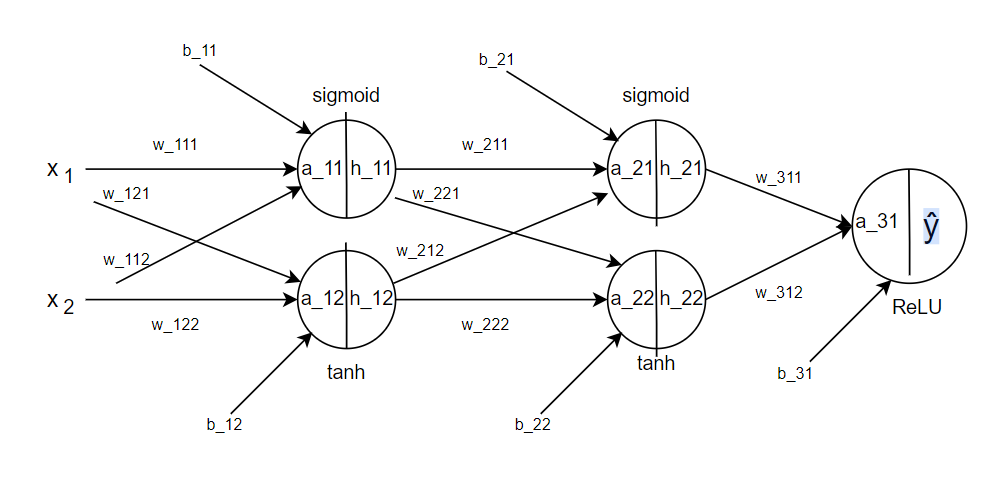
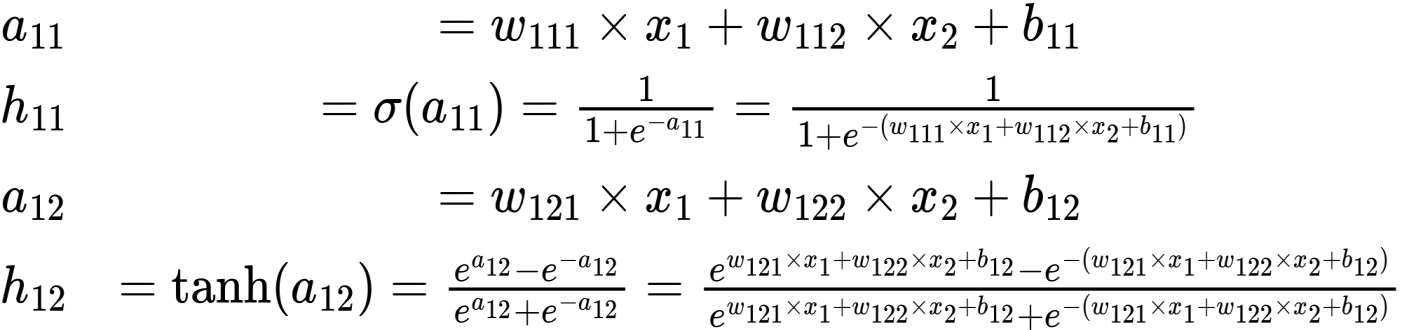
1. Consider a network with two inputs x1 and x2. It has two hidden layers, each of which contain two units. Assume that the weights in each layer are set so that top unit in each layer applies sigmoid activation to the sum of its inputs and the bottom unit in each layer applies tanh activation to the sum of its inputs. Finally, the single output node applies ReLU activation to the sum of its two inputs. Write the output of this neural network in closed form as a function of x1 and x2. This exercise should give you an idea of the complexity of functions computed by neural networks.

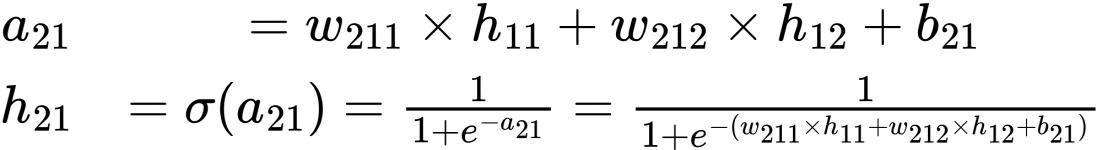
**Solution:**

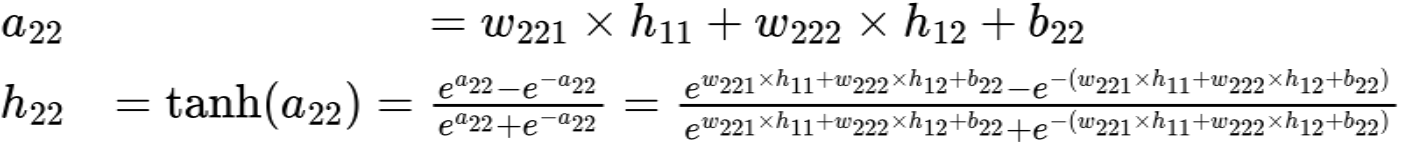
****

**Hidden Layer 1:**

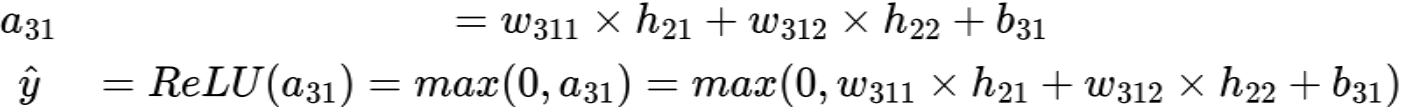


**Hidden Layer 2:**





**Output Layer:**

****

C:/Users/GOD WORLD/AppData/Local/Temp/wps.dbRUrqwps

1. Consider a 2-dimensional data set in which all points with x1 > x2 belong to the positive class, and all points with x1 < x2 belong to the negative class. Therefore, the true separator of the two classes is linear hyperplane (line) defined by x1 - x2 = 0. Now create a training data set with 20 points randomly generated inside the unit square in the positive quadrant. Label each point depending on whether or not the first coordinate x1 is greater than its second coordinate x2.

(a) Implement the perceptron algorithm without regularization, train it on the 20 points above, and test its accuracy on 1000 randomly generated points inside the unit square. Generate the test points using the same procedure as the training points.

(b) Change the perceptron criterion to hinge-loss in your implementation for training, and repeat the accuracy computation on the same test points above. Regularization is not used.

(c) In which case do you obtain better accuracy and why?

(d) In which case do you think that the classification of the same 1000 test instances will not change significantly by using a different set of 20 training points?

