Deep learning HT24: Project 4

Joakim Andersson Svendsen & Max Brehmer

January 2025

Contents

1 Task description **2**
1.1 Dataset . . . 2
1.2 Task . . . 2

2 Methods **3**
2.1 CNN . . . 3
2.1.1 Data augmentation . . . 4
2.1.2 Vanilla CNN architecture . . . 4
2.2 MobileNet v2 . . . 5

3 Results **7**

4 Discussion **9**

5 Appendix **11**
5.1 Code . . . 11
5.1.1 Create data pipeline . . . 11
5.1.2 Image preprocessing . . . 12
5.1.3 Data augmentation . . . 13
5.1.4 Create MobileNetV2 models . . . 13
5.1.5 Create vanilla CNN model . . . 14
5.1.6 Train the networks . . . 15
5.1.7 Plot loss and accuracy curves . . . 15
5.1.8 Vanilla CNN curves . . . 16

1 Task description

1.1 Dataset

In this project we are working with a comprehensive dataset that encompasses a collection of chest X-rays collected during the COVID-19 pandemic, with cases from both sick and unaffected individuals. The dataset contains 324 images in total, with an equal number (162) of COVID-19 affected and normal cases. The dataset provides a valuable resource for benchmarking image classification ML algorithms which can improve medical research and assist medical professionals in the context of chest radiology.

Training and validation data is split 75/25 since there are relatively few images in this dataset. After splitting there are 324 images belonging to 2 classes, of which 243 are training images, and 81 are validation images. The batch size used is 32.

The images are loaded as 224 by 224 RGB images, meaning the input tensors we are working with have $(224, 224, 3)$ dimensions. Since the images are RGB coded, each data point is on a range $[0, 255]$. Model-specific pre-processing is performed on the data so that each RGB value is represented on a scale $[-1, 1]$ or $[0, 1]$.

1.2 Task

Our dataset consists of chest X-ray images, which are two-dimensional grids of pixel values. The goal is to perform binary classification on these X-ray images into the categories: **COVID-19 positive** and **Normal**. Given the spatial nature of the data and the importance of extracting hierarchical patterns (e.g., abnormalities in lung structure), we figured that a 2D CNN is a natural fit for this task.

2 Methods

2.1 CNN

A Convolutional Neural Network (CNN) is a type of deep learning model specifically designed to process data with a grid-like structure, such as images. Unlike traditional neural networks, CNNs preserve the spatial structure of the data and focus on learning patterns like edges, textures, and shapes directly from the input images. This makes CNNs highly effective for tasks involving image analysis.

One of the key features of a CNN is the use of convolutional layers, which apply filters (kernels) that slide over the input image to extract features. These filters are much smaller than the original image (e.g., 3x3 or 5x5) and allow the model to detect patterns across the image efficiently. These patterns can range from simple edges in the early layers to more complex shapes and structures in deeper layers.

After the convolution operation, an activation function (commonly ReLU) is applied. This introduces non-linearity, enabling the network to learn complex, non-linear relationships in the data. Following this, a pooling layer is often applied to reduce the spatial dimensions of the feature maps. Pooling retains the most important information while reducing computational complexity and preventing overfitting. Near the end of the network, fully connected layers combine the extracted features to make a final prediction. [1]

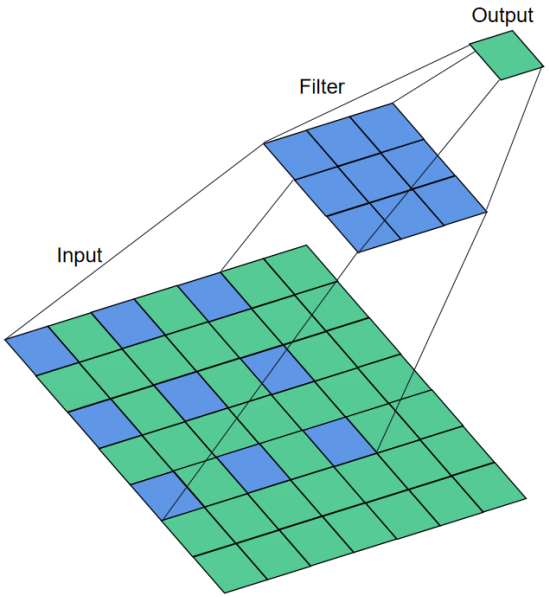

Figure 1: Convolutional layers (Image borrowed from DO:10.3390/s21217319)

