Computer Vision Assignment IV

Muhammed Görkem KOLA

January 5, 2024

Abstract

This is the Report of Last Assignment of Computer Vision Laboratory (AIN433).

Contents

1 Introduction and Terminologies

1.1 Overview

In this Project Assignment, we are asked to implement a Convolutional Neural Network, Fine Tune VGG16 on MIT indoor scenes dataset, and Object Detection on Racoon Dataset using ResNet-18 using TensorFlow.

The purpose of implementing these models, understanding the nature of Deep Learning and Experimenting with different hyperparameters, such as batch size, learning rate, and dropout rates.

1.2 Convolutional Neural Networks

Convolutional Neural Networks gained popularity after 2010 with AlexNet, then it widespread among all machine learning models. The idea behind this network is simple, instead of multiplying long matrices that doesn't contain local information, we can capture this local information with a kernel and it can reduce the computational requirements as it has less parameters.

There are several important parameters for these kind of networks:

- **In Channels:** Input channels show how many channels input tensor have (For example for an RGB image there are 3 channels.)
- **Out Channels:** The output channels show how many kernel we have.
- **Stride:** step size for each step CNN passed.
- **Padding:** Padding input frame with a fill value, mostly it is 0.
- **Pooling:** Selecting a value from a position that kernel shows.
- **Kernel Size:** Shows how wide a kernel.
- **Activation Function:** It makes model have non-linearity.

1.3 Chosen Classes

There are 15k images on MIT indoors Dataset, but this dataset is imbalanced so they are filtered using a 250 threshold and 15 of them selected.

Selected Classes and Number of instances:

- toystore - 347
- bedroom - 662
- pantry - 384
- kitchen - 734
- laundromat - 276
- livingroom - 706
- inside_subway - 457
- bookstore - 380
- warehouse - 506
- bar - 604
- restaurant - 513
- casino - 515
- airport_inside - 608
- winecellar - 269
- corridor - 346

1.4 Activation Functions

- **ReLU (Rectified Linear Unit)**

  ReLU is one of the most used activation function, it provides non-linearity to a model and reduces overfit risks. Its architecture is simple and effective. we can show it as a simple function, "max(x, 0)", x stands for the instance value.

- **Sigmoid**

  Sigmoid activation function is commonly used to make logistic regression. It compresses values between 0 and 1, and it also can be used in bounding box detection.

- **Softmax**

  Softmax activation function is commonly used in multi class classification, and also Transformers' Attention Mechanism to understand how much a value is important.

1.5 Loss Functions

- **Cross Entropy**

  Cross Entropy loss is mostly used in multi class classification as Categorical Cross Entropy loss or in binary classification as Binary Cross Entropy loss. It is easy to implement and effective so it is commonly used to train Neural Networks.

- **L2**

  It is a loss function to understand how a model made a mistake, it can be got by summation of square of differentiating True and Predicted labels.

- **Mean Squared Error**

  It is similar to L2 loss, the only difference is it takes average of Squared differences instead of summing them.

1.6 Adam Optimizer

Adam Optimizer is one of the mostly used optimizers, it provides an adaptive momentum and it can make easier and faster to backpropagate, and it is also useful not stucking in the local minimums.

1.7 Callback Functions

- **Early Stopping**

  Early Stopping is a callback function that when training a model, model is not going better for a decided time, it stops the training process.

- **Checkpoint**

  It checkpoints the best model according to a metric.

- **Tensorboard**

  It logs the training process to a logdir and it can be called using tensorboard.

2 PART 1 - Modeling and Training a CNN classifier from Scratch

In this section we are expected to implement a Convolutional Neural Network using TensorFlow and then Fine Tune a VGG16 with 3 different learning rate and 2 different batch sizes. The chosen learning rates are **0.01, 0.001, 0.0001**, and the chosen batch sizes are **16, 128**

