

DEEP LEARNING ASSIGNMENT 2:

Convolutional Network

Part A: Convolutional Neuronal Network

The objective of this assessment is to build a range of deep learning models using convolutional neural networks.

The data we will be using is the Flowers 17 dataset. This is a multi-class classification problem with 17 possible classes (17 different classes of flowers). The images have significant variation in pose and lighting and there is also significant variation within certain classes and close similarity between other distinct classes. What makes this dataset even more challenging is that there are only 80 images for each class. Therefore, there are just 1360 images

In this assignment, the original dataset divided so that 75% will be used for training and 25% will be used as validation data. As there is such a limited amount of data we will just use training and validation datasets.

Moreover, pre-processing has already been applied on the dataset: All images are now of an equal size (128*128*3) and the data has also been normalized.

The shape of the feature training data is (1020, 128, 128, 3), while the shape of the validation data is (340, 128, 128, 3). Therefore, the data is divided into 1020 training images and 340 validation images.

1) Section I: CNN Building

In this part, the task is to build a baseline CNN, which contains various convolutional layer and pooling layer. (We will start with 1 convolutional layer and 1 pooling layer).

As optimizer, we are using Stochastic Gradient Descent (SGD) with a Learning Rate of 0.01.

a. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL_Assignment2_Part_A.ipynb .

2 functions have been defined:

`_loadDataH5`: This function reads the data1.h5 file dataset. It returns at the end, 4 tensorflow objects which corresponds to the training images, the training labels, the test images, and the test labels.

`_class ShallowNet, build`: This function defines the architecture of the CNN. It returns at the end the CNN model.

Beside the 2 functions, we launch the 3 required steps to train our CNN model:

`_model.build` : builds the CNN architecture defined in the ShallowNet class

`_model.compile`: compilation of the CNN model with a specific optimizer (in our case we use the SGD optimizer with a learning rate of 0.01)

`_model.fit`: this step trains our model given the previous defined CNN architecture and optimizer strategy.

Finally, after training our model, we plot the loss and accuracy (for training and validation data) results.

For more information on the code implementation, please, refers to the comments present in the ipynb file.

b. Result Analysis:

In this section, we have built 3 different CNN architecture:

- 1st architecture: 1 convolution + 1 Max pooling layer
- 2nd architecture: 2 convolutions + 2 Max pooling layers
- 3rd architecture: 6 convolutions + 3 Max Pooling layers

The result of those architecture is explained below.

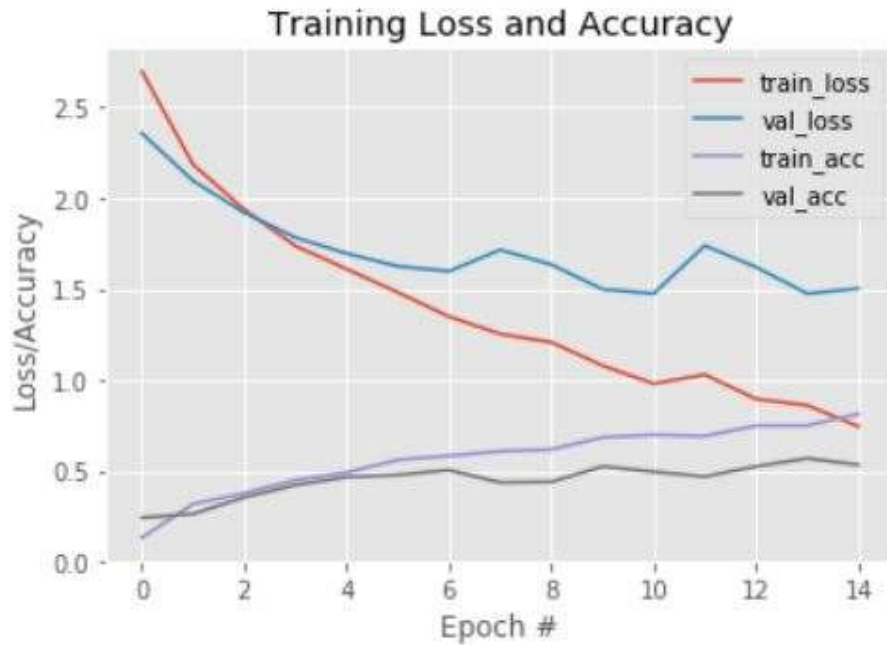
1. 1st architecture: 1 convolution + 1 Max pooling layer

The architecture of the model has been implemented as followed:

- a. Layer 1: 2D Convolution of size 3x3 and with 64 filters (ReLU activation function and padding same)
- b. Layer 2: Max Pooling of size 2x2 (padding same)
- c. Data in flattened
- d. Activation Layer: Softmax activation with 17 classes

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 15.

The result of this training has been plotted:



In this model, we can clearly see that a divergence of the training loss a validation loss occurs at around epoch 4. The validation starts to be higher than the training loss. This means that the model is overfitting after 4 epochs.

We can clearly see this overfitting when we compare the result at epoch 7 and at epoch 15:

At epoch 4: loss: 1.7372 - acc: 0.4480 - val_loss: 1.7834 - val_acc: 0.4206

At epoch 15: loss: 0.7434 - acc: 0.8118 - val_loss: 1.5027 - val_acc: 0.5324

As we can see, when the loss for training and validation is approximatively similar at epoch 4 (around 1.75), at epoch 15 we have a difference of almost 1 for those losses. Moreover, at epoch 15, the difference of the accuracy is extremely high between the training (0.8118) and the validation(0.5324): The accuracy of the “unseen” data is far low compared to the trained data and indicates that the CNN model is starting to take into account meaningless features to make its prediction.

If we just rely on the result before overfitting (result of epoch 4), we can say that with this CNN model, we are able to reach to an accuracy of around 0.43.

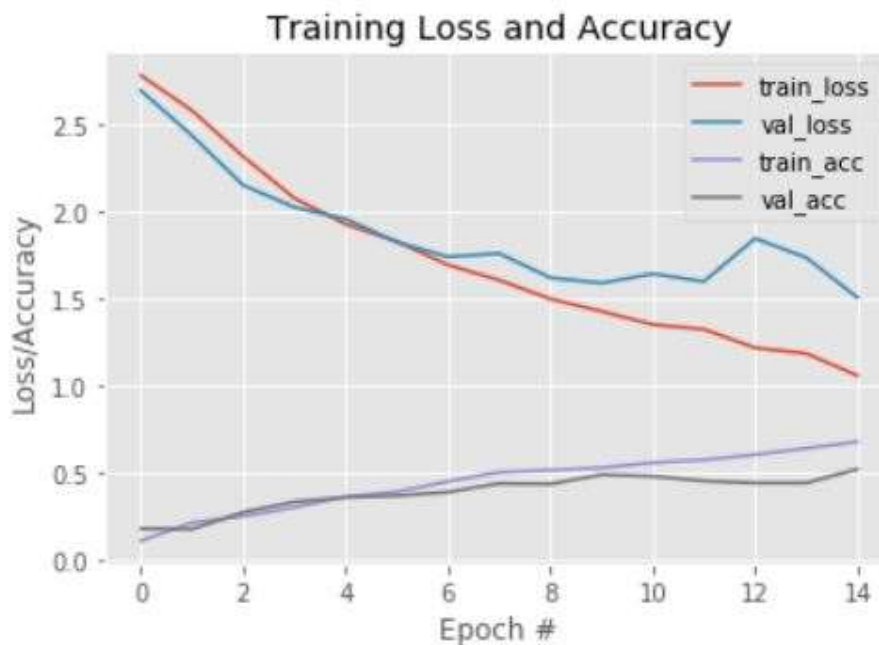
II. 2nd architecture: 2 convolutions + 2 Max pooling layers

The architecture of the model has been implemented as followed:

- Layer 1: 2D Convolution of size 3x3 and with 64 filters (ReLU activation function and padding same)
- Layer 2: Max Pooling of size 2x2 (padding same)
- Layer 3: 2D Convolution of size 3x3 and with 64 filters (ReLU activation function and padding same)
- Layer 4: Max Pooling of size 2x2 (padding same)
- Data is flattened
- Activation Layer: Softmax activation with 17 classes

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 15.

