a) Increasing the number of neurons to (64, 128) in the two convolutional layers may improve the model's ability to learn more complex features from the input images. However, it also increases the number of trainable parameters, which can lead to overfitting if the model is not properly regularized. To investigate the effect of this change, we can modify the code as follows:

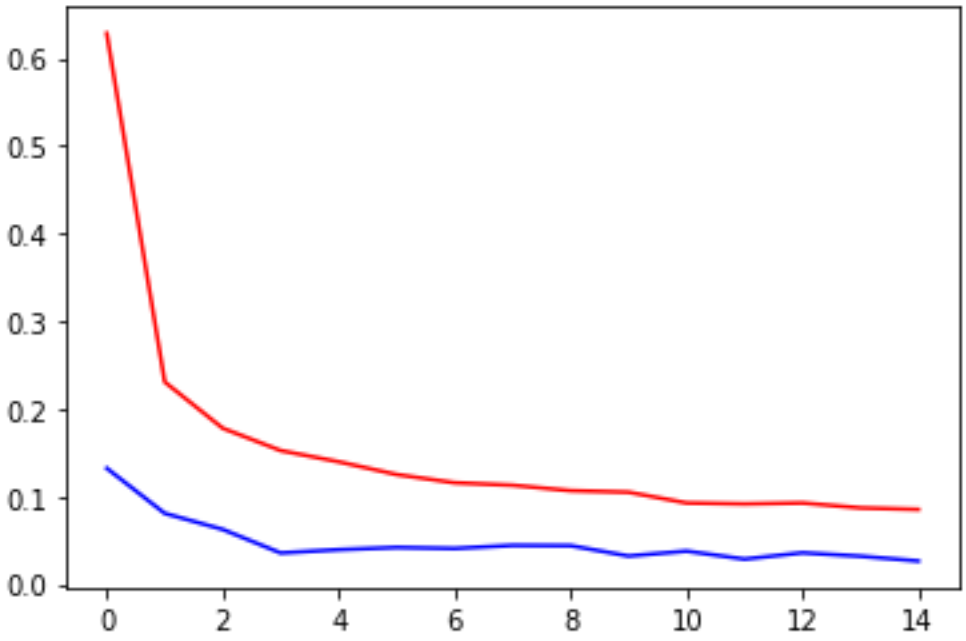
Change the number of neurons in two convolutional layers from (32, 64) to (64, 128):

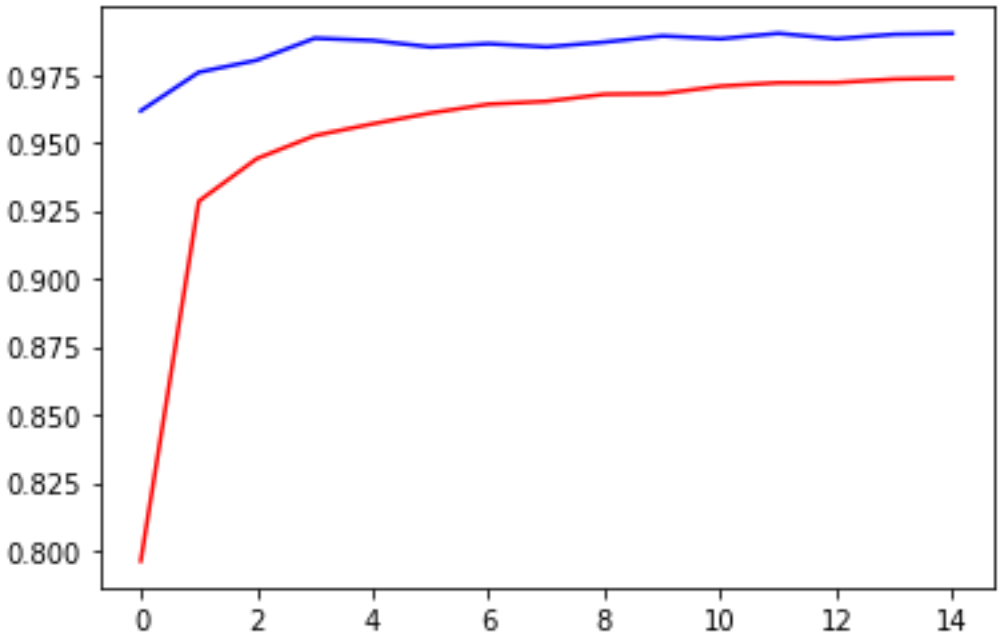
```
#Preparing a CNN model architecture
model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(64, (5,5), activation='relu', kernel_initializer='he_uniform',input_shape=(28, 28, 1)),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Conv2D(128, (5,5), activation='relu'),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Dropout(0.25),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(10, activation='softmax')
    ])

[16]  ✓ 0.1s
```