2.1 Custom Model

Our CNN architecture has 3 Convolutional Layers and 2 Dense layers. First convolutional layer has (7, 7) kernel size and (2, 2) strides, that way it captures wider informations

and make the size of input smaller. The second and third convolutional layer has (3, 3) kernel Size and MaxPooling layers. And, first dense layer has 128 output neurons, and the other one has 15 (number of chosen classes) output neurons. ReLU is chosen activation function for our CNN architecture. It is shown that ReLU outperforms other activation functions when used in CNN's. Our choice of loss function is Categorical Cross Entropy, as it is a multiclass classification problem, it is more reliable. And, the choice of Optimizer is Adam Optimizer, because of its performance.

```
def create_custom_model(
        logit_size,
        input_shape=(128, 128, 3),
        activation_function='relu',
        p=.0):
    model = models.Sequential()

    model.add(layers.Conv2D(32, (7, 7), activation=
   activation_function, input_shape=input_shape, strides=(2, 2)))
    model.add(layers.BatchNormalization())

    model.add(layers.Conv2D(64, (3, 3), activation=
   activation_function))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(128, (3, 3), activation=
   activation_function))
    model.add(layers.BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.GlobalAveragePooling2D())

    model.add(layers.Dense(128, activation=activation_function))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(p))

```

```
    model.add(layers.Dense(logit_size, activation='softmax'))
```

Listing 1: Custom Model Creating Code

The implementation above is our custom model's creating code. We implemented it using BatchNormalization in various parts and dropout if wanted in top hidden layer. This way even though there are less number of parameters we achieved more generalizing.

2.2 VGG16 (Visual Geometry Group 16)

VGG16 consists of 13 Convolution and 3 Dense Layers. Convolution Layers are mostly consist (3x3) convolutional kernels. We did not include top layers (3 Dense Layers) while training because they have lots of parameters, so we decided to implement one hidden layer and one output layer instead of these top layers. The Hidden Dense Layer Consists 256 Neurons. And output consist 15 (number of chosen classes) neurons.

```
def get_VGG(
        logit_size,
        activation_function='relu',
        unfreeze_convs=False,
        p = .0
            ):
    base_model = VGG16(weights='imagenet',
                       include_top=False,
                       input_shape=(128, 128, 3))
    for layer in base_model.layers:
        layer.trainable = False

    if unfreeze_convs:
        for layer in list(reversed(base_model.layers))[:3]:
            layer.trainable = True

    model = models.Sequential()
    model.add(base_model)
    model.add(layers.Flatten())
```

```
    model.add(layers.Dense(256, activation=activation_function))
    model.add(layers.Dropout(p))
    model.add(layers.Dense(logit_size, activation='softmax'))

    return model
```

Listing 2: VGG model

2.3 Data Preprocessing

In prerprocessing, train and test txt files are updated so that there will be 15 large classes, images are converted to RGB format and resized to be shaped (128, 128, 3), and rescaled dividing by 255, and the labels are one hot encoded.

2.4 Training

In training part, first iterated over batch sizes, then datasets are created:

```
train_dataset = create_dataset(txt_path=train_paths_file,
                        batch_size=batch_size,
                        data_dir=data_dir)

test_dataset = create_dataset(txt_path=test_paths_file,
                            batch_size=batch_size,
                            data_dir=data_dir)
```

Listing 3: Datasets

then iterated over learning rates to get all the combinations, then learning rate schedular and optimizer created.

```
# After every 10k steps learning rate slightly decrease.
learning_rate_schedule = tf.keras.optimizers.schedules.
   ExponentialDecay(
    initial_learning_rate=learning_rate,
```

```
    decay_steps=10000,
    decay_rate=0.96
)

optimizer = tf.keras.optimizers.Adam(learning_rate=
   learning_rate_schedule)
```

Listing 4: Optimizer

Then train function called

```
train(
    model=model,
    dataset=train_dataset,
    optimizer=optimizer,
    metrics=metrics,
    epochs=100,
    checkpoint=checkpoint,
    log_dir=log_dir,
    hist_file=hist_file,
)
```

Listing 5: Optimizer

In this train function model compiled and callbacks are set

```
model.compile(optimizer=optimizer,
              loss=loss_fn,
              metrics=metrics)

early_stopping = EarlyStopping(
    monitor='val_loss',
    mode='min',
    patience=30,
    restore_best_weights=True
)

```

```
checkpoint_callback = ModelCheckpoint(
    filepath=checkpoint,
    save_best_only=True
)

tensorboard_callback = TensorBoard(
    log_dir=log_dir
)

all_callbacks = [early_stopping, checkpoint_callback,
   tensorboard_callback]
```

Listing 6: Optimizer

And then split data into train and validation

```
split = int(0.8 * len(dataset))
train_dataset = dataset.take(split)
val_dataset = dataset.skip(split)
```

