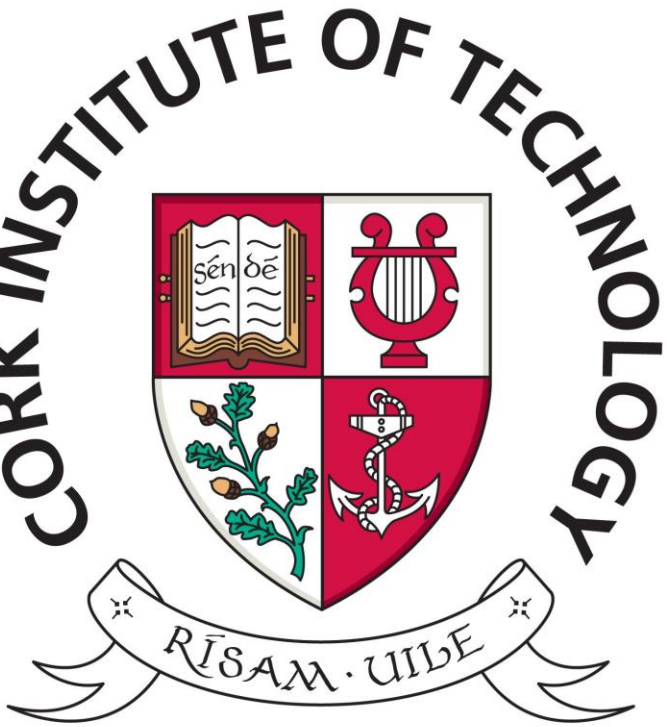


Mike Leske
R00183658

COMP9074 – Deep Learning
Assignment 2 – Convolutional Neural Networks

Overview

This assignment led us to investigate different approaches for getting classification results for the difficult Flowers image dataset. It is difficult as it contains 17 classes of flowers with only 1370 images including high intra-class variations.

As a preview for Part A and Part B of the assignment the following classification results (accuracy) have been achieved:

Part	Approach	Best Accuracy
A(i)	“Simple” Baseline CNN	0.582
A(i)	Deeper “Simple” CNNs	0.600
A(i)	Deeper “Simple” CNNs /w Data Augmentation	0.774
A(ii)	CNN Ensemble /w Data Augmentation	0.835
B(i)	Feature Extraction: DenseNet121 + LogisticRegression MobileNetV2 + PCA + LogisticRegression	0.926
B(ii)	Fine-Tuning: VGG19 /w Data Augmentation	0.956
B(ii)	Fine-Tuning: Fine-Tuned Ensemble /w Data Augmentation	0.971

Part C covers Capsule Networks.

1 Part A – Convolutional Neural Networks

1.1 CNNs and Data Augmentation

In this part of the assignment we implement a basic CNN network, extend it with additional convolutional layers and finally perform data augmentation to increase diversification of the training data. In total 12 experiments were executed in order to analyse the CNN performance on the provided “flowers” dataset (17 classes).

Like in Assignment 1 I created a function to dynamically build, compile and return the Keras model. For all experiments in this section layers like BatchNormalization, SpatialDropout or Dropout are not taken into consideration.

cnn_model function

```
def cnn_model(width, height, depth, classes, convBlocks=[]):

    inputShape = (width, height, depth)

    # Instantiate a Sequential model
    model = tf.keras.models.Sequential()

    # Add an InputLayer to allow for dynamic neural assembly
    model.add(tf.keras.layers.InputLayer(input_shape=inputShape))

    for convBlock in convBlocks:
        for convFilter in convBlock:
            model.add(tf.keras.layers.Conv2D(convFilter, (3, 3), padding='same', activation='relu'))
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(1024, activation='relu'))
    model.add(tf.keras.layers.Dense(classes, activation='softmax'))

    opt = tf.keras.optimizers.SGD(learning_rate=0.01)

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

    # Print the model summary
    model.summary()

    # return the constructed network architecture
    return model
```

Every model starts with a separate InputLayer. This allows to then flexibly build architectures without bothering about the `input_shape` parameter.
The core of the CNN builder is the `convBlocks` object, which provides the CNN architecture as nested lists. [[64]] creates a single convolutional layer with 64 filters, followed by a MaxPooling layer. [[64], [128]] would create a convolutional layer with 64 filter, followed by a MaxPooling layer, followed by a convolutional layer with 128 filters, followed by another MaxPooling layer. [[64, 64]] would create 2 subsequent convolutional layers with 64 filters each, followed by a MaxPooling layer.
To each model a 1024 neuron wide Dense layer with ReLU activation is added. Each model is completed with a Softmax layer.
As per assignment brief SDG optimizer with `learning_rate=0.01` is used. Because we’re dealing with 17 classes, the loss is set to `'sparse_categorical_crossentropy'`.

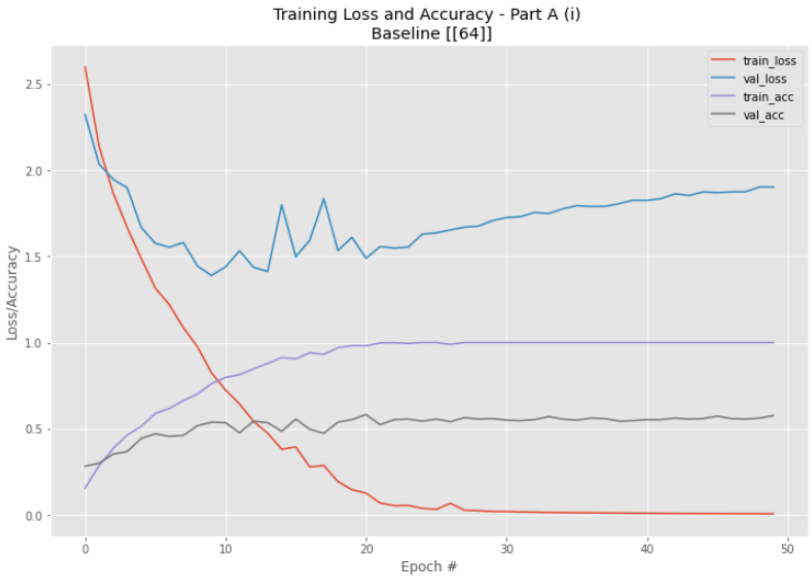
Experiments without data augmentation were trained with batch_size 32 for 50 epochs.
Experiments with data augmentation were trained with batch_size 32 for 100 epochs.

Baseline CNN

```
'''
Baseline model

----------------------------
|           Input          |
----------------------------
|   Conv2D  - 64 Filter    |
----------------------------
|        MaxPooling        |
----------------------------
|          Flatten         |
----------------------------
|        Dense (1024)      |
----------------------------
|        Softmax (17)      |
----------------------------

'''

convBlocks = [[64]]
model = cnn_model(width, height, depth, classes, convBlocks)
history = model.fit(trainX, trainY, batch_size=BATCH_SIZE, epochs=EPOCHS, validation_data=(valX, valY))
print('\n\nMax validation accuracy:', max(history.history["val_accuracy"]))
plot_loss(history, EPOCHS, 'Part A (i) \n Baseline ' + str(convBlocks))
```

The baseline model was built with a single convolutional layer with 64 filters, followed by a MaxPooling layer, followed by a single Dense layer and the final Softmax layer. During the 50 epochs the maximum validation accuracy seen was 58.2%. The model started to clearly overfit around epoch 5.