The result of this training has been plotted:



In this model, we can clearly see that a divergence of the training loss and validation loss occurs at around epoch 6 or 7. The validation starts to be higher than the training loss. This means that the model is overfitting after 7 epochs.

We can clearly see this overfitting when we compare the result at epoch 7 and at epoch 15 :

At epoch 7: loss: 1.6972 - acc: 0.4529 - val_loss: 1.7438 - val_acc: 0.3912

At epoch 15: loss: 1.0615 - acc: 0.6804 - val_loss: 1.5096 - val_acc: 0.5235

As we can see, when the loss for training and validation is approximatively similar at epoch 7 (around 1.7), at epoch 15 we have a difference of almost 0.5 for those losses. Moreover, at epoch 15, the difference of the accuracy is relatively high between the training (0.6804) and the validation (0.5235): The accuracy of the “unseen” data is far low compared to the trained data and indicates that the CNN model is starting to take into account meaningless features to make its prediction.

If we just rely on the result before overfitting (result of epoch 7), we can say that with this CNN model, we are able to reach to an accuracy of around 0.4.

If we compare with the previous model, we can see that adding an extra convolutional layer and max pooling layer does not make a significant improvement in term of accuracy and losses. The only visible impacts by adding those extra layers is that the number of iteration (nb of epochs) which is required to get the optimal accuracy has increased. (for the last model it was after 4 epochs. For this model, it is now 7 epochs).

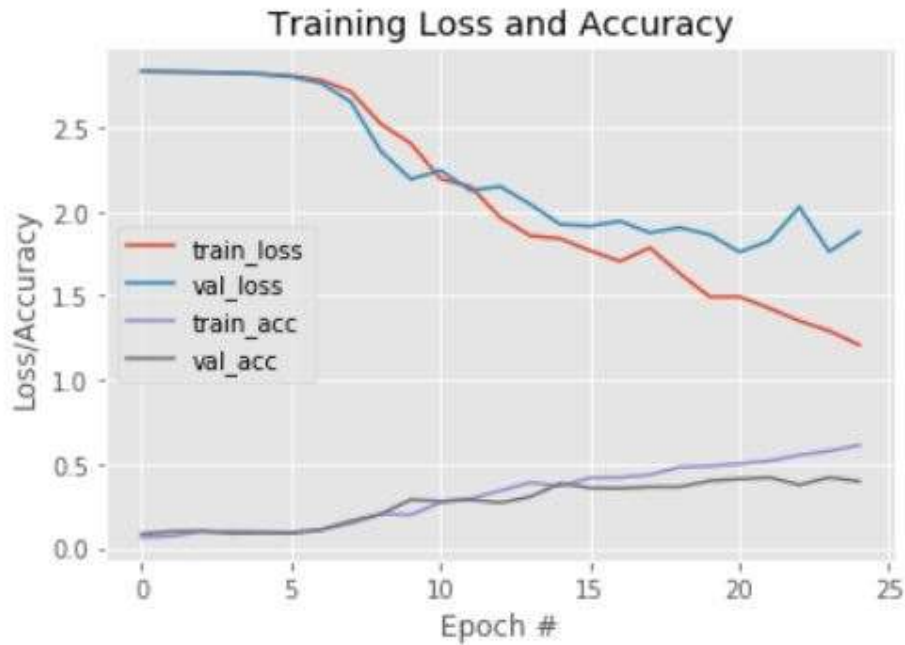
III. 3rd architecture: 6 convolutions + 3 Max pooling layers

The architecture of the model has been implemented as followed:

- a. Layer 1: 2D Convolution of size 3x3 and with 32 filters (ReLU activation function and padding same)
- b. Layer 2: 2D Convolution of size 3x3 and with 32 filters (ReLU activation function and padding same)
- c. Layer 3: Max Pooling of size 2x2 (padding same)
- d. Layer 4: 2D Convolution of size 3x3 and with 64 filters (ReLU activation function and padding same)
- e. Layer 5: 2D Convolution of size 3x3 and with 64 filters (ReLU activation function and padding same)
- f. Layer 6: Max Pooling of size 2x2 (padding same)
- g. Layer 7: 2D Convolution of size 3x3 and with 128 filters (ReLU activation function and padding same)
- h. Layer 8: 2D Convolution of size 3x3 and with 128 filters (ReLU activation function and padding same)
- i. Layer 9: Max Pooling of size 2x2 (padding same)
- j. Data is flattened
- k. Activation Layer: Softmax activation with 17 classes

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 25.

The result of this training has been plotted:



In this model, we can clearly see that a divergence of the training loss and validation loss occurs at around epoch 15. The validation starts to be higher than the training loss. This means that the model is overfitting after 15 epochs.

We can clearly see this overfitting when we compare the result at epoch 7 and at epoch 15 :

At epoch 15: loss: 1.8396 - acc: 0.3686 - val_loss: 1.9248 - val_acc: 0.3853

At epoch 25: loss: 1.2080 - acc: 0.6118 - val_loss: 1.8787 - val_acc: 0.3971

As we can see, when the loss for training and validation is approximatively similar at epoch 7 (around 1.85), at epoch 25 we have a difference of almost 1 for those losses. Moreover, at epoch 25, the difference of the accuracy is relatively high between the training (0.6118) and the validation (0.3971): The accuracy of the “unseen” data is far low compared to the trained data and indicates that the CNN model is starting to take into account meaningless features to make its prediction.

If we just rely on the result before overfitting (result of epoch 15), we can say that with this CNN model, we are able to reach to an accuracy of around 0.37.

If we compare with the previous model, we can see that adding an extra convolutional layer and max pooling layer does not make a significant improvement in term of accuracy and losses. (It is even slightly less performant; but the difference is not significant). Once again, the only visible impacts by adding those extra layers is that the number of iteration (nb of epochs) which is required to get the optimal accuracy has increased. (for the 1st model it was after 4 epochs; for the 2nd, it was after 7 epochs; and now, for the 3rd, it is at 15 epochs).

c. General Conclusion:

We have studied the impact of increasing the number of convolutional layers and max layer in our CNN model. After analysis, it seems that, we do not gain any accuracy by adding extra layers.

Indeed, adding layers increases the number of weights in the network and the model complexity. But this complexity is not required in our dataset since the dataset is quite small (1020 training images and 340 validation images): We don't need to extract more and more feature. This is the reason why we don't gain any accuracy by adding extra layers in our current case. Moreover, because of the small dataset, we tend to overfit very quickly the model when we are training it.

2) Section II: Data augmentation

In this part, the task is to build a baseline CNN, which contains various convolutional layer and pooling layer.

As optimizer, we are using Stochastic Gradient Descent (SGD) with a Learning Rate of 0.01.

a. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL_Assignment2_Part_A.ipynb .