**Answer:**

Description: The hinge loss is a loss function used for training classifiers. It is used for the de-facto maximum-margin classification. The x-axis represents the distance from the boundary of any single instance, and the y-axis represents the loss size, or penalty, that the function will incur depending on its distance. High hinge loss indicates datapoints being on the wrong side of the boundary, and hence are misclassified; while a positive distance calls for low (or zero) hinge loss and correct classification. Regularization involves adjusting a learning algorithm to prioritize simpler prediction rules, thus preventing overfitting. Typically, this is done by modifying the loss function to penalize large weights. In short, regularization penalizes large weights to promote simpler models and prevent overfitting.

import numpy as np

class CustomPerceptron:

    def \_\_init\_\_(self, learning\_rate=0.01):

        self.learning\_rate = learning\_rate

        self.weights = None

    def fit(self, X, y):

        self.weights = np.random.rand(X.shape[1])

        converged = False

        epochs = 0

        while not converged:

            errors = 0

            for i, sample in enumerate(X):

                prediction = np.dot(self.weights, sample)

                if prediction <= 0:

                    self.weights += self.learning\_rate \* sample

                    errors += 1

            converged = errors == 0 or epochs >= 2000

            epochs += 1

    def predict(self, X):

        return np.where(np.dot(X, self.weights) > 0, 1, 0)

def generate\_dataset(num\_samples, square\_size=1.0):

    X = np.random.rand(num\_samples, 2) \* square\_size

    y = np.where(X[:, 0] > X[:, 1], 1, 0)

    return X, y

X\_train, y\_train = generate\_dataset(20)

X\_test, y\_test = generate\_dataset(1000)

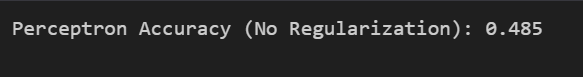
perceptron = CustomPerceptron()

perceptron.fit(X\_train, y\_train)

y\_pred = perceptron.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print("Perceptron Accuracy (No Regularization):", accuracy)



import numpy as np

class LinearPerceptron:

    def \_\_init\_\_(self, learning\_rate=0.01):

        self.learning\_rate = learning\_rate

        self.weights = None

    def train(self, X, y):

        self.weights = np.zeros(X.shape[1])

        for \_ in range(100):

            for i in range(len(X)):

                pred = np.dot(X[i], self.weights)

                self.weights += self.learning\_rate \* y[i] \* (1 - y[i] \* pred) \* X[i]

    def predict(self, X):

        return np.sign(np.dot(X, self.weights))

def generate\_data(num\_samples, scale=1.0):

    X = np.random.rand(num\_samples, 2) \* scale

    y = np.where(X[:, 0] > X[:, 1], 1, 0)

    return X, y

X\_train, y\_train = generate\_data(20)

X\_test, y\_test = generate\_data(1000)

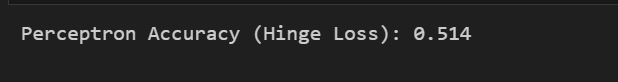
perceptron = LinearPerceptron()

perceptron.train(X\_train, y\_train)

y\_pred = perceptron.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print("Perceptron Accuracy (Hinge Loss):", accuracy)



c) We will obtain better accuracy in case of hinge loss because in case of hinge loss, the penalization for the incorrect classification or misclassification is greater forcing the model to adjust its weights more significantly to correctly classify that point. Because of larger margin, hinge loss can lead to a more robust model that generalizes better and potentially achieves higher accuracy on unseen data compared to the perceptron with no regularization.

d)Without regularization, the perceptron tends to memorize the training data, resulting in a decision boundary that closely follows the training points. This can lead to overfitting, where the model performs well on the training data but poorly on unseen data. In contrast, the hinge-loss criterion encourages the perceptron to prioritize a larger margin between classes, even with a small dataset. This wider margin promotes better generalization, as it allows for more flexibility in separating the classes. Therefore, when using hinge-loss, the classification of the same 1000 test instances may not change significantly with different sets of 20 training points, as the model focuses on maximizing the margin rather than fitting the training data precisely.

1. Implement strategies such as oversampling, undersampling, or using weighted loss functions to handle class imbalances. Implement a hyperparameter optimization pipeline for your MLP model using techniques like grid search or random search.

a. Explore the impact of varying hyperparameters (e.g., learning rate, number of hidden layers, batch size) on model performance.

b. Compare the training time and performance of your GPU-accelerated implementation with the CPU-based version.

**Answer:**

Description: To address imbalanced data, over sampling boosts the minority class, often done with SMOTE (Synthetic Minority Oversampling Technique), generating synthetic samples by combining characteristics from existing minority instances. Under sampling, on the other hand, reduces the majority class, balancing it with the minority. Hyperparameter tuning optimizes the values of hyperparameters in a neural network, which are set before training and affect the model's performance. It explores different combinations of hyperparameter values, such as epochs, batch size, and learning rate, to find the best-performing combination. Techniques like manual tuning, grid search, and random search are commonly used to achieve this optimization, ultimately improving the accuracy and efficiency of the neural network model. In deep learning, CPUs are used for data preprocessing and managing the neural network flow, while GPUs excel at training large models due to their parallel processing power. During inference and hyperparameter tuning, both CPUs and GPUs may be employed based on the specific task requirements and available hardware resources.

import numpy as np

import pandas as pd

from sklearn.datasets import load\_breast\_cancer

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import classification\_report, accuracy\_score

from sklearn.utils import resample

import time

import tensorflow as tf

data = load\_breast\_cancer()

features = data.data

labels = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, labels, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

X\_train\_resampled, y\_train\_resampled = resample(X\_train\_scaled[y\_train == 0],

                                               y\_train[y\_train == 0],

                                               n\_samples=len(y\_train[y\_train == 1]),

                                               random\_state=42)

X\_train\_balanced = np.vstack((X\_train\_scaled[y\_train == 1], X\_train\_resampled))

y\_train\_balanced = np.hstack((y\_train[y\_train == 1], y\_train\_resampled))

mlp = MLPClassifier()

param\_grid = {

    'hidden\_layer\_sizes': [(50,), (100, 50), (100, 100)],

    'alpha': [0.0001, 0.001, 0.01],

    'learning\_rate\_init': [0.001, 0.01, 0.1],

    'batch\_size': [16, 32, 64]