Listing 7: Optimizer

then train the model

```
with tf.device('/GPU:0' if tf.config.list_physical_devices('GPU')
    else '/CPU:0'):
    # Train the model
    history = model.fit(train_dataset,
                epochs=epochs,
                validation_data=val_dataset,
                validation_steps=len(val_dataset),
                callbacks=all_callbacks)
```

Listing 8: Optimizer

then test the model after training and chose best weights according to validation

```
test(
    model=model,
```

```
    dataset=test_dataset,
    test_file=f'src/results/part1_{model_name}_{p}_{learning_rate}_{batch_size}.txt'
)
```

Listing 9: Optimizer

2.5 Results

HyperParameter Tuning

After training, we run the optimal_hyper_parameters function in src/parts/part1.py file and get these results: Optimal Hyperparameters for VGG is:

- Learning Rate: 0.0001
- Batch Size: 16

Optimal Hyperparameters for Custom Model is:

- Learning Rate: 0.01
- Batch Size: 128

This results show us the best models with respect to test cases.

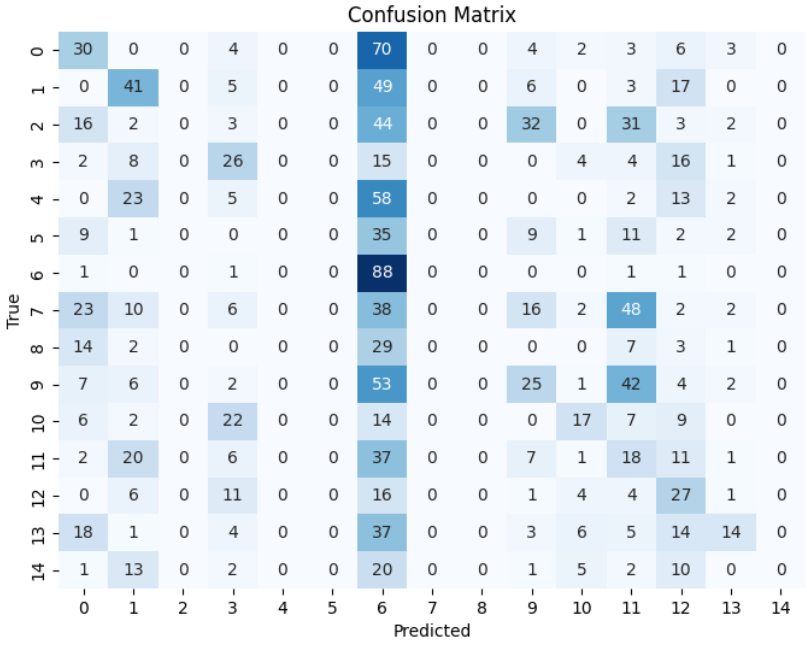

Figure 1: Custom Confusion Matrix using Optimal Hyperparameters

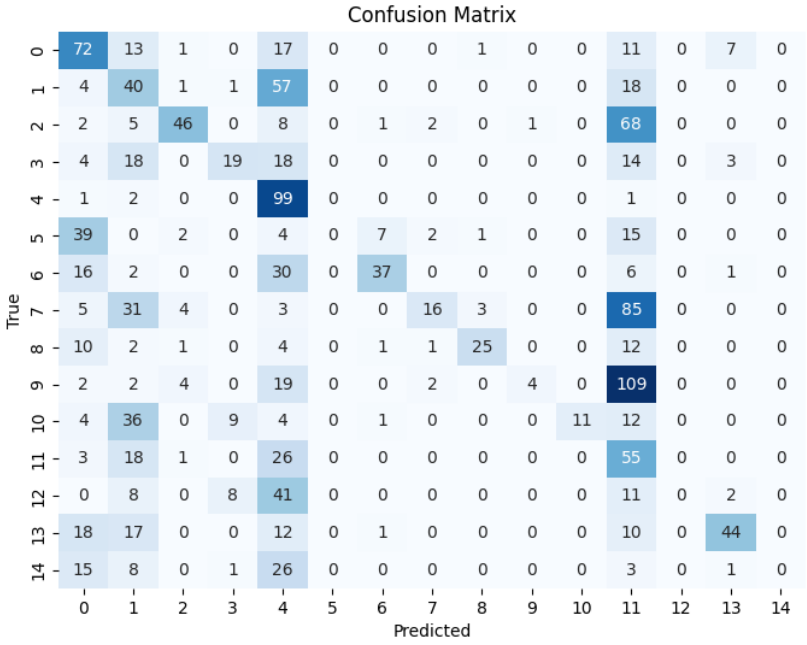

Figure 2: VGG Confusion Matrix using Optimal Hyperparameters

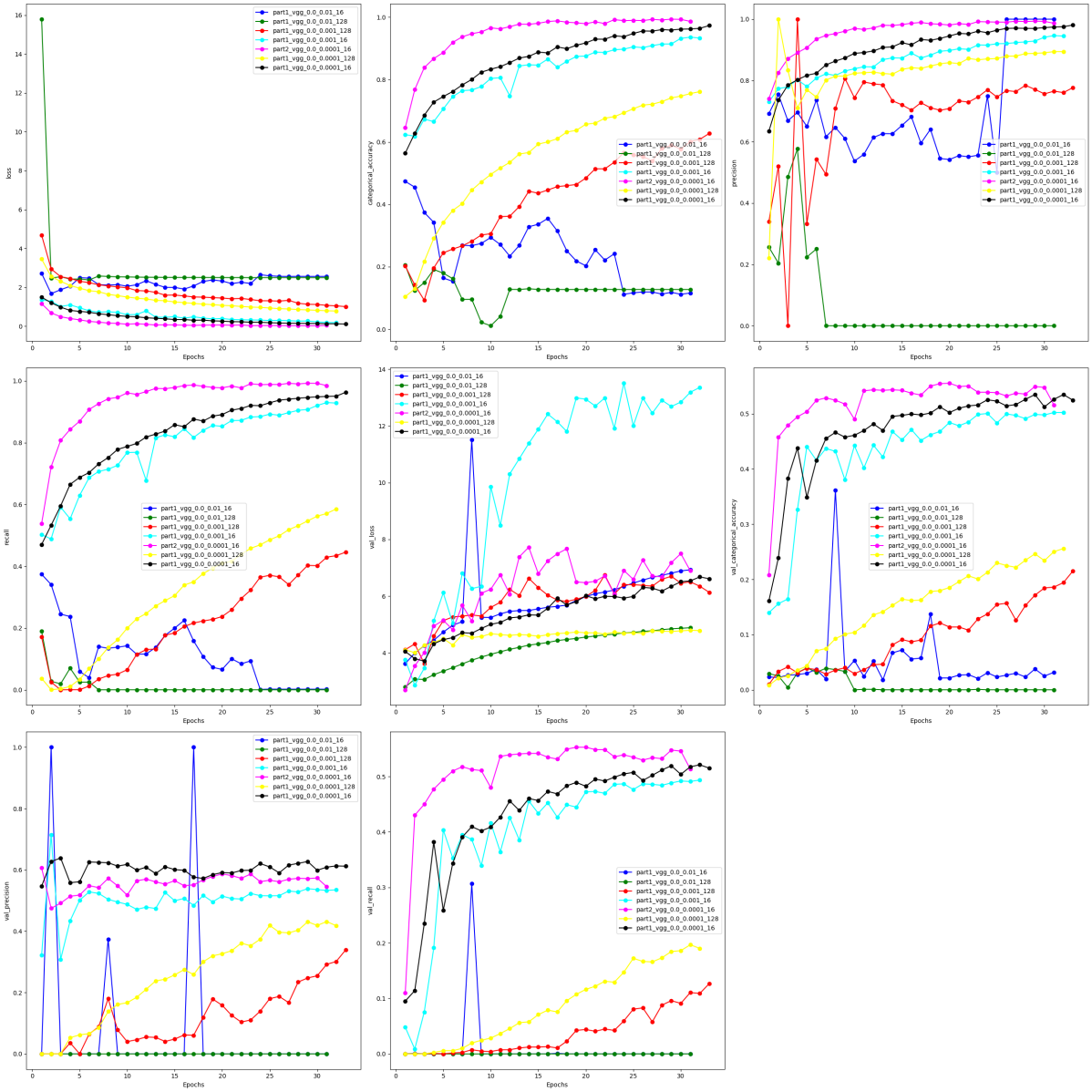

Figure 3: Custom plots related to Loss, Accuracy, Precision, Recall

Figure 4: Custom plots related to Loss, Accuracy, Precision, Recall

It seems the Optimal Hyperparameters and The plot above are not related. Its because optimal hyperparameters chosen by test data, probably they look different so test data may not be enough. Or, there is another probability that we have made a mistake while training or testing but it is unlikely.

