[7 marks] Question 1: Universal Approximation Theorem

Universal Approximation Theorem states that a feed-forward neural network with a single hidden layer, a finite number of neurons, and a non-constant, bounded activation function can approximate any continuous function on a compact domain to arbitrary accuracy. Mathematically, this is represented as follows. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuous function on a compact set $K \subset \mathbb{R}^n$. Then, for any $\epsilon > 0$, there exists a neural network with weights W_1, W_2, biases b_1, b_2, and an overall network function $\phi(x)$ such that:

$$\sup_{x \in K} |f(x) - \phi(x)| < \epsilon$$

[2 marks] Q1.1: Imagine the following data distributions for a 2-class problem.

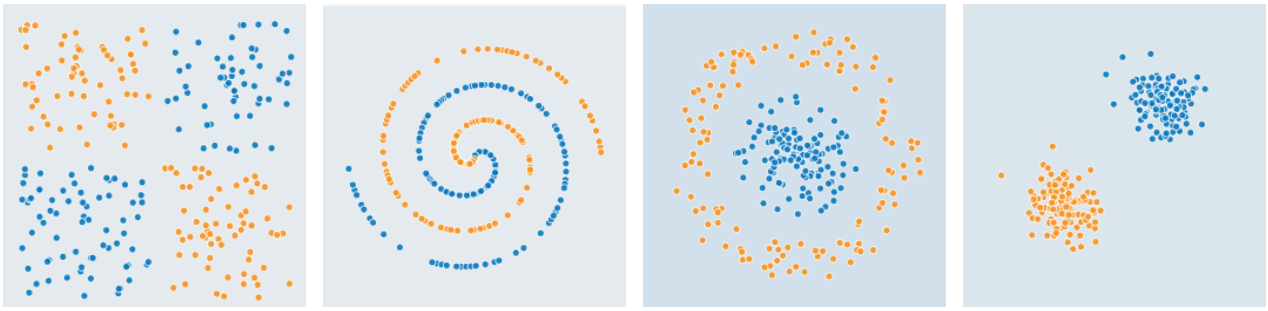


Figure 1: Left to right: dataset 1; dataset 2; dataset 3; dataset 4.

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

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

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

[2 marks] Q1.2: 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.

[3 marks] Q1.3: 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 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 .

Q 1.1:

X_1	X_2	$X_1 \cdot X_2$	Class
0	0	0	0
1	1	1	0
1	0	0	1
0	1	0	1

Dataset 1: XOR: It is not linear separable by only using X_1, X_2.
Add a new feature: $X_1 \cdot X_2$, it can achieve linear separability.

Dataset 2: Spiral pattern is not linear separable by only using X_1, X_2. But adding non-linear transformations like $\sin(X_1)$ and $\sin(X_2)$ can help in achieving separability. Because $\sin(x)$ is cyclical function and the dataset is cyclical.

Dataset 3: Concentric circles: X_1^2, X_2^2 can help in this situation. The circle function is $X_1^2 + X_2^2 = r^2$ which can use to separate this dataset.

Dataset 4: Two clusters: it is linear separable by only using X_1, X_2.

Q1.2: Dataset 2: Spiral pattern is not separable without activation function, no matter what features will be used.

Adding a non-linear activation function: Sigmoid, Tanh can allow the neural network to capture the complex non-linear relationships

Q1.3: A good model I found for dataset 2: Features: X_1, X_2, $\sin(X_1)$, $\sin(X_2)$. Number of neurons: 6. Number of hidden layers: 1. Learning Rate: 0.01. Epochs: 1500. Activation functions: Sigmoid. ⇒ Result: Test loss: 0.015, Training loss: 0.005

Change of Activation Functions with other parameters unchange: ① ReLu Training loss: 0.134, Testing loss: 0.333 ② Tanh: Testing loss: 0.015. Training loss: 0.001 ③ Linear: Testing loss: 0.524, Training loss: 0.432
⇒ Result: the activation function is very important. Inappropriate activation function will result in poor model.

Change of Hidden layer: Increasing hidden layers will improve the result but the computation time for each epoch will increase. It can also achieve a same performance with less epoch compared to the less hidden layer model. Too many hidden layers may lead to overfitting.

Change of neurons: Increasing the number of neurons will also improve the model performance and achieve a same performance with less epoch. If the number of neurons is too little, the model cannot capture the features.

Change of learning rate: If the learning rate is too small, then it will take forever to train. If the learning rate is too large, the model will fail to converge.

Change of features: ① X_1, X_2, X_1X_2: Testing loss: 0.195. ② X_1, X_2: Testing loss: 0.443 ③ $\sin(x_1)$, $\sin(x_2)$: Testing loss: 0.448.
⇒ Good features will improve the model. With poor features, the model will fa

Pattern observed: the universal approximation theorem implied that with sufficient neurons and an appropriate activation function, a neural network can approximate any continuous function. The activation function played a crucial role. Increasing the number of neurons and layers will improve the model but it will also increase the risk of overfitting. An appropriate features and learning rate is also important.

[15 marks] Question 2: Gradient Descent for Softmax Regression

Using only NumPy, implement batch gradient descent for softmax regression. Train this classifier on the Iris dataset, demonstrating the process without relying on Scikit-learn.

[1 mark] Q2.1: 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 ($x_0 = 1$).

[2 marks] Q2.2: The targets are class indices (0, 1 or 2). They have to turn to class probabilities to train the Softmax Regression model. Each instance must show a probability equal to 0.0 for all classes except for the target class (1.0) (so the class probability vectors should be a one-hot vector). Write a function to convert the vector of class indices to a matrix of one-hot vector for each instance.

[2 marks] Q2.3: Normalize the data using Z-Score Normalization and define the softmax function to be used later.

[8 marks] Q2.4: Implement the gradient step using numpy. Make sure about the dimensions and the correctness of your calculations. The cost function is given by:

$$J(\mathbf{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

and the gradients are:

$$\nabla_{\theta^{(k)}} J(\mathbf{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

to avoid problems with getting `nan` values regarding $\hat{p}_k^{(i)} = 0$ you can add a tiny value ϵ to $\log\left(\hat{p}_k^{(i)}\right)$.

[2 marks] Q2.5: 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.

Q2:

```
import matplotlib.pyplot as plt
```
[2] ✓ 1.0s

#Using only NumPy, implement batch gradient descent for softmax regression. Train this classifier on the Iri

1. Load the Iris data as provided in the notebook of the ass instance (x0 = 1).

```
from sklearn.datasets import load_iris
import numpy as np
iris = load_iris(as_frame=True)
```
[50] ✓ 0.0s

load the data (take the petal length and petal width).

```
X = iris.data.iloc[:, 2:4].values # ---- fill here    (take the petal length and petal width)
y = iris.target.values# --- fill here
```
[48] ✓ 0.0s

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

```
X_with_bias = np.c_[np.ones([len(X), 1]), X] #----- fill here
```
[38] ✓ 0.0s

2. The targets are class indices (0, 1 or 2). The Softmax Regression model. Each instance mu except for the target class (1.0) (so the class a function to convert the vector of class indic

```
def to_one_hot(y):
    n_classes = y.max() + 1
    m = len(y)
    Y_one_hot = np.zeros((m, n_classes))
    Y_one_hot[np.arange(m), y] = 1
    return Y_one_hot
    #---- fill here
```
[40] ✓ 0.0s

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

```
y_train[:10]
```
[41] ✓ 0.0s

```
array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1])
```

```
to_one_hot(y_train[:10])
```
[61] ✓ 0.0s

```
array([[0., 1., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.]])
```

```
Y_train_one_hot = to_one_hot(y_train)
Y_valid_one_hot = to_one_hot(y_valid)
Y_test_one_hot = to_one_hot(y_test)
```
[55] ✓ 0.0s

3. Normalize the data using Z-Score Normali later.

```
# fill the following lines
mean = X_train[:, 1:].mean(axis=0)
std = X_train[:, 1:].std(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
```
] ✓ 0.0s

```
def softmax(logits):
    exps = np.exp(logits)
    exp_sums = np.sum(exps, axis=1, keepdims=True)
    return exps / exp_sums
```
] ✓ 0.0s

```
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)
```
] ✓ 0.0s

```
n_inputs, n_outputs
```
] ✓ 0.0s

```
(3, 3)
```

4. Implement the gradient step using numpy. Make sure about the din of your calculations.

turn the math equations into Python code.