Train the model for 15 epochs and plot the training and validation accuracy over epochs:

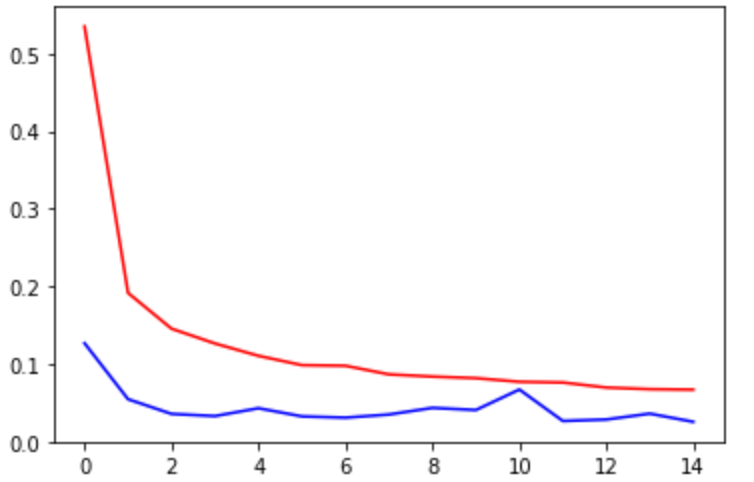
```
Epoch 1/15
525/525 [==============================] - 70s 134ms/step - loss: 0.5528 - accuracy: 0.8203 - val_loss: 0.0783 - val_accuracy: 0.9768
Epoch 2/15
525/525 [==============================] - 71s 135ms/step - loss: 0.1870 - accuracy: 0.9423 - val_loss: 0.0584 - val_accuracy: 0.9835
Epoch 3/15
525/525 [==============================] - 74s 141ms/step - loss: 0.1431 - accuracy: 0.9565 - val_loss: 0.0498 - val_accuracy: 0.9858
Epoch 4/15
525/525 [==============================] - 72s 137ms/step - loss: 0.1221 - accuracy: 0.9611 - val_loss: 0.0505 - val_accuracy: 0.9849
Epoch 5/15
525/525 [==============================] - 72s 138ms/step - loss: 0.1104 - accuracy: 0.9653 - val_loss: 0.0534 - val_accuracy: 0.9857
Epoch 6/15
525/525 [==============================] - 68s 129ms/step - loss: 0.1009 - accuracy: 0.9690 - val_loss: 0.0475 - val_accuracy: 0.9860
Epoch 7/15
525/525 [==============================] - 72s 138ms/step - loss: 0.0941 - accuracy: 0.9705 - val_loss: 0.0329 - val_accuracy: 0.9904
Epoch 8/15
525/525 [==============================] - 75s 143ms/step - loss: 0.0840 - accuracy: 0.9745 - val_loss: 0.0322 - val_accuracy: 0.9906
Epoch 9/15
525/525 [==============================] - 72s 137ms/step - loss: 0.0843 - accuracy: 0.9738 - val_loss: 0.0367 - val_accuracy: 0.9883
Epoch 10/15
525/525 [==============================] - 65s 124ms/step - loss: 0.0783 - accuracy: 0.9761 - val_loss: 0.0358 - val_accuracy: 0.9907
Epoch 11/15
525/525 [==============================] - 65s 124ms/step - loss: 0.0741 - accuracy: 0.9766 - val_loss: 0.0315 - val_accuracy: 0.9896
Epoch 12/15
525/525 [==============================] - 65s 124ms/step - loss: 0.0749 - accuracy: 0.9771 - val_loss: 0.0412 - val_accuracy: 0.9881
Epoch 13/15
...
Epoch 14/15
525/525 [==============================] - 79s 151ms/step - loss: 0.0655 - accuracy: 0.9791 - val_loss: 0.0260 - val_accuracy: 0.9926
Epoch 15/15
525/525 [==============================] - 78s 148ms/step - loss: 0.0660 - accuracy: 0.9794 - val_loss: 0.0466 - val_accuracy: 0.9864
```