Dropout comparing

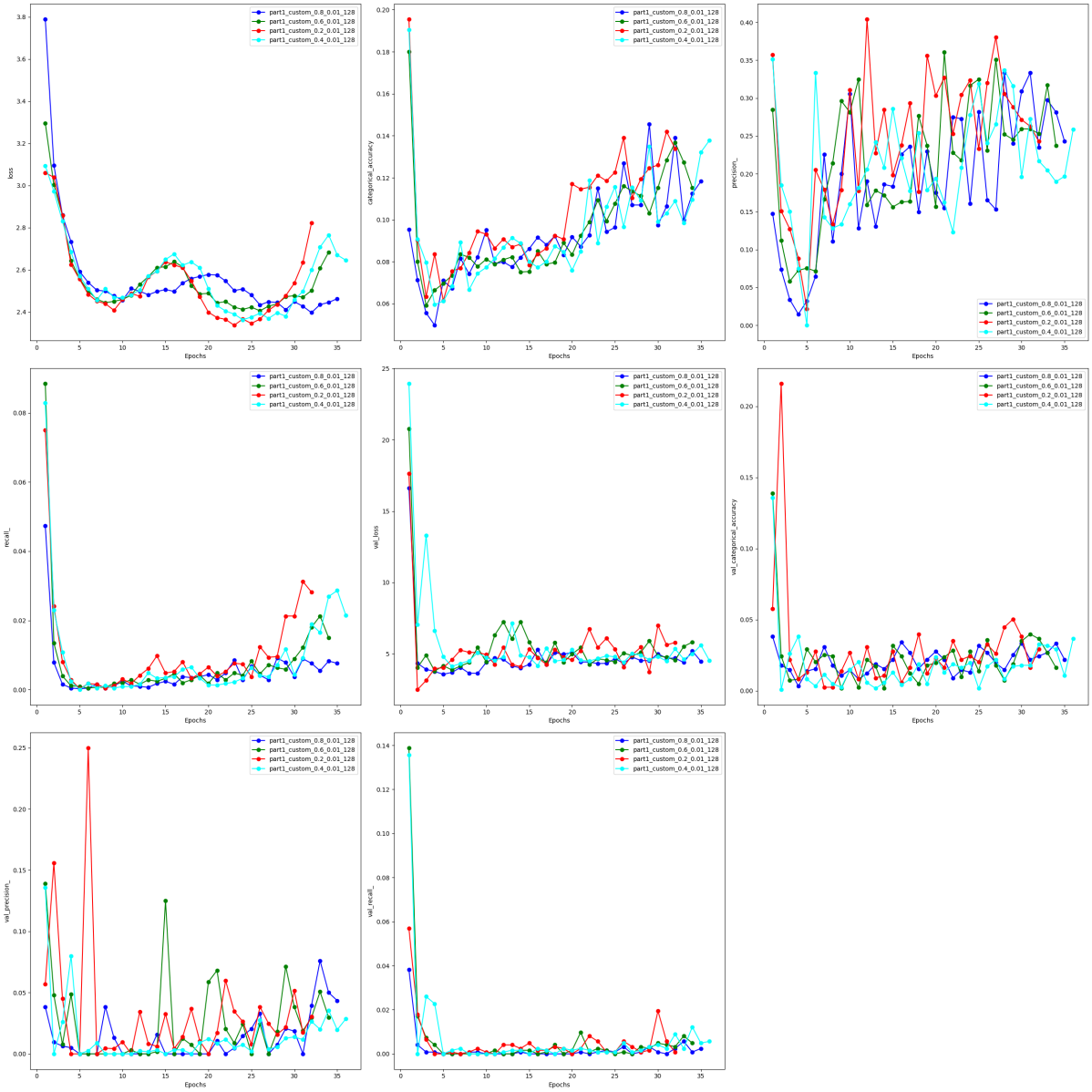

Figure 5: Custom plots related to Loss, Accuracy, Precision, Recall

Even though dropout rates change the model give similar results for all. But we still can see dropout 0.2 is a bit better than the others in some cases. But, these graphs above are not enough to say anything about how dropout affects the results.

3 PART 2 - Transfer Learning with CNNs

In this part we are expected to compare just top layers fine tuned VGG16 and the last 2 convolutional layers and top layers fine tuned VGG16.