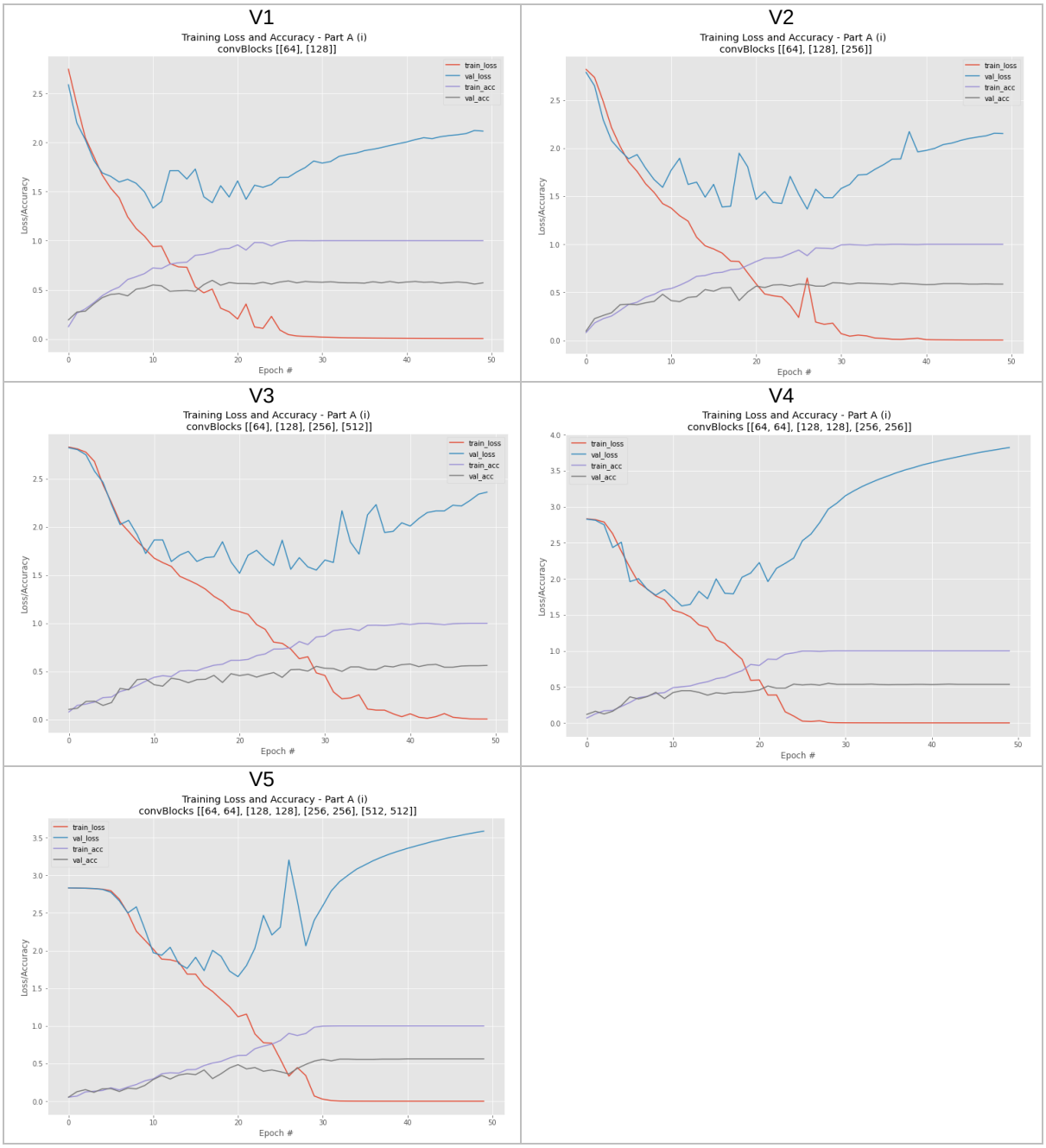
```
Max validation accuracy: 0.5823529362678528
```



Next, 5 different CNN variants, called V1 to V5, were built and trained.

Model	Architecture (convBlocks)	Train acc + loss (after 50 epochs)	Val acc + loss (after 50 epochs)	Best val accuracy
Base	[[64]]	1.0000, 0.0065	0.5765, 1.9034	0.582
V1	[[64], [128]]	1.0000, 0.0041	0.5706, 2.1156	0.597
V2	[[64], [128], [256]]	1.0000, 0.0035	0.5853, 2.1514	0.600
V3	[[64], [128], [256], [512]]	1.0000, 0.0054	0.5618, 2.3606	0.576
V4	[[64, 64], [128, 128], [256, 256]]	1.0000, 3.5177e-04	0.5353, 3.8197	0.550
V5	[[64, 64], [128, 128], [256, 256], [512, 512]]	1.0000, 2.8613e-04	0.5618, 3.5857	0.562

It becomes apparent, that the increased capacity of the CNNs are not generally improving the predictive accuracy of the models. All models V1 to V5 overfit on the training data and the more complex the model is, the larger the validation loss gets after 50 epochs.



Despite their different neural network capacity all CNN variants show very similar overfitting behaviour like the baseline model. On a general note, the deeper the neural network was, the longer it took until overfitting became apparent, e.g. in variant V5 train_acc and val_acc start to diverge around epoch 15. For each variant the train_loss approaches 0 indicating the neural networks are able to completely fit to the training data, which adversely manifests in an increasing val_loss curve.

Hence, despite variants V1 and V2 result in a slight validation accuracy improvement over the simple baseline network the results do not justify the additional computational complexity for the given dataset.

Next, data augmentation was implemented on variants V3, V4 and V5 as these most complex neural networks were expected to benefit most from more variation in the training data. When applied to image datasets, data augmentation supports the CNN to learn location-invariant features. This is done by a number of distortions that are applied to each mini-batch of images during the training process. Especially for datasets with a relatively small number of images this largely increases the variation of images the CNN is trained on – overall and for every single class. Typical augmentations for images include: rotation, zoom, brightness, shifts, and flips.

Variants V3-V5 were trained with the following image augmentation parameters:

