.jpg

```
img_path = '/content/test/test_4.jpg'  
load_display_predict_image(img_path, model)
```



```
1/1 [=====] - 0s 65ms/step  
[[1.]]  
Polar Bear
```

Prediction is Incorrect

▼ Inference for test_5.jpg

```
img_path = '/content/test/test_5.jpg'  
load_display_predict_image(img_path, model)
```



```
1/1 [=====] - 0s 63ms/step  
[[1.]]  
Polar Bear
```

Prediction is Incorrect

▼ Inference for test_6.jpg

```
img_path = '/content/test/test_6.jpg'  
load_display_predict_image(img_path, model)
```



```
1/1 [=====] - 0s 61ms/step  
[[1.]]  
Polar Bear
```

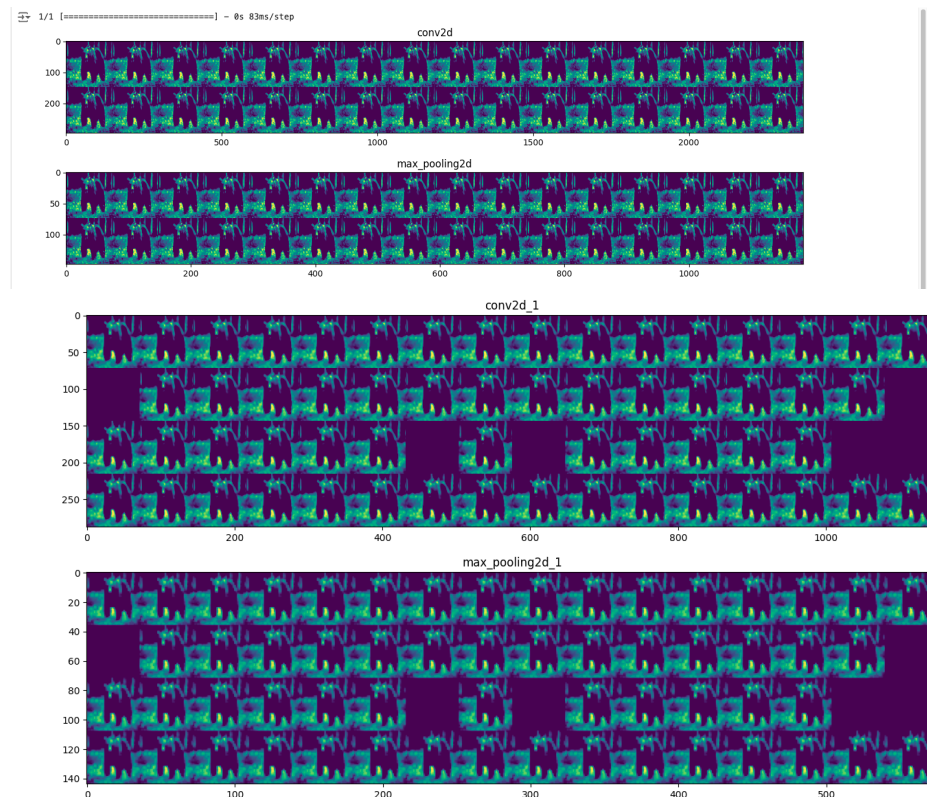
Prediction is Correct

The model might be too simple to capture the complexity or there might not have been sufficient training data.

Solution: *Experiment with different architectures, ensuring an appropriate balance between model complexity and the amount of training data.*

4.5: Upload the visualizations of the activations for test_4.png and comment

on increasing abstractness as we go deeper into the model.



Comments on increasing abstractness:

Low-Level Features: Early layers (e.g., first convolutional layer) focus on basic features such as edges, gradients, and textures. These features are still closely related to the original image and are relatively easy to interpret.

High-Level Features: Deeper layers (e.g., second convolutional layer and beyond) combine these basic features to form higher-level abstractions, such as shapes, patterns, and object parts. These features are more abstract and less visually similar to the original image.

Spatial Reduction: MaxPooling layers reduce the spatial dimensions of the activations, focusing on the most important features and reducing the computational complexity of subsequent layers. This makes the representation more abstract and compact, preserving only the essential information.