2.1.1 Data augmentation

For our own CNN model we attempt to avoid overfitting by augmenting images. Random horizontal flips, accompanied by randomized rotations, zooms and contrast variations of 10% are introduced to the training data to avoid overfitting, forcing the model to learn the more important features and not simply memorize features from the training set.

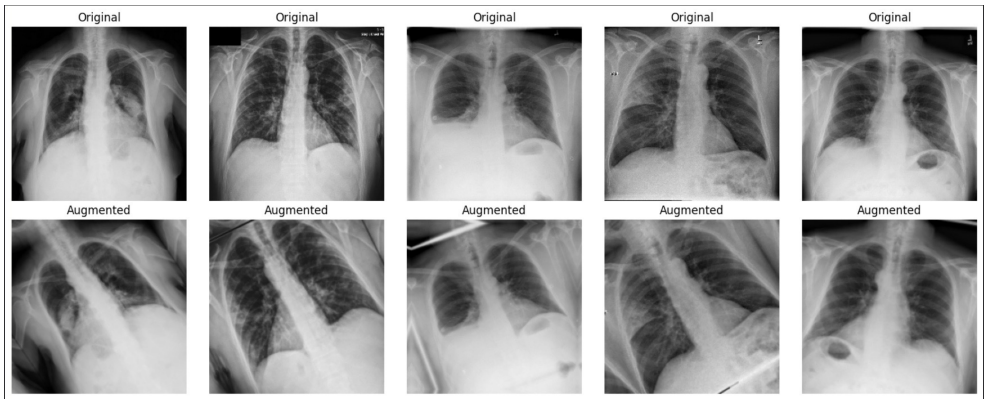

Figure 2: Original and augmented images for comparison

2.1.2 Vanilla CNN architecture

Image classification is commonly applied by using a convolutional neural network, so we begin by constructing a simple CNN using `tensorflow.keras`. The model we construct contains 4 convolutional layers, starting with 32 filters of size $(5, 5)$ and a MaxPooling layer of size $(3, 3)$. Following are $64, (3, 3)$, $128, (3, 3)$, and $128, (3, 3)$ convolutions with $(2, 2)$ MaxPooling.

The output shape before flattening is $(7, 7, 128)$, which is the same as for MobileNetV2. The activation function used in each of the non-final layers is ReLU, while the output layer uses sigmoid and a single output channel for binary classification. Dropout regularization of 20% ensures the network does not get overly reliant on certain neurons and thus prevents overfitting.

Layer type	Filters	# Param
Input (Sequential)	(224, 224, 3)	
Conv2D	(32, 5, 5)	2,432
Activation	ReLU	
MaxPooling2D	(3, 3)	
Conv2D	(64, 3, 3)	18,496
Activation	ReLU	
MaxPooling2D	(2, 2)	
Conv2D	(128, 3, 3)	73,856
Activation	ReLU	
MaxPooling2D	(2, 2)	
Conv2D	(128, 3, 3)	147,584
Activation	ReLU	
MaxPooling2D	(2, 2)	
Dense	(128)	802,944
Activation	ReLU	
Dropout	20%	
Dense	(1)	129
Activation	Sigmoid	
		1,045,441

Table 1: Simple CNN Model summary

2.2 MobileNet v2

To compare the performance of two different approaches, we implemented a 2D CNN using transfer learning with the pre-trained transfer learning network MobileNetV2. Since MobileNetV2 is pre-trained on a dataset where the images are of size 224x224x3, we resized the images in our dataset to match this size to avoid shape mismatch issues.