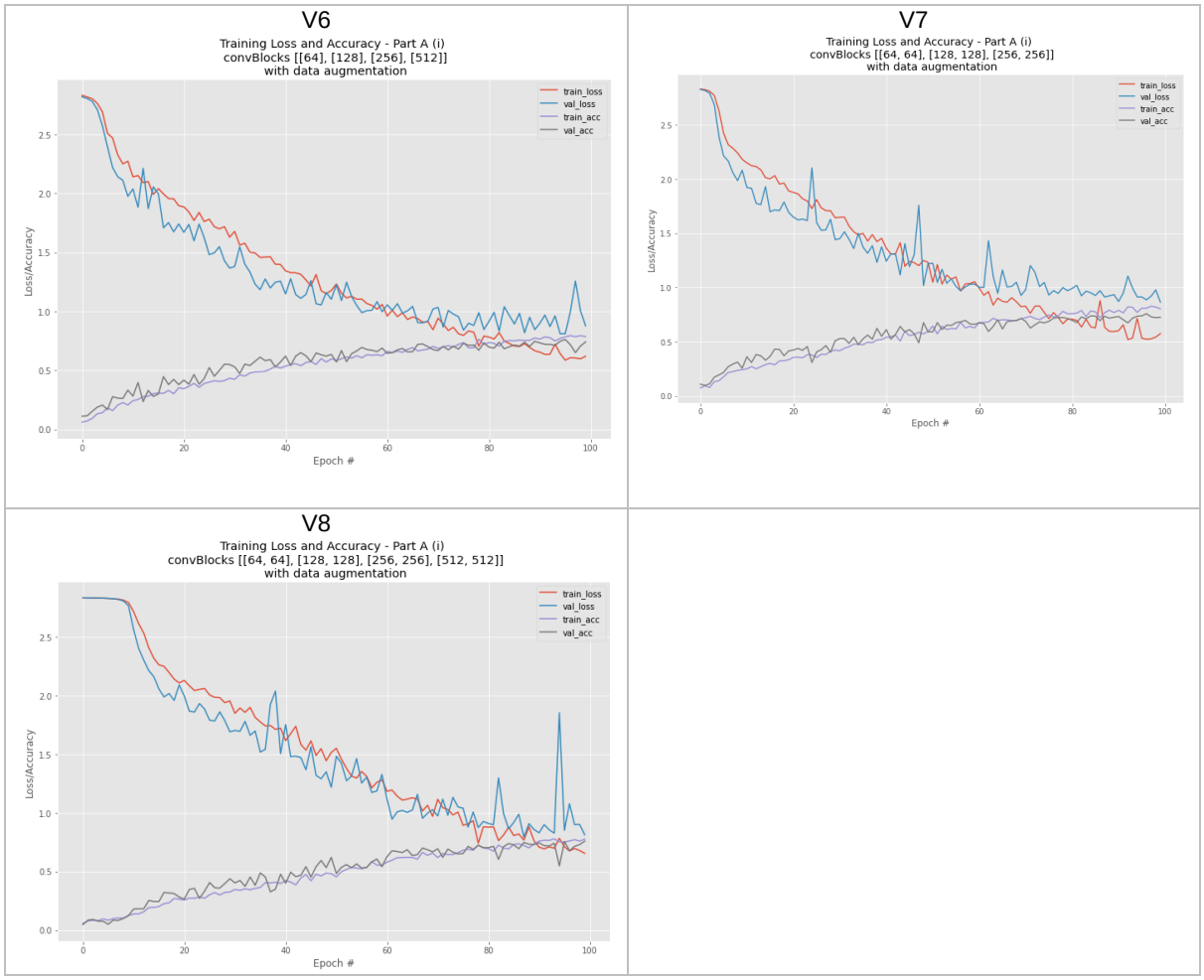
Image data augmentation

```
data_gen_args = dict(
        rotation_range=30,
        zoom_range=0.2,
        shear_range=0.2,
        width_shift_range=0.2,
        height_shift_range=0.2,
        horizontal_flip=True,
        #vertical_flip=True,
    )
```

Due to the nature of the dataset, vertical_flip was left at its default False. Flowers being upside down are not expected during the test (validation) predictions and they might result in the CNN learning unnecessary concepts.

Applying data augmentation resulted in variants V6 to V8:

Model	Architecture (convBlocks)	Train acc + loss (after 100 epochs)	Val acc + loss (after 100 epochs)	Best val accuracy
V6	[[64], [128], [256], [512]]	0.7854, 0.6195	0.7412, 0.8773	0.762
V7	[[64, 64], [128, 128], [256, 256]]	0.8036, 0.5750	0.7265, 0.8648	0.759
V8	[[64, 64], [128, 128], [256, 256], [512, 512]]	0.7753, 0.6551	0.7588, 0.8136	0.759



Both, the absolute accuracy and loss values as well as the training history diagrams clearly demonstrate the regularizing effect data augmentation has on the training process:

- Best validation accuracy improved to ~76%
- Extremely low overfitting in V8 (validation loss becomes “expressive” after 80 epochs)
- V7 shows medium overfitting starting from epoch 60
- Models are far from 100% training accuracy due to constantly changing training data, hence the resulting training loss helps to further improve the neural network weights

The curves for V6 to V8 also clearly show the impact of data augmentation on the training data, which constantly changes throughout the training process. The training accuracy curves now became spiky as each epoch was presented with unseen images. Overall, this training variation drastically helped the networks to generalize better and ultimately improve predictive quality on the validation data.

The chosen data augmentation parameters were expected to give good results for the flowers dataset. However, 4 additional data augmentation experiments were executed to document the impact of either leaving out some options or using higher/lower values. For the sake of computational cost these 4 additional experiments were conducted against architecture V3.

- V9 - like V6, but no horizontal_flip
- V10 - like V6, but no horizontal_flip and no shift (width & height)
- V11 - like V6, but all values 0.1 and rotation=20
- V12 - like V6, but all values 0.3 and rotation=45

Model	Architecture (convBlocks)	Train acc + loss (after 100 epochs)	Val acc + loss (after 100 epochs)	Best val accuracy
V9	[[64], [128], [256], [512]]	0.8553, 0.4182	0.6912, 1.0469	0.741
V10	[[64], [128], [256], [512]]	0.9808, 0.0599	0.7147, 1.3329	0.724
V11	[[64], [128], [256], [512]]	0.9271, 0.2087	0.7353, 1.1156	0.774
V12	[[64], [128], [256], [512]]	0.7176, 0.8375	0.7000, 0.9600	0.735

Findings for V9 to V12:

- As expected V9 and V10 show stronger overfitting due to less variation within the training data
- V9 shows noticeable overfitting starting epoch ~40
- V10 shows noticeable overfitting starting epoch ~30
- The small scale of the data augmentation parameters in V11 lead to larger overfitting than seen in V6 due to smaller amount of data variation
- Only extremely small overfitting is seen in V12 towards epochs 90-100. The network could potentially train for another 20 epochs and then level at ~75% validation accuracy.

The ~75-77% validation accuracy achieved with data augmentation can now be treated as “target-to-beat” for CNN ensembles and transfer learning.

1.2 CNN Ensembles

A CNN Ensemble is a collection of multiple CNN classifiers. Once fully trained, the average predictions per class can be calculated to – hopefully – get a better prediction than those of the individual ensemble members. It is imperative to introduce as much variability into the CNN ensemble training process as possible in order to let each CNN member learn slightly different features. If all ensemble learners have the same structure and are trained on the same data, it is very likely that they will make the same predictions.

As CNNs with Softmax layers provide probabilities across the possible classes, CNN ensembles do not rely on majority voting, but can use the detailed information for the ensemble prediction, e.g. calculate the numerical average for all classes per sample. This way a single learner with a very strong prediction can outperform the majority voting.

In my approach I introduced variability mainly by 2 design choices:

1. Build 10 different CNN learners
2. Train each learner with different data augmentation parameters

For creating 10 distinct CNN architectures I re-used the flexible CNN builder developed in Part A (i).