5 word2vec

5.1 & 5.2: What is the total number of words? What is the vocab size, number of contexts and number of targets?

```
➡ Number of words: 280000
   Vocabulary size: 28
   Length of contexts array: 279996
   Length of targets array: 279996
```

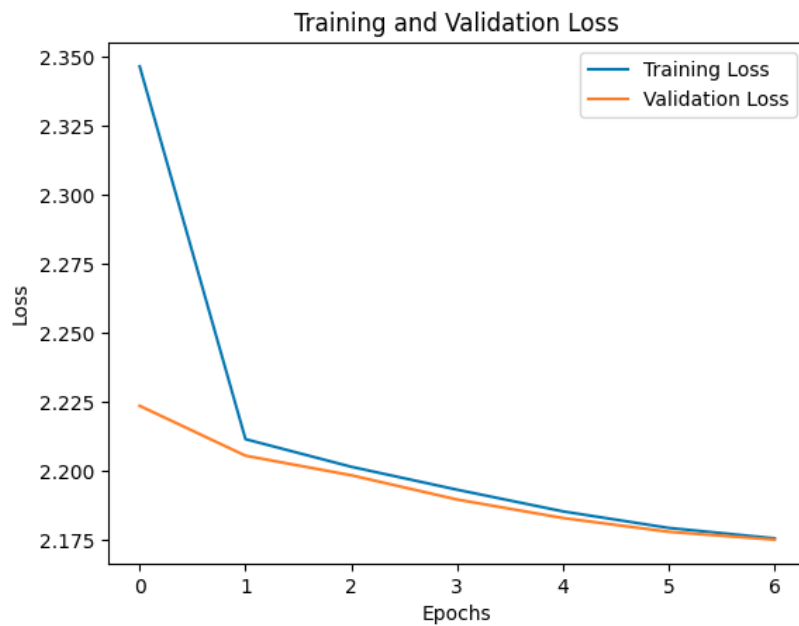
5.3: Print the model summary.

```
➡ Model: "model"
```

Layer (type)	Output Shape	Param #
context_words (InputLayer)	[(None, 4)]	0
embedding (Embedding)	(None, 4, 2)	56
averaging (Lambda)	(None, 2)	0
output (Dense)	(None, 28)	84

```
=====
Total params: 140 (560.00 Byte)
Trainable params: 140 (560.00 Byte)
Non-trainable params: 0 (0.00 Byte)
```

5.4: Provide the plot in your submission.



5.5: Use these functions to find and return the top 3 similar countries for each of the following: Poland, Thailand, and Morocco..

```

0 s  query_words = ['poland', 'thailand', 'morocco']

    for query_word in query_words:
        find_similar_words(query_word, vocab, embeddings)
        print("\n")

```

```

Words most similar to 'poland':
germany: 0.9997
uk: 0.9997
switzerland: 0.9994

```

```

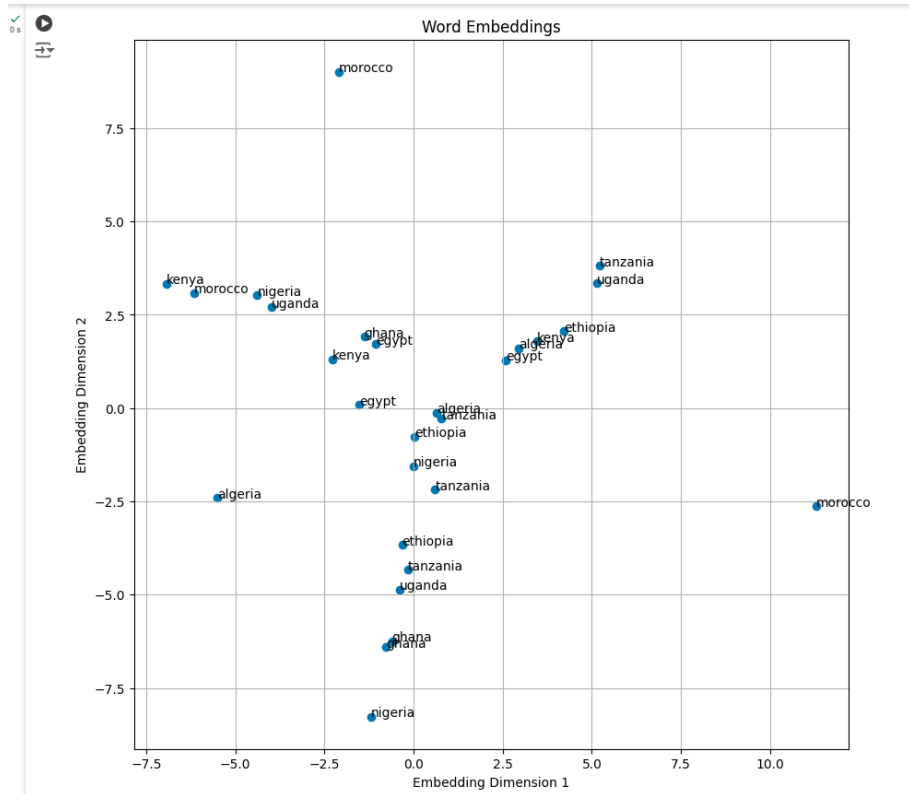
Words most similar to 'thailand':
philippines: 1.0000
indonesia: 0.9999
vietnam: 0.9994