As we are reusing the CNN models implemented in Section 1, only the part related to Data generation has been implemented:

`_trainDataGenerator`: this variable is using the `tf.keras.preprocessing.image.ImageDataGenerator` function to generate new images for training data. In our case, the data generated has the following characteristics:

- `rotation_range=40`: It indicates the range of degree of rotation allowed when images are generated
- `width_shift_range=0.1`: fraction of the total width for which shift is allowed
- `shear_range=0.2` : it represents the shear intensity
- `zoom_range=0.4` : it stores the range for random zoom
- `horizontal_flip=True`: Specify the possibility of doing an horizontal flip to the image

`_trainDataGenerator.fit`: this step sets the set of images which will be used for generation the data augmentation.

`_trainDataGenerator.flow`: This step generates real-time data augmentation

Finally, after training our model, we plot the loss and accuracy (for training and validation data) results.

For more information on the code implementation, please, refers to the comments present in the ipynb file.

b. Result Analysis:

In this section, we use the same 3 CNN architecture which have been implemented in section A-1:

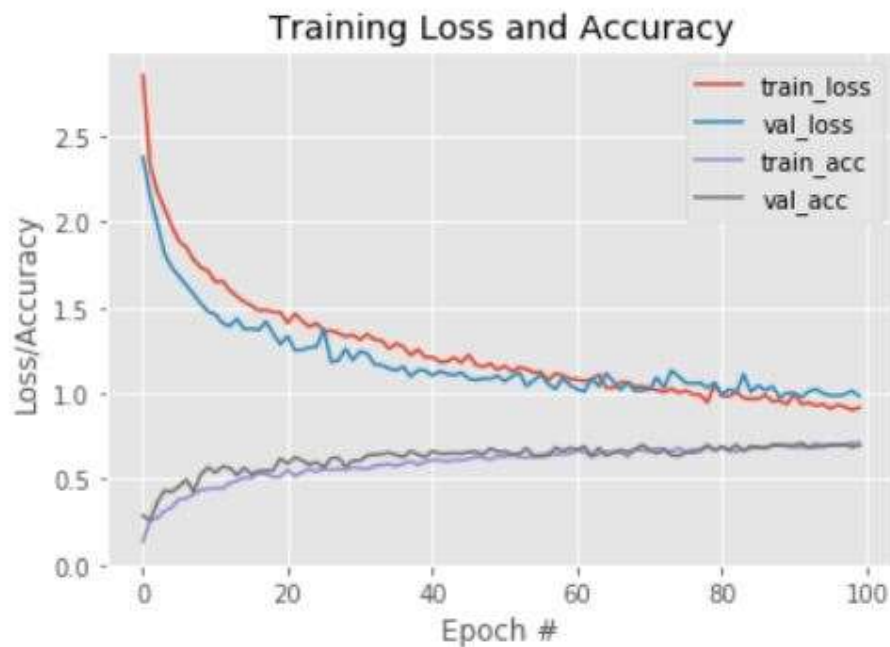
- 1st architecture: 1 convolution + 1 Max pooling layer
- 2nd architecture: 2 convolutions + 2 Max pooling layers
- 3rd architecture: 6 convolutions + 3 Max Pooling layers

The result of those architecture are explained below.

1. 1st architecture: 1 convolution + 1 Max pooling layer

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 100.

The result of this training has been plotted:



In this model, we can see that even after 100 iterations (nb of epoch), there is no divergence of the training loss and validation loss occurs at around epoch 15. Therefore, no overfitting is observed.

Typically, we can observe the following result at epoch 100:

At epoch 100: `loss: 0.9159 - acc: 0.7088 - val_loss: 0.9828 - val_acc: 0.6941`

As we can see, the values for the losses for training and the validation is almost similar (around 0.91 and 0.98) and their respective accuracy are in the same range (0.70 and 0.69) which confirms that we don't have any overfitting.

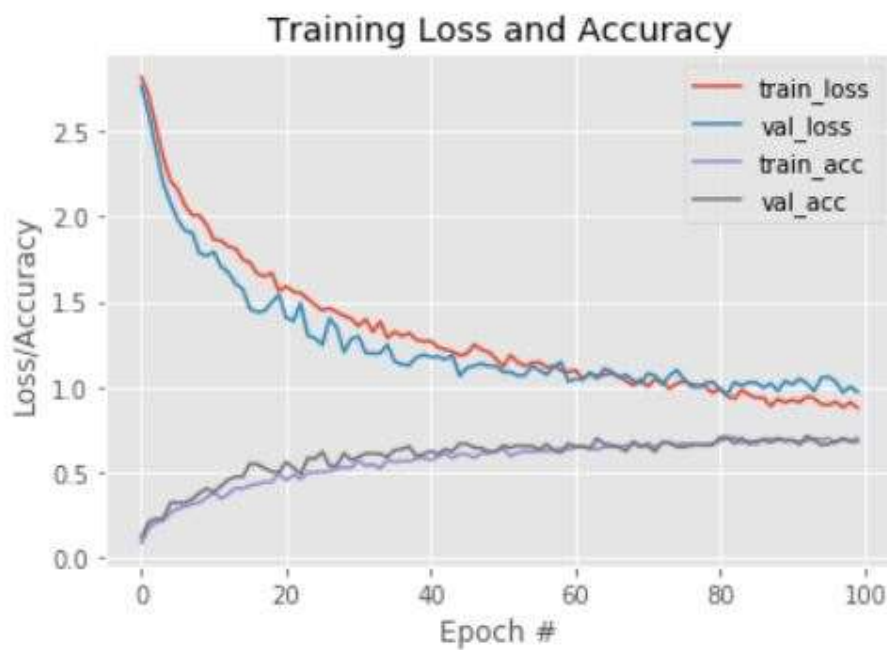
Moreover, the accuracy which have been achieved is pretty decent with around 0.7.

If we compare the result with the previous training model (without any data augmentation), we can notice a significant improvement in term of accuracy (from around 0.4 to 0.7) and losses (from around 1.7 to 0.9). This increase of accuracy is expected since the generation of the data gives new images with little transformation of the originals without changing the item to be identified on itself. (It provides basically more context of the same object). Therefore, the data augmentation is very useful technique in order to improve the accuracy of our model

II. 2nd architecture: 2 convolutions + 2 Max pooling layers

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 100.

The result of this training has been plotted:



In this model, we can see a slight divergence of the training loss and validation loss occurs at around epoch 82. The validation starts to be higher than the training loss. This means that the model is overfitting after 82 epochs.

We can clearly see this overfitting when we compare the result at epoch 61 and at epoch 100:

At epoch 82: loss: 0.9465 - acc: 0.7010 - val_loss: 0.9610 - val_acc: 0.7088

Epoch 83/100

At epoch 100: loss: 0.8801 - acc: 0.7029 - val_loss: 0.9731 - val_acc: 0.6824

As we can see, when the loss for training and validation is approximatively similar at epoch 82 (around 0.95), at epoch 100 we have a difference of almost 0.3 for those losses. Moreover, at epoch 100, the difference of the accuracy is relatively high between the training (0.70) and the validation(0.6824): The accuracy of the “unseen” data is far low compared to the trained data and indicates that the CNN model is starting to take into account meaningless features to make its prediction.