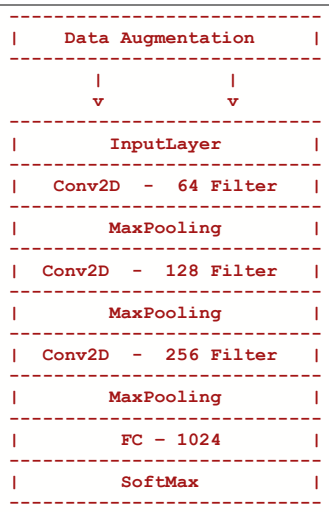
Randomize CNN Creation

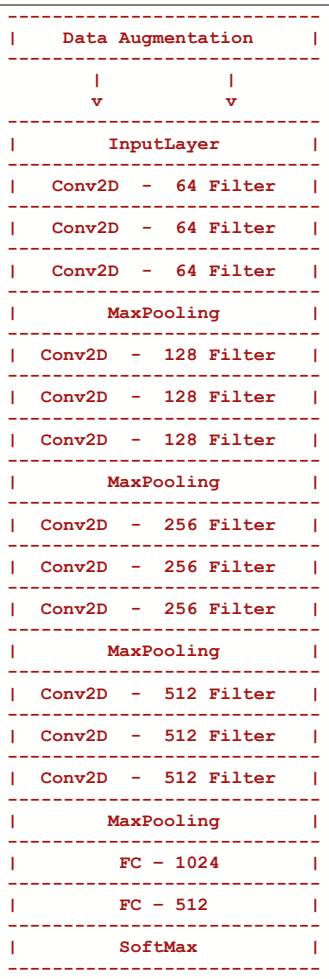
```
    def build(self):

        filters_sizes = {0:  64, 1: 128, 2: 256, 3: 512}
        dense_layers = {0: [1024, 512], 1: [1024]}

        # Instantiate a Sequential model
        model = tf.keras.models.Sequential()

        # Add an InputLayer to allow for dynamic neural assembly
        model.add(tf.keras.layers.InputLayer(input_shape=self.inputShape))

        for convBlock in range(np.random.choice([3, 4])):
            filters = filters_sizes[convBlock]
            conv = np.random.choice([3, 5])
            for _ in range(np.random.choice([1, 2, 3])):
                model.add(tf.keras.layers.Conv2D(filters, (conv, conv), padding='same', activation='relu
'))

            model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

        model.add(tf.keras.layers.Flatten())

        for size in dense_layers[np.random.choice([0, 1])]:
            model.add(tf.keras.layers.Dense(size, activation='relu'))

        model.add(tf.keras.layers.Dense(self.classes, activation='softmax'))

        <…>

        # return the constructed network architecture
        return model
```

The build() function generates varying CNN networks. Variation is achieved by letting randomness decide

1. Number of convolutional blocks (Conv2D + MaxPooling)
2. Number of Conv2D Layers part of a single conv block
3. Convolution size
4. Number of Dense Layers following the convolution blocks

The potentially most different CNN architectures to be created look as follows:





In addition to the distinct CNN architectures the data augmentation process was randomized. This way each learner was trained with distinct image data.

Randomize Data Augmentation

```
    def data_augmentation(self):
        data_gen_args = dict(
                rotation_range=np.random.randint(20, 75),
                zoom_range=np.random.uniform(0.1, 0.5),
                shear_range=np.random.uniform(0.1, 0.5),
                width_shift_range=np.random.uniform(0.1, 0.5),
                height_shift_range=np.random.uniform(0.1, 0.5),
                horizontal_flip=True
            )

        train_gen = tf.keras.preprocessing.image.ImageDataGenerator(**data_gen_args)
        train_generator = train_gen.flow(self.trainX, self.trainY, batch_size=self.BATCH_SIZE)

        print('\nData augmentation with the following parameters:')
        pp.pprint(data_gen_args)

        return train_generator
```

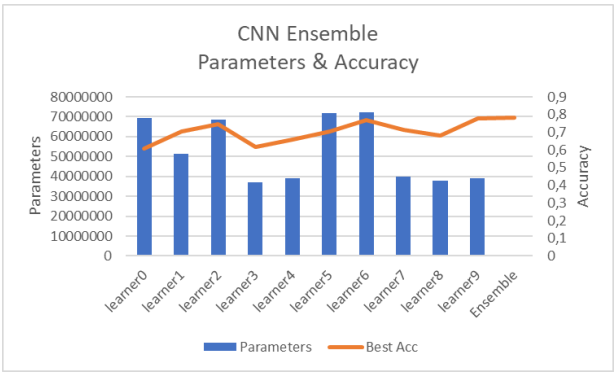
The ensemble was trained with a simple for-loop. Each loop created a new Learner object with random architecture and random data augmentation. After training each learner for 100 epochs, the checkpointed weights producing the lowest validation_loss / highest validation accuracy are loaded in order to create the final validation set predictions.

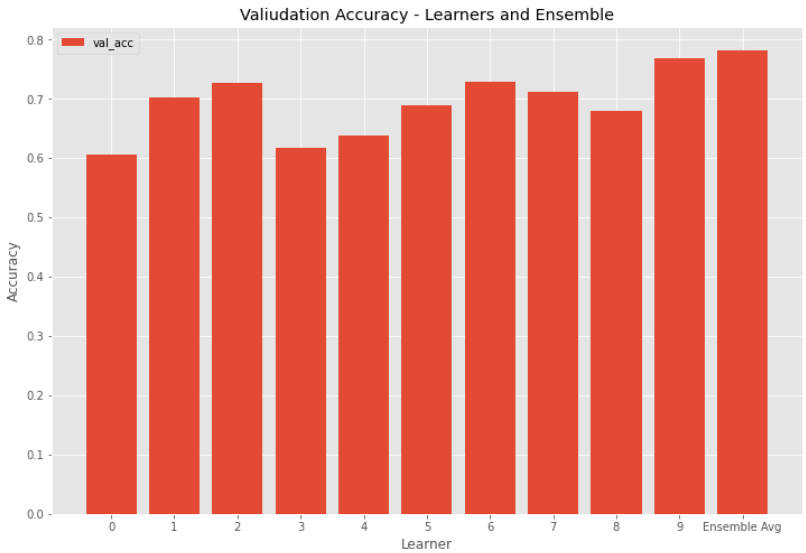
Note:
I noticed that checkpointing on highest validation accuracy resulted in a better ensemble performance than checkpointing on lowest validation loss. This was caused as for some learners the lowest validation loss did not lead to the highest classification accuracy. Both results will be presented.

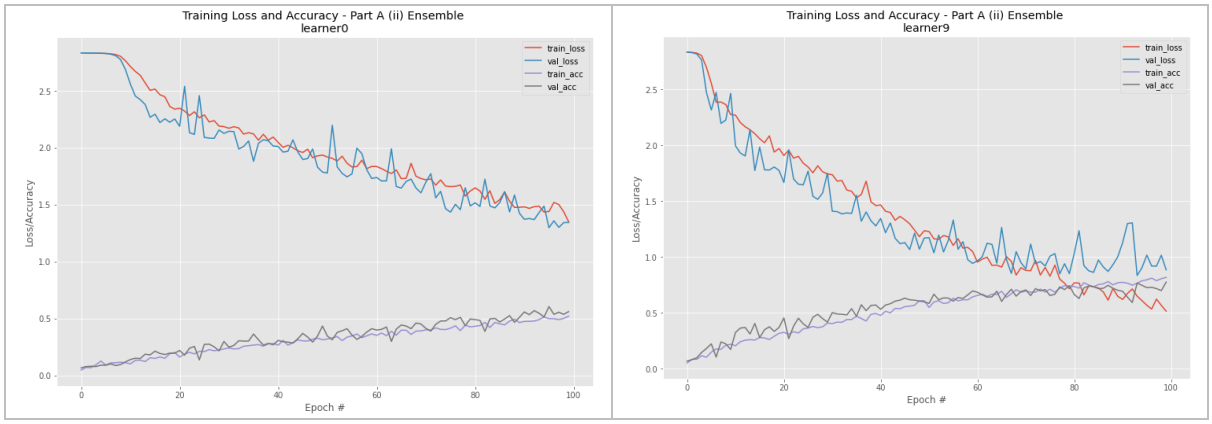
Ensemble 1 – Learner and checkpointing on <u>min validation loss</u>

The best accuracies of the ensemble members ranged between 0.606 and 0.776. The final ensemble accuracy was only slightly higher at 0.782.

