CSC 450 – Honors Research Project

Student: Dennis Krupitsky

dennis.krupitsky@gmail.com

Mentor: Dr. Natacha Gueorguieva

Deep Learning Image Recognition and Detection: Architectures, Learning and Applications

December 2019

Deep Learning Image Recognition and Detection: Architectures, Learning and Applications

Abstract

Development of machine learning algorithms allows for solving classic problems such as image classification, detection, and recognition. Utilizing the ability of Deep Learning algorithms, the construction of Deep Neural Networks is possible, which will allow learning of powerful features from huge amounts of data by extracting features of the data layer by layer. The goal of this research is to propose and develop different solutions using Convolutional Neural Networks, by experimenting with different training approaches including batch training, gradient and stochastic gradient descent methods and different activation and loss functions, augmentation, pooling and dropout. Experiments done within this paper use the Flowers data set from Kaggle, which are plotted and analyzed in order to see evaluate the performance of the different applications and architectures using validation procedures.

Keywords— Machine learning, neural network, deep learning neural networks, CNN, gradients, optimizers, activation functions, K-fold cross validation, supervised learning

Table of Contents:

1. Introduction to Deep Learning:

With machine learning being an ever-growing, popular field within the artificial intelligence world, a subset of machine learning is **Deep Learning.** This specific subset is a learning technique for computers involving algorithms, and neural networks inspired by the human brain to learn from huge amounts of data. The process of creating a model could grow quite extensive as there are many facets to account for. A simple definition for a deep learning model can be described as an algorithm repeatedly performing a task, and each time have certain tweaks in order to improve the outcome. The "*deep*" within deep learning is a reference to the number of successive layers of representations, which could also he described as the *depth* of the model. A model could range from one to hundreds of successive layers, all learned during the exposure to training data. Deep learning has only come to surface as one of the most useful AI techniques in the last few decades, as we now have access to large amounts of labeled data for training (over 2 quintillion bytes of data is generated daily), and substantial computing power to train our models. Overall, deep learning allows modern machines to solve complex problems, by learning from experience.

1.1. Purpose and use of Deep Learning

Deep learning is one of the areas that has attracted a lot of attention due to its potential for real world applications. It is widely used in real life applications, such as image classification, aerospace and defense, medical research, self-driving cars, robots, etc. There are several forms in which this type of machine learning can be trained. One allows for a training method with data that is pre-labeled, and through training the model is comparing the label it assigned the data to the actual label, to see if its prediction was correct or not, also known as supervised learning. There is tons of data that is collected every day, but most of this data is not labeled, so we are not able to use it in supervised learning. This is where unsupervised learning comes in to play, we are still able to show the data to our deep learning networks, and it will learn to identify the data's label.

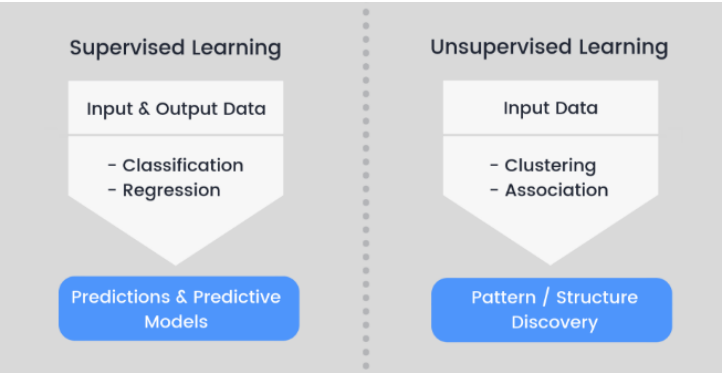


Fig 1. Supervised vs Unsupervised learning

These networks can be successfully applied to huge amounts of data for knowledge discovery, application of this knowledge, predictions based off the knowledge, etc. Deep learning is used to create actionable results. Deep learning allows us to advance and innovate within the real world, and is one of the most powerful aspects of machine learning.

1.2. Neural Networks:

Deep learning got its name for another reason, more specifically due to the neural networks it is comprised of have various deep layers that enable learning. A neural network is a set of algorithms, which as stated earlier are based off the human brain, which we design in order to recognize patterns, whether it be in images, text, etc. Neural networks assist us in clustering and classifying data. These neural networks are a set of layers that are stacked on top one-another that adjust to the properties of the training data.