```

```

Words most similar to 'morocco':
tanzania: 0.9999
nigeria: 0.9983
ghana: 0.9913

```




5.6: Consider a small window size, e.g., 2 or 3. In this scenario, is there a possibility that antonyms (opposite words) might end up with similar embeddings? Explain your views.


Yes, there is a possibility that antonyms (opposite words) might end up with similar embeddings when using a small context window size (e.g., 2 or 3) in models like Continuous Bag of Words (CBOW). This happens because these models learn word embeddings based on the local context in which words appear, and antonyms often appear in similar contexts due to their semantic relationships.


For example, the words "hot" and "cold" might both appear in contexts related to weather, temperature, or sensations (e.g., "The weather is very hot/cold today"). In a small window size, the context captured around these words would be very similar. For example, in the sentences "The soup is hot" and "The soup is cold", the context words around "hot" and "cold" (like "The", "soup", and "is") are identical.

6 Next Word Prediction


6.1: Print the length of the final processed text obtained.


 0 s


 `print(len(text))`

 140269


6.2: Print the total number of words.


 0 s


 `print(total_words)`

 2751

6.3: For the following steps print the number of input sequences finally created for the actual given text.

 0 s

 `print(len(input_sequences))`

 23693

6.4: Print the size of the train and validation subsets for the features and targets.

```

➡ Training features size: (18954, 15)
   Validation features size: (4739, 15)
   Training labels size: (18954, 2751)
   Validation labels size: (4739, 2751)

```

6.5: After defining the model, print its summary. Build and train the model for 20 epochs. After training, visualize the performance by plotting the training and validation accuracy and loss over the epochs.

```

➡ Model: "sequential"

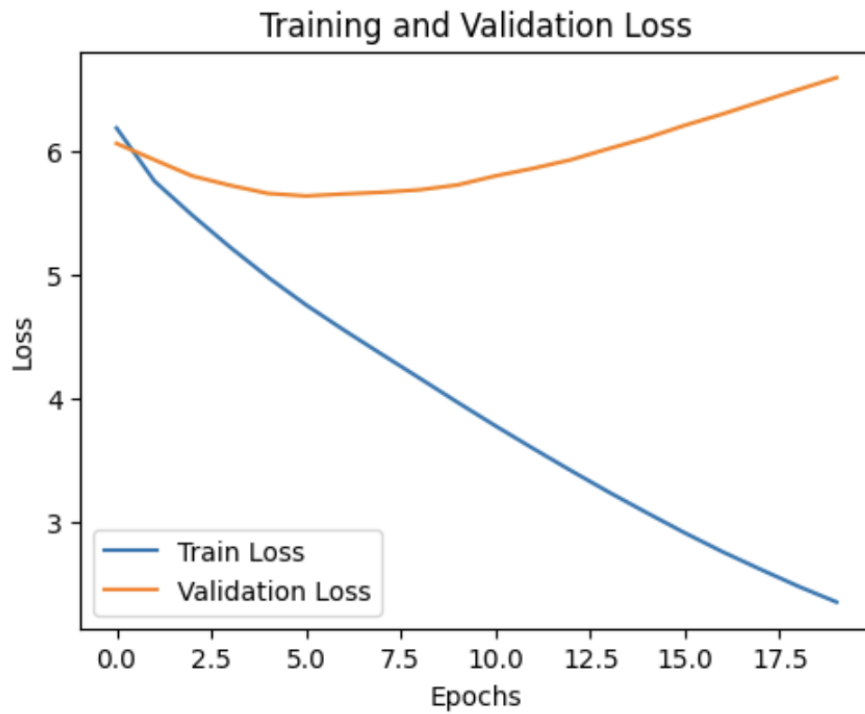
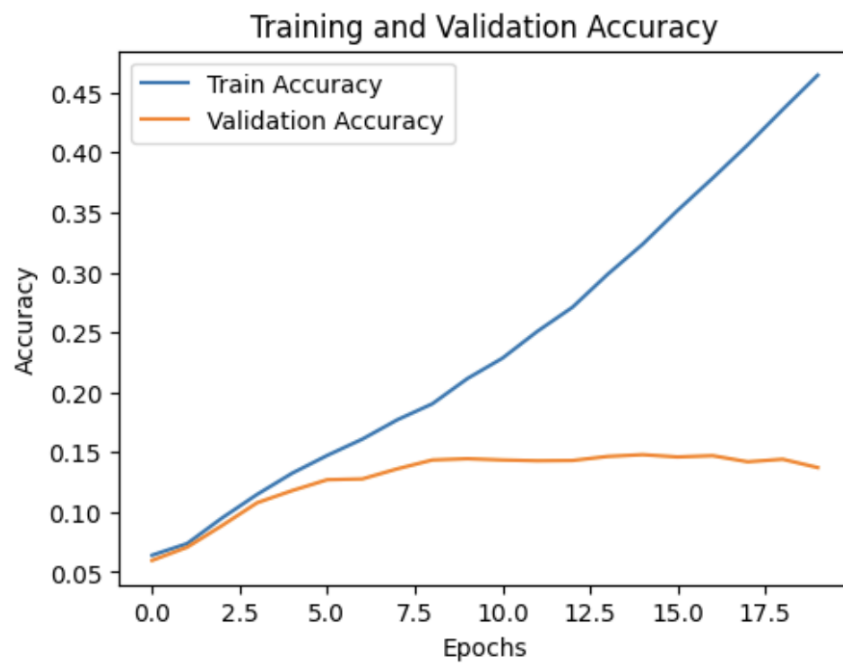
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 15, 100)	275100
lstm (LSTM)	(None, 150)	150600
dense (Dense)	(None, 2751)	415401