So the equations we will need are the cost function:

$J(\mathbf{\Theta}) =

- \dfrac{1}{m}\sum\limits_{i=1}^{m}\sum\limits_{k=1}^{K}{y_k^{(i)}\log\left(\hat{p}_k^{(i)}\right)}$

And the equation for the gradients:

$$\nabla_{\theta(k)} J(\mathbf{\Theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) \mathbf{x}^{(i)}$$

Note that $\log\left(\hat{p}_k^{(i)}\right)$ may not be computable if $\hat{p}_k^{(i)} = 0$. So we will add a tiny value ϵ to $\log\left(\hat{p}_k^{(i)}\right)$ to avoid getting `nan` values.

```
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)


# fill the following lines:
for epoch in range(n_epochs):
    logits = np.dot(X_train, Theta) #----
    Y_proba = softmax(logits) #----
    if epoch % 1000 == 0:
        Y_proba_valid = softmax(np.dot(X_valid, Theta))#(----)
        xentropy_losses = -np.sum(Y_valid_one_hot * np.log(Y_proba_valid + epsilon), axis=1) #-----
        print(epoch, xentropy_losses.mean())
    error = Y_proba - Y_train_one_hot#-----
    gradients = 1/m * X_train.T.dot(error)# -----
    Theta = Theta - eta * gradients
```
] ✓ 0.0s

```
0 3.7085808486476917
1000 0.14519367480830644
2000 0.13013095755040877
3000 0.12009639326384533
4000 0.11372961364786881
5000 0.11002459532472428
```

```
Theta
```
[79] ✓ 0.0s

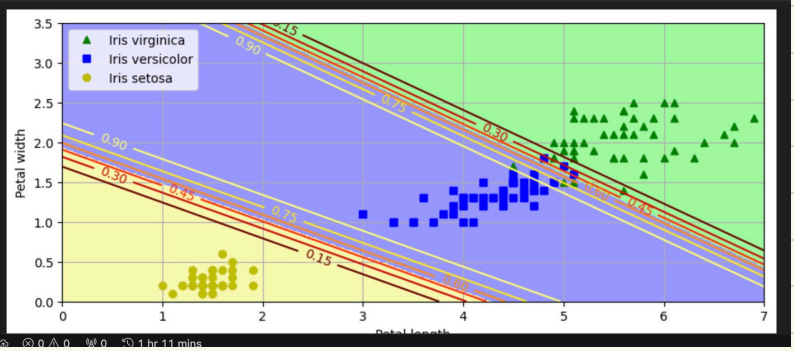
```
array([[ 0.41931626,  6.11112089, -5.52429876],
       [-6.53054533, -0.74608616,  8.33137102],
       [-5.28115784,  0.25152675,  6.90680425]])
```

```
logits = X_valid @ Theta
Y_proba = softmax(logits)
y_predict = Y_proba.argmax(axis=1)

accuracy_score = (y_predict == y_valid).mean()
accuracy_score
```
[80] ✓ 0.0s

```
np.float64(0.9333333333333333)
```

5. Document and plot the results of your model.

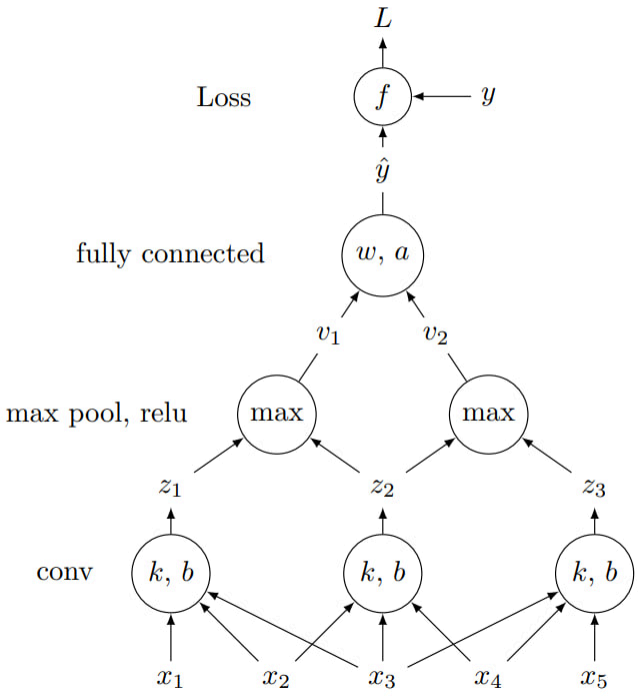
```
import matplotlib as mpl

custom_cmap = mpl.colors.ListedColormap(['#fafab0', '#9898ff', '#a0faa0'])

x0, x1 = np.meshgrid(np.linspace(0, 8, 500).reshape(-1, 1),
                     np.linspace(0, 3.5, 200).reshape(-1, 1))
X_new = np.c_[x0.ravel(), x1.ravel()]
X_new = (X_new - mean) / std
X_new_with_bias = np.c_[np.ones(len(X_new)), X_new]

logits = np.dot(X_new_with_bias, Theta)#---- fill here
Y_proba = Y_proba = softmax(logits)#---- fill here
y_predict = Y_proba.argmax(axis=1) #---- fill here

zz1 = Y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y == 2, 0], X[y == 2, 1], "g^", label="Iris virginica")
plt.plot(X[y == 1, 0], X[y == 1, 1], "bs", label="Iris versicolor")
plt.plot(X[y == 0, 0], X[y == 0, 1], "yo", label="Iris setosa")

plt.contourf(x0, x1, zz, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap="hot")
plt.clabel(contour, inline=1)
plt.xlabel("Petal length")
plt.ylabel("Petal width")
plt.legend(loc="upper left")
plt.axis([0, 7, 0, 3.5])
plt.grid()
plt.show()
```
1] ✓ 0.1s



ble ⊗ 0 ⚠ 0 0 1 hr 11 mins

[10 marks] Question 3: Backpropagation

Consider the following 1-dimensional convolutional neural network (ConvNet) where all variables are scalars:



$$L = \frac{1}{2}(y - \hat{y})^2$$

$$\hat{y} = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + a$$

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \max\{z_1, z_2, 0\} \\ \max\{z_2, z_3, 0\} \end{bmatrix}$$

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} k_1 & k_2 & k_3 & 0 & 0 \\ 0 & k_1 & k_2 & k_3 & 0 \\ 0 & 0 & k_1 & k_2 & k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix}$$

Figure 2: 1D convolutional net

[1 mark] Q3.1: Identify all the trainable parameters within this network.