}

grid\_search = GridSearchCV(mlp, param\_grid, cv=5, n\_jobs=-1)

start\_time = time.time()

grid\_search.fit(X\_train\_balanced, y\_train\_balanced)

end\_time = time.time()

best\_params = grid\_search.best\_params\_

mlp\_best = MLPClassifier(\*\*best\_params, max\_iter=200)

start\_time\_mlp = time.time()

mlp\_best.fit(X\_train\_balanced, y\_train\_balanced)

end\_time\_mlp = time.time()

y\_pred\_mlp = mlp\_best.predict(X\_test\_scaled)

accuracy\_mlp = accuracy\_score(y\_test, y\_pred\_mlp)

print("MLP Performance:")

print("Accuracy:", accuracy\_mlp)

print(classification\_report(y\_test, y\_pred\_mlp))

print(f"MLP Training Time: {end\_time\_mlp - start\_time\_mlp:.2f} seconds")

if tf.test.is\_gpu\_available():

    tf.keras.backend.clear\_session()

    X\_train\_tf = tf.constant(X\_train\_balanced, dtype=tf.float32)

    y\_train\_tf = tf.constant(y\_train\_balanced, dtype=tf.float32)

    X\_test\_tf = tf.constant(X\_test\_scaled, dtype=tf.float32)

    mlp\_gpu = tf.keras.Sequential([

        tf.keras.layers.Dense(100, activation='relu', input\_shape=(X\_train\_tf.shape[1],)),

        tf.keras.layers.Dense(50, activation='relu'),

        tf.keras.layers.Dense(1, activation='sigmoid')

    ])

    mlp\_gpu.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=best\_params['learning\_rate\_init']),

                    loss='binary\_crossentropy',

                    metrics=['accuracy'])

    start\_time\_gpu = time.time()

    mlp\_gpu.fit(X\_train\_tf, y\_train\_tf, epochs=200, batch\_size=best\_params['batch\_size'], verbose=0)

    end\_time\_gpu = time.time()

    y\_pred\_gpu = mlp\_gpu.predict(X\_test\_tf)

    y\_pred\_gpu = np.round(y\_pred\_gpu).flatten().astype(int)

    accuracy\_gpu = accuracy\_score(y\_test, y\_pred\_gpu)

    print("\nGPU Performance:")

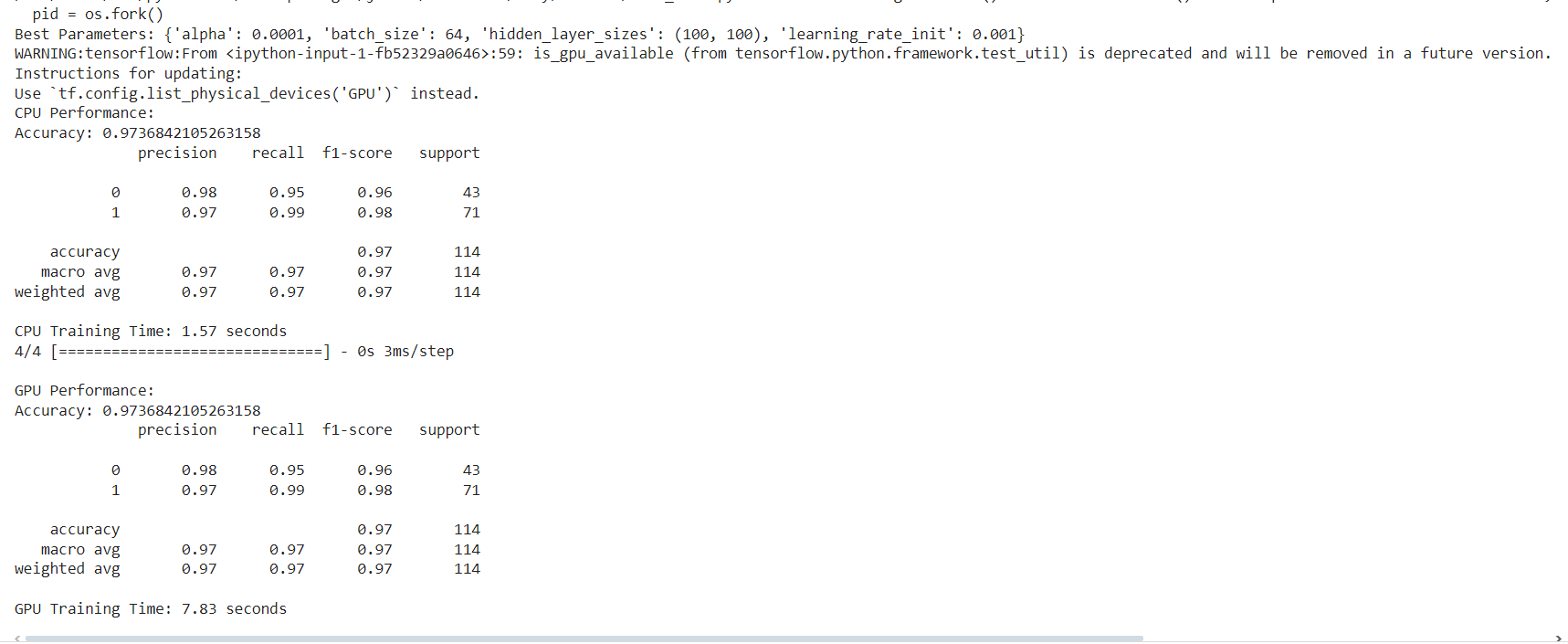
    print("Accuracy:", accuracy\_gpu)

    print(classification\_report(y\_test, y\_pred\_gpu))

    print(f"GPU Training Time: {end\_time\_gpu - start\_time\_gpu:.2f} seconds")

else:

    print("GPU is not available. Install TensorFlow with GPU support to compare GPU performance.")



4) Design a deep CNN architecture for image classification with a specific focus on handling small datasets.

a. Implement and experiment with advanced architectures such as Inception, ResNet, or DenseNet.

b. Analyze the impact of different architectures on training time, convergence, and model performance.

**Answer:**

Description: DenseNet enhances feature reuse and gradient flow by linking each layer to all subsequent layers, improving model performance. GResNet addresses the vanishing gradient issue through residual connections, enabling successful training of deep neural networks. ResNet addresses the vanishing gradient issue through residual connections, enabling successful training of deep neural networks. In terms of training time, Inception models are slower due to their complex Inception modules, while ResNets are faster thanks to efficient residual connections, and DenseNets fall in between. All benefit from residual connections for convergence, though Inception's multi-scale feature extraction may add complexity, potentially affecting convergence speed.