MobileNetV2 is a pre-trained convolutional neural network (CNN) consisting of 155 layers, trained on the ImageNet dataset, which contains millions of natural images. This training allows MobileNetV2 to learn general features, such as edges, textures, and shapes, which can be transferred to other tasks. We opted for transfer learning because our dataset is relatively small (324 images). By using the pretrained model, we can use these general features as a starting point, reducing the risk of overfitting and improving computational efficiency, as the model does not have to learn all features from scratch. We do not deem data augmentation to be necessary for this model. [2]

One important decision when using transfer learning is whether or not to "freeze" the imported layers. Freezing layers means keeping the weights of the pretrained model unchanged during training, which prevents the model from fine-tuning these layers to the specific dataset. We implemented three approaches:

- Completely frozen layers: All 155 layers were frozen, and only the custom classification head was trained.
- Partially frozen layers: The first 135 layers were frozen, leaving the final layers trainable to fine-tune specific features relevant to our data.
- Completely unfrozen layers: All layers were trainable, allowing the model to update the weights across the entire architecture.

For all models, including the vanilla CNN, the `tensorflow.keras` package with the highly efficient optimizer `Adam` was used, and the custom classification head consisted of the following layers:

- A global average pooling layer to condense spatial features into a single vector.
- A dense layer with 128 neurons and ReLU activation to learn complex patterns.
- A dropout layer (50%).
- A final output layer with 1 neuron and sigmoid activation for binary classification.

A summarized table of the hyperparameters settings can be found in **Table 1**

Model	**Adam learning rate**	**DropOut rate**
Model 1 (Vanilla CNN)	`(10e-5)`	0.2
Model 2 (Freeze all layers)	`(10e-4)`	0.5
Model 3 (Freeze first 135 layers)	`(10e-6)`	0.5
Model 4 (No frozen layers)	`(10e-6)`	0.5

Table 2: Model hyperparameters

3 Results

All of the models where evaluated using the accuracy metric and a Binary Cross-entropy loss function. Each model was trained for 500 epochs with early stopping patience of 30 epochs to prevent excessive training.

Model	Tr. accuracy	Tr. loss	Val. accuracy	Val. loss	Epoch runtime
Model 1	88.89%	0.2608	81.48%	0.4318	331
Model 2	100%	0.0273	92.59%	0.1873	127
Model 3	100%	0.0609	95.06%	0.1767	500
Model 4	99.59%	0.0939	88.89%	0.4017	288

Table 3: Model Performances

The first model, the CNN we built from scratch performed the worst with a validation accuracy of 81.48%. The model's loss and accuracy curves, particularly the validation curves are significantly more chaotic than the more sophisticated pre-trained models. The model appears to reach an accuracy of $\sim 81\%$ within 50 epochs.

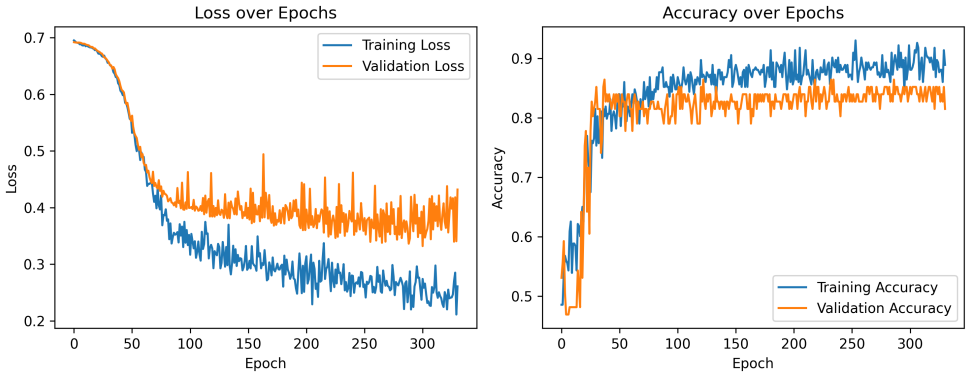

Figure 3: Model 1 (Vanilla CNN) performance

All models using the pre-trained network MobileNetV2 performed reasonably well, however we found significant variations in performance between the models $2-4$. Most notably the model with 135 frozen layers achieved an accuracy of 95.06% (model 3). The epoch runtime was the maximum 500, suggesting that the model could have improved even further given more training time.