[3 marks] Q3.2: Calculate the gradients of the loss function (L) with respect to the parameters w_1, w_2, and a. That is, find $\frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_2}$, and $\frac{\partial L}{\partial a}$.

[3 marks] Q3.3: 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). Specifically, if $\frac{\partial L}{\partial v_1} = \delta_1$ and $\frac{\partial L}{\partial v_2} = \delta_2$, find $\frac{\partial L}{\partial z_1}$, $\frac{\partial L}{\partial z_2}$, and $\frac{\partial L}{\partial z_3}$.

[3 marks] Q3.4: 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). Specifically, if $\frac{\partial L}{\partial z_1} = \delta_1$, $\frac{\partial L}{\partial z_2} = \delta_2$, and $\frac{\partial L}{\partial z_3} = \delta_3$, find $\frac{\partial L}{\partial k_1}$, $\frac{\partial L}{\partial k_2}$, $\frac{\partial L}{\partial k_3}$, and $\frac{\partial L}{\partial b}$.

Q3.1: W_1, W_2: the weights of fully connected layer, a: the bias term of fully connected layer.
K_1, K_2, K_3: weights of convolutional filter, b: bias term of the convolutional layer.

Q3.2: $L = \frac{1}{2}(y-\hat{y})^2$, $\hat{y} = W_1 V_1 + W_2 V_2 + a$

$\frac{\partial L}{\partial W_1} = (y-\hat{y}) \cdot V_1$, $\frac{\partial L}{\partial W_2} = (y-\hat{y}) \cdot V_2$, $\frac{\partial L}{\partial a} = (y-\hat{y})$

Q3.3: $\frac{\partial L}{\partial V_1} = \delta_1$ $\frac{\partial L}{\partial V_2} = \delta_2$

$V_1 = \max(Z_1, Z_2, 0)$, $V_2 = \max(Z_3, Z_3, 0)$

For V_1:

$$\frac{\partial V_1}{\partial Z_1} = \begin{cases} 1, & \text{if } Z_1 = \max(Z_1, Z_2, 0) \\ 0, & \text{o.w.} \end{cases} \qquad \text{for } V_2: \quad \frac{\partial V_2}{\partial Z_2} = \begin{cases} 1, & \text{if } Z_2 = \max(Z_2, Z_3, 0) \\ 0, & \text{o.w.} \end{cases}$$

$$\frac{\partial V_1}{\partial Z_2} = \begin{cases} 1 & \text{if } Z_2 = \max(Z_1, Z_2, 0) \\ 0 & \text{o.w.} \end{cases} \qquad \frac{\partial V_2}{\partial Z_3} = \begin{cases} 1 & \text{if } Z_2 = \max(Z_2, Z_3, 0) \\ 0 & \text{o.w.} \end{cases}$$

$$\frac{\partial V_1}{\partial 0} = \begin{cases} 1 & \text{if } 0 = \max(Z_1, Z_2, 0) \\ 0 & \text{o.w.} \end{cases} \qquad \frac{\partial V_2}{\partial 0} = \begin{cases} 1 & \text{if } 0 = \max(Z_1, Z_2, 0) \\ 0 & \text{o.w.} \end{cases}$$

$\therefore \frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial V_1} \cdot \frac{\partial V_1}{\partial Z_1} = \delta_1 \frac{\partial V_1}{\partial Z_1}$; $\frac{\partial L}{\partial Z_2} = \frac{\partial L}{\partial V_1} \cdot \frac{\partial V_1}{\partial Z_2} + \frac{\partial L}{\partial V_2} \cdot \frac{\partial V_2}{\partial Z_2} = \delta_1 \cdot \frac{\partial V_1}{\partial Z_2} + \delta_2 \cdot \frac{\partial V_2}{\partial Z_1}$; $\frac{\partial L}{\partial Z_3} = \frac{\partial L}{\partial V_2} \cdot \frac{\partial V_2}{\partial Z_3} = \delta_2 \cdot \frac{\partial V_2}{\partial Z_3}$

Q3.4: $\frac{\partial L}{\partial Z_1} = \delta_1$, $\frac{\partial L}{\partial Z_2} = \delta_2$, $\frac{\partial L}{\partial Z_3} = \delta_3$.

$$\begin{cases} Z_1 = K_1 X_1 + K_2 X_2 + K_3 X_3 + b \\ Z_2 = K_1 X_2 + K_2 X_3 + K_3 X_4 + b \\ Z_3 = K_1 X_3 + K_2 X_4 + K_3 X_5 + b \end{cases}$$

$$\Rightarrow \begin{cases} \frac{\partial L}{\partial K_1} = \frac{\partial L}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial K_1} + \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial K_1} + \frac{\partial L}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial K_1} = \delta_1 \cdot X_1 + \delta_2 \cdot X_2 + \delta_3 \cdot X_3 \\ \frac{\partial L}{\partial K_2} = \frac{\partial L}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial K_2} + \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial K_2} + \frac{\partial L}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial K_2} = \delta_1 \cdot X_2 + \delta_2 \cdot X_3 + \delta_3 \cdot X_4 \\ \frac{\partial L}{\partial K_3} = \frac{\partial L}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial K_3} + \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial K_3} + \frac{\partial L}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial K_3} = \delta_1 \cdot X_3 + \delta_2 \cdot X_4 + \delta_3 \cdot X_5 \\ \frac{\partial L}{\partial b} = \frac{\partial L}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial b} + \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial b} + \frac{\partial L}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial b} = \delta_1 + \delta_2 + \delta_3 \end{cases}$$

[20 marks] Question 4: Convolutional Neural Networks

The dataset can be found here. Convolutional Neural Networks (CNNs) are a class of deep neural networks highly effective for analyzing visual imagery. In this exercise, you will work with an image dataset divided into two main folders: 'train' and 'test'. Within the 'train' dataset, there are two subfolders named 'polar' and 'not_polar', containing images of polar bears and images without polar bears, respectively. Your task will involve training a CNN to distinguish between these two categories. The skeleton code can be found here. Insert code wherever you see a 'TODO' comment in the section.

[4 marks] Q4.1: The ImageDataGenerator class in Keras is a tool for real-time data augmentation, which can enhance the number and diversity of your training images available for training deep learning models. Data augmentation automatically introduces random transformations to the training data (such as rotation, scaling, and horizontal flipping), creating variations of the images. This helps prevent the model from overfitting and generalizes better, thus improving its performance on new, unseen data. Implement the `create_image_generators` function to create image data generators for training and validation:

- The function should take parameters: `train_dir` (e.g., "/path/to/data"), `target_size` ((150, 150)), `batch_size` (e.g., 32), and `val_split` (0.2).
- Initialize an `ImageDataGenerator` object with parameters for rescaling (1./255), rotation (40 degrees), width shift (0.2), height shift (0.2), shear (0.2), zoom (0.2), flipping (horizontal), fill mode ('nearest'), and validation split (0.2).
- Use `flow_from_directory` with `subset` set to 'training', `class_mode` set to 'binary', and `seed` for reproducibility (25).
- Similarly, create a validation generator with `subset` set to 'validation'. Use the same settings for `class_mode` and `seed`.
- The function should return both the training and validation generators.

Report the 'Training Generator Info' and 'Validation Generator Info' that you got when you executed `print_generator_info` function.