from tensorflow.keras.models import Sequential, Model

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, GlobalAveragePooling2D

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

import tensorflow as tf

import time

import matplotlib.pyplot as plt

import numpy as np

def create\_simple\_cnn(input\_shape, num\_classes):

    model = Sequential()

    model.add(Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=input\_shape))

    model.add(MaxPooling2D(pool\_size=(2, 2)))

    model.add(Conv2D(64, kernel\_size=(3, 3), activation='relu'))

    model.add(MaxPooling2D(pool\_size=(2, 2)))

    model.add(Conv2D(128, kernel\_size=(3, 3), activation='relu'))

    model.add(MaxPooling2D(pool\_size=(2, 2)))

    model.add(Flatten())

    model.add(Dense(64, activation='relu'))

    model.add(Dropout(0.5))

    model.add(Dense(num\_classes, activation='softmax'))

    return model

def create\_resnet50(input\_shape, num\_classes):

    base\_model = tf.keras.applications.ResNet50(weights='imagenet', include\_top=False, input\_shape=input\_shape)

    for layer in base\_model.layers:

        layer.trainable = False

    x = base\_model.output

    x = GlobalAveragePooling2D()(x)

    x = Dense(256, activation='relu')(x)

    x = Dense(num\_classes, activation='softmax')(x)

    model = Model(inputs=base\_model.input, outputs=x)

    return model

def train\_model(model, model\_name, x\_train, y\_train):

    print(f"Training {model\_name}...")

    start\_time = time.time()

    model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

    history = model.fit(x\_train, y\_train, batch\_size=64, epochs=10, validation\_split=0.2, verbose=0)

    end\_time = time.time()

    train\_time = end\_time - start\_time

    print(f"{model\_name} training time: {train\_time:.2f} seconds")

    plt.plot(history.history['loss'], label='Training Loss')

    plt.plot(history.history['val\_loss'], label='Validation Loss')

    plt.title(f'{model\_name} Loss')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.legend()

    plt.show()

    return train\_time, history

def evaluate\_model(model, model\_name, x\_test, y\_test):

    print(f"Evaluating {model\_name}...")

    loss, accuracy = model.evaluate(x\_test, y\_test, verbose=0)

    print(f"{model\_name} Test Loss: {loss:.4f}")

    print(f"{model\_name} Test Accuracy: {accuracy:.4f}")

    return loss, accuracy

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

input\_shape = (32, 32, 3)

num\_classes = 10

simple\_cnn\_model = create\_simple\_cnn(input\_shape, num\_classes)

resnet50\_model = create\_resnet50(input\_shape, num\_classes)

models = {

    "Simple CNN": simple\_cnn\_model,

    "ResNet50": resnet50\_model,

}

results = {}

for model\_name, model in models.items():

    train\_time, \_ = train\_model(model, model\_name, x\_train, y\_train)

    test\_loss, test\_accuracy = evaluate\_model(model, model\_name, x\_test, y\_test)

    results[model\_name] = {"Train Time": train\_time, "Test Loss": test\_loss, "Test Accuracy": test\_accuracy}

print("\nResults:")

for model\_name, metrics in results.items():

    print(f"{model\_name}:")

    for metric, value in metrics.items():

        print(f"{metric}: {value}")

    print()

import numpy as np

import keras

from keras.datasets import cifar10

from keras.preprocessing.image import ImageDataGenerator

from keras.applications import InceptionV3

from keras.layers import Dense, GlobalAveragePooling2D, Dropout

from keras.models import Model

(train\_images, train\_labels), (test\_images, test\_labels) = cifar10.load\_data()

percent\_data = 1

num\_samples = int(len(train\_images) \* percent\_data)

train\_images = train\_images[:num\_samples]

train\_labels = train\_labels[:num\_samples]

num\_classes = 10

train\_labels = keras.utils.to\_categorical(train\_labels, num\_classes)

test\_labels = keras.utils.to\_categorical(test\_labels, num\_classes)

from skimage.transform import resize

resized\_train\_images = np.array([resize(img, (75, 75)) for img in train\_images])

resized\_test\_images = np.array([resize(img, (75, 75)) for img in test\_images])

train\_datagen = ImageDataGenerator(

    rescale=1./255,

    shear\_range=0.2,

    zoom\_range=0.2,

    horizontal\_flip=True

)

test\_datagen = ImageDataGenerator(rescale=1./255)

val\_split = 0.2

val\_size = int(val\_split \* len(resized\_train\_images))

val\_images = resized\_train\_images[:val\_size]

val\_labels = train\_labels[:val\_size]

train\_images = resized\_train\_images[val\_size:]

train\_labels = train\_labels[val\_size:]

train\_generator = train\_datagen.flow(train\_images, train\_labels, batch\_size=32)

validation\_generator = test\_datagen.flow(val\_images, val\_labels, batch\_size=32)

def create\_inception\_model(include\_top=False):

    base\_model = InceptionV3(weights='imagenet', include\_top=include\_top, input\_shape=(75, 75, 3))

    if not include\_top:

        x = base\_model.output

        x = GlobalAveragePooling2D()(x)

        x = Dense(1024, activation='relu')(x)

        x = Dropout(0.5)(x)

        predictions = Dense(num\_classes, activation='softmax')(x)

        model = Model(inputs=base\_model.input, outputs=predictions)

    else:

        model = base\_model

    return model

# Inception Model

inception\_model = create\_inception\_model(include\_top=False)

# Freeze base model layers for transfer learning

inception\_model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

inception\_model.fit(

    train\_generator,

    epochs=10,

    validation\_data=validation\_generator

)

test\_loss, test\_acc = inception\_model.evaluate(test\_datagen.flow(resized\_test\_images, test\_labels, batch\_size=32))

print("Test Loss:", test\_loss)

print("Test Accuracy:", test\_acc)

import numpy as np

import keras

from keras.datasets import cifar10

from keras.preprocessing.image import ImageDataGenerator

from keras.applications import InceptionV3, DenseNet121

from keras.layers import Dense, GlobalAveragePooling2D, Dropout

from keras.models import Model

def create\_densenet\_model(include\_top=False):

    base\_model = DenseNet121(weights='imagenet', include\_top=include\_top, input\_shape=(75, 75, 3))

    if not include\_top:

        x = base\_model.output

        x = GlobalAveragePooling2D()(x)