Model 2, the model with fully frozen layers apart from the classification head performed second best, however its validation curves stagnated after only 20 epochs, suggesting that some layers should perhaps be left open for training as the fully frozen model cannot adjust the weights beside the final layers,

compromising the performance.

The worst performing model of the MobileNetV2 approaches was the fully trainable model 4. Its validation curves indicate a high discrepancy between training and validation. The accuracy plot shows that the model does not converge monotonously. The relatively poor performance of 88.89% achieved with model 4 may be due to the indiscriminant update of weights across the network, perhaps undoing a lot of the information aquired during pre-training which can also lead to overfitting.

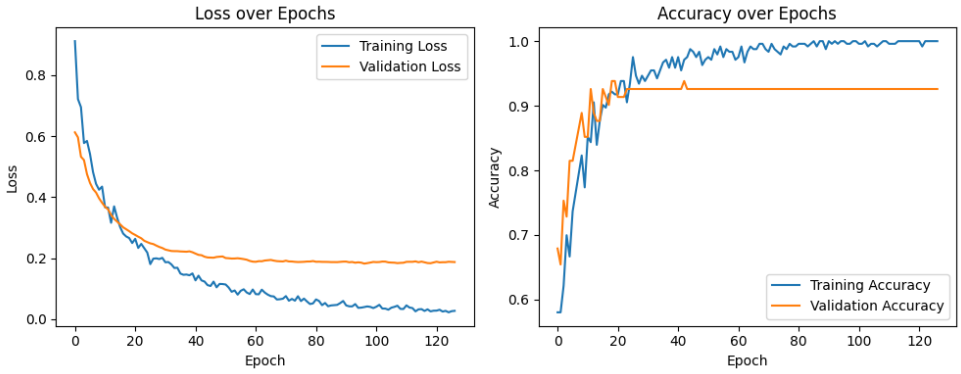

Figure 4: Model 2 (Freeze all layers) performance

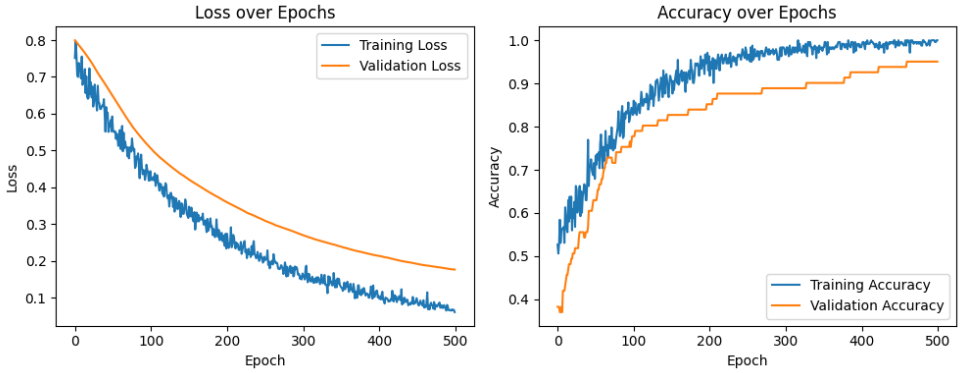

Figure 5: Model 3 (Freeze first 135 layers) performance

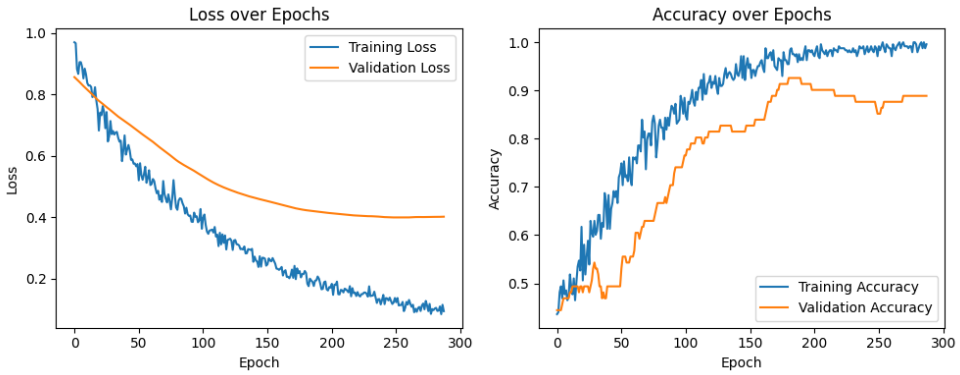

Figure 6: Model 4 (No frozen layers) performance

4 Discussion

In this analysis, we applied a Convolutional Neural Network (CNN) to classify whether an image showed symptoms of COVID-19 or not. We used two different approaches: a vanilla CNN and transfer learning with MobileNetV2.

The vanilla CNN performed worse compared to the transfer learning models, which is expected given the small dataset size. Without access to pretrained features, the vanilla CNN struggled to learn meaningful patterns from scratch, leading to suboptimal performance. On the other hand, the transfer learning models achieved significantly better results, leveraging the general features learned during training on the ImageNet dataset.
However, a notable difference between the two approaches was the dropout rates used during training. The vanilla CNN used a dropout rate of 0.2, whereas the transfer learning models used a higher dropout rate of 0.5. While dropout is commonly employed to prevent overfitting, we saw no immediate signs of overfitting in the vanilla CNN, suggesting that its lower performance came from limited feature extraction capabilities rather than overfitting. In contrast, the transfer learning models, which started with pretrained weights, faced a higher risk of overfitting to the small dataset, which is why we used a higher dropout rate to regularize the training process.