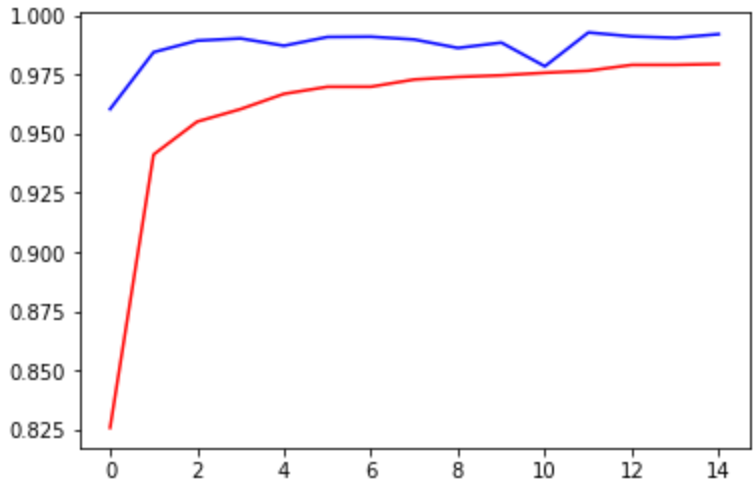
Original:





After:



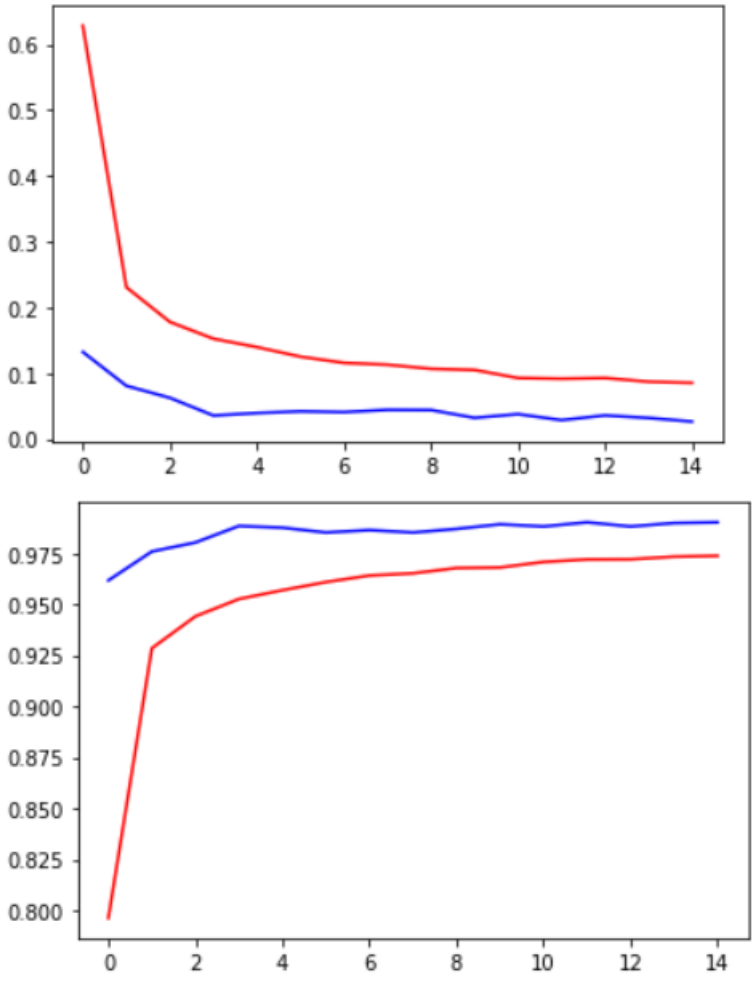


b) To increase the number of convolutional layers to 3, we can add another Conv2D layer to the existing architecture. We will use (32, 64, 32) neurons with 3x3 convolution filters in the third layer. The updated model architecture will be:

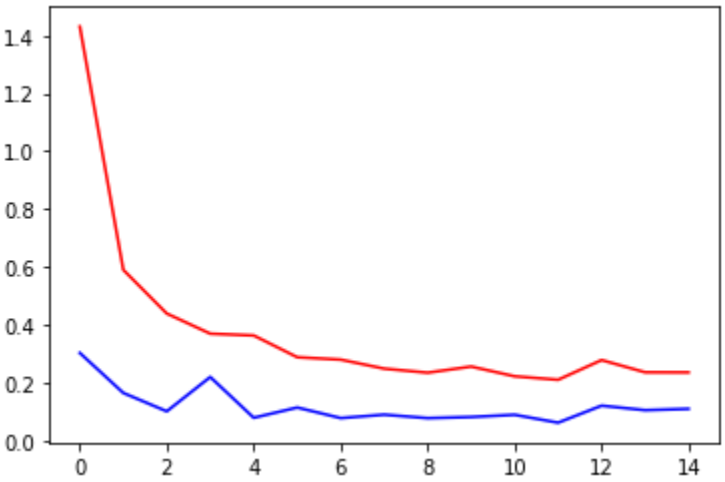
```
#Preparing a CNN model architecture
model = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, (5,5), activation='relu', kernel_initializer='he_uniform', input_shape=(28, 28, 1)),
            tf.keras.layers.MaxPooling2D(2, 2),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.Conv2D(64, (5,5), activation='relu'),
            tf.keras.layers.MaxPooling2D(2, 2),
            tf.keras.layers.Dropout(0.25),
            tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
            tf.keras.layers.MaxPooling2D(2, 2),
            tf.keras.layers.Dropout(0.3),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.Dense(10, activation='softmax')
    ])
[25]  ✓ 0.1s
```