        # Add custom layers for classification (adjust units as needed)

        x = Dense(1024, activation='relu')(x)

        x = Dropout(0.5)(x)

        predictions = Dense(num\_classes, activation='softmax')(x)

        model = Model(inputs=base\_model.input, outputs=predictions)

    else:

        model = base\_model

    return model

densenet\_model = create\_densenet\_model(include\_top=False)

densenet\_model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

densenet\_model.fit(

    train\_generator,

    epochs=10,

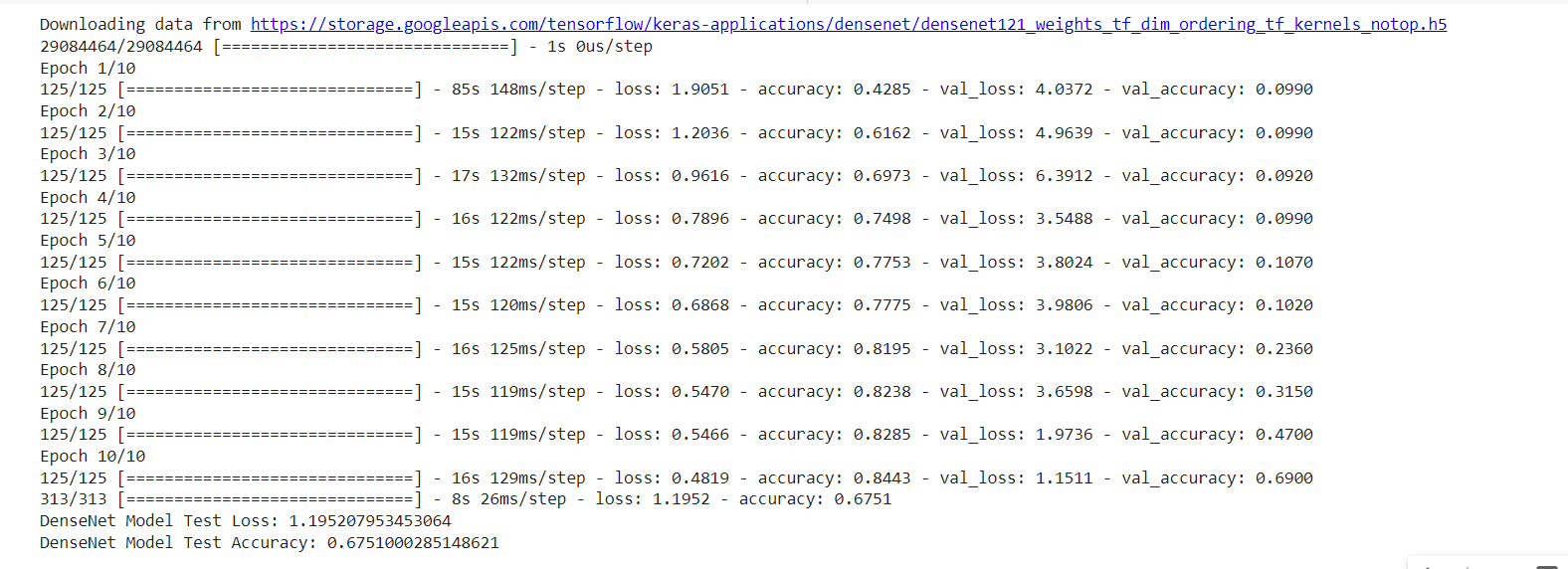
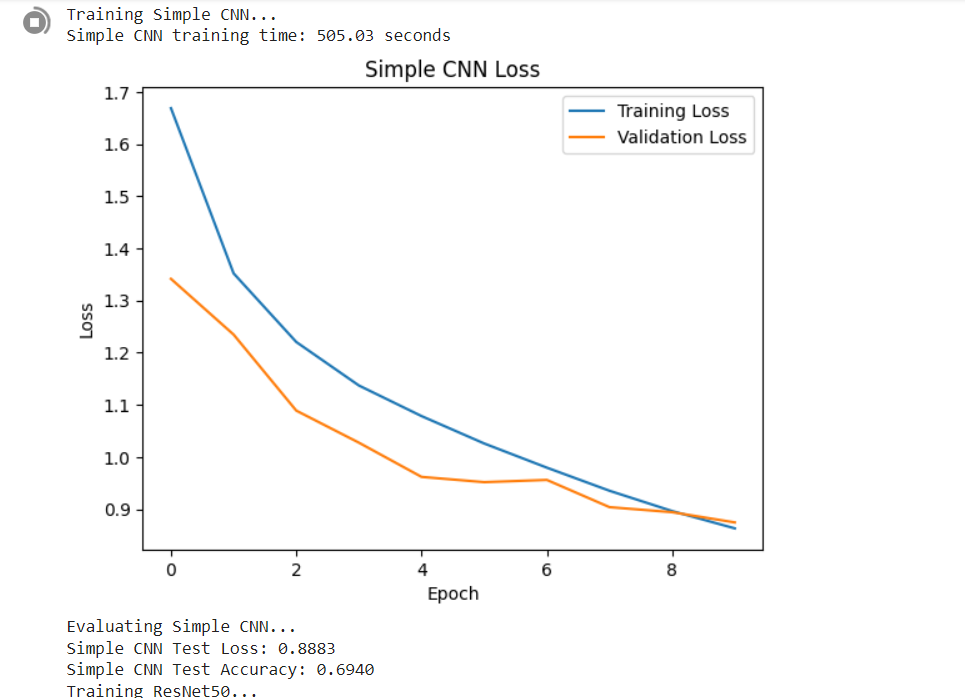
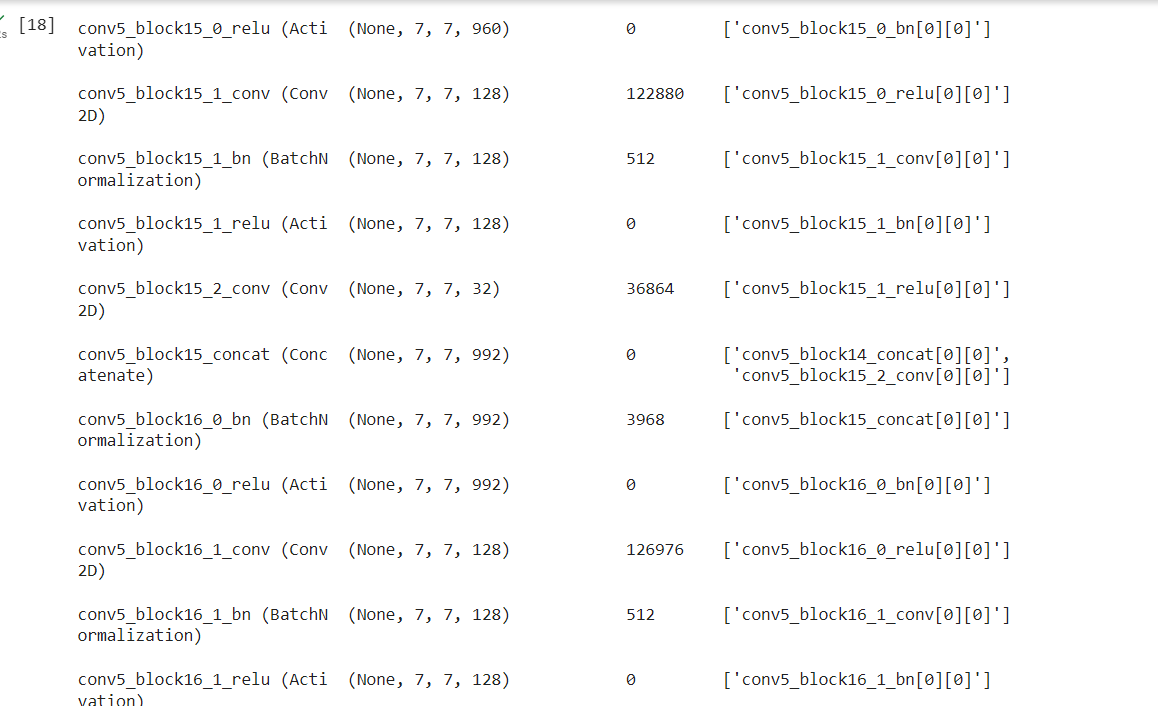
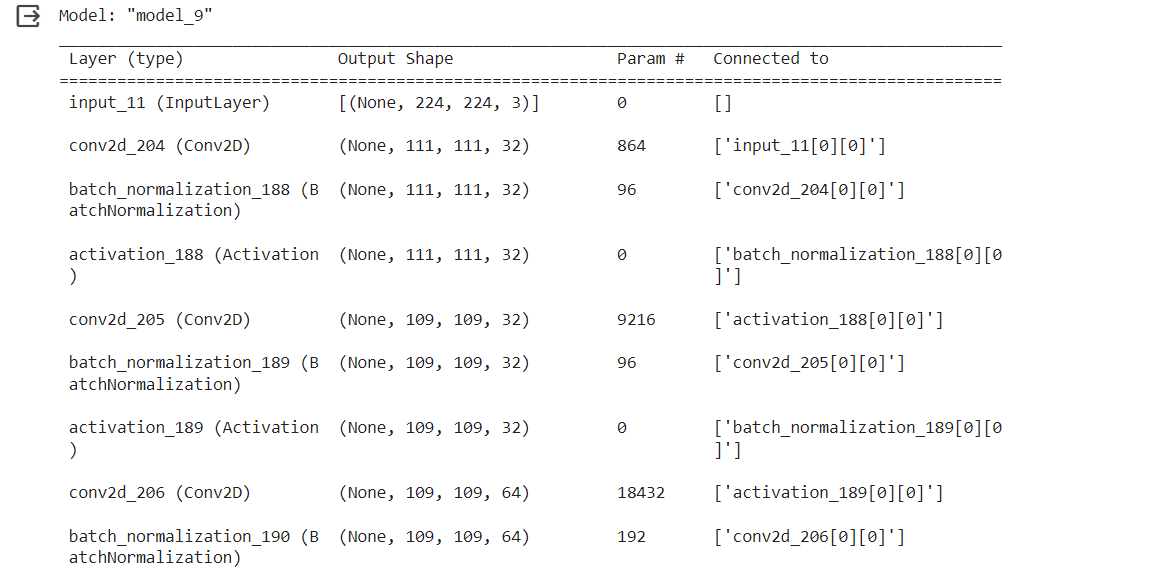
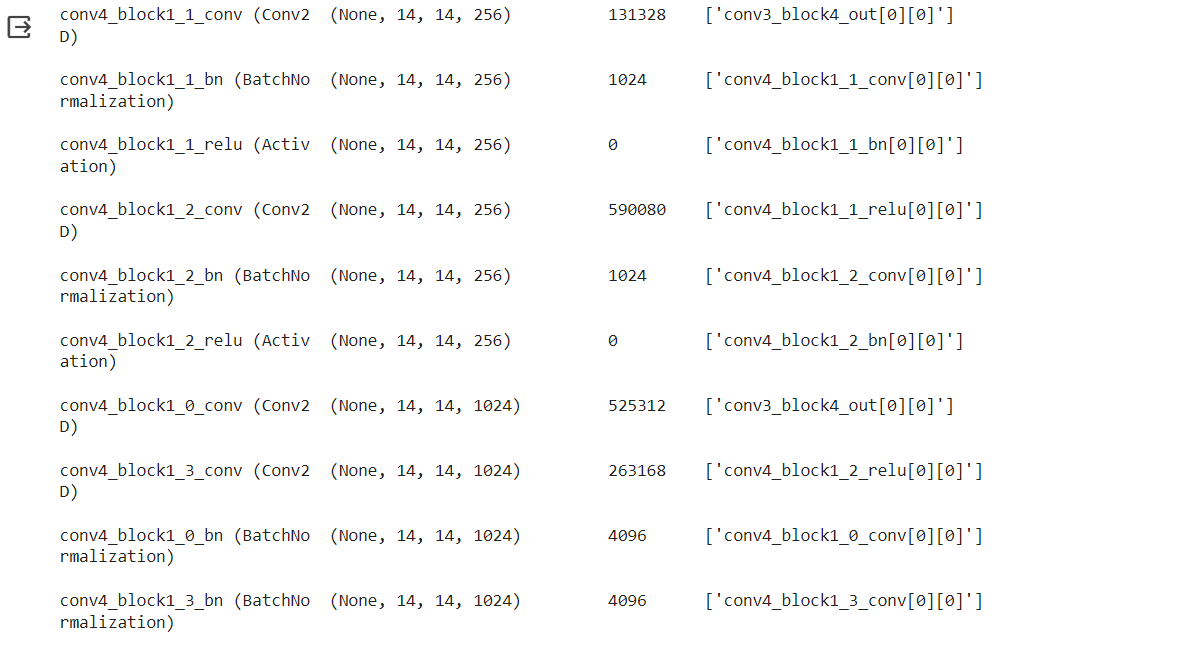
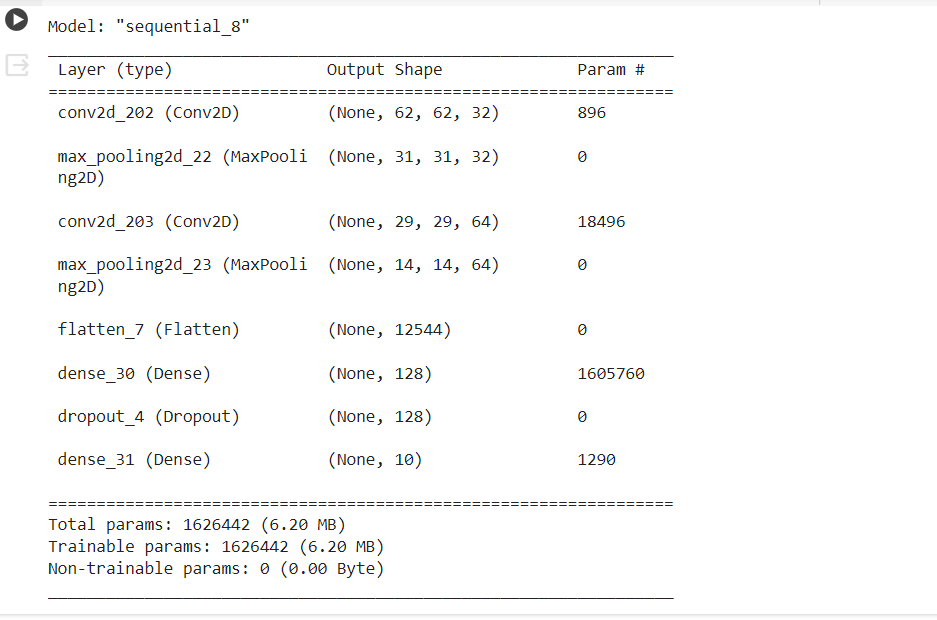
    validation\_data=validation\_generator