Among the transfer learning models, Model 2 (all frozen layers) emerged as perhaps the most "practical" choice. While Model 3 (partially frozen layers) achieved slightly better accuracy, it came at the cost of significantly higher training time. Model 2 balanced performance and computational efficiency, achieving nearly the same results in a fraction of the time. However, in a medical context, where accuracy directly impacts patient outcomes, the marginal improvement of Model 3 could still justify its use despite the increased computational cost.

It is possible that we could have achieved better performance using a different architecture, such as ResNet. While MobileNetV2 is highly efficient, it may lack the capacity to capture complex features that ResNet, with its deeper architecture and skip connections, could handle. However, the higher computational cost of ResNet makes it less practical for tasks prioritizing efficiency.

Overall, the analysis was successful and highlighted the trade-off between computational efficiency and performance in deep learning. Compared to a vanilla CNN, transfer learning reduced computational cost(in most cases), enabled faster convergence, and generalized better on the small dataset by leveraging pretrained features from ImageNet. However, transfer learning posed challenges, particularly in determining the optimal freezing strategies and hyperparameters, which often relied on trial and error. This shows the need for systematic and intuitive approaches to fine-tuning and highlights the balance required between generalization and adaptation.

In conclusion, this study demonstrates that transfer learning is effective for small datasets, offering strong performance with limited resources. While alternative architectures like ResNet may provide further improvements, the balance between efficiency and performance makes MobileNetV2 a practical choice for this task. Future work could explore more systematic hyperparameter tuning and alternative models, as well as expanding the dataset to further enhance performance.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Mark Sandler. Mobilenetv2: Inverted residuals and linear bottlenecks. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

5 Appendix

To run code yourself, view our Github repo. We ran our code using Kaggle GPU T4×2 accelerator.