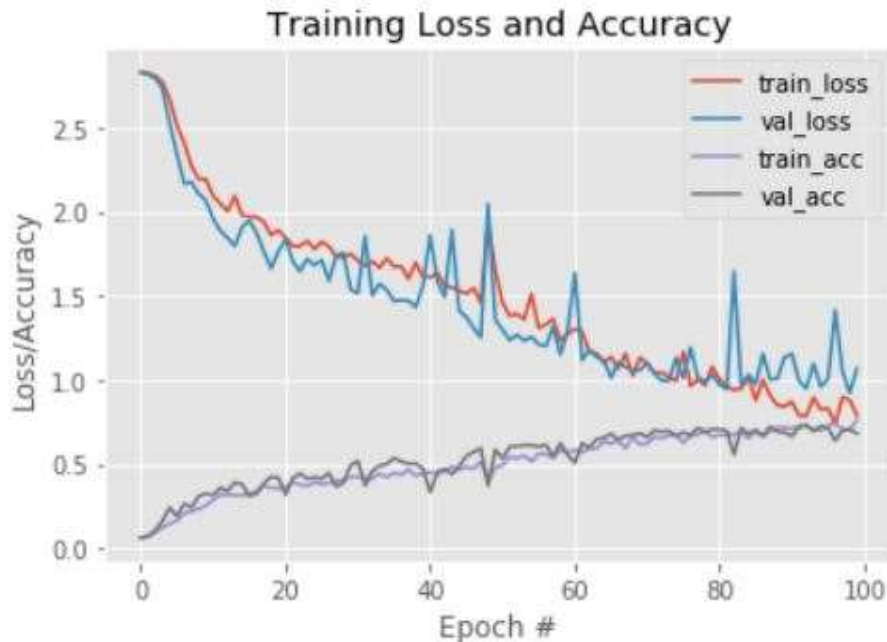
If we just rely on the result before overfitting (result of epoch 82), we can say that with this CNN model, we are able to reach to an accuracy of around 0.7. This result is almost similar to the 1st model but we are able to get it after 82 epochs instead of 100 epochs for the 1st model.

If we compare the result with a training without any data augmentation for the same model, we can notice a significant improvement in term of accuracy (from around 0.4 to 0.7) and losses (from around 1.7 to 0.9). The data augmentation technique is again effective for this model.

III. 3rd architecture: 6 convolutions + 3 Max pooling layers

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 100.

The result of this training has been plotted:



In this model, we can see a slight divergence of the training loss and validation loss occurs at around epoch 82. The validation starts to be higher than the training loss. This means that the model is overfitting after 82 epochs.

We can clearly see this overfitting when we compare the result at epoch 61 and at epoch 100:

At epoch 82: loss: 0.9646 - acc: 0.6725 - val_loss: 0.9466 - val_acc: 0.6971

At epoch 100: loss: 0.7890 - acc: 0.7598 - val_loss: 1.0672 - val_acc: 0.6824

As we can see, when the loss for training and validation is approximatively similar at epoch 82 (around 0.95), at epoch 100 we have a difference of almost 0.3 for those losses. Moreover, at epoch 100, the difference of the accuracy is relatively high between the training (0.759) and the validation (0.6824): The accuracy of the “unseen” data is far low compared to the trained data and indicates that the CNN model is starting to take into account meaningless features to make its prediction.

If we just rely on the result before overfitting (result of epoch 82), we can say that with this CNN model, we are able to reach to an accuracy of around 0.7. This result is almost similar to the 1st model but we are able to get it after 82 epochs instead of 100 epochs for the 1st model.

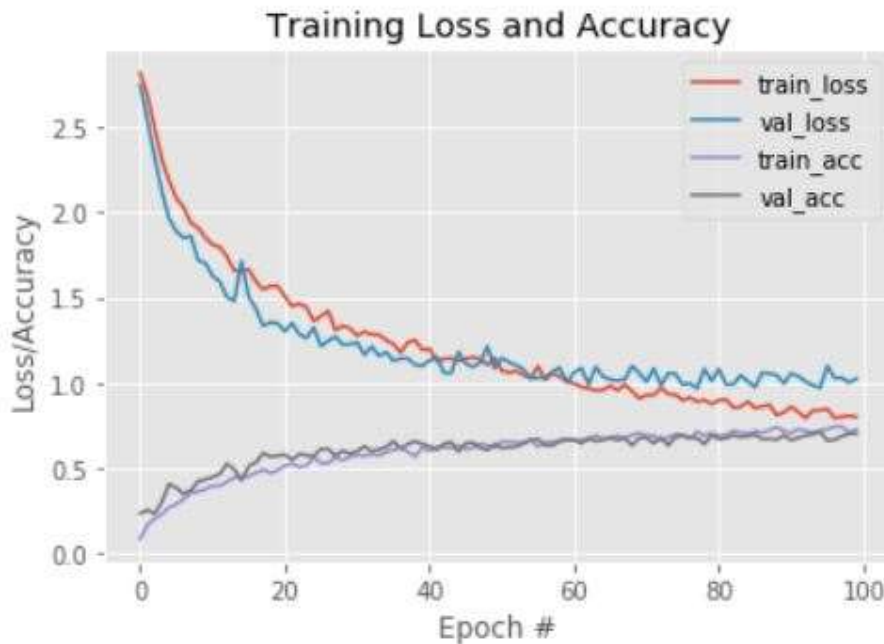
If we compare the result with a training without any data augmentation for the same model, we can notice a significant improvement in term of accuracy (from around 0.4 to 0.7) and losses (from around 1.7 to 0.9). The data augmentation technique is again effective for this model.

IV. 2nd architecture: 2 convolutions + 2 Max pooling layers (Data augmentation without flip horizontal)

For this particular case, we are studying the data augmentation without applying the horizontal flip to see its impact on the model accuracy.

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 100.

The result of this training has been plotted:



In this model, we can see a slight divergence of the training loss and validation loss occurs at around epoch 64. The validation starts to be higher than the training loss. This means that the model is overfitting after 64 epochs.

Typically, we can observe the following result at epoch 63 (before the overfitting):

At epoch 64: loss: 0.9545 - acc: 0.6931 - val_loss: 1.0940 - val_acc: 0.6588

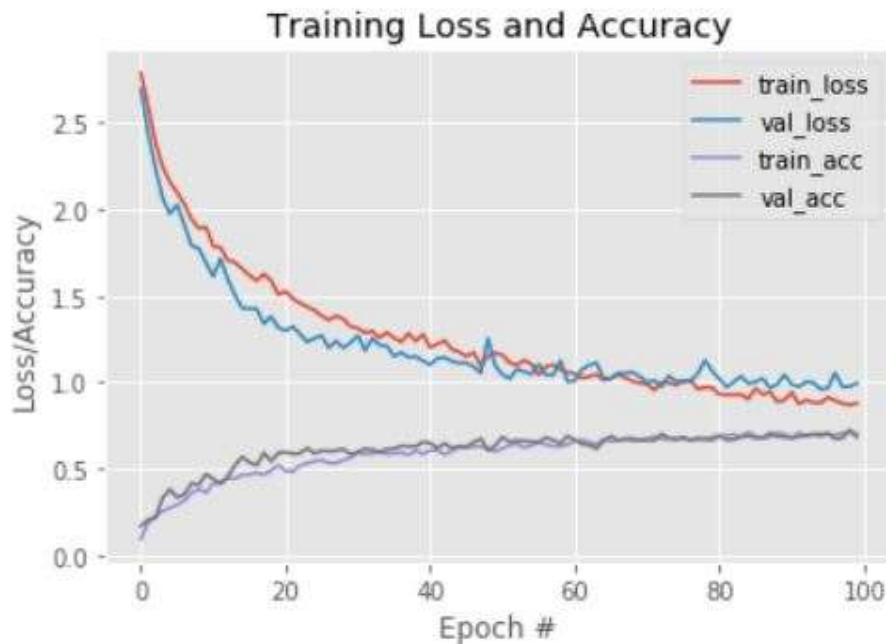
If we just rely on the result, we can say that with this CNN model trained on augmented (without flip) data has reached to an accuracy of around 0.66. This result is slightly lower than the accuracy of the same model trained with augmented data containing horizontal flip (0.69). Removing flip on the augmented data for this particular dataset does affect the accuracy a little by reducing it slightly.