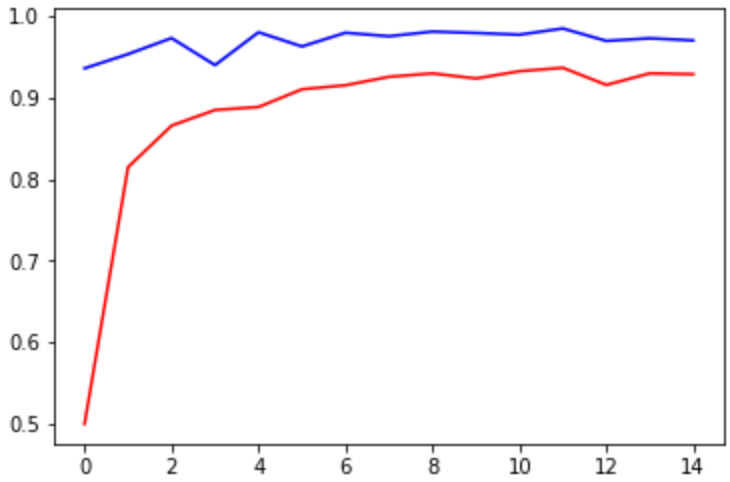
Now, we will train the updated model and compare the training, validation, and testing accuracy with the previous model.

Original:



After:

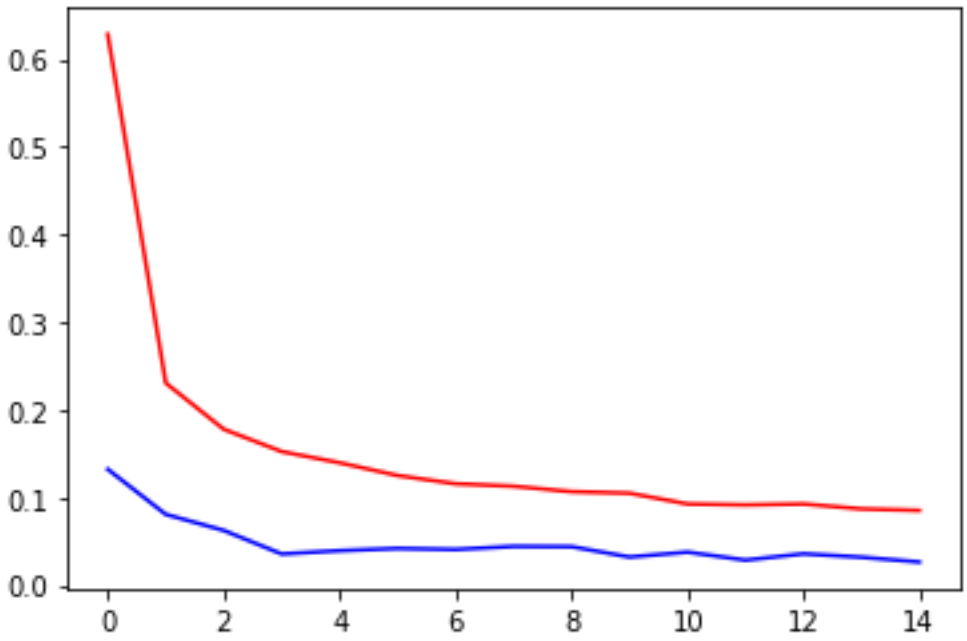


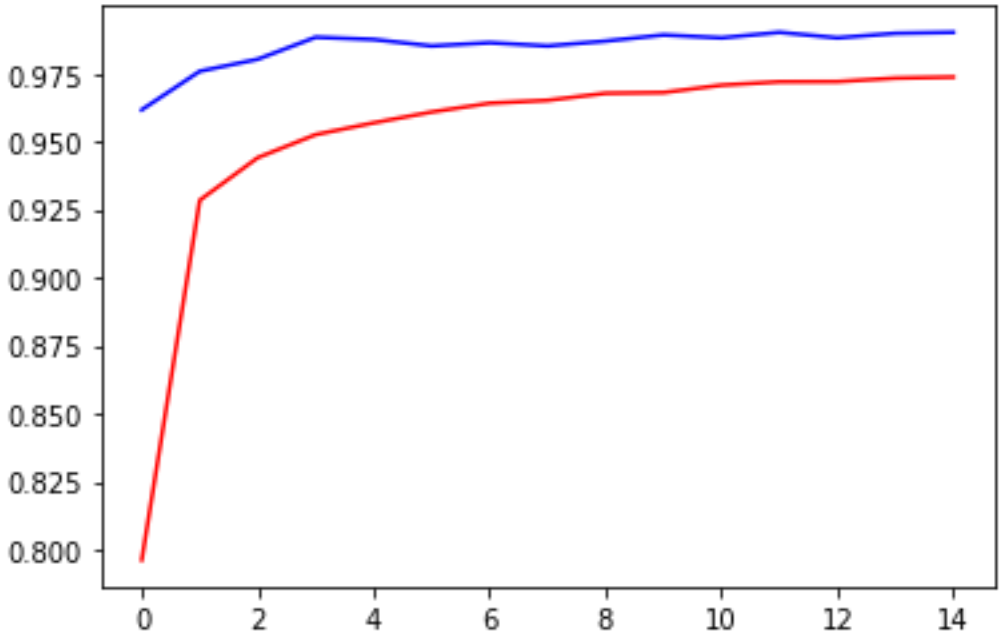


c) The dropout layer is used to prevent overfitting in the model by randomly dropping out some neurons during training. To investigate the effect of the dropout layer on the test accuracy, we can train the original model without the dropout layer and compare its test accuracy with the model that includes the dropout layer.
Here is the modified code without the dropout layer:

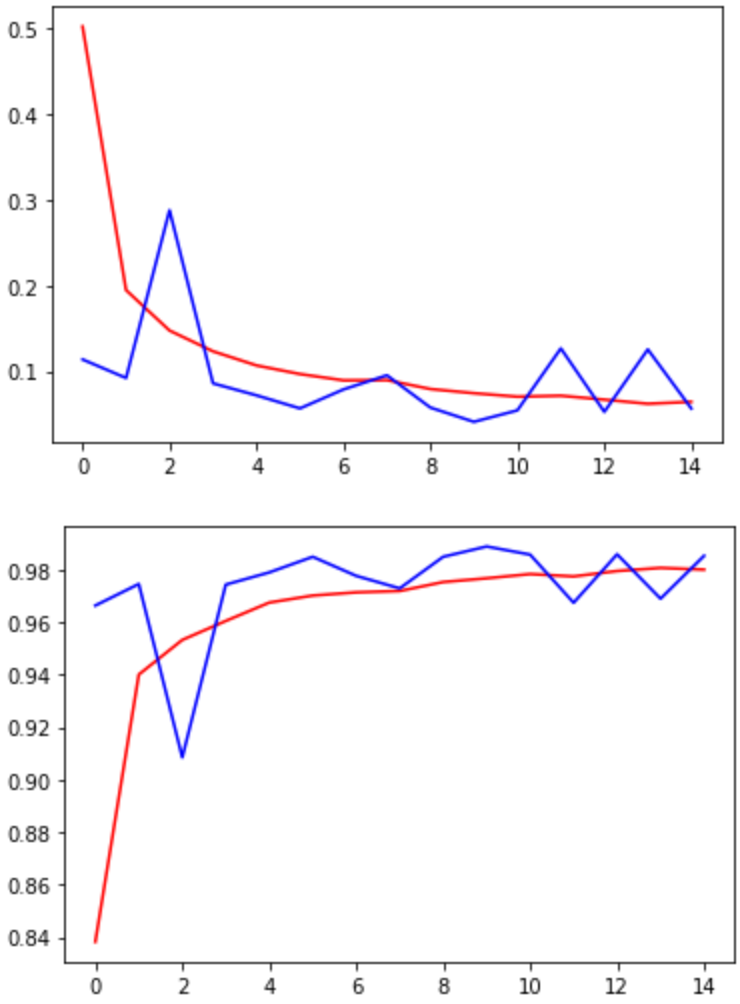
```
#Preparing a CNN model architecture without dropout layer
model = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, (5,5), activation='relu', kernel_initializer='he_uniform',input_shape=(28, 28, 1)),
            tf.keras.layers.MaxPooling2D(2, 2),
            tf.keras.layers.Conv2D(64, (5,5), activation='relu'),
            tf.keras.layers.MaxPooling2D(2, 2),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.Dense(10, activation='softmax')
    ])
[39]  ✓ 0.2s
```

Original:


0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10
12
14


0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
2
4
6
8
10
12
14

After:

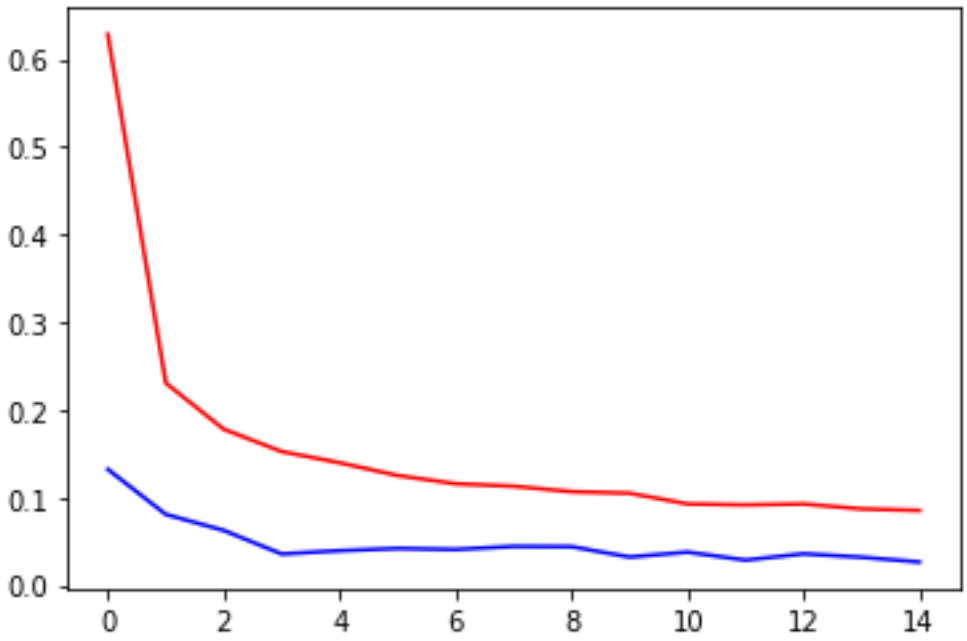


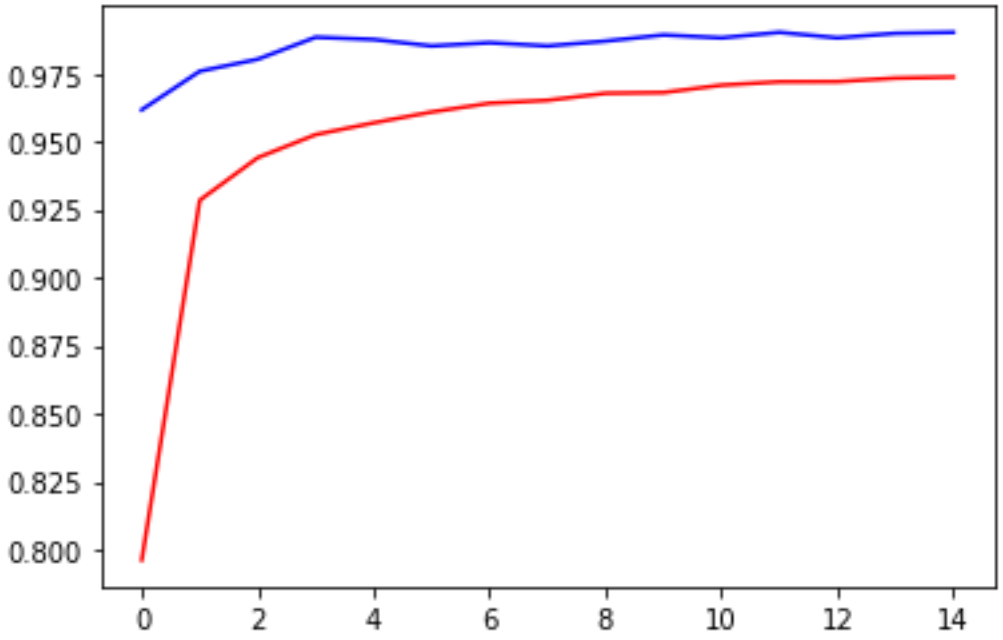
d) Train the model with a learning rate of 0.2:

```
#Compiling the model
model.compile(optimizer= tf.keras.optimizers.SGD(lr=0.2, momentum=0.9), loss='categorical_crossentropy', metrics=['accuracy'])
```