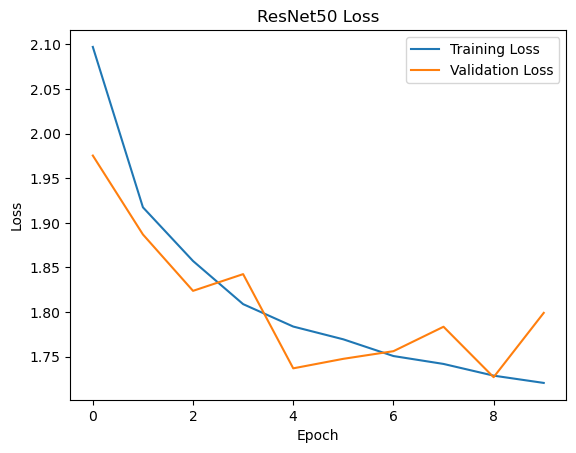
)

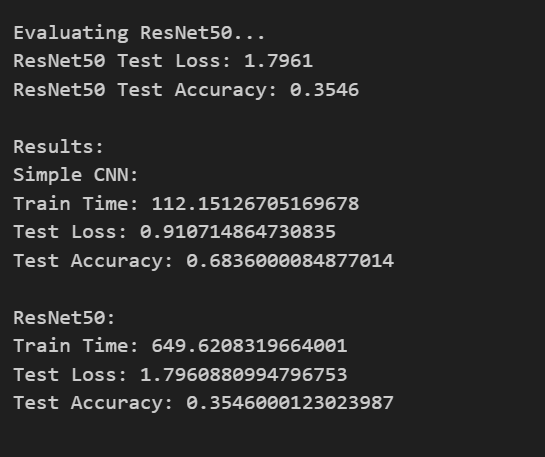
test\_loss\_densenet, test\_acc\_densenet = densenet\_model.evaluate(test\_datagen.flow(resized\_test\_images, test\_labels, batch\_size=32))

print("DenseNet Model Test Loss:", test\_loss\_densenet)

print("DenseNet Model Test Accuracy:", test\_acc\_densenet)







5)Address the challenges of transfer learning when dealing with small datasets.

a. Choose a small dataset and demonstrate the effectiveness of transfer learning compared to training a CNN from scratch.

b. Experiment with data augmentation techniques and analyze their impact on model

generalization.

**Answer:**

Description: Data augmentation involves generating synthetic training data by modifying existing images, such as rotating, flipping, or changing colors. This technique enhances the robustness of deep learning models, enabling them to handle variations in real-world data. By introducing these variations during training, data augmentation helps prevent overfitting and improves the model's ability to generalize to new information. Transfer learning adapts pre-trained models to new tasks, leveraging prior knowledge to enhance performance and accelerate learning.

Creating a CNN model from scratch involves designing and building the architecture, including convolutional, pooling, and fully connected layers. Then, the model is trained on a dataset using techniques like backpropagation to learn the features and optimize parameters for specific tasks such as image classification.

import numpy as np

from tensorflow.keras.applications import VGG16

from tensorflow.keras.layers import Dense, Flatten, Input

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

import tensorflow as tf

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

X\_train = X\_train.astype('float32') / 255.0

X\_test = X\_test.astype('float32') / 255.0

y\_train = to\_categorical(y\_train, num\_classes=10)

y\_test = to\_categorical(y\_test, num\_classes=10)

def create\_custom\_cnn():

    model = tf.keras.Sequential([

        tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)),

        tf.keras.layers.MaxPooling2D((2, 2)),

        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),

        tf.keras.layers.MaxPooling2D((2, 2)),

        tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),

        tf.keras.layers.MaxPooling2D((2, 2)),

        tf.keras.layers.Flatten(),

        tf.keras.layers.Dense(64, activation='relu'),

        tf.keras.layers.Dense(10, activation='softmax')

    ])

    model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

def create\_transfer\_learning\_model():

    vgg16\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(32, 32, 3))

    for layer in vgg16\_model.layers:

        layer.trainable = False

    x = vgg16\_model.output

    x = Flatten()(x)

    x = Dense(1024, activation='relu')(x)

    predictions = Dense(10, activation='softmax')(x)

    model = Model(inputs=vgg16\_model.input, outputs=predictions)

    model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

    return model

custom\_cnn\_model = create\_custom\_cnn()

transfer\_learning\_model = create\_transfer\_learning\_model()

custom\_cnn\_model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))