As VGG models fine tuned in Part1, we did not train it again, just trained the two last conv layers unfreezed version. This way we can understand how it affects the results with less computational power.

3.1 Results

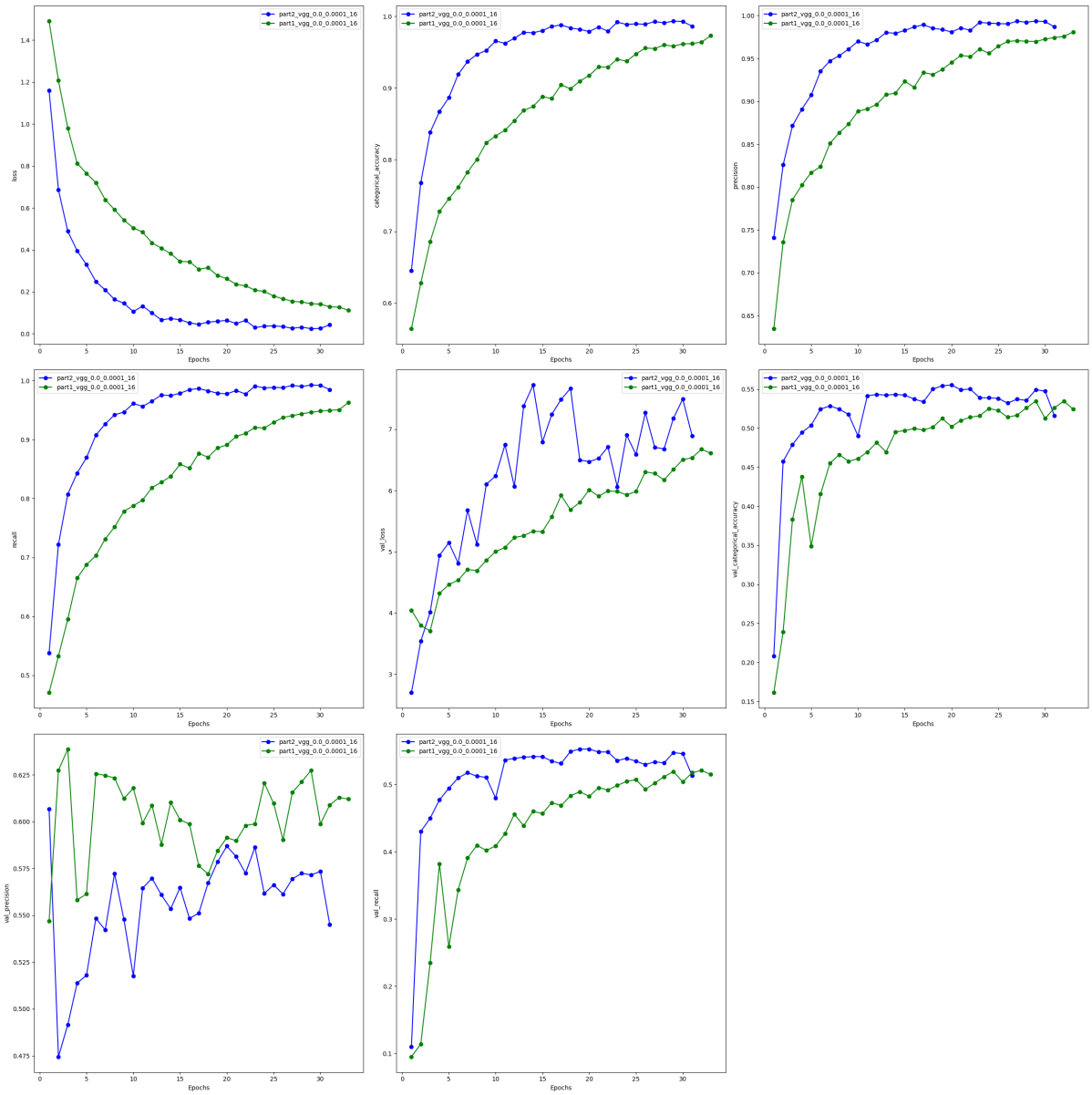

Figure 6: Freezed conv layers (Part1) - Unfreezed conv layers (Part2)

We can see how superior the unfreezed conv layers, as they capture the local information unfreezing them can make model understand what are in the images more accurately and as a result convolutional layers outperform the non-convolutional ones.