Learner	Parameters	Best Acc
learner0	69415825	0,606
learner1	51282833	0,703
learner2	68341329	0,747
learner3	36893969	0,618
learner4	39249361	0,656
learner5	71946065	0,703
learner6	72052433	0,768
learner7	39854609	0,712
learner8	37884369	0,679
learner9	39059601	0,776
Ensemble		0,782







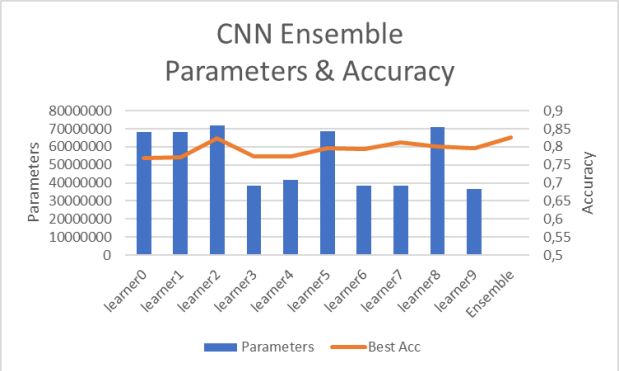
Learner0 was identified as the weakest learner in the ensemble. It shows very little overfitting, but also a slow learning. More training iterations would have been helpful. Learner9 was the strongest learner from the ensemble. Starting epoch 80 however, it shows signs overfitting as the validation loss flattens out.

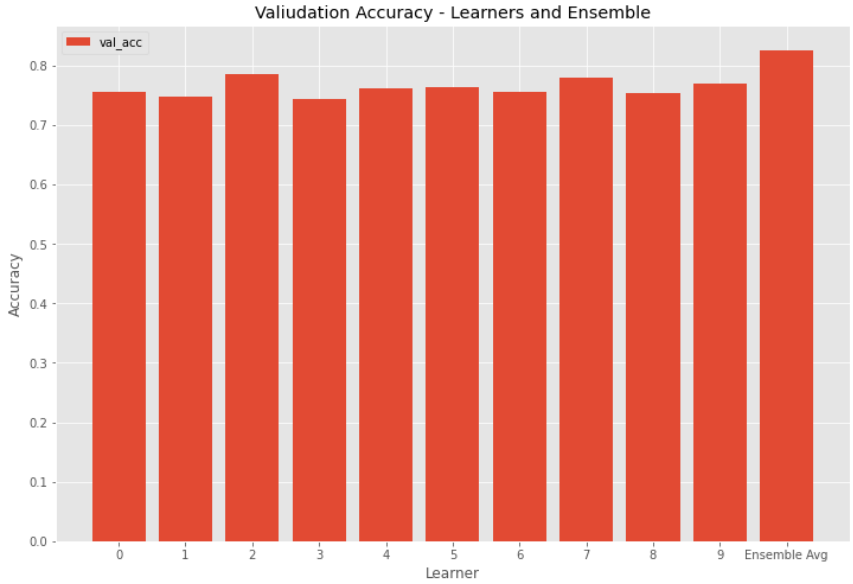
The slower training process of learner0 may be explained by it much larger number of trainable parameters (~ 70 million). Learner9 only had to train ~40 million parameters.

Ensemble 2 – Learner and checkpointing on max validation accuracy

To gain the best ensemble accuracy each learner was trained **for 200 epochs** now.
The best accuracies of the ensemble members ranged between 0.771 and 0.824. The final ensemble accuracy resulted in 0.826 (simple average).

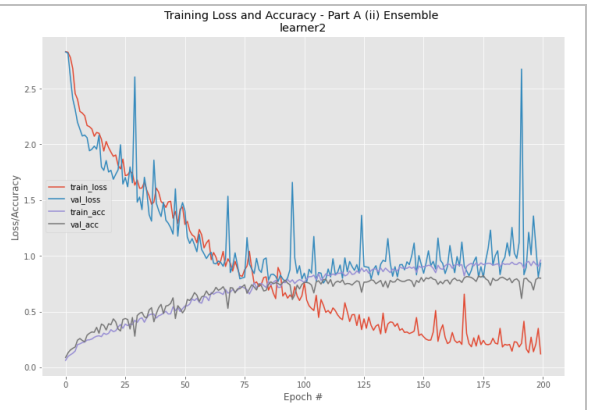
Learner	Parameters	Best Acc
learner0	68238865	0,768
learner1	68088209	0,771
learner2	71638801	0,824
learner3	38294929	0,774
learner4	41670801	0,774
learner5	68689169	0,797
learner6	38600465	0,794
learner7	38412049	0,812
learner8	70721041	0,8
learner9	36810577	0,797
Ensemble		0,826





Let’s again visualize the learning process of the weakest and strongest learner from the ensemble.

Data augmentation with the following parameters:
{'height_shift_range': 0.14158673592756013,
'horizontal_flip': True,
'rotation_range': 57,
'shear_range': 0.4186730962843158,
'width_shift_range': 0.49498429859087556,
'zoom_range': 0.30956238088887555}



Data augmentation with the following parameters:
{'height_shift_range': 0.10846063510517197,
'horizontal_flip': True,
'rotation_range': 63,
'shear_range': 0.18237653081548033,
'width_shift_range': 0.22104016292896175,
'zoom_range': 0.2269947262784604}

The training and validation curves over 200 epochs look roughly equal with noticeable overfitting after 100 epochs. Both learners had a comparable number of trainable parameters, hence similar chances of learning similar representations. Very likely the variance in predictive performance between these 2 learners was introduced by the random data augmentation.

Catastrophic Neural Breakdown

Throughout training the 10 ensemble members a very strong phenomenon was seen. After ~105 epochs the accuracies values dropped significantly, while the loss jumped to values only seen during the initial epochs of the training. Noticeably, the neural network recovered to pre-event performance within ~20 epoch, indicating that only parts of the network were affected by this catastrophic event. Another event happened around epoch 165.

The Deep Learning research domain knows this behaviour as **Catastrophic Forgetting**.

2 Part B – Transfer Learning

2.1 Feature Extraction

The domain of image classification using Convolutional Neural Networks has seen incredible advances over the past years. Processing capacities, especially provided by GPUs and TPUs, allowed the researchers to build and train extremely deep CNNs. DenseNet201 for example is 201 layers deep. A common problem is, though, training such networks with scarce access to GPUs.

The overall structure of CNNs includes

- several **convolutional layers** to extract features from images
- followed by 1 or more **dense layers** for classification.

After researches started to make available these extremely deep CNNs fully trained on the ImageNet dataset, the computational burden to deploy such a deep network was drastically lowered – as long as the classifications to be made were covered by the ImageNet classes.