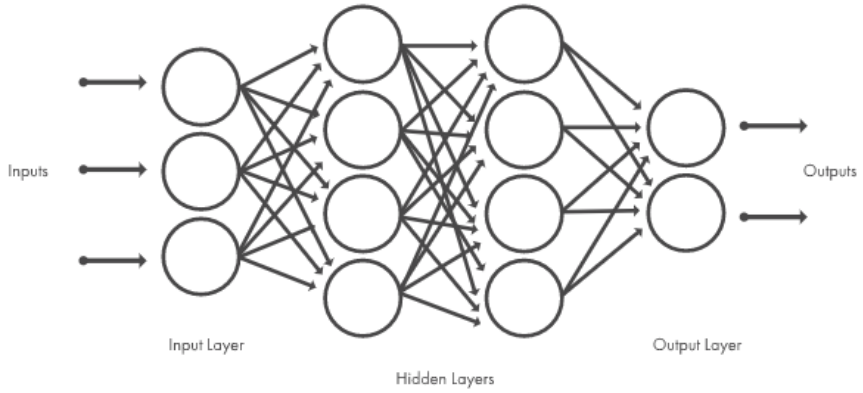


Fig 2. Representation of a Neural Network

The input layer is the initial data being brought in that will be used by the neural network. The hidden layers within neural networks make this one of the superior machine learning algorithms. These hidden layers are not visible to external systems, and are private to the neural network. The amount of these hidden layers can also range from zero to hundreds. Each hidden layer is comprised of neurons that receive an input from the previous layer, and does some sort of manipulation or transformation to the data before sending it to the next layer/neuron. The final layer in the flow chart is the output layer, which produces the result for the original inputs. Essentially each layer, start at the hierarchy, is combining information into something more and more complex, depending on the number of layers you utilize. Each node in a layer relates to each node in the following layer, and each arrow in the connection holds a certain weight. This could also be perceived as the impact that the node has on the next layers node.

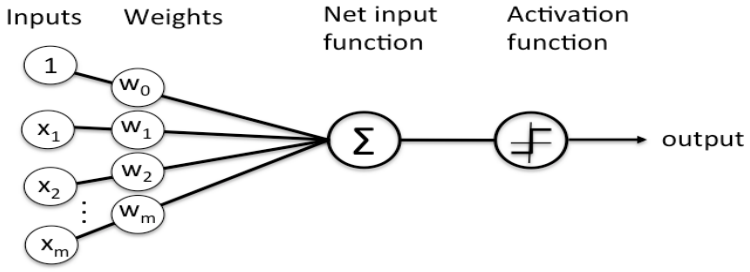


Fig. 3 Representation of a single node/layer

1.3. Data Preprocessing

There is a crucial step that is taken before the training of a machine learning model, which helps improve both the quality of our data, and result of the model. This step is called data preprocessing, a data mining technique used to transform raw data into usable formats. Within it there are several steps, such as data cleansing, data transformation, data distribution, etc. These steps allow our data to take a form in which they can create a model. We use this in order to combat data inefficiencies which could have negative effects on our experiments, such as inaccurate data, noisy data, inconsistent data, etc. If this step is skipped over, there is a possibility that a percentage of the results will be false.

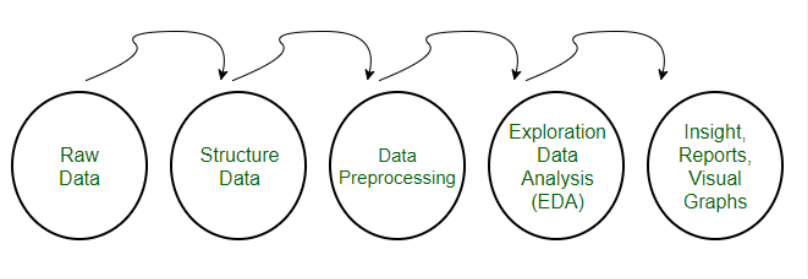


Figure 4: Data Preparation

Some of the widely used techniques include removing null data, rescaling data, standardizing data, binarizing data, and label encoding. Missing or null values should be handled properly during data preprocessing, in order to avoid altered results that will differ from the proper data. Rescaling data is the process of converting data that is comprised of attributes with varying scales, into common values which range between 0 and 1. This method is very useful in optimization algorithms. Standardizing data allows the transformation of values with a Gaussian distribution of differing means and of differing standard deviations into a standard Gaussian distribution that has a mean of 0 and standard deviation of 1. The formula looks as following:

$$S'_j = (S_j - \text{mean}(S_j)) / \text{std}(S_j))$$

Binarizing the data allows all values over the threshold to be marked 1, while all equal to or below the threshold are marked as 0. This method is useful when dealing with probabilities, as it allows the data to be transformed into crisp data. Normalization of data is the process of scaling individual samples in a dataset to have a unit norm between 0 and 1. Normalizing data can be useful when data has input values with differing scales, and is even required by some algorithms when creating neural networks. It requires the estimate of the minimum and maximum observable values of data. The formula looks as following:

$$X_{new} = (x - x_{min}) / (x_{max} - x_{min})$$

Label encoding transforms data labels, which are usually labeled with words in order to make it readable, into numbers or binary labels for the algorithms to be able to work with them. Another crucial step is to distribute the original collection of raw data into separate sets, more specifically training, validation, and testing sets. The training dataset contains data that will be used to train the model, as the model sees and learns from this data, therefore this set should have the biggest ratio of data. The validation dataset is used to evaluate a given model during its training, this data is used to fine-tune the hyperparameters, this set should have slightly more data than the testing from the remaining available data after the training set is allocated. The testing set is used to fully evaluate the model, after it has completed training (using test and validation sets). It should receive the remaining undistributed data. It is good practice to not have 2 sets containing the same data. Once the data has been preprocessed visual graphs, and reports are generated for the researcher to get a sense of the altered data's values.

2. Convolutional Neural Network Structure

One of the main components of this image processing research experiment involves convolutional neural networks. A convolutional neural network (CNN) is a deep learning algorithm which is comprised of neurons that can take an image as input, assign certain weights, and biases to the aspects/objects within the image, and successfully differentiate from one another. Like a standard neural network, a CNN also consists of an input layer, an output layer, additionally adding on pooling layers, convolutional layers, normalization layers, fully connected layers, etc. CNN architectures make the explicit assumption that the input it is going to be receiving will be images, which then allow for it to encode certain properties into the architecture. Allowing for the architecture to be focused to a certain type of data allows for an increase in efficiency for image processing results. There are a few differences between CNNs and regular neural networks. As mentioned earlier a neural network will transform an input by sending it through a series of hidden layers within the model. These hidden layers are of course comprised of sets of neurons, where the layers are connected to the layer preceding it. A main difference in Convolutional Neural Networks is that the layers within it are organized into 3 different dimensions of width, height, and depth. Another difference is that the neurons within a layer do not all connect to neurons in the next layer, but instead only to a small region of it. Instead within this 3-dimensional structure, each set of neurons analyzes is set to analyze a specific region of the image. Within the figure pictured below the red input layer represents the image, with the width and height being the dimensions, and the depth having a value of 3 (Red, green, blue) channels.