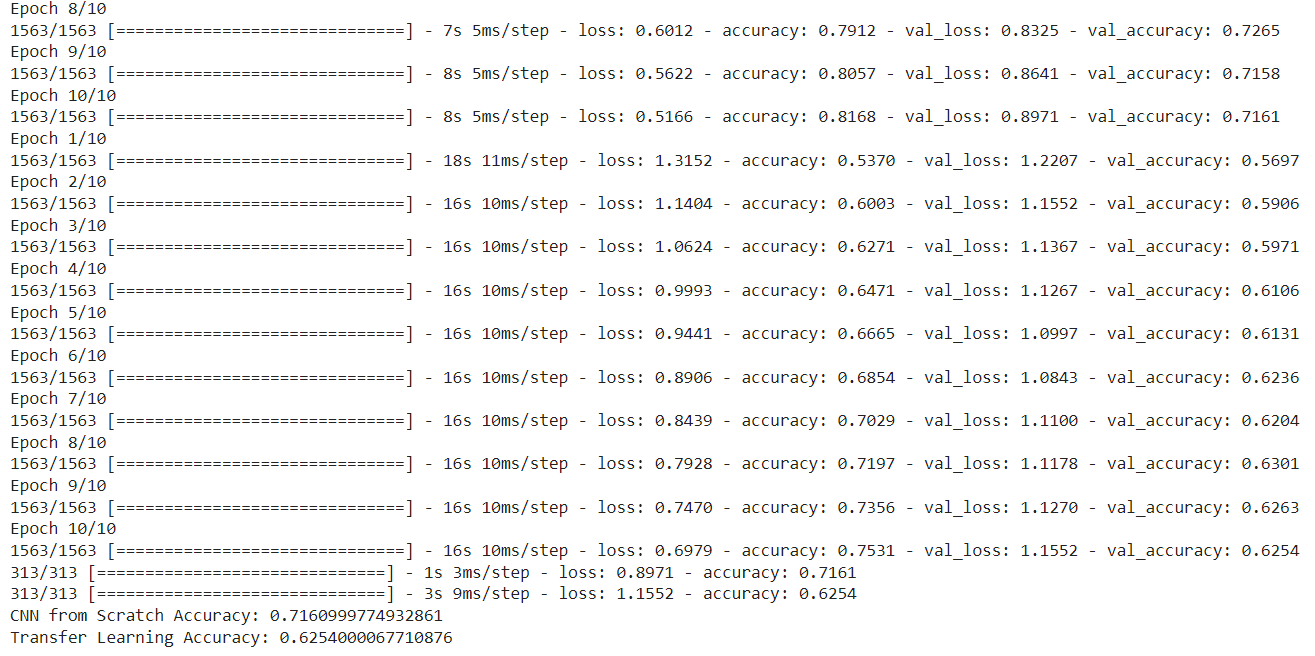
transfer\_learning\_model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))

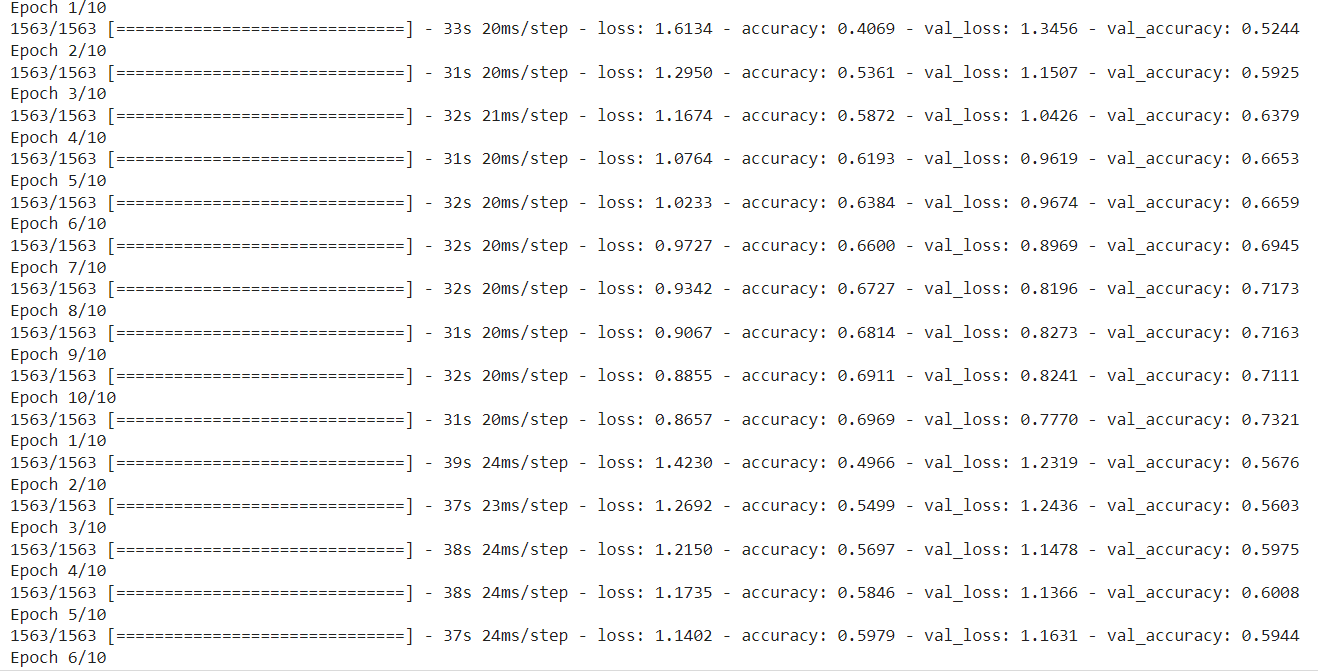
custom\_cnn\_loss, custom\_cnn\_accuracy = custom\_cnn\_model.evaluate(X\_test, y\_test)

transfer\_learning\_loss, transfer\_learning\_accuracy = transfer\_learning\_model.evaluate(X\_test, y\_test)

print("CNN from Scratch Accuracy:", custom\_cnn\_accuracy)

print("Transfer Learning Accuracy:", transfer\_learning\_accuracy)

****

****

**b)**

from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(

    rotation\_range=15,

    width\_shift\_range=0.1,

    height\_shift\_range=0.1,

    horizontal\_flip=True,

    fill\_mode='nearest'

)

custom\_cnn\_model = create\_custom\_cnn()

transfer\_learning\_model = create\_transfer\_learning\_model()

custom\_cnn\_model.fit(datagen.flow(X\_train, y\_train, batch\_size=32),

                     epochs=10, validation\_data=(X\_test, y\_test))

transfer\_learning\_model.fit(datagen.flow(X\_train, y\_train, batch\_size=32),

                             epochs=10, validation\_data=(X\_test, y\_test))

cnn\_from\_scratch\_loss\_new, cnn\_from\_scratch\_acc\_new = cnn\_from\_scratch.evaluate(X\_test, y\_test)

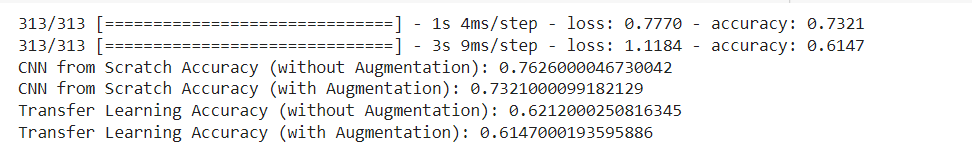
transfer\_learning\_loss\_new, transfer\_learning\_acc\_new = transfer\_learning\_model.evaluate(X\_test, y\_test)

print("CNN from Scratch Accuracy (without Augmentation):", cnn\_from\_scratch\_acc)

print("CNN from Scratch Accuracy (with Augmentation):", cnn\_from\_scratch\_acc\_new)

print("Transfer Learning Accuracy (without Augmentation):", transfer\_learning\_acc)

print("Transfer Learning Accuracy (with Augmentation):", transfer\_learning\_acc\_new)

****