Feature Extraction describes a very cost-efficient way, where the final dense and softmax layers are removed from the pre-trained CNNs so that the output of the last convolutional block becomes “visible”. Because all these networks have been trained on the ImageNet dataset the sequence of convolutions is expected to extract very meaningful image features when images are sent through their convolutional layers – no matter what category. These extracted features can then be fed into classical machine learning algorithms.

This way, within a few minutes very impressive results can be achieved, e.g. DenseNet121 and Logistic Regression were able to correctly predict **92.6%** of the Flowers validation set. However, this approach shows a strong tendency to overfit the classical machine learning algorithms on the training data (not augmented). Only SVC was less affected by this phenomenon.

Model	Features*	LogReg Train/Val Acc	SVC Train/Val Acc	SGD Train/Val Acc	Random Forest Train/Val Acc
VGG16	8192	1.0 / 0.876	0.899 / 0.844	1.0 / 0.812	1.0 / 0.862
VGG19	8192	1.0 / 0.859	0.877 / 0.812	0.992 / 0.812	1.0 / 0.835
InceptionV3	8192	1.0 / 0.865	0.957 / 0.841	1.0 / 0.832	1.0 / 0.824
DenseNet121	16384	1.0 / 0.926	0.979 / 0.897	1.0 / 0.914	1.0 / 0.903
DenseNet201	30720	1.0 / 0.865	0.949 / 0.820	1.0 / 0.835	1.0 / 0.806
ResNet152V2	32768	1.0 / 0.865	0.899 / 0.844	1.0 / 0.832	1.0 / 0.824
MobileNetV2	20480	1.0 / 0.924	0.999 / 0.912	1.0 / 0.903	1.0 / 0.894

* Features = number of neurons after last convolutional block

The selection of above base-models was driven by the following rationale:

- General popularity of VGG and InceptionV3
- Deepness of DenseNet201 and ResNet152V2
- Lower computational cost of DenseNet121 over DenseNet201
- MobileNetV2 due to their applicability to edge AI applications

The 4 classical machine learning algorithms were selected due to their general applicability to classification tasks. SVC suffered from long computation due to its sensitivity to feature size. A KNN classifier was omitted as well due to its susceptibility to high dimensional feature space.

Interestingly, the classical machines learning algorithms performed significantly worse with lower numbers of CNN feature output (e.g. 8192) and when very large features vectors were extracted from the CNNs (> 30000). This may just be a coincidence.

GradientBoostingClassifier and AdaBoostClassifier suffered from prohibitively long computation times and were excluded from the experiments.

No hyper-parameter optimization was executed for the secondary classifiers.

Overall, the feature extraction and classification process was split into 4 steps:

1. Download the pre-trained CNNs without final dense and softmax layers
2. Send training and validation data through the networks get extract features
3. Train secondary machine learning model with extracted feature
4. Get final predictions for extracted validation set features from secondary models

Sample Feature Extraction process

```
def get_nn_out(nn, trainX, valX):
    featuresTrain = nn.predict(trainX)
    featuresTrain = featuresTrain.reshape(trainX.shape[0], -1)

    featuresVal = nn.predict(valX)
    featuresVal = featuresVal.reshape(valX.shape[0], -1)

    return featuresTrain, featuresVal

def fit_and_evaluate(model, featuresTrain, trainY, featuresVal, valY):
    model.fit(featuresTrain, trainY)

    # evaluate the model
    results_train = model.predict(featuresTrain)
    print('Training accuracy:', metrics.accuracy_score(results_train, trainY))

    results_val = model.predict(featuresVal)
    print('Validation accuracy:', metrics.accuracy_score(results_val, valY))


nn = tf.keras.applications.VGG16(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
featuresTrain, featuresVal = get_nn_out(nn, trainX, valX)

model = LogisticRegression(max_iter=1000)
fit_and_evaluate(model, featuresTrain, trainY, featuresVal, valY)
```

Sample process for learning using Feature Extraction. Here, first the VGG16 model including imagenet weights is loaded without the final dense layers. Next, training and validation data is sent through the remaining CNN layer to extract image features. Next, a LogisticRegression model is established. Finally, the LogisticRegression model is trained on the extracted training features, before classification predictions for the validation dataset features are fetched and evaluated.

As seen above, the number of output features of the pre-trained models differs vastly between 8192 and 32768. This leads to long run times for some of the secondary classifiers. Using Principle Component Analysis (PCA) we can further shrink the size of the extracted feature space and the input to the secondary classifiers can be made of the same width for each pre-trained model output.

PCA allows maximum 1020 components to be calculated. The selection of the principle components affects the performance of the different secondary classifiers differently. However, a very large number of principle components resulted in the best performance. The table below lists the validation accuracy without PCA (taken from previous experiments) and PCA with 1000 components. Green color indicates improvement after introducing PCA, red color signals performance deterioration.

Model	Features*	LogReg Val Acc NOPCA / PCA	SVC Val Acc NOPCA / PCA	SGD Val Acc NOPCA / PCA	Random Forest Val Acc NOPCA / PCA
VGG16	8192	0.876 / 0.876	0.844 / 0.859	0.812 / 0.859	0.862 / 0.573
VGG19	8192	0.859 / 0.859	0.812 / 0.835	0.812 / 0.817	0.835 / 0.650
InceptionV3	8192	0.865 / 0.847	0.841 / 0.835	0.832 / 0.826	0.824 / 0.626
DenseNet121	16384	0.926 / 0.926	0.897 / 0.900	0.914 / 0.888	0.903 / 0.791
DenseNet201	30720	0.865 / 0.912	0.820 / 0.912	0.835 / 0.885	0.806 / 0.809
ResNet152V2	32768	0.865 / 0.868	0.844 / 0.826	0.832 / 0.838	0.824 / 0.647
MobileNetV2	20480	0.924 / 0.926	0.912 / 0.915	0.903 / 0.894	0.894 / 0.656

The best classification remains at 92.6%. Logistic Regression and SVC benefitted most from introducing PCA between CNN feature output and the secondary classifier. The largely reduce feature space also heavily reduced the SVC runtime. Random Forest performance dropped strongly by introducing PCA.

Introduction of PCA

```
def get_nn_out(nn, trainX, valX, pca=None):
    featuresTrain = nn.predict(trainX)
    featuresTrain = featuresTrain.reshape(trainX.shape[0], -1)

    featuresVal = nn.predict(valX)
    featuresVal = featuresVal.reshape(valX.shape[0], -1)

    if pca:
        pca = decomposition.PCA(n_components=pca)
        pca.fit(featuresTrain)
        featuresTrain = pca.transform(featuresTrain)
        featuresVal = pca.transform(featuresVal)

    return featuresTrain, featuresVal
```

Finally, lets measure how long it takes for getting from “Zero to Hero” for the best feature extractor + secondary classifier combination seen so far.

MobileNetV2 + PCA + Logistic Regression

```
%%time

nn = tf.keras.applications.MobileNetV2(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
featuresTrain, featuresVal = get_nn_out(nn, trainX, valX, pca=1000)

fit_and_evaluate(LogisticRegression(max_iter=1000), featuresTrain, trainY, featuresVal, valY, pca)


Classifier: <class 'sklearn.linear_model._logistic.LogisticRegression'>
PCA: 1000
  Training accuracy: 1.0
  Validation accuracy: 0.9264705882352942

CPU times: user 22.8 s, sys: 2.92 s, total: 25.8 s
Wall time: 16.4 s
```

If one is in a real hurry to develop a reasonably good classifier for the Flowers dataset, it takes roughly **16.4 seconds** to get a classifier with **92.6% accuracy**.