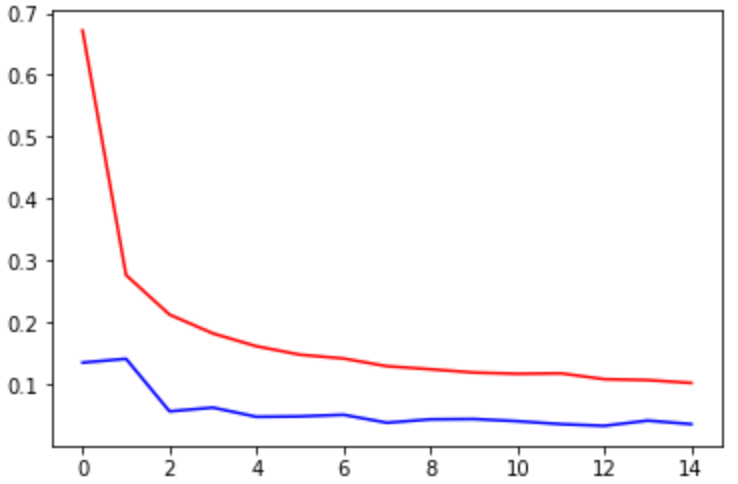
[111] ✓ 0.0s Python

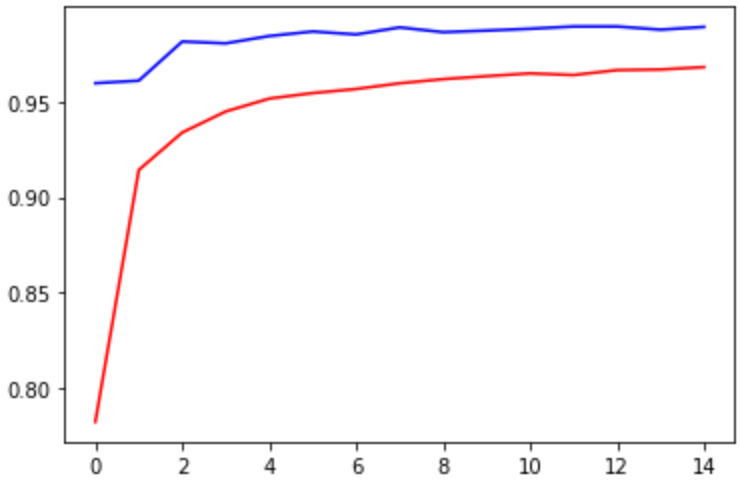
Original:


0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10
12
14


0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
2
4
6
8
10
12
14

After:

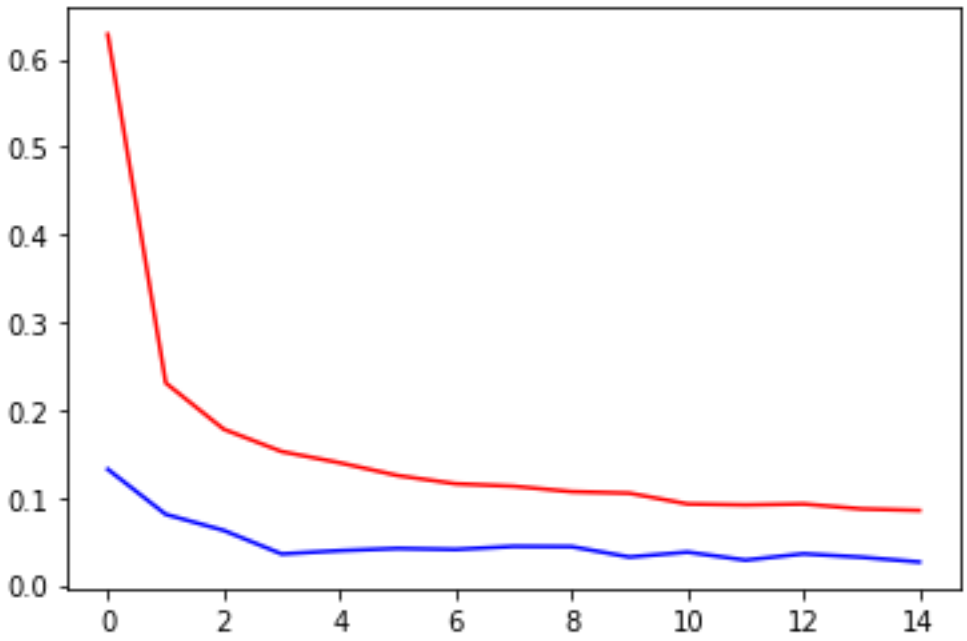


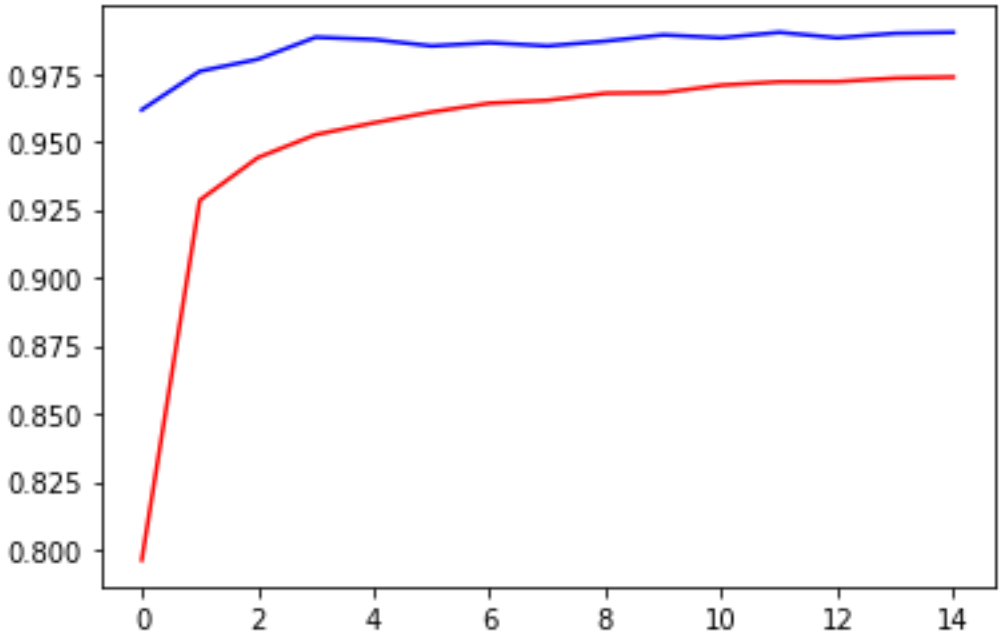


e) In the original code, data augmentation is applied during training to prevent overfitting of the model. Without data augmentation, the model may overfit to the training data and perform poorly on the test set. To evaluate the effect of data augmentation on the performance of the model, we can retrain the model without data augmentation and compare the test accuracy with the original model.
To train the model without data augmentation, we can simply remove the datagen argument from the fit_generator() method and use the fit() method instead. Here is the modified code:

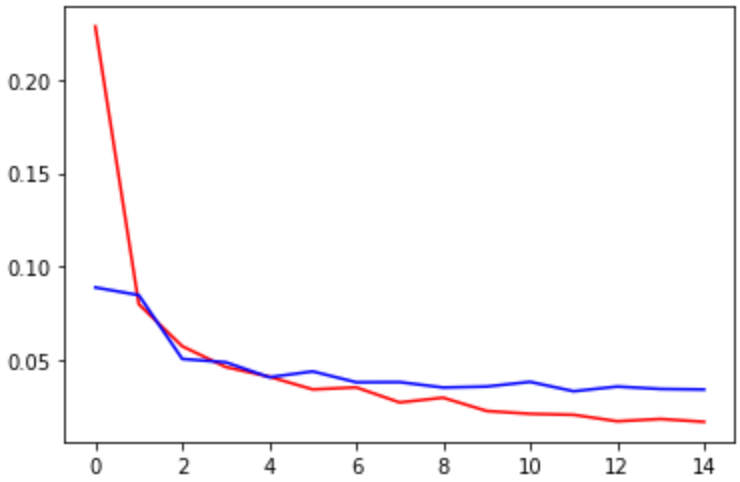
```
#Training the model without data augmentation
history = model.fit(X_train, Y_train, batch_size=64, epochs=15, validation_data=(X_val, Y_val))
```
29.6s Python

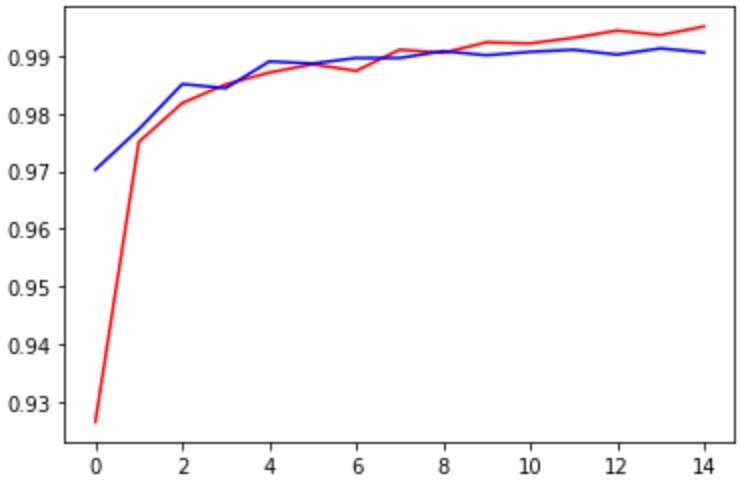
Original:


0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10
12
14


0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
2
4
6
8
10
12
14

After:


0.20
0.15
0.10
0.05
0
2
4
6
8
10
12
14


0.99
0.98
0.97
0.96
0.95
0.94
0.93
0
2
4
6
8
10
12
14