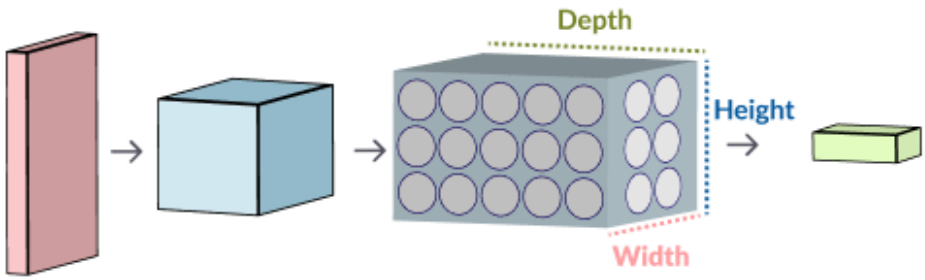


Fig. 5: Convolutional Neural Network Structure

2.1. Convolutional Layers

The first layer following the input layer is the convolutional layer that allows a feature map to be produced. Convolution is a mathematical operation on two function that will produce a third function. The purpose of this layer is to be able to figure out certain features that an image contains, for instance, the vertical/horizontal edges, gradients, etc. Input to this layer is the (m x m x r) image, where m represents the height and the width, and r is the depth or number of channels, usually this input will be as an array of pixel values. This layer will also define a filter/kernel of size (n x n x q), where n is defined to be smaller than the dimension of the image, and q is defined to have the same channels r. Essentially this convolutional layer allows the filter to slide across the input, and at every location a matrix multiplication will occur and then sums the result onto the feature map with the process then repeating for every location on the input volume **(Fig. 7)**. Within a CNN, there could be several Convolutional layers of varying kernel sizes.

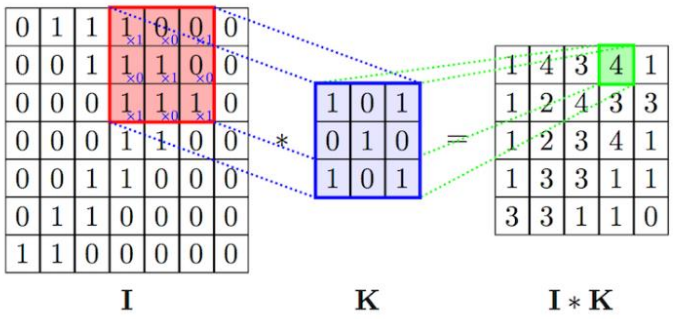


Fig. 6: Feature Map

2.2. Pooling Layers

Following a convolutional layer it is usually common to add a pooling layer in between CNN layers. It serves the function of reducing the size of the matrix in order to reduce the amount of spatial size produced from the convolved feature map. This allows for a decrease in the number of parameters and computation power required to process the data. The most common used form of pooling is max-pooling. Max pooling takes the maximum value amongst each kernel of the feature map, which in turn allows the feature map to decrease in size (**Fig. 8)**, but also retain the significant information. Max pooling also serves as a noise suppressant, because it discards the noisy activations while also performing dimensionality reduction. Similar to convolutional layers, you can include several of these pooling layers within your network, and as a result there will be deeper extraction of features within the images.

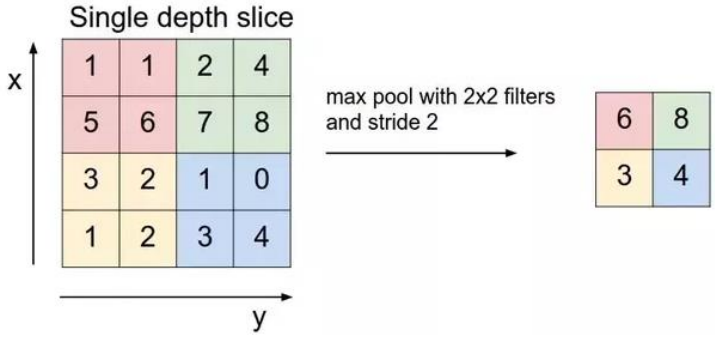


Fig. 7: Max Pooling

2.3 Batch Normilization

Another layer that is often included within CNNs, and used within this experiment is a batch normalization layer. This normalizes each of the input channels across a mini batch by adjusting and scaling the activations. Additionally, batch normalization allows for each individual layer of the network to learn by itself a bit more independently from other layers. During training of the network, any activation and distribution changes within a layer due to the changing weights and biases will cause rapid changes in the layer above it, and cause training to slow down. As an example, when there are features that could range from 0-1, and then features that could range from 1-1000. The normalization of these values will allow of an increase in speed during training. Along with speed it also reduces the sensitivity of network initialization when training convolutional neural networks. Batch normalization is usually placed between each convolution layer within the network. Including batch normilization is always good, as it serves as almost a preprocessing step at every layer within the network. The formula to calculate the normalization values goes as following:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