This is drastically faster than fine-tuning full CNNs (or even parts thereof) to this accuracy as we’ll see in the next section.

2.2 Fine-Tuning

Another powerful way to leverage the performance of large pre-trained CNN models is Transfer Learning. With Transfer Learning the final dense Layers of the pre-trained CNNs used for image classification are replaced by (a) new dense layer(s) where the new final layer implements a Softmax activation over the new classes to predict. This way the extremely strong feature extractors of the pre-trained CNNs can be used to predict the new classes. General consensus exists that image features used to classify cars, airplanes, bicycles, etc. are also useful to e.g. classify between dog breeds.

Training the CNN with the new dense layers should be executed in 2 phases:

- Phase 1: Only train the new dense layers. This way the output of the CNN convolutions remains stable to allows the dense layers learn to classify the extracted features to classes. The original trained CNN layers will remain “frozen”.
- Phase 2: In this phase one additionally “fine-tunes” either the full CNN or selected layers only to further increase predictive accuracy of the network. A lower learning-rate is used to prevent too drastic changes to the feature extractors.

If the full CNN would be trained immediately, i.e. skipping phase 1, the completely untrained dense layers would initially create close-to-random predictions leading to a high loss. This loss would then be back-propagated through the whole CNN and likely “break” the already well-trained feature detectors.

In this task several pre-trained models were fine-tuned with to the Flowers dataset using phase 1 and phase 2 for 25 epochs each. After completion of phase 1 the weights that produced the highest accuracy are loaded. For each CNN, the final Dense and Softmax Layers have been removed and replaced by

- Dense Layer with 1024 neuron
- BatchNormalization Layer
- Dropout Layer with 25% drop probability
- Softmax Layer with 17 neurons

Model	Phase 1 Accuracy	Phase 2 Accuracy
VGG19	0.853	0.956
InceptionV3	0.829	0.838
DenseNet121	0.909	0.935
DenseNet201	0.921	0.932
MobileNet	0.938	0.932
MobileNetV2	0.906	0.915

These experiments showed that almost each fine-tuned network benefitted from both phases of the fine-tuning process. Only the MobileNet network showed a behaviour where accuracy dropped upon starting phase 2. For each network data augmentation was used during the training phases.

As the focus of this task lies on finding the highest accuracy we can find, I chose to work with the **Adam optimizer**.

The selection of above base-models was driven by the following rationale:

- General popularity of VGG and InceptionV3
- Deepness of DenseNet201
- Lower computational cost of DenseNet121 over DenseNet201
- MobileNet and MobileNetV2 due to their applicability to edge AI applications

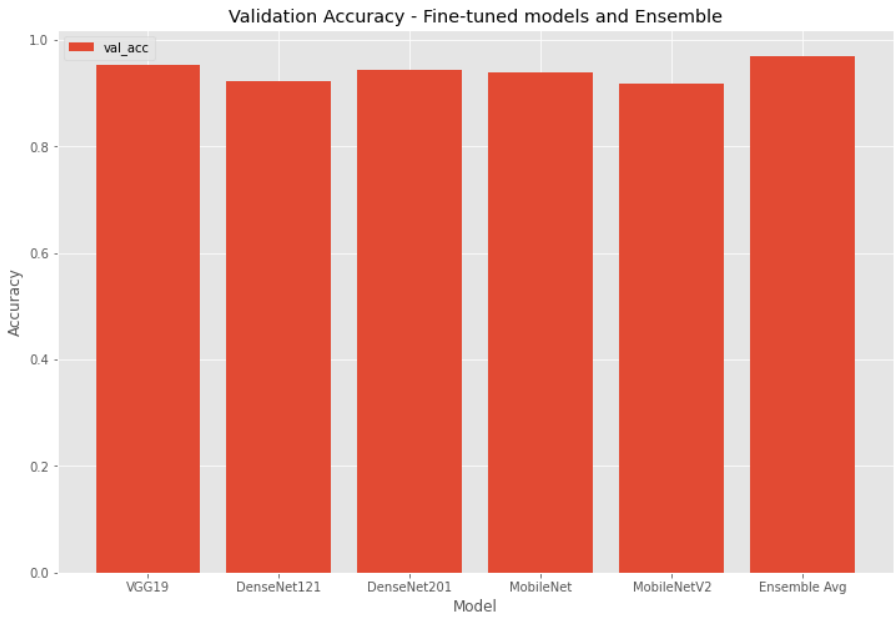
The best individual fine-tuned model was VGG with a max accuracy of **0.956** seen during phase 2 training.

Lastly, the models were trained again to form a fine-tuned CNN ensemble. InceptionV3 was included due to its low accuracy after 2x25 epochs fine-tuning.

Model	Phase 1 Accuracy	Phase 2 Accuracy
VGG19	0.841	0.953
DenseNet121	0.918	0.924
DenseNet201	0.912	0.944
MobileNet	0.935	0.938
MobileNetV2	0.888	0.918
Ensemble	**0.97058**	

The ensemble classification, however, resulted in an impressive **97.06%**.

Note: Individual Phase 1 and 2 train/val loss/accuracy graphs can be found in PartB.ipynb.



Another ensemble run with 3 Dense Layers after the pre-trained CNN layers resulted in slightly worse ensemble accuracy of 96.8%.

3 Part C – Research: Capsule Networks

3.1 Limitations of CNNs

Convolutional Neural Networks (CNN) have been the backbone of the image classification and object detection developments seen over the past years. To achieve improving error rates on the famous ImageNet classification contest, CNN architectures became deeper and deeper and introduced skip layers (residual layers) between near layers and layers “deeper into” the model.

However, besides all successes CNNs have major drawbacks. On a high level these limitations are:

1. Focus on object existence, not localization
2. Overlapping objects / Segmentation is a hard task
3. Low 3D viewpoint variation

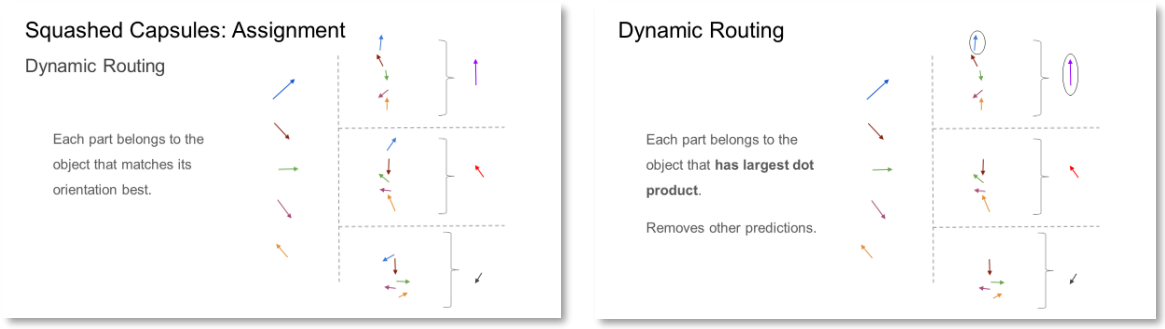
These limitations can be explained by the overall architecture used in CNNs. In each convolutional layer so called Feature Maps try to learn image features (e.g. lines, corners) based on image gradients. Before the output of one convolutional layer is fed into the next convolutional layer, a MaxPooling layer typically shrinks the feature resolution by factor 4 by only keeping the maximum value across a 2x2 pooling window. While this progressing feature map shrinkage allows the CNN to learn higher-level concepts in deeper layers and reduces computational complexity, we achieve viewpoint invariance, but we also lose the pose and location information of the object the detected by the CNN. Especially, we lose the relative spatial relationships between features. So far object detection/localization algorithms (SSD, YOLOv3) circumvented this limitation by performing image classification in parallel in multiple pre-selected frame within the original image. [1][2]