```

=====
Total params: 841101 (3.21 MB)
Trainable params: 841101 (3.21 MB)
Non-trainable params: 0 (0.00 Byte)

```



Is the model overfitting? Explain your observation.

Yes, the model is overfitting. The increasing gap between the training accuracy and validation accuracy indicates that the model performs well on the training data but struggles with the validation data. The same pattern is observed with the loss curves, where the training loss decreases, but the validation loss increases after a certain point.

Create one more model of your choice (you may explore Bidirectional, LayerNormalization, Dropout, Attention and GRU etc) that improves upon the previous model in terms of overfitting. Print this new model summary, train for 20 epochs, and then plot the training and validation accuracy and loss.

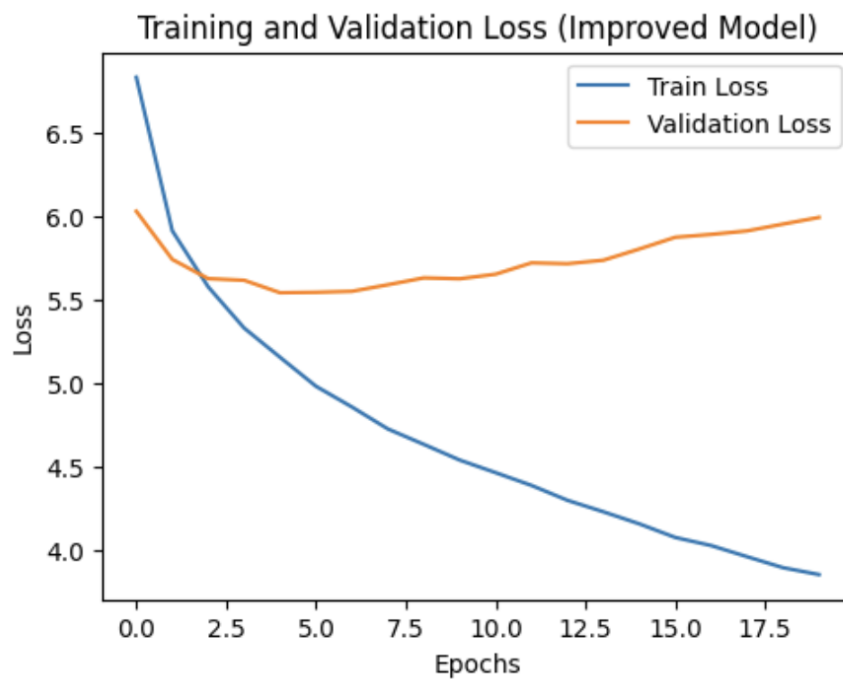
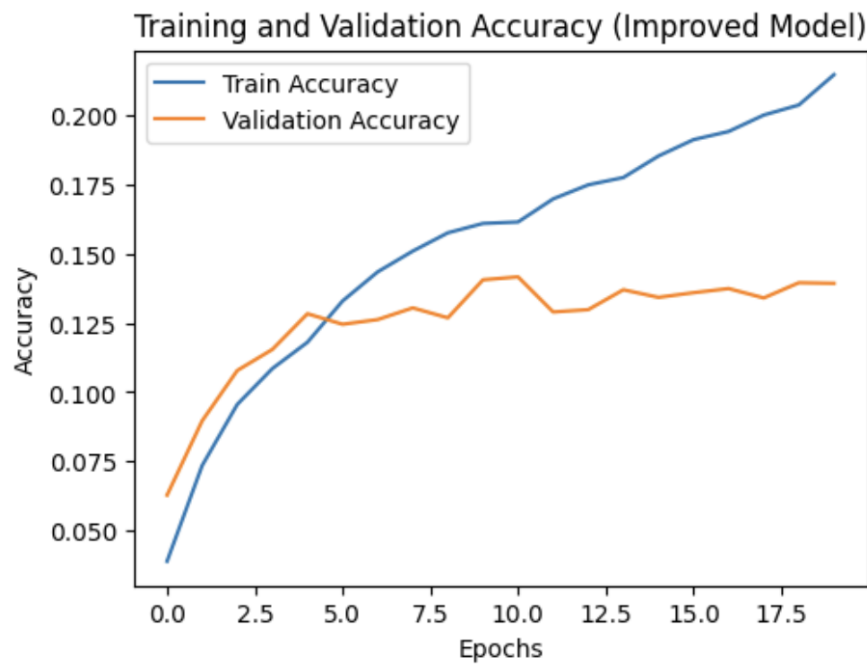
4 min

Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 15, 200)	550200
bidirectional (Bidirectional)	(None, 15, 512)	703488
batch_normalization (Batch Normalization)	(None, 15, 512)	2048
dropout (Dropout)	(None, 15, 512)	0
gru_1 (GRU)	(None, 15, 256)	591360
batch_normalization_1 (Batch Normalization)	(None, 15, 256)	1024
dropout_1 (Dropout)	(None, 15, 256)	0
gru_2 (GRU)	(None, 256)	394752
batch_normalization_2 (Batch Normalization)	(None, 256)	1024
dropout_2 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 2751)	707007

Total params: 2950903 (11.26 MB)
Trainable params: 2948855 (11.25 MB)
Non-trainable params: 2048 (8.00 KB)

Epoch 1/20



6.6: Demonstrate the function by generating text with temperature values 0.05 and 1.5 using the previously created model with less overfitting.