[4 marks] Q4.2: Build the following convolutional neural network architecture using the sequential model approach:

- Start with a **Conv2D layer** with 32 filters, a 3x3 kernel size, and 'relu' activation. Specify the input shape as (150, 150, 3) for images that are 150x150 pixels with 3 color channels.
- Add a **MaxPooling2D layer** with a pool size of 2x2 to reduce the spatial dimensions of the output from the previous layer.
- Include another **Conv2D layer** with 64 filters and a 3x3 kernel size, using 'relu' activation to introduce non-linearity.
- Follow this with another **MaxPooling2D layer** with a pool size of 2x2.
- Flatten the output to a single vector with a **Flatten layer**, preparing it for the dense layers that follow.
- Introduce a **Dropout layer** with a rate of 0.5 to reduce overfitting by randomly setting input units to 0 during training.
- Add a **Dense layer** with 512 units and 'relu' activation to process the vector of features coming from the flattened output.
- Conclude with a **Dense layer** with 1 unit and 'sigmoid' activation for binary classification, which will output probabilities indicating the presence or absence of a polar bear.

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

[4 marks] Q4.3: To compile and train a Keras model, first, compile the model using the `compile` method, specifying the loss function as `binary_crossentropy`, the optimizer as `adam`, and tracking the `accuracy` metric. Then, train the model using the `fit` method, passing the `train_generator` for training data, setting the number of epochs to 20, and including the `validation_generator` for validation data. Plot the training and validation accuracies and losses. Does this model appear to be efficient at this stage?

[4 marks] Q4.4: There are just six images in the test folder. Load them one by one and infer by seeing what the model classifies. 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?

[4 marks] Q4.5: One of the intriguing aspects of deep learning is that the models we train are far more than just enigmatic boxes outputting predictions or classification results; instead, your network operates as an intricate, multi-layered system that transforms input data into a series of sophisticated encodings. Define the `get_layer_outputs` function, begin by specifying its signature, which includes two parameters: 'model', representing the Keras model, and 'img_tensor', the input tensor for the model. Within the function, extract the output tensors for all layers up to the last MaxPooling layer (index 3) in the model. Next, create a new model named 'activation_model' using the 'Model' class from Keras, which takes the original model's input and the tensors extracted in the previous step as output. Finally, utilize the 'predict' method of the 'activation_model' to predict the activations for the input 'img_tensor', and return these activations as a list of numpy arrays, where each array corresponds to the activations of one layer. These activations are then used by `display_layer_activations` function. Upload the visualizations of the activations for test_4.png and comment on increasing abstractness as we go deeper into the model.

Q4.1:

Load the dataset

```python
# TODO: Populate the create_image_generators() function
def create_image_generators(train_dir, target_size=(150, 150), batch_size=20, val_split=0.2):
    """
    Create training and validation generators for image data.

    Parameters:
    - base_dir: Path to the base directory where the 'train' folder is located.
    - target_size: Tuple of integers, the dimensions to which all images found will be resize
    - batch_size: Integer, size of the batches of data.
    - val_split: Float, the fraction of images reserved for validation.

    Returns:
    - train_generator: Training data generator.
    - validation_generator: Validation data generator.
    """

    # TODO: Creating Image Data Generator for both training and validation
    datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest',
        validation_split=val_split
    )

    # TODO: Create a training data generator
    train_generator = datagen.flow_from_directory(
        train_dir,
        target_size=target_size,
        batch_size=batch_size,
        class_mode='binary',
        subset='training',
        seed=seed_value
    )
    # TODO: Create a validation data generator
    validation_generator = datagen.flow_from_directory(
        train_dir,
        target_size=target_size,
        batch_size=batch_size,
        class_mode='binary',
        subset='validation',
        seed=seed_value
    )
    return train_generator, validation_generator

train_dir = './train'
train_generator, validation_generator = create_image_generators(train_dir)
```

```
Found 148 images belonging to 2 classes.
Found 36 images belonging to 2 classes.
```

Q4.2:

```python
def print_generator_info(generator):
    """
    Print information about a data generator.

    Parameters:
    - generator: The data generator (train or validation).
    """
    # Number of images
    num_images = generator.samples
    # Batch size
    batch_size = generator.batch_size
    # Class indices
    class_indices = generator.class_indices
    # Number of classes
    num_classes = generator.num_classes
    # Filenames
    filenames = generator.filenames

    print(f"Number of images: {num_images}")
    print(f"Batch size: {batch_size}")
    print(f"Class indices: {class_indices}")
    print(f"Number of classes: {num_classes}")
    print(f"Number of filenames loaded: {len(filenames)}")

# Example usage:
print("Training Generator Info:")
print_generator_info(train_generator)
print("\nValidation Generator Info:")
print_generator_info(validation_generator)
```

✓ 0.0s

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

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

Define the CNN model

```python
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])# TODO: Define the CNN architecture

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

[34]

```
/opt/homebrew/lib/python3.10/site-packages/keras/src/layers/convolutional/base_conv.py:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
conv2d_10 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_10 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_11 (Conv2D)	(None, 72, 72, 64)	18,496
max_pooling2d_11 (MaxPooling2D)	(None, 36, 36, 64)	0
flatten_5 (Flatten)	(None, 82944)	0
dropout_5 (Dropout)	(None, 82944)	0
dense_10 (Dense)	(None, 512)	42,467,840
dense_11 (Dense)	(None, 1)	513

```
Total params: 42,487,745 (162.08 MB)

Trainable params: 42,487,745 (162.08 MB)

Non-trainable params: 0 (0.00 B)

None
```

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

[6]

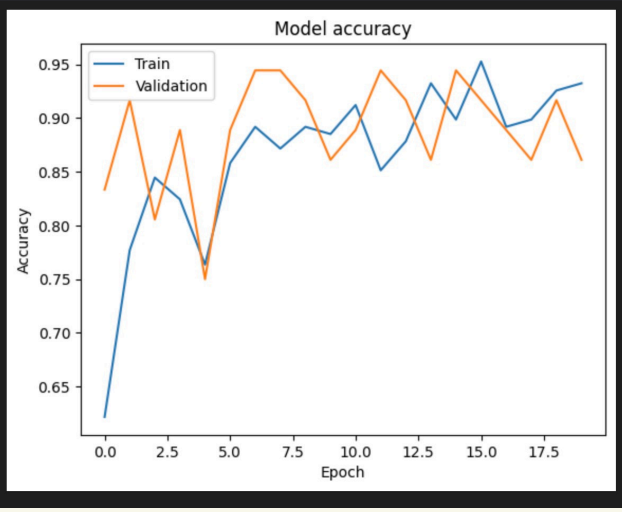
```
Total number of parameters: 42487745
```

Q4.3:

Plot Training and Validation accuracies

```python
# TODO: Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

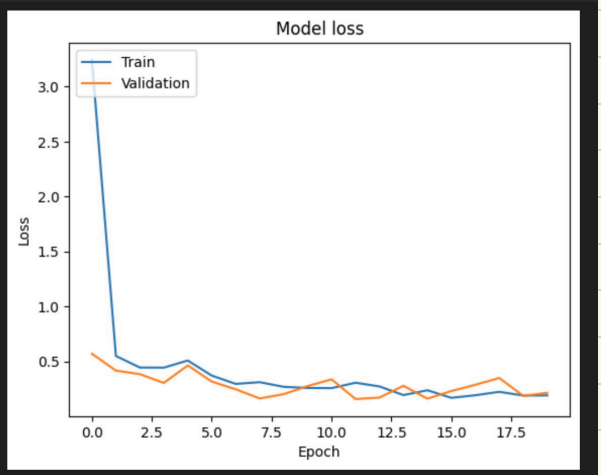
✓ 0.1s



Plot Training and Validation losses