3.2 Capsule Networks

To provide a solution Sara Sabour, Nicholas Frosst and Geoffrey E Hinton proposed the use of Capsules in their seminal paper “Dynamic Routing Between Capsules” [3]. In the most high-level notion, Capsule Networks do not forward individual neuron activations from one layer to the next layer, but instead each capsule represents a small nested neural network which outputs a whole vector. The total length of a capsule’s output vector encodes the probability that a certain feature has been detected. The direction of the vector lengths helps representing the state of the detected feature (e.g. location, pose, scale). When a feature moves across the image, the length of the vector should stay the same (as the feature will till be detected), but the vector’s direction will change. This behavior was called activity equivariance by Hinton.

A capsule s_j then does not forward its output vector v_j blindly to every capsule in the next layer. Instead, a capsule predicts the output of all capsules in the next layer given its own output vector v_j and the respective coupling coefficient c_{ij} and forwards its output only to that capsule whose predicted output results in the largest vector. By this “next-layer output prediction” capsule s_j ensures that it selects the most appropriate capsule for a given higher-

level feature. Depending on the resulting next-layer output vector, the coupling coefficient c_{ij} be updated.



Source: Sara Sabour @ CVPR2019 – Capsule Tutorial [4]

Each capsule's state s_j is calculated as the weighted sum of the matrix multiplication of output/prediction vectors of the capsules from the lower layer with the coupling coefficient c_{ij} between s_j and the respective lower-level capsule s_i.

$$\mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}\ , \qquad \hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$$

Of course, capsules in the first capsule layer of a capsule network calculate their activation based on the input from the previous convolution layer. In this case, no coupling coefficient c_{ij} exists.

As the capsule's output vector indicates the probability of having detected a certain feature, capsule s_j's output vector v_j is "squashed", so that long vectors sum up to 1 max and short vectors are close to zero.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1+||\mathbf{s}_j||^2}\frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

The coupling coefficients c_{ij} defines the "activation routing" between a capsule and all potential parent capsules in the next layer and sum to 1. The softmax-like calculation ensure that the most likely "parent" capsule gets the "most" of capsule s_j's output.

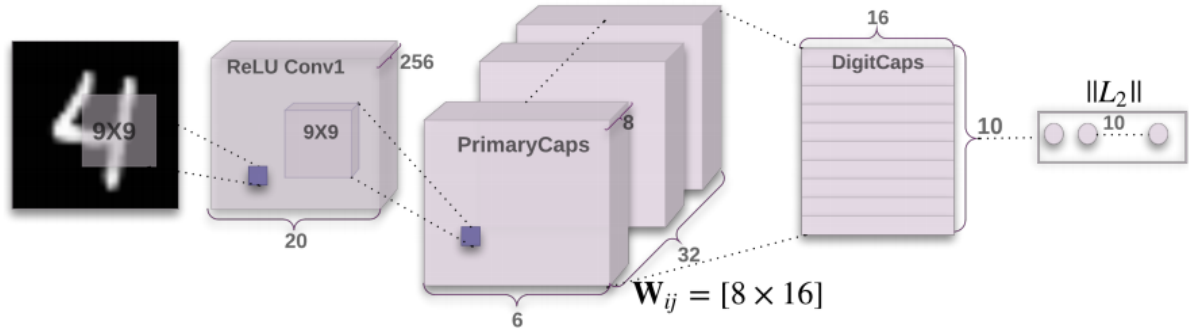
$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$$

By following the presented calculations, the routing preferences between capsules and the prediction of next layer activations, Capsule Network claim to address the CNN limitations listed above, especially modelling stronger feature relationships then CNN could represent which is a very strong tool to boost image segmentation.

3.3 Capsule Networks Use Cases

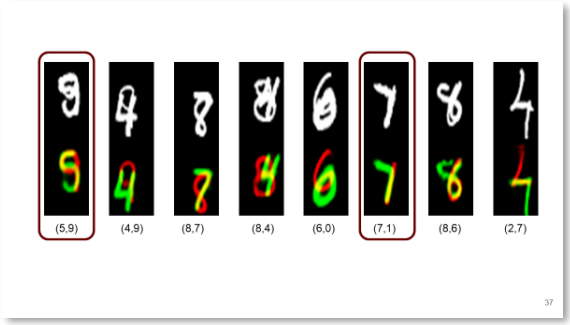
The CapsNet architecture represent a baseline implementation of Capsule Networks for the MNIST dataset. CapsNet not only achieves state-of-the-art CNN performance in digit classification, but it especially excels in the MultiMNIST dataset where each image includes 2 overlapping digits.

The CapsNet was designed with an initial classical convolutional layer, followed by 2 capsule layers.

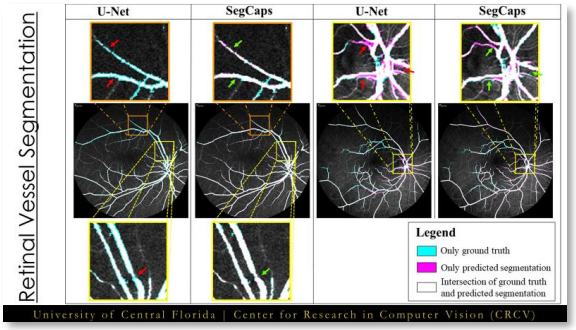


Source: Dynamic Routing Between Capsules, Sabour, Frosst, Hinton [3]

At the CVPR 2019 conference several capsule use cases were presented. The left image below demonstrates how CapsNet is able to correctly detect overlapping digits due to the feature bonding between capsules. The right image demonstrates how SegCaps, a segmentation network with capsules, achieves more accurate results in segmenting granular retinal vessels.



Source: Sara Sabour @ CVPR2019 – Capsule Tutorial [4]



Source: Rodney Lalonde @ CVPR2019 – Capsule Tutorial [4]

References:

[1] SSD: Single Shot MultiBox Detector, Liu et al., https://arxiv.org/abs/1512.02325
[2] YOLOv3: An Incremental Improvement, Joseph Redmon, Ali Farhadi, https://arxiv.org/abs/1804.02767
[3] “Dynamic Routing Between Capsules”, Sara Sabour et al., https://arxiv.org/abs/1710.09829
[4] CVPR 2019 Capsule Tutorial, https://www.crcv.ucf.edu/cvpr2019-tutorial/
[5] Capsule Networks (CapsNets) – Tutorial, Aurélien Géron, https://www.youtube.com/watch?v=pPN8d0E3900

Appendix – Adversarial Machine Learning

Despite my choice on Capsule Networks in Part 3 of the assignment, I came across a very a remarkable adversarial machine learning related use case that meets the 2020 zeitgeist on multiple layers:

Source: https://github.com/BruceMacD/Adversarial-Faces