V. *2nd architecture: 2 convolutions + 2 Max pooling layers (Data augmentation without crop)*

For this particular case, we are studying the data augmentation without applying the crop to see its impact on the model accuracy.

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 47.

The result of this training has been plotted:



In this model, we can see a slight divergence of the training loss and validation loss occurs at around epoch 79. The validation starts to be higher than the training loss. This means that the model is overfitting after 64 epochs.

Typically, we can observe the following result at epoch 79 (before the overffiting):

At epoch 79: loss: 0.9701 - acc: 0.6745 - val_loss: 1.1266 - val_acc: 0.6676

If we just rely on the result, we can say that with this CNN model trained on augmented (without flip) data has reached to an accuracy of around 0.66. This result is slightly lower than the accuracy of the same model trained with augmented data containing horizontal flip (0.69). Removing crop on the augmented data for this particular dataset does affect the accuracy a little by reducing it slightly.

c. General Conclusion:

We have studied the impact of the data augmentation technique in our CNN models. After analysis, this technique offers a significant gain in term of accuracy when it is activated with various parameters to generate the data (such as crop, flip, zoom,...). if one of those parameters is missing, then we should expect a diminution of the model accuracy.

Indeed, adding new data with little variations on those data helps the model to catch meaningful features to make its classification.

3) Section III: CNN Ensemble

In this part, the task is to build a basic CNN ensemble containing a maximum of 10 base learners.

For this part, we will take the same architecture for the 10 base learners.

a. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL_Assignment2_Part_A.ipynb .

2 classes functions have been defined:

`_class nin_cnn_model, function build:` This function defines the architecture of the CNN. It returns at the end the CNN model.

`_class train_model, function train:` This function ensures that the CNN model is build and compiled with a given optimizer and afterward, fitted to a specified dataset. It returns at the end the built model, the history of the fitted model and the number of epoch for training

The next step is to train 10 times this model in order to get 10 trained models which will be our base learners.

At the end prediction is calculated for each base learner in order to build the average accuracy of the ensemble of these 10 base learners.

b. Result Analysis:

For this part, `nin_cnn_model:`

- `batch_size = 100`
- `batch_size = 500`
- `batch_size = 1000`

l. Preliminary result with the nin_cnn_model

The architecture of the model has been implemented as followed:

- g. Layer 1: 2D Convolution of size 3x3 and with 32 filters (ReLu activation function and padding same)
- h. Layer 2: Max Pooling of size 2x2 (padding same)
- i. A dropout layer (0.5)
- j. Layer 3: 2D Convolution of size 3x3 and with 64 filters (ReLu activation function and padding same)
- k. Layer 4: Max Pooling of size 2x2 (padding same)
- l. A dropout layer (0.5)
- m. Data is flattened
- n. Activation Layer: Softmax activation with 17 classes

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 10.

The nin_cnn_model has been trained 10 times to get 10 base learners: The accuracy of each base have been calculated. Then, the mean accuracy of the ensemble of those 10 bases have been computed.

The result of the accuracy of each trained model and its ensemble is as following:

```
Accuracy of trained model 0 : 0.5147059
Accuracy of trained model 1 : 0.5117647
Accuracy of trained model 2 : 0.5382353
Accuracy of trained model 3 : 0.5441176
Accuracy of trained model 4 : 0.5323529
Accuracy of trained model 5 : 0.5
Accuracy of trained model 6 : 0.45882353
Accuracy of trained model 7 : 0.46764705
Accuracy of trained model 8 : 0.5264706
Accuracy of trained model 9 : 0.49117646
```

```
Accuracy of the ensemble: 0.5647059
```

As we can see, the accuracy we get for the ensemble (0.56) is significantly higher than the accuracy we used to get in Part A section 1 (accuracy of around 0.4). The ensemble is very meaningful technique to improve the accuracy of the model. We can notice as well that the accuracy of the ensemble is the slightly higher compared to the accuracy of each base learner taken separately. Typically, here the highest accuracy achieved for the base learner is 0.54 (trained model 3) whereas the accuracy of the ensemble is 0.56. The gain is not that significant if we compare those 2 metrics.

Indeed, as our ensemble is a very basic one containing the same base learner model and same parameter, we can see that there is not a significant improvement in overall performance between the base learner model and its ensemble.

One of the reasons for this, may be the lack of diversity in the base models. Indeed, we have taken the same CNN architecture for the base learner. This mean that we are trying to extract always the same type of features (since the topology is identical) for each of our base learner. Therefore, we end to have approximatively the same result for each Base learner. The small variation between them is only due to a different initialization of the parameters in the convolutional layer.

There are many steps which can be taken in order to introduce additional diversity into the base models:

- 1) Varying the initial parameters : We can use different technique to initialize parameters in the convolutional layers.
- 2) Varying the optimizer, with different algorithm and learning parameters
- 3) Varying the CNN architecture: This is probably the most important technique for an ensemble in order to capture different feature between each base model
- 4) Varying regularization and drop out in the same base model.

Those steps would allow us to get heterogeneous base model which would capture differently the features in the CNN and therefore give complementarity between each base. This could increase significantly the overall average of the ensemble.

II. Varying some parameters in the nin_cnn_model

Here, we will study the impact of the 2 first steps mentioned above for having a better ensemble with a quite heterogeneous base model. Ideally it would be better to vary the CNN architecture but for our study, we will focus only on:

- Varying the initial parameters
- Varying the optimizer, with different algorithm and learning parameters

The Base model will be the same for all the 10 base learners.

For supporting the above implementation, we have given one parameter in the `nin_cnn_model` to allows us to select different initializers and we have also added one parameter in the `Train_model` to allow us to select different optimizer with different learning rate.

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 10.

The `nin_cnn_model2` has been trained 10 times to get 10 base learners: The accuracy of each base have been calculated. Then, the mean accuracy of the ensemble of those 10 bases have been computed.

The result of the accuracy of each trained model and its ensemble is as following:

```
Accuracy of trained model 0 : 0.5117647
Accuracy of trained model 1 : 0.5411765
Accuracy of trained model 2 : 0.5294118
Accuracy of trained model 3 : 0.57941175
Accuracy of trained model 4 : 0.58235294
Accuracy of trained model 5 : 0.49117646
Accuracy of trained model 6 : 0.43235293
Accuracy of trained model 7 : 0.5352941
Accuracy of trained model 8 : 0.59411764
Accuracy of trained model 9 : 0.5588235
```

```
Accuracy of the ensemble: 0.62058824
```

As we can see, the accuracy we get for the ensemble (0.62) is significantly higher than the accuracy we used to get earlier without adding any variation (accuracy of around 0.56).

Of course, this accuracy could be improve again if we explore the other variation such as the variation of the CNN architecture and variation of regularization and dropout.

Part B: Transfer Learning

The objective of Part B is to work on transfer learning. For this purpose, we will investigate and explore the impact of feature extraction and fine-tuning techniques.

1) Section I: Feature extraction

In this part, the task is to use a pre-trained CNN model as a feature extractor and pair its output with a secondary (standard) machine learning algorithm.

a. Quick code explanation:

The python code using tensorflow library has been implemented using Google collab in the file DL_Assignment2_Part_B.ipynb .