5.1 Code

5.1.1 Create data pipeline

```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
import cv2

# Find directory paths for images
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Assign directory path for images
train_data_dir = "/kaggle/input/covid-19-vs-normal-chest-x-rays/"
train_normal_dir = train_data_dir + "NORMAL/NORMAL/"
train_covid_dir = train_data_dir + "COVID/COVID/"
```

```
import matplotlib.pyplot as plt
import seaborn as sns

def display_images(directory, num_images=5, image_size=(100, 100),
    title=''):
    plt.figure(figsize=(15, 3))
    plt.title(title)
    plt.axis('off')

    for i, image_name in enumerate(os.listdir(directory)[:
    num_images]):
        image_path = os.path.join(directory, image_name)
        img = cv2.imread(image_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, image_size)

        plt.subplot(1, num_images, i + 1)
        plt.imshow(img)
```

```
        plt.axis('off')
        plt.title(image_name)

    plt.show()

display_images(train_normal_dir, title='NORMAL Images')
display_images(train_covid_dir, title='COVID Images')
```

5.1.2 Image preprocessing

```
import tensorflow as tf
from tensorflow.keras import layers, models, backend, Sequential
from tensorflow.keras.preprocessing import
    image_dataset_from_directory
from tensorflow.keras.applications.mobilenet_v2 import
    preprocess_input
from tensorflow.keras.layers import Rescaling
from tensorflow.data import AUTOTUNE

# Resize and scale images
img_height = 224
img_width = 224
batch_size = 32

# Load training and validation datasets
train_ds = image_dataset_from_directory(
    train_data_dir,
    validation_split=0.25,
    subset="training",
    seed=42,
    image_size=(img_height, img_width),
    batch_size=batch_size
)

val_ds = image_dataset_from_directory(
    train_data_dir,
    validation_split=0.25,
    subset="validation",
    seed=42,
    image_size=(img_height, img_width),
    batch_size=batch_size
)

# Apply MobileNetV2-specific preprocessing
train_ds = train_ds.map(lambda x, y: (preprocess_input(x), y),
    num_parallel_calls=AUTOTUNE)
val_ds = val_ds.map(lambda x, y: (preprocess_input(x), y),
    num_parallel_calls=AUTOTUNE)

# Prefetch data to improve performance
train_ds = train_ds.prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.prefetch(buffer_size=AUTOTUNE)

# Apply CNN-specific preprocessing
AUTOTUNE = tf.data.AUTOTUNE
```

```
train_ds_cnn = train_ds.map(lambda x, y: (data_augmentation(x,
    training=True), y))
train_ds_cnn = train_ds_cnn.prefetch(buffer_size=AUTOTUNE)
val_ds_cnn = val_ds.prefetch(buffer_size=AUTOTUNE)

train_ds_cnn = train_ds.map(lambda x, y: (x / 255.0, y))
val_ds_cnn = val_ds.map(lambda x, y: (x / 255.0, y))
```

5.1.3 Data augmentation

```
# Data augmentation pipeline
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
    layers.RandomContrast(0.1)
])

# Function to display original and augmented images
def display_augmented_images(dataset, augmentation_model,
    num_images=5):
    for images, labels in dataset.take(1):  # Take one batch
        augmented_images = augmentation_model(images)  # Apply
    augmentation

        plt.figure(figsize=(15, 6))
        for i in range(num_images):
            # Display original image
            ax = plt.subplot(2, num_images, i + 1)
            plt.imshow(images[i].numpy().astype("uint8"))
            plt.axis("off")
            plt.title("Original")

            # Display augmented image
            ax = plt.subplot(2, num_images, i + 1 + num_images)
            plt.imshow(augmented_images[i].numpy().astype("uint8"))
            plt.axis("off")
            plt.title("Augmented")

        plt.tight_layout()
        plt.show()

# Display the images
display_augmented_images(train_ds, data_augmentation, num_images=5)
```