```python
# TODO: Plot training & validation accuracy losses
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

✓ 0.0s

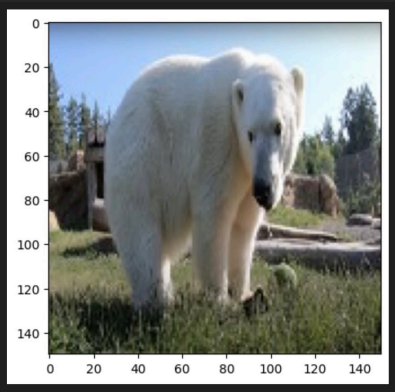


Accuracy: Both training and validation accuracy show a general increasing over each epoch. The accuracy fluctuates but remain high, indicating that the model is not stable but performing reasonable well.

Loss: Both training and validation loss decrease over each epoch. It indicates that the model is learning efficiently. There are some fluctuations for validation loss while training loss is stable, indicating that it may have some overfitting.

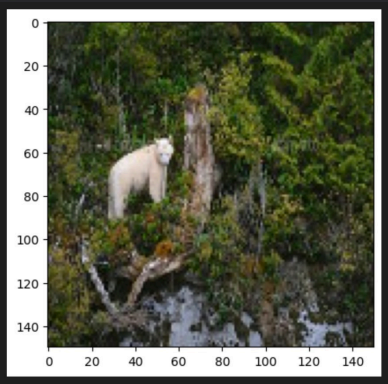
Overall, in this stage, the model appears to be fairly efficient.

```
img_path = 'test/test_1.jpg'# TODO: set path to test_1.jpg
load_display_predict_image(img_path, model)
✓ 0.1s
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
[[0.583148]]
Polar Bear
```

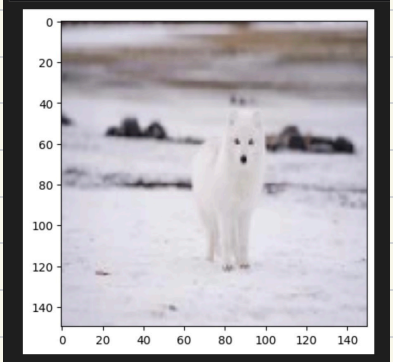
```
img_path = 'test/test_2.jpg'# TODO: set path to test_2.jpg
load_display_predict_image(img_path, model)
✓ 0.1s
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
[[2.1169275e-05]]
Not Polar Bear
```

Inference for test_3.jpg

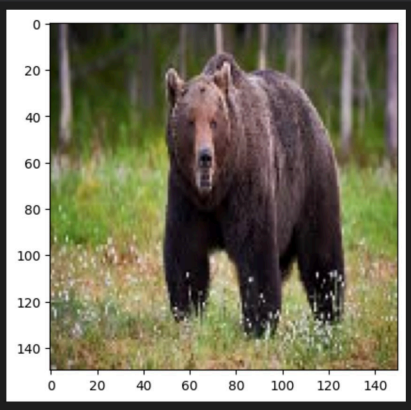
```
img_path = 'test/test_3.jpg'# TODO: set path to test_3.jpg
load_display_predict_image(img_path, model)
✓ 0.1s
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
[[0.9877873]]
Polar Bear
```

Inference for test_4.jpg

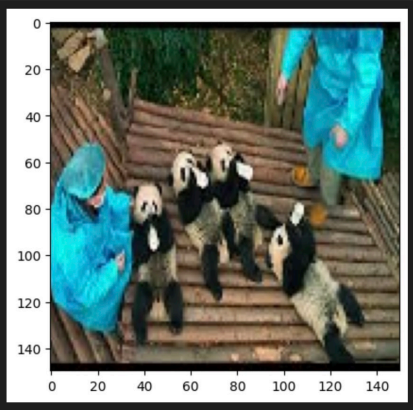
```
img_path = 'test/test_4.jpg'# TODO: set path to test_4.jpg
load_display_predict_image(img_path, model)
✓ 0.1s
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
[[3.038277e-06]]
Not Polar Bear
```

Inference for test_5.jpg

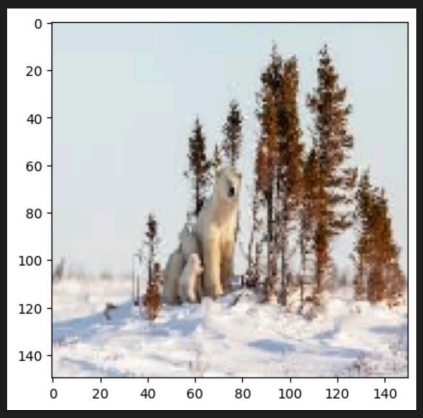
```
img_path = 'test/test_5.jpg'# TODO: set path to test_5.jpg
load_display_predict_image(img_path, model)
✓ 0.1s
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
[[0.0035461]]
Not Polar Bear
```

Inference for test_6.jpg

```
img_path = 'test/test_6.jpg'# TODO: set path to test_6.jpg
load_display_predict_image(img_path, model)
✓ 0.1s
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
[[0.8971442]]
Polar Bear
```

Q4.4: Test image 2 and 3 are misclassified.

Potential Reasons: 1. Image quality and variability. The training images may have different quality like the size of the polar bear, background may be varied. For example, for test image 2, the polar bear is to small in the frame compared to the others.

2. Model: the model is not complex enough to distinguish the polar bear. For example, test image 3 is a Arctic fox, which is already hard for human eye to distinguish when the image in a low resolution.

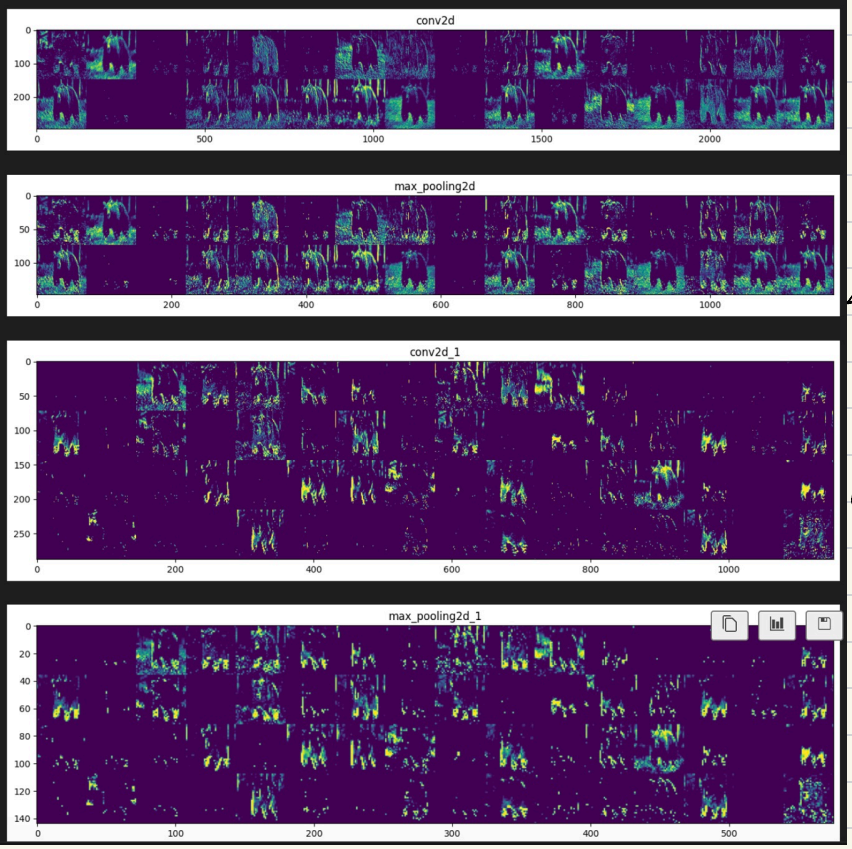
Possible solutions: 1: Preprocess the data beforehand: Eg. Resize the animal to a reasonable size and refine the image quality before processing.
2 Increase data agumments and layers the cnn model: increase the complexity of the model for a better performance.