The implementation of the code is quite straightforward:

- 1) A pre-trained CNN model is loaded with `tf.keras.applications`.
- 2) `featuresTrain`: The model (`model.predict`) is used to make prediction on the training data and then its output values are reshaped
- 3) `featuresVal`: The model (`model.predict`) is used to make prediction on the test data and then its output values are reshaped
- 4) `model`: A ML classifier is selected from `sklearn`.
- 5) `model.fit`: This ML model is trained
- 6) `results`: a prediction (`model.predict(featuresVal)`) is made on the test data
- 7) `metrics.accuracy_score(results, testY)`: the accuracy of the prediction is done

For more information on the code implementation, please, refer to the comments present in the ipynb file.

b. Result Analysis:

In this section, we are considering 2 pre-trained CNN architectures:

- **VGG16**: This CNN model is proposed in 2014. The model achieves 92.7% top-5 test accuracy in ImageNet. As our dataset is a set of images to classify. It makes sense to use this CNN which was able to achieve such a high score in ImageNet as a feature extractor.
- **InceptionV3**: This CNN has been shown to attain greater than 78.1% accuracy on the ImageNet dataset. The particularity of this CNN is that it uses very few parameters compared to VGGNet. It could be interesting to test this CNN as a feature extractor.

After extraction of the features from a CNN, we will use the output scalars to train our ML classifier and make a prediction on the test dataset. The ML classifier we will work on are:

- Random Forest: This classifier has reached the highest score in term of wine classification (cf. Machine Learning Assignment 2 in December 2018). A very high accuracy is expected for this classifier
- Decision Tree: This classifier is probably not as strong as the Random Forest since the Decision Tree is just a “subset” of a Random Forest and itself, the Decision Tree tends to overfit the data.
- KNN: This classifier is also interesting to consider since it was also one of the ML classifier which got a very high score in term of wine classification (cf. Machine Learning Assignment 2 in December 2018). A very high accuracy is expected for this classifier
- Naïve Bayes: This classifier is probably not as strong as the KNN. But it could be tested to see if it can give some result after a CNN feature extractor.
- SVM: This ML classifier could be interesting to study since the result obtained with it for wine classification was not that bad.

The result of those feature extractor test paired with the above ML classifier are visible below.

1. CNN Model: VGG16

The result for this CNN feature extractor is displayed below:

```
### CNN Model: VGG16
Accuracy Evaluation of different ML classifier
Random Forest: 0.8529411764705882
Decision Tree: 0.4588235294117647
KNN           : 0.7088235294117647
Naïve Bayes   : 0.4676470588235294
SVM           : 0.7617647058823529
```

For the VGG16 model extractor, we are able to achieve a very high score of accuracy using the Random Forest Classifier with an accuracy of nearly 0.85. This high level of accuracy for Random Forest Classifier is not surprising. Indeed, Unlike Decision Tree, Random Forest combines multiple Decision Tree and avoid by doing this overfitting. Moreover, It is very powerful to handle large features parameters.

SVM with an accuracy of 0.76 seems very powerful as well. This can be explained by the fact that SVM is quite effective in case of unknown (or non-regularly) distribution data which is the case here with our image dataset. Indeed, this classifier handles nonlinear separation between the classes and could be therefore extremely powerful between 2 images which separation could be difficult (such as 2 different flowers just like in our database) . This may explain why at the end SVM is more powerful in this context than KNN which relies on neighbors similarities.

I. CNN Model: InceptionV3

The result for this CNN feature extractor is displayed below:

```
### CNN Model: InceptionV3
Accuracy Evaluation of different ML classifier
Random Forest: 0.8294117647058824
Decision Tree: 0.4764705882352941
KNN           : 0.638235294117647
Naïve Bayes   : 0.6794117647058824
SVM           : 0.8088235294117647
```

For the InceptionV3 model extractor, we are able to achieve a very high score of accuracy using the Random Forest Classifier with an accuracy of nearly 0.82. Just like for VGG16, this results is expected.

SVM with an accuracy of 0.86 seems again a very powerful technique in this context as we have already explained above.

It is interesting to notice here that all ML classifiers which are known as quite inefficient for overfitted data or for parameters dependencies (Decision Tree, Naïve Bayes) perform better than in VGG16. This is due to the fact that InceptionV3 uses very less parameters in comparison to VGG16. Therefore the features are unlikely dependent from each other which explains a better score for Naïve Bayes.

c. General Conclusion:

We have done a comparison study between to pre-trained CNN VGG16 and InceptionV3 as feature extractors pairing with various ML classifier. After analysis, it seems that we have the best result with VGG16 paired with the Random Forest Classifier (0.86 of accuracy). This can be explained by the fact that VGG16 exploits much more parameters than InceptionV3.

However, InceptionV3 has the advantage to offer a very good accuracy with much more less parameters (GPU consumption will be very less compared to VGG16).

Random Forest seems to be the ML classifier the most efficient given its strength to handle large parameters and overfitted data.

2) Section II: Fine-tuning Techniques

In this part, the task is to explore the application of fine tuning as a method of transfer learning for the Flowers dataset.

a. Quick code explanation:

The implementation of the code is quite straightforward:

- 1) A pre-trained CNN model is loaded with `tf.keras.applications`.
- 2) `model.trainable= False`: The model is set to non trainable
- 3) A new model is created on the basis of the previous one with extra layers which are added so that it is compliant for a 17 classes classification
- 4) `Model.compile`: The model is set for a specific optimizer.
- 5) `model.fit`: The model is trained and accuracy is displayed
- 6) `vggModel.trainable= True`: the pre-trained CNN model is set to trainable
- 7) `layer.trainable = True/False` : Selection of the layers of the pre-trained CNN which should be trained
- 8) The model is again trained (with a low learning rate).
- 9) The new accuracy of the model is checked

For more information on the code implementation, please, refers to the comments present in the ipynb file.

b. Result Analysis:

For the fine-tuning, we will consider the following experiments:

- 1) Experiment 1: CNN model VGG16 with SGD optimizer
- 2) Experiment 2: CNN model VGG16 with Nadam optimizer
- 3) Experiment 3: VGG16 Fine-tuning from block4
- 4) Experiment 4: VGG16 Fine-tuning from block3
- 5) Experiment 5: CNN model InceptionV3 with Nadam optimizer

1. Experiment 1: CNN model VGG16 with SGD optimizer

The idea of the experiment is to build a model based on the CNN VGG16 with extra convolutional layer. Here is the model built:

- VGG16
- Data in Flattened
- 2D Convolution of size 3x3 and with 256 filters (ReLU activation function)

- Drop out layer
- Activation Layer: Softmax activation with 17 classes

The VGG16 is set to non-trainable.

Therefore, only the convolution layer after VGG16 is trained.

For the experiment, for the compilation of the model, we use the SGD optimizer with a learning rate of 0.01.

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 12.

The result of this experiment is as follow:

```
CNN model VGG16, SGD optimizer accuracy: 0.8264706
```

The model reaches a very decent result of 0.82 of accuracy.

However, we used to have better result by changing the optimizer to Nadam (See Part A section III for the ensemble). An experiment should be carried on this.