```
# Generate text
seed_text = "Forest is"
next_words = 10
generated_text_1 = generate_text(seed_text, next_words, model, max_sequence_length, temperature=0.05)
generated_text_2 = generate_text(seed_text, next_words, improved_model, max_sequence_length, temperature=1.5)
generated_text_3 = generate_text(seed_text, next_words, model, max_sequence_length, temperature=0.05)
generated_text_4 = generate_text(seed_text, next_words, improved_model, max_sequence_length, temperature=1.5)

print(generated_text_1)
print(generated_text_2)
print(generated_text_3)
print(generated_text_4)
```

Forest is great begged the relief accidentally reeling tree lie down tittered
Forest is yourself flinging wonder move ropewill fight seeneverything mouseof steady swimming
Forest is hush oblong and change the position in sugar actually pictures
Forest is im loud welcome least provoking yards whiting fryingpan sighing moments

6.7: In the preprocessing step for NLP, removing stop words is often considered important. We did not perform stop word removal in our text generation task. Should we have done that? Explain reasons to support your answer.

For this specific text generation task, the inclusion of stop words ensures that the generated text is coherent, grammatically correct, and natural-sounding. While stop word removal is useful in tasks focused on content analysis or information retrieval (where individual word importance is critical), it is less appropriate for generating fluid and natural language text. Therefore, in this case, keeping stop words is a great approach to achieve more realistic and better quality text generation.