Q4.5

```python
def get_layer_outputs(model, img_tensor):
    # Extracts outputs for all layers up to the last MaxPooling layer (
    layer_outputs = [layer.output for layer in model.layers[:4]]
    # Creates a new model that will return these outputs, given the model input
    activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
    # Returns a list of numpy arrays, one array per layer activation
    activations = activation_model.predict(img_tensor)
    return activations


# Load and preprocess an image
img_path = 'test/test_4.jpg'
img_tensor = load_image(img_path, size=(150, 150))
# Get activations and display them
activations = get_layer_outputs(model, img_tensor)
display_layer_activations(activations, model)
```

✓ 0.5s

```
<keras.src.engine.sequential.Sequential object at 0x173fd4cd0>
1/1 [==============================] - 0s 26ms/step
```



1. First convolutional layer: this layer captures basic features of the image such as the contects and the edges. And it highlight specific features.

2. First Max Poding: this layer is increasing the features' highlight from the first layer and reducing the other contexd's sensitivity.

3. Second Convolutional layer: this layer capture the features from the previous layer and transform them into a more complex pattern. This layer increase the abstraction a lot and it is hard to explain the patterns visually.

4. Second max pooling: this layer increases the level of abstraction by highlighting important features and contexts To let the CNN model focus on the specific details.

Overall as the data go through those layers, abstraction level increases. It enables the CNN to focus on what is essential for making accurate classification, ignoring irrelevant data in the input data.

[25 marks] Question 5: word2vec

Create Python script for creating Word2vec from scratch by training a Continuous Bag of Words (CBOW) model using TensorFlow and Keras. Do not use the Gensim library. The dataset can be found here, and the skeleton code can be found here. You are required to complete the **TODO** sections in the skeleton code. The invocation of functions which are to completed will be made later in the code; do not change the function call and parameter values like window size and embedding size. Set seeds to ensure reproducibility in your code. Use the **same seed value (25)** for all random number generators in your environment, including libraries like NumPy, TensorFlow, and Python's built-in random module.

[2 marks] Q5.1: Complete the `preprocess()` function by converting the text in lowercase and splitting it into words. What is the total number of words?

[5 marks] Q5.2: Complete the `build_and_prepare_data()` function. This function preprocesses a list of words to create a vocabulary and generate training data. It builds a vocabulary by assigning a unique index to each word and generates context-target pairs by considering a specified window of words around each target word. The contexts and targets are then extracted and prepared for training by ensuring uniform length for contexts and encoding the targets in a suitable format. The processed vocabulary, context sequences, and target data are then returned, ready to be used in training tasks like word embeddings or other natural language processing models. The **window size should be 2**. What is the vocab size, number of contexts and number of targets?

[4 marks] Q5.3: Complete the `build_cbow_model` function that constructs a Continuous Bag of Words (CBOW) model using TensorFlow and Keras libraries. The CBOW model architecture includes an input layer for context words, an embedding layer to convert these words into dense vectors, an averaging layer to combine these vectors, and an output layer with a softmax function to predict the target word. The **embedding size should be 2**. Your function should return the constructed model. Print the model summary.

[4 marks] Q5.4: Extract the embeddings from the embedding layer. Since the size the embedding size is 2, you should plot the embeddings and visualize them. Provide the plot in your submission.

[6 marks] Q5.5: Complete the two functions, `cosine_similarity()` and `find_similar_words()`, to analyze word similarities using vector embeddings. The `cosine_similarity` function should measure how similar two vectors are based on their orientation. The `find_similar_words` function should identify and return the most similar words to a given `query_word` from a set of word embeddings, ranking them by their similarity. Use these functions to find and return the top 3 similar countries for each of the following: `Poland`, `Thailand`, and `Morocco`.

[2 marks] Q5.6: 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.

Q5.1 & 5.2:

TODO: Preprocess

```
# Preprocess the text
def preprocess(text):
    # TODO
    words = text.lower().split()
    return words
```

0.0s

TODO: Build Vocabulary and training data

```
# TODO: Build vocabulary and generate training data
def build_and_prepare_data(words, window_size=2):
    # Build vocabulary
    word_counts = defaultdict(int)
    for word in words:
        word_counts[word] += 1
    vocab = {word: idx for idx, (word, _) in enumerate(word_counts.items())}

    # Generate context-target pairs
    contexts = []
    targets = []

    # Extract contexts and targets from data
    for idx, word in enumerate(words):
        context = []
        for i in range(-window_size, window_size + 1):
            if i != 0 and 0 <= idx + i < len(words):
                context.append(words[idx + i])
        if len(context) == 2 * window_size:
            contexts.append(context)
            targets.append(word)
    contexts = [[vocab[word] for word in context] for context in contexts]
    targets = [vocab[word] for word in targets]
    # Prepare contexts and targets for training by padding and one-hot encoding
    contexts = pad_sequences(contexts, maxlen=2 * window_size, padding='post')
    targets = to_categorical(targets, num_classes=len(vocab))
    return vocab, contexts, targets
```

0.0s

```
# Read the file
with open(file_path, 'r', encoding='utf-8') as file:
    text = file.read()

words = preprocess(text)

# Print vocabulary size
print(f"Number of words: {len(words)}")

# Model parameters
window_size = 2

# Prepare dataset
vocab, contexts, targets = build_and_prepare_data(words, window_size)

vocab_size = len(vocab)
# Print vocabulary size
print(f"Vocabulary size: {vocab_size}")

# Print lengths of contexts and targets
print(f"Length of contexts array: {len(contexts)}")
print(f"Length of targets array: {len(targets)}")
```

0.5s

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

Q5.3:

TODO: Build CBOW model

```
# Define CBOW model function
def build_cbow_model(vocab_size, embed_size, window_size):
    # TODO
    input_layer = Input(shape=(window_size * 2,))
    embedding = Embedding(vocab_size, embed_size, input_length=window_size*2)(input_layer)
    hidden = Lambda(lambda x: K.mean(x, axis=1))(embedding)
    output = Dense(vocab_size, activation='softmax')(hidden)
    model = Model(inputs=input_layer, outputs=output)
    model.compile(optimizer='adam', loss='categorical_crossentropy')
    return model
```

0.0s

Split the data inton training and validatio

```
# Splitting the data
contexts_train, contexts_val, targets_train, targets_val = train_test_split(contexts, targets, te

embed_size = 2
```