II. Experiment 2 : CNN model VGG16 with Nadam optimizer

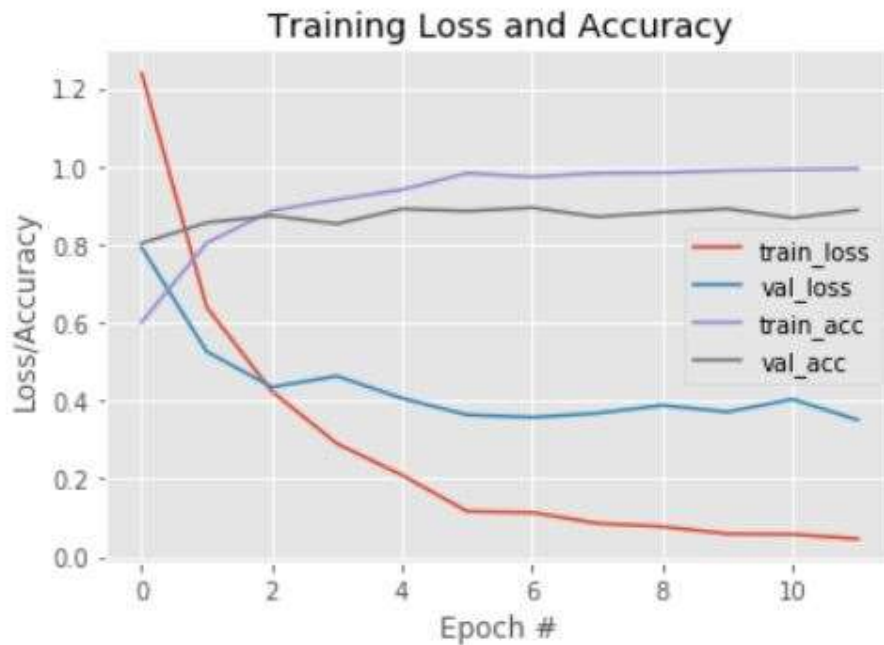
We use in this experiment the same model as presented in Experiment 1 The only difference is to use another optimizer during the compilation of the model. Indeed, this time we will use the Nadam optimizer with a learning rate of 0.001.

If we keep the exact parameters as in Experiment 1 (except for the optimizer), we get the following result :

```
CNN model VGG16, Nadam optimizer: 0.8882353
```

By changing the optimizer, we are able to increase the accuracy. We can get an accuracy of 0.88.

Moreover if we check the training and validation curve , we can see the following:



The training and validation curves for accuracy and loss does not show a big overfitting since the curves are not really divergent.

We can now try to proceed to the fine-tuning using the above model.

III. Experiment 3: VGG16 Fine-tuning from block4

In this experiment, we freeze all the blocks for training except the blocks from block 4. Therefore those following convolution layers will be trained:

- Block4_conv1
- Block4_conv2
- Block4_conv3
- Block5_conv1
- Block5_conv2
- Block5_conv3

we use again the Nadam optimizer but as it is a fine-tuning the learning rate is very low this time: $1e-5$

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 60. As the learning rate is low and as we are doing a fine-tuning, there is no chance that the model will overfit.

The result after 60 epochs is given:

At epoch 60: loss: $4.1054e-04$ - acc: 1.0000 - val_loss: 0.4280 - val_acc: 0.9118

The accuracy has increased from 0.88 to 0.91.

The fine-tuning from block4 was successful since we have a slight improvement in term of accuracy. However, it could be interesting to see if this accuracy can be improved a bit if we unfreeze the block3. This will be our next experiment.

IV. Experiment 4: VGG16 Fine-tuning from block3

In this experiment, we freeze all the blocks for training except the blocks from block 3. Therefore, those following convolution layers will be trained:

- Block3_conv1
- Block3_conv2
- Block3_conv3
- Block4_conv1
- Block4_conv2
- Block4_conv3
- Block5_conv1
- Block5_conv2
- Block5_conv3

we use again the Nadam optimizer with a low learning rate: 1e-5

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 60. As the learning rate is low and as we are doing a fine-tuning, there is no chance that the model will overfit.

The result after 60 epochs is given:

At epoch 60: loss: 8.4034e-04 - acc: 1.0000 - val_loss: 0.3229 - val_acc: 0.9235

Here is the final result of the fine-tuning:

```
CNN model VGG16, Nadam optimizer:  
Fine-tuning using Nadam optimiser from block3_conv1  
Accuracy after fine-tuning: 0.9235294
```

At the end of our fine-tuning process, we are able to get an accuracy of 0.92.

V. Experiment 5: CNN model InceptionV3 with Nadam optimizer

The idea of the experiment is to build a model based on the CNN InceptionV3 with extra convolutional layer. Here is the model built:

- InceptionV3
- Data in Flattened

- 2D Convolution of size 3x3 and with 256 filters (ReLU activation function)
- Drop out layer
- Activation Layer: Softmax activation with 17 classes

The InceptionV3 is set to non-trainable.

Therefore, only the convolution layer after InceptionV3 is trained.

For the experiment, for the compilation of the model, we use the Nadam optimizer with a learning rate of 0.001.

The model has been trained with a batch size of 51 and a number of iteration (number of epoch) of 12.

The result of this experiment is as follow:

```
CNN model InceptionV3, Nadam optimizer: 0.66764706
```

This model does not give a satisfying result. The accuracy is 0.67 which is very low compared to the VGG16 model where the accuracy was 0.88.

As a consequence, it is not necessary to continue to do a fine-tuning on this model since that there is a limited chance that a fine-tuning will improve much this result.

c. General Conclusion:

We have done an exploratory study to search the best accuracy we could reach with a pre-trained CNN completed with a fine-tuning.

This research has led us to get our best accuracy to 0.92 using the CNN VGG16 with a Nadam optimizer and a fine-tuning from Block3.

```
CNN model VGG16, Nadam optimizer:
Fine-tuning using Nadam optimiser from block3_conv1
Accuracy after fine-tuning: 0.9235294
```

Part C: Research

Deep convolutional networks have achieved exceptional levels of accuracy when applied to image related machine learning problems.

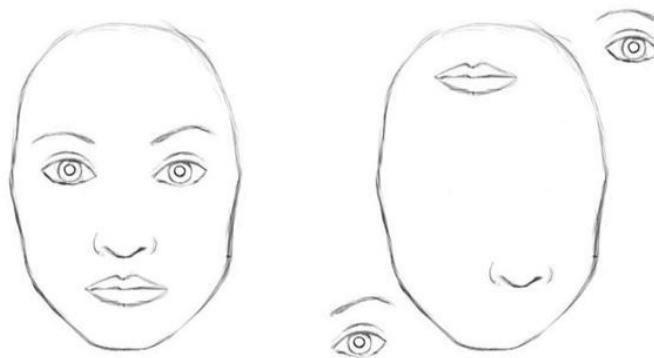
However, convolutional networks do have some significant limitations and vulnerabilities, some of which may undermine their long-term success and viability. Geoffrey Hinton, one of the leading figures in deep learning research, has argued that despite the success of CNNs they have some have significant disadvantages that may be difficult to solve.

One of the techniques which is proposed to overcome the CNN limitation is the **Capsule Neural Network** which is a category of an artificial neural networks proposed by Geoffrey E. Hinton in November 2017.

1) Capsule Neuronal Network context: Why Capsule Network?

Currently CNN (Convolutional Neuronal Networks) is widely used in various domains and has proven to be extremely powerful for making classification and prediction among all AI techniques in the field of Deep Learning. However, CNN has a flaw which lies in its architecture itself.

Indeed, a CNN is based on the detection of significant features in a data. Lower level of neurons detects basic feature (like edge or color) and upper level of neurons associate these simple features to get a more complex and broader features which will at the end be classified into categories to make a prediction out of the data. But in this process, information about spatial consideration between features is not taking into account. Therefore, an image containing all the features but disposed randomly in the space could fool CNN in its prediction. Typically, for a CNN, both images below could be identified as a face!