5.1.4 Create MobileNetV2 models

```
optimizer = Adam(learning_rate=0.000001)

# Build the model
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.optimizers import Adam

def create_model():
```

```
    # Load MobileNetV2 pretrained on ImageNet
    base_model = MobileNetV2(input_shape=(224, 224, 3),
                             include_top=False,  # Exclude the top
    layer for custom classification
                             weights='imagenet')

    # Freeze the base model
    #base_model.trainable = False
    base_model.trainable = True

    # Freeze only the first 100 layers
    #for layer in base_model.layers[:135]:
        #layer.trainable = False
    #for layer in base_model.layers[135:]:
        #layer.trainable = True

    # Add custom classification head
    model = models.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(1, activation='sigmoid')  # Binary
    classification
    ])

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

    return model

# Reset the model by clearing the backend and creating a new
    instance
backend.clear_session()
model = create_model()

model.summary()
```

5.1.5 Create vanilla CNN model

```
model1 = models.Sequential()
model1.add(layers.Input(shape=(224, 224, 3)))
model1.add(data_augmentation)

model1.add(layers.Conv2D(32, (5, 5)))
model1.add(layers.ReLU())
model1.add(layers.MaxPooling2D((3, 3)))

model1.add(layers.Conv2D(64, (3, 3)))
model1.add(layers.ReLU())
model1.add(layers.MaxPooling2D((2, 2)))

model1.add(layers.Conv2D(128, (3, 3)))
model1.add(layers.ReLU())
```

```
model1.add(layers.MaxPooling2D((2, 2)))

model1.add(layers.Conv2D(128, (3, 3)))
model1.add(layers.ReLU())
model1.add(layers.MaxPooling2D((2, 2)))

model1.add(layers.Flatten())
model1.add(layers.Dense(128))
model1.add(layers.ReLU())
model1.add(layers.Dropout(0.2))

model1.add(layers.Dense(1, activation='sigmoid'))

model1.summary()
```

5.1.6 Train the networks

```
from tensorflow.keras.callbacks import EarlyStopping

# Early stopping to prevent overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=30,
    restore_best_weights=True)

# Train MobileNetV2 models
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=500,
    callbacks=[early_stopping]
)

# Compile vanilla CNN model
model1.compile(optimizer=Adam(learning_rate=0.00001),
               loss='binary_crossentropy',
               metrics=['accuracy'])

# Train vanilla CNN model
history1 = model1.fit(
    train_ds_cnn,
    validation_data=val_ds_cnn,
    epochs=500,
    callbacks=[early_stopping]
)
```

5.1.7 Plot loss and accuracy curves

```
# Extract the loss and accuracy values for each epoch
train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_acc = history.history.get('accuracy', history.history.get('
    acc', []))
val_acc = history.history.get('val_accuracy', history.history.get('
    val_acc', []))

```

```
# Plot loss curves
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label='Training Loss')
plt.plot(val_loss, label='Validation Loss', linestyle='-')
plt.title('Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plot accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy', linestyle='-')
plt.title('Accuracy over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
#plt.savefig("ModelX.png")
plt.show()
```

5.1.8 Vanilla CNN curves

```
# Extract the loss and accuracy values for each epoch
train_loss_cnn = history1.history['loss']
val_loss_cnn = history1.history['val_loss']
train_acc_cnn = history1.history.get('accuracy', history1.history.
    get('acc', []))
val_acc_cnn = history1.history.get('val_accuracy', history1.history
    .get('val_acc', []))

# Plot loss curves
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss_cnn, label='Training Loss')
plt.plot(val_loss_cnn, label='Validation Loss', linestyle='-')
plt.title('Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Plot accuracy curves
plt.subplot(1, 2, 2)
plt.plot(train_acc_cnn, label='Training Accuracy')
plt.plot(val_acc_cnn, label='Validation Accuracy', linestyle='-')
plt.title('Accuracy over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
#plt.savefig("ModelX.png")
plt.show()
```