0.0s Python

Train the model

```
# Create and train the model
model = build_cbow_model(vocab_size, embed_size, window_size)
print(model.summary())
history = model.fit(contexts_train, targets_train, validation_data=(contexts_val, targets_val), e
```

15.6s Python

```
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 4)	0
embedding_1 (Embedding)	(None, 4, 2)	56
lambda_1 (Lambda)	(None, 2)	0
dense_1 (Dense)	(None, 28)	84

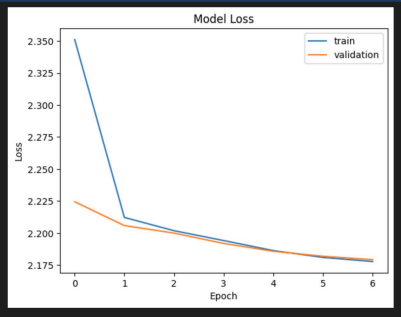
```
Total params: 140 (560.00 B)

Trainable params: 140 (560.00 B)

Non-trainable params: 0 (0.00 B)

None
```

Q5.4:

TODO: Visualise the Training and Validation loss

```
# Plotting the training and validation loss
#TODO
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='validation')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

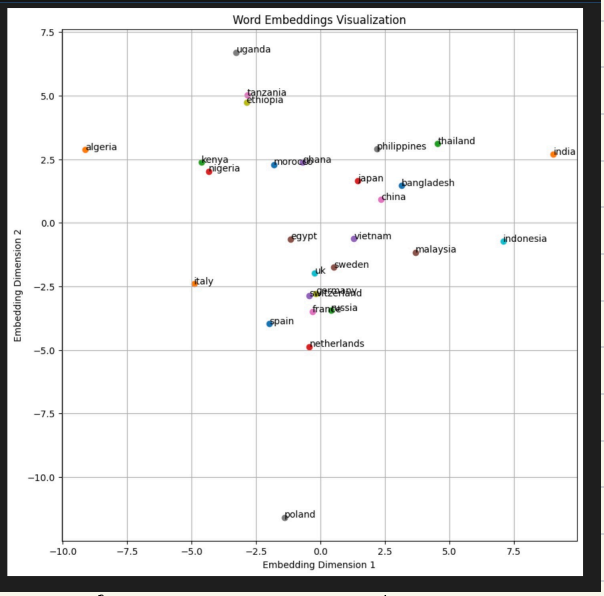
0.0s



TODO: Visualise the embeddings

```
# Create a scatter plot of the embeddings
# TODO
plt.figure(figsize=(10, 10))
for word, idx in vocab.items():
    plt.scatter(embeddings[idx, 0], embeddings[idx, 1], label=word)
    plt.annotate(word, (embeddings[idx, 0], embeddings[idx, 1]))

plt.xlabel('Embedding Dimension 1')
plt.ylabel('Embedding Dimension 2')
plt.title('Word Embeddings Visualization')
plt.grid(True)
plt.show()
```

0.2s



Q5.5:

TODO: Find similar words

```
def cosine_similarity(vec_a, vec_b):
    """Calculate the cosine similarity between two vectors."""
    #TODO
    dot_product = np.dot(vec_a, vec_b)
    norm_a = np.linalg.norm(vec_a)
    norm_b = np.linalg.norm(vec_b)
    similarity = dot_product / (norm_a * norm_b)
    return similarity

def find_similar_words(query_word, vocab, embeddings, top_n=3):
    """Find the top_n words most similar to the query_word based on the embeddings."""
    similarities = []

    #TODO populate the similarities list
    query_idx = vocab[query_word]
    query_vec = embeddings[query_idx]
    similarities = []
    for word, idx in vocab.items():
        if word != query_word:
            word_vec = embeddings[idx]
            sim = cosine_similarity(query_vec, word_vec)
            similarities.append((word, sim))

    # Sort based on similarity scores
    similarities.sort(key=lambda x: x[1], reverse=True)

    # Print top similar words
    print(f"Words most similar to '{query_word}':")
    for word, similarity in similarities[:top_n]:
        print(f"{word}: {similarity:.4f}")
```

```
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':
uk: 1.0000
france: 0.9996
switzerland: 0.9996


Words most similar to 'thailand':
bangladesh: 0.9864
china: 0.9735
japan: 0.9691


Words most similar to 'morocco':
ethiopia: 0.9924
tanzania: 0.9882
uganda: 0.9766
```

Q5.6: For a small window size, it is possible that antonyms end up with similar embedding. Specially, when the antonyms happens in a position that very close to each other, like "true or false". And the model cannot understand the each word, but just capture the co-occurances. If window size is small and antonyms are closed to each other, then the model may think they are highly relevant and they will end up with similar embedding.

[30 marks] Question 6: Next Word Prediction

"Alice in Wonderland" is a whimsical tale by Lewis Carroll about a young girl named Alice who falls through a rabbit hole into a fantastical world full of peculiar creatures and surreal adventures. You can find the entire text here. Your task is to build a next-word prediction model, trained only on the Alice in Wonderland textual data. **Note:** We do not aim to achieve high accuracy in this exercise, the goal is to observe whether the model is learning or not and how to address overfitting. Skeleton code may be found here. Use the same seed value (25) for all random number generators in your environment, including libraries like NumPy, TensorFlow, and Python's built-in random module.

[4 marks] Q6.1: For this task, you'll be working with a text file containing data that needs to be preprocessed for further analysis. Start by accessing the file from its location on your drive. Once opened, read its contents into a string, ensuring that the text is handled in a case-*insensitive* manner by converting it to lowercase. To remove punctuations in the text, apply a regular expression that filters out all characters that are not letters, digits, underscores, or whitespace (not $\backslash w$ and $\backslash s$ in regex). For example:

Sample text: "Hello, World! Welcome to 2024. Let's preprocess this text: #ECE-657 @UWaterloo"
Preprocessed sample text: "hello world welcome to 2024 lets preprocess this text ece657 uwaterloo"

This preprocessing step simplifies the text, making it uniform and easier to analyze in subsequent tasks. Print the length of the final processed text obtained.

[2 marks] Q6.2: Initiate the process of text tokenization which is vital for preparing data for natural language processing models. Utilize the Tokenizer from the TensorFlow Keras library used for preparing text data for deep learning models to analyze the text and identify unique words. By fitting the tokenizer to the text, it constructs a comprehensive dictionary of these unique words. Subsequently, calculate the total number of unique words, which is essential for configuring various model parameters, such as input dimensions in neural networks. This total also includes an additional count to accommodate the tokenizer's indexing method. Print the total number of words.