Source image from "A simple and intuitive explanation of Hinton's Capsule Networks", George Seif

Max pooling layer is usually used in CNN to try to have this "broader view" of the data. But in reality Max Pooling loses important information since again, it does not describe the hierarchical spatial relationship of the lower feature within the upper features.

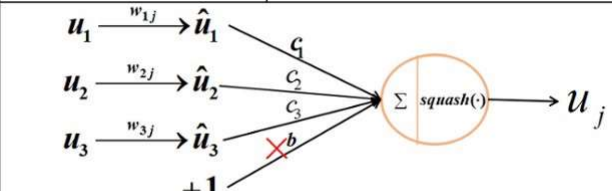
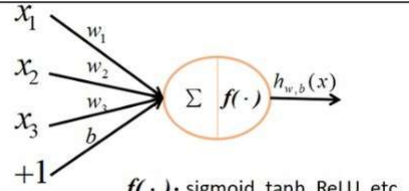
2) Capsule Neuronal Network Overview

In this context, Geoffrey E. Hinton proposes with the capsule network, a way to embed this hierarchical spatial information by providing interrelationship between capsules and better depict hierarchical relationships in a model.

Indeed, instead of relying on scalars for a feature (just like CNN does), the capsule network works with vectors.

Due to this subtlety, the steps to process feature information between a standard CNN and a Capsule network at neuron level is slightly different.

We can see below a table which illustrates the difference between the neuron activity in the Capsule network and in a regular CNN:

		capsule	vs.	traditional neuron
Input from low-level neuron/capsule		vector(u_i)		scalar(x_i)
Operation	Affine Transformation	$\hat{u}_{j i} = W_{ij} u_i$ (Eq. 2)		—
	Weighting	$s_j = \sum_i c_{ij} \hat{u}_{j i}$ (Eq. 2)		$a_j = \sum_{i=1}^3 W_i x_i + b$
	Sum			
	Non-linearity activation fun	$v_j = \frac{\ s_j\ ^2}{1 + \ s_j\ ^2} \frac{s_j}{\ s_j\ }$ (Eq. 1)		$h_{w,b}(x) = f(a_j)$
output		vector(v_i)		scalar(h)
				

Capsule = New Version Neuron!
vector in, vector out VS. scalar in, scalar out

Source table from "A simple and intuitive explanation of Hinton's Capsule Networks", George Seif

In a Capsule Network, the basic operation at neuron level of a regular CNN is kept just like Weighting, Summing and Non-linear activation function with a little transformation to support vectors. The real new addition in Capsule Network lies in the 1st operation: "Affine Transform". Therefore, the steps which are performed in a Capsule Network are:

1. **Affine Transform:** matrix multiplication of input vectors
2. **Weighting:** scalar weighting of input vectors
3. **Summing:** sum of weighted input vectors
4. **Activation:** vector-to-vector nonlinearity

We will go through these steps to explain the operations.

d. Affine Transform:

This step ensures the matrix multiplication of all the input vectors as illustrated in the table with the following equation:

$$\hat{u}_{j|i} = W_{ij}u_i$$

This is an important step in the Capsule network since it is in this step that is encoded the hierarchical spatial information.

In the equation, the element u_i corresponds to input vectors which comes from previous neurons. The length of those vectors represents the probability of a feature detection at lower level (example nose in a face). In short, the vector length is basically identical to the scalar in a CNN neuron. But the direction of those vectors shows one more indication: It indicates the internal state of the detected features its orientation and size.

These vectors are multiplied to W_{ij} which is a matrix encoding spatial and relationship between low level feature and the current level feature.

After multiplication, we get a new vector $\hat{u}_{j|i}$ which represents the predicted position of the current level feature based on the position of the lower level feature.

e. Scalar Weighting and Summing:

Those 2 steps are described in the table with this equation:

$$s_j = \sum_i c_{ij} \hat{u}_{j|i}$$

Basically, the previous calculated vectors $\hat{u}_{j|i}$ is multiplied with the weight c_{ij} . The result s_j will be transformed and scaled to generate the output vector of the capsule network.

The capsule has also to adjust the weight c_{ij} to determine to which higher level capsule its output has to be sent: This is the concept of dynamic routing based on agreement which is another innovation of Capsule Network.

f. Activation:

In this step, capsule network introduces another non-linear activation function: the “squash” function in order to support vectors instead of scalars:

$$V_j = \frac{\|s_j\|^2}{1 + \|s_j\|^2} \frac{s_j}{\|s_j\|}$$

The squash function is a non-linear function using the above calculated vector s_j to create another vector V_j which will be the output of the capsule.

You can notice that V_j length will not exceed 1 thanks to the scaling done in the function and its direction is kept relatively to the vector s_j .

This non-linear function is also involved in the process of the dynamic routing based on agreement.

g. Dynamic Routing based on agreement:

The Dynamic Routing based on agreement is an important concept in Capsule Network since it allows to feed the upper level capsule with appropriate features and participates by doing so to give the notion of spatial hierarchy between lower capsules and upper ones.

Let's have a look to the pseudocode of the routing algorithm:

Procedure 1 Routing algorithm.

```

1: procedure ROUTING( $\hat{\mathbf{u}}_{j|i}, r, l$ )
2:   for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow 0$ .
3:   for  $r$  iterations do
4:     for all capsule  $i$  in layer  $l$ :  $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$  ▷ softmax computes Eq. 3
5:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$ 
6:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$  ▷ squash computes Eq. 1
7:     for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$ 
   return  $\mathbf{v}_j$ 

```

Pseudocode from original paper “Dynamic Routing Between Capsules”, Geoffrey E. Hinton

In this pseudo code, we can see that routing algorithm is performed for all capsules at a particular layer l and the higher level $l+1$.

- 1) in line 3, the process is iterated r times (where r is the number of routing iterations)
- 2) In line 4, all capsules at current level l has their weight c_i affected. Due to the softmax function, this weight has a positive value within 0 to 1 and the sum of all of them is 1.
- 3) In line 5, the vector \mathbf{s}_j is computed for scalar weighting as described in part C-2-b.
- 4) In line 6, the output vector \mathbf{v}_j is computed using the new nonlinear activation function “squash” (see part C-2-c)
- 5) In line 7, the new value of the weight b_{ij} (which will be, later on, affected to c_{ij} on line 4) is updated based on the dot product of $\hat{\mathbf{u}}_{j|i}$ and \mathbf{v}_j . Typically if vectors $\hat{\mathbf{u}}_{j|i}$ and \mathbf{v}_j “goes” to the same direction then the product will be high otherwise the value will be low.

At the end of the iteration, the output vector \mathbf{v}_j of the capsule is returned.

As we can see the whole purpose of the routing algorithm is to update the weighting coefficient c_{ij} in order to give more importance to predicted vectors $\hat{\mathbf{u}}_{j|i}$ which is “going” to the same direction as per expectation of upper capsule (weighting W_{ij} in the Affine transformation – see Part C-2-a). As a result, the output vector \mathbf{V}_j will have a small or big length value (from 0 to 1) depending to which upper-level capsule it goes.

This whole process introduces spatial hierarchy in the neuronal network and gives an advantage from CNN.

3) Conclusion

Capsule networks seem quite promising since it has introduced the notion of spatial hierarchy in CNN. It has shown good result for MNIST and medical image datasets. However, its performance on CIFAR-10 was not satisfying. A deeper research on this technique has to be done to improve its performance.