[4 marks] Q6.3: In this task, you'll prepare input sequences for training by first splitting the preprocessed text on newline character and converting each line into a list of tokenized words. For each line, generate n-gram sequences of increasing length to create a comprehensive set of training samples. These n-grams, which consist of consecutive tokens, help the model learn contextual relationships within the data. After constructing these sequences, identify the maximum sequence length and standardize all sequences to this length using padding. This padding, typically added to the beginning of sequences, ensures that all input data fed into the model maintains a consistent format, crucial for effective training of sequence-based neural networks like LSTMs or RNNs. For example for the given text,

```
"Hello world
How are you"
```

Following steps will occur:
Tokenizer mapping: {`"hello"`: 1, `"world"`: 2, `"how"`: 3, `"are"`: 4, `"you"`: 5}
After tokenization and creating n-gram sequences:
For `"Hello world"`: `[1, 2]`
For `"How are you"`: `[3, 4], [3, 4, 5]`
Combining all n-gram sequences:[[1, 2], [3, 4], [3, 4, 5]]
Maximum sequence length: `3`
After padding:[[0, 1, 2], [0, 3, 4], [3, 4, 5]]

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

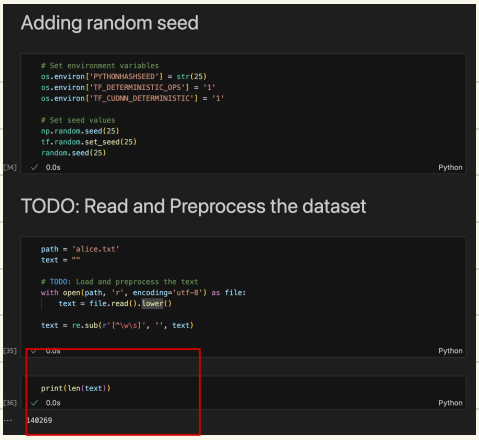
[3 marks] Q6.4: In this phase of preparing your data for machine learning models, you'll separate the previously formatted input sequences into predictors (features) and labels (targets). By slicing the sequences, the last token of each sequence becomes the label, while the preceding tokens form the predictors. Convert the label tokens into one-hot encoded vectors using TensorFlow's utility function, facilitating effective categorical output prediction. Subsequently, divide your dataset into training and validation subsets using a 20% split for validation. Print the size of the train and validation subsets for the features and targets.

[8 marks] Q6.5: Create a simple LSTM-based model by defining a sequential architecture. Begin with an embedding layer that uses the total number of words as the input dimension, and an output dimension of 100. Follow this with an LSTM layer containing 150 units. Then, add a dense layer with the total number of words as the output dimension, using the softmax activation function. Compile the model with categorical cross-entropy as the loss function, the Adam optimizer, and accuracy as the metric. 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. Is the model overfitting? Explain your observation. 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.

[6 marks] Q6.6: Define the `generate_text()` function that takes a starting text, the desired number of additional words, a predictive model, the maximum sequence length, and a temperature parameter as inputs. Within the function, predict the subsequent word iteratively based on the evolving text. In each iteration, convert the current text into tokens, pad these tokens to the required sequence length, and use the model to predict the logits (unnormalized predictions generated by the last layer of a neural network before applying an activation function) for the next word. Adjust the logits by the temperature parameter, apply the softmax function to get probabilities, and sample the next word's index based on these probabilities. Map this index back to the corresponding word using the tokenizer, and append the word to the current text. Ultimately, return the expanded text that now includes the newly generated words, effectively extending the original text input. The temperature parameter in NLP controls the randomness of the predictions by adjusting the probability distribution of the next word. Lower temperatures make the model more confident and deterministic, often resulting in more repetitive and conservative text, while higher temperatures increase randomness, producing more diverse and creative outputs but also raising the risk of generating incoherent text. Demonstrate the function by generating text with temperature values 0.05 and 1.5 using the previously created model with less overfitting.

[3 marks] Q6.7: 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.

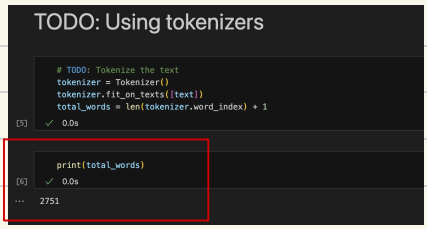
Q6.1:



length of the text: 140269

Q6.2:

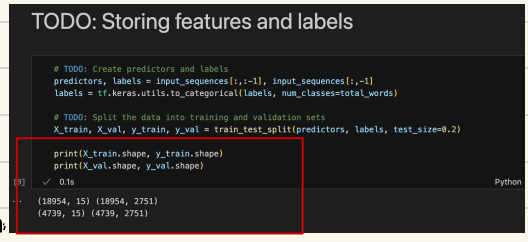
Total number of word: 2751.



Q6.3:

Total number of sequences: 23,693

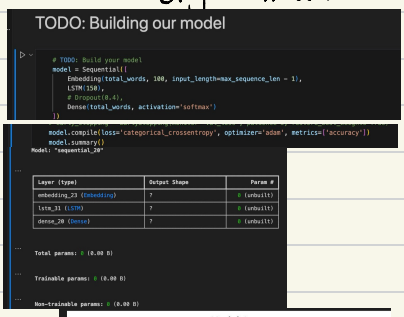
Q6.4:

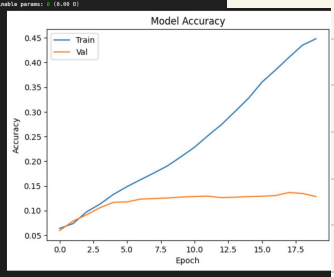


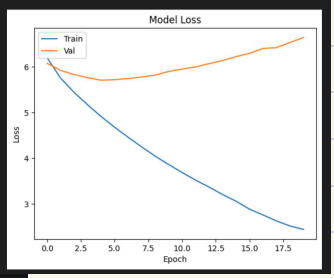
Train:

Validation:

Q6.5: Original model:





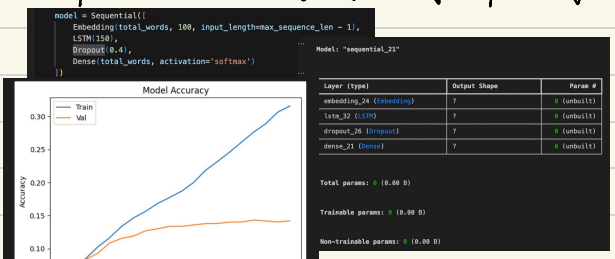


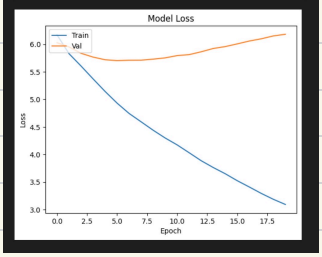
According to the graphs, the training accuracy is steadily increasing, means it is getting better after each epoch. But the validation accuracy remains relatively low.

And when the training loss is steadily decreasing after each epoch, but the validation loss even increases.

Overall, there is an overfitting.

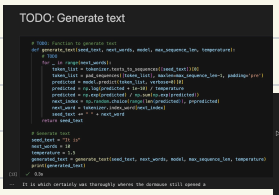
Improve the overfitting by adding dropout layer:





According to the model accuracy and model loss graphs, the absolute differences between training and validation have reduced.

Q6.6: Temperature 1.5:



Temperature 0.05:

According to the generated outputs, with a low temperature the model will be more deterministic, it will only generate the text with very high probabilities. It will resolve in keep generate the same result of the same seed_text.

With a high temperature, the texts get generated are more random, it will see less repeatedly word. You can see the result is more creative and it won't generate the same result for the same seed_text.

And something it doesn't make sense.

Q6.7: Stop word contains: "a", "I", "of", "the", etc.

In this text-generation, We shouldn't remove the stop word because:

1. These stop words are crucial for understanding and grammatical correct. If we remove the stop words, the sentence get generated maybe disjointed.

2. These stop words can play a crucial role in the model training as well. Eg. The phrase "fall in" with the stop word "in", the model will know it is likely followed by "love" instead of "apple".

3. Because "Alice Wonderland" is a novel, the stop words maybe important for the readers to understand the scenes or the contents that the writer wants to deliver.