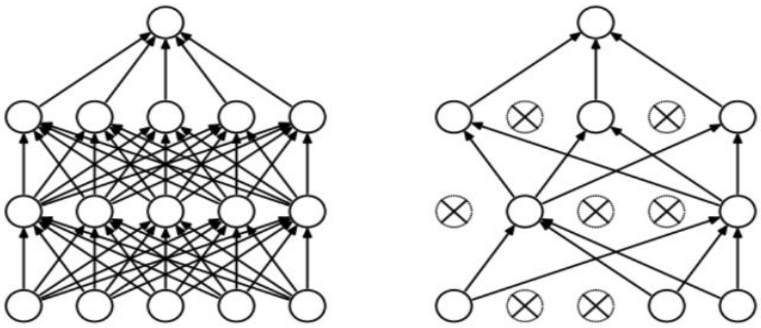
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

2.4. Dropout

Dropout is one of the most effective and common used regularization methods used within machine learning. It is applied to a layer, and consists of dropping out (changing value to 0) a number of output features of that layer (Fig. 9). The rate of dropout is the fraction of features that are to be zeroed out, which is usually set to be between 0.2 and 0.5. Dropout also effects the testing, but instead of zeroing values out, the layers output values will be scaled down by a factor with the same value as the dropout rate. Dropout is a great approach to regularization in neural networks, as it helps reduce interdependent learning amongst the neurons.



(a) Standard Neural Net (b) After applying dropout.

Fig. 8: Dropout Effect

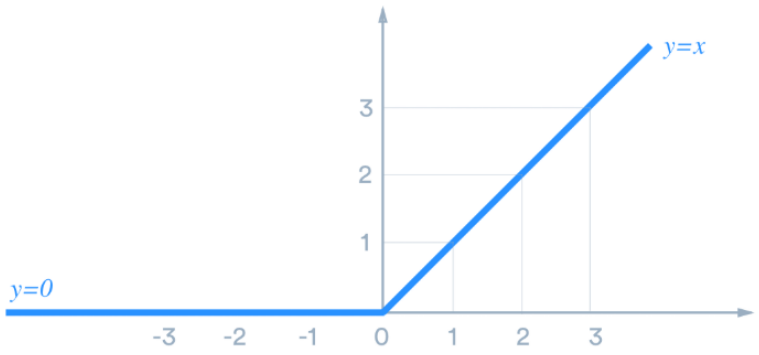
3. Algorithms and Functions in CNN

3.1. Activation Functions

Activation functions, also known as transfer functions, are very important to an artifical neural network in order for it to be learn and make sense of non-linear complex function mappings between the input and the output variables. These functions allow the introduction of of non-linear properties to the network. They serve a main purpose of converting input signals of a node into an output signal to be sent along to the next layer. It essentially decides whether a neuron should be "activated" or not, in other words it decides if the information being received is relevant to the for the model's prediction. Without activation functions, a neural network would simply be a linear regression model. In the case that there is no activation function, the model would not be able to solve complex problems, such as image classification or language translations. That would steer from the main goal of the nerual network, which is to learn complex non-linear functions. Without non-linearity, the seperation of classes is attempted to be done with a linear hyper plane. The initial input to the neuron is a weighted sum of the previous inputs plus the bias. The bias value allows the activation function to be shifted to the left or right , in order to better fit that data. It could also be though of as how likely a node will "fire" off. The linear operation being performed is the multiplication of the weight, addition of bias and summation across all inputs that arrive to the neuron. In some cases, the output of this operation could be very large, and as it is fed to more layers it will grow larger and larger, causing computation issues. This is where the activation comes in play, as it will transform to a fixed interval. The general activation function operation looks like this:

$$\mathbf{Y = activation(\Sigma\ (weights * input) + bias)}$$

There are many different activations functions that are used, such as Sigmoid, Tanh, Parametric, Softmax etc. The one that is most widely used within CNNs and also used in this expirement is **Rectified Linear Unit (ReLu).** Mathematically it is defined as **y = max(0,x),** and is represented as:



ReLu differs from other activation functions as it does not acitvate all the neurons simultaneously. As you can see in the graphic, negative inputs get converted to zero, and the neuron does not get activated. Positive data is left unmodified, which means it does not have any constraints on the values, so the range spans from [0, ∞). This allows computational effeciency, as only a few neurons will be activated at a time, thus introducing sparsity within the model. Sparsity is useful to have within a model, as you do not want certain neurons to fire off when they are unecessary. Within a sparse network it is more likely that the neurons are being utilized to process meaningful aspects of the problem at hand. Additionally, computational effeciency allows less time for training and running. Both the original function, and its derivative are monotonic (varying in such a way that it either never decreases or increases). A disadvantage of ReLu is that on range from (−∞,∞), half of the gradient is always 0, and could lead to a bias shift due to it not being centered around 0. Batch normalization is used to avoid this issue as it normalizes the inputs with zero mean and variance.

3.2. Optimizers

During the training of a CNN model, we tweak and change the parameters in order to minimize the loss function of the model, so our predictions will be as accurate as possible. This is done by using an optimizer, as they tie together a models loss function and its parameters by updating the model in response to the output of the loss function. In another view, optimizers serve the purpose of shaping and molding your model into the most accurate form by manipulating the weights. The loss function tells the optimizer when it is moving in the right or wrong direction. It is impossible to know what the model's weights should be right away, so the optimizer allow for trial and error based on the loss function to reach the correct weights. The optimizer implements a specific variant of stochastic gradient descent. The gradient descent algorithm is used across all types of machine learning. Gradients represent the effect a small change within a weight or parameter would have on the loss function. They are partial derivatives, and a measure of change. The algorithms used to optomize gradient descent are referred to as black-box optimizers, seeing as the explanations of their strengths and weaknesses are hard to come by. The formula for paramter updates is as following: $\theta=\theta-\eta\cdot\nabla J(\theta)$, where η is the learning rate, $\nabla J(\theta)$ is the gradient of loss function. Momentum is another learning method that helps accelerate gradient descent in the relevant direction and dampens oscillations. It achieves this by adding a new variable γ of the update vector of the past update to the current update vector. It is represented as: $vt=\gamma vt-1+\eta\nabla\theta J(\theta)$. Momentum speeds up learning by increasing learning when the gradients point in the same direction, and slow learning down when the gradient is oscillating vigorously.

The optimizer used within this research is **Adaptive Moment Estimation (Adam)**. It is an optimization algorithm that can be used instead of classic stochastic gradient descent to update network weights. Adam utilizes the concept of momentum by keeping track of the decaying average of past averages in order to scale the learning rate. This algorithm implements an exponential moving average of the past gradients, and the past squared gradient. It works by first updating the exponential moving averages of the gradient (call that m_t), and the squared gradient (call that v_t) which is estimates of the first and second moment. Hyper-parameters $\beta1, \beta2 \in [0, 1)$ control the exponential decay rates within the formulas of the moving averages as shown:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance). As these moving avergaes are initilialized it was noticed that they are biased towards 0, especially when the decay rates are small. In order to counteract these biases, the first and second moment bias-corrected estimates are calculated as following:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the parameter is then updated once the bias-correction has occured, it is calculated as following:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

Adam is computationally efficient, and has become one of the most popular gradient descent algorithms. It compares favorably to other adaptive optimizers as it rectifies problems, such as vanishing learning rate, slow convergence or high variance, etc.

3.3. Confusion Matrix

A Confusion Matrix (CM) is among one of the metrics that is used to measure the performance of a classification algorithm on a set of test data, for which the actual values are known. It is an N x M matrix, where each row represents the true classification value of a piece of data, and each column represents the predicted classification (could be vice-versa). When looking at the confusion matrix for a multi-class classification problem, the accuracy of the model can be determined by looking at the diagonal values, to evaluate the number of correct classifications (Fig.10). A model with good accuracy will have high values along the diagonal and low values off the diagonal. There are 4 values within a CM, they are true positive (TP), false positive (FP), true negative (TN), false negative (FN). Besides accuracy, a confusion matrix also allows us to compute several other performance measures, such as precision, recall, f1-score, etc. Precision calculates the amount instances that were correctly predicted, the formula is: **P = TP/ (TP + FP)**. Recall calculates the proportions of actual positives that were identified correctly, the formula is: **R = TP/ (TP +FN)**. The f1-score is the harmonic mean of precision and recall, the formula for that is: **F1 = (P * R) / (P + R).** Analyzing a confusion matrix will allow for a sense of where your classifier/model is predicting incorrectly. A multi-class confusion matrix will usually look something like:

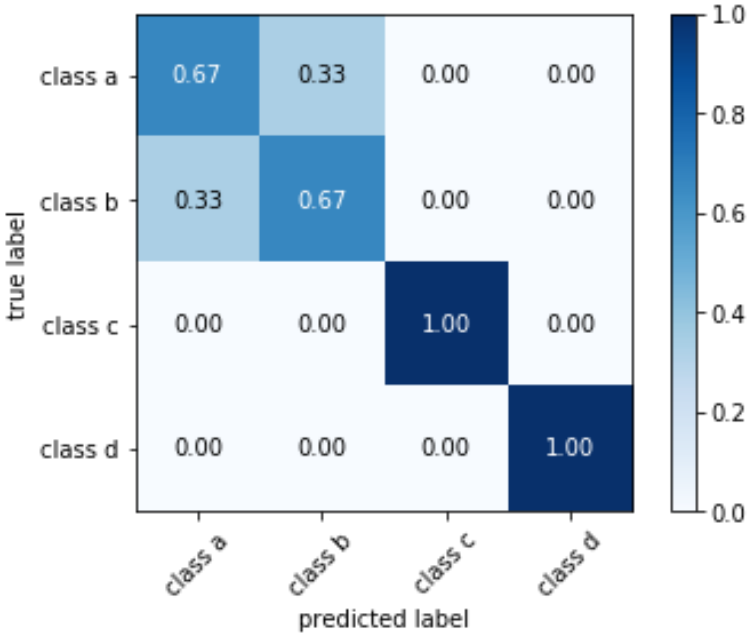


Fig. 9: Confusion Matrix

3.4. K-Fold Validation

K-Fold validation is the process of evaluating machine learning models on a limited data sample, by splitting a given data set into a K number of sections/folds. The value of K will determine how many sets to be split into. This sort of validation is primarily used in machine learning to estimate the skill of a machine learning model on unseen data. In other words, it is used to see how the model will perform when making predictions on data that was not used for the training of the model. The general procedure works by splitting the dataset into equal K splits, then using K-1 splits to train the neural network, and then using the remaining set is used to test the model. This method is performed K times, as we need each of the splits to serve as the testing split. Once this is complete, the average performance metrics (accuracy, error, accuracy during training) give the overall performance of the model. A good representation of K-Fold validation is as follows:

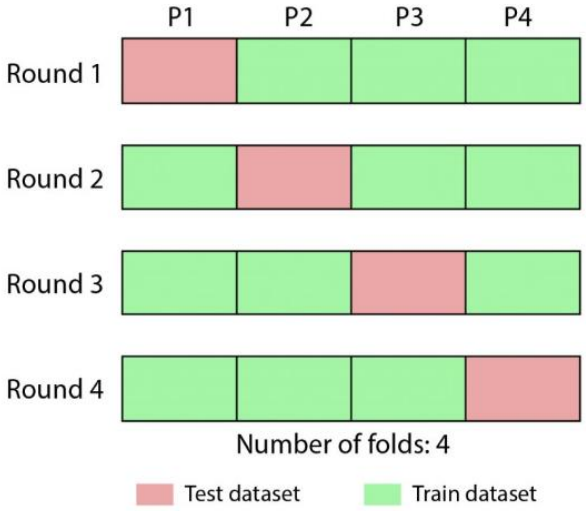


Fig. 10: K-Fold validation with K=4

3.5. Imbalanced Datasets

When training a machine learning model, it is likely to come across a dataset that has an unequal distribution of classes. At times this imbalance could be a small percentage, and at other times it could be a sizeable difference. Learning from datasets where there is a large imbalance between the sets of data will lead to the training of a model which is bias and inaccurate, due to insufficient exposure all the classifications. Machine learning algorithms are designed to improve accuracy by reducing error, so they do not take class distribution/proportion into account. To combat the inefficiencies that arise due to distribution imbalance, there are several data level techniques that can be used. The first of the data level techniques is a resampling technique called oversampling. Oversampling refers to increasing the number within an under presented class in the dataset. The process of oversampling involves generating synthetic data that tries to randomly generate a sample of attributes from observations made in the minority class. There are several techniques that could be used to oversample a dataset for classification problems, and the most common used is **SMOTE** (Synthetic Minority Over-Sampling Technique). The instances created from SMOTE are not copies of existing minority case, but instead this algorithm takes samples of the feature space for each of the target classes and its nearest neighbors, and then generates new samples that combine features of the target and its neighbors (Fig. 12). The input to SMOTE is the entire dataset, but only increases the percentage of the minority cases. Another resampling technique that could be used is undersampling. This technique does the opposite oversampling, it is the process of removing a subset of

samples within a majority class based on observations in order to match the minority class. It is a great way to balance your dataset, but you risk potentially losing relevant information from the samples that are left out.

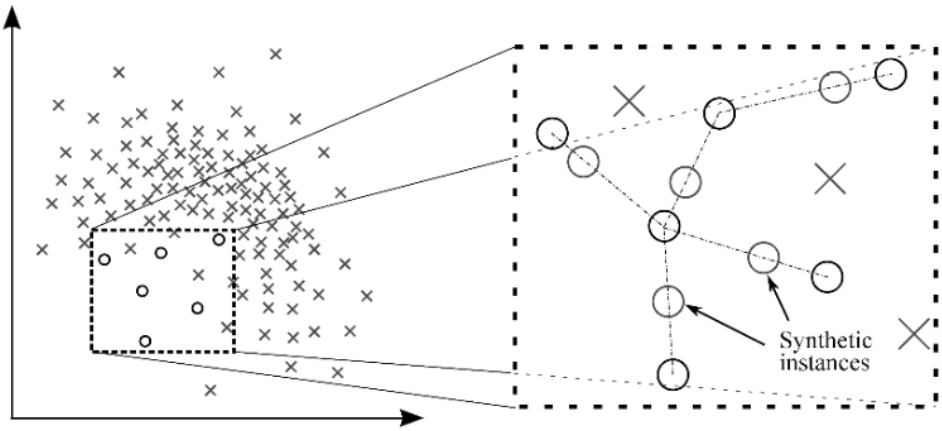


Fig. 11: Generation of Synthetic Instances using SMOTE

3.6. Experiments for overfitting

Overfitting is a problem in machine learning models occurs when the error on training sets are driven to a very small value, but when exposed to new data within the testing set the error is large. The goal for a machine learning model is to generalize well from the training data to any similar data in the problem domain. The model is intended to make predictions on data that it has never seen before. There are several approaches to prevent overfitting. One of the approaches is to simplify the model, in other words to remove layers or reduce the number of neurons within the neural network in order to make it smaller. Size of a neural network cannot be pre-determined, but if a network is overfitting, reducing the size of it could help combat that issue. An additional approach is to stop the training of a model earlier. If it is visible that after some iterations that the testing error begins to increase while the training error is still decreasing, training of this model should be stopped at the point where this begins to happen. Another great approach, which is also used within this research is Data Augmentation. This method refers to increasing the size of the data that is being utilized for training, it is not done using new images, but instead it is the alteration of existing images for new copies. Techniques used for augmentation involves cropping, padding, flipping, translation, rotation, etc. With addition of new samples, the model is unable to overfit, but instead is forced to generalize.

4. Experimentation

4.1. Dataset Information

The dataset used within the experiments is the **Flowers Recognition** dataset from Kaggle. The details of the original dataset with 4,326 photos. There are five classes within this dataset: daisy, dandelion, rose, sunflower, and tulip. The splits of the data used in the experiment are presented in the table below:

	Daisy	Dandelion	Rose	Sunflower	Tulip	Total
Training	499	685	509	476	638	2807
Validation	154	211	157	147	197	866
Testing	116	156	118	111	149	650

4.2. Experiment Details

The experiments were designed to evaluate the classification results on a medium sized dataset by using different approaches, such as data normalization, data augmentation, adaptive learning rates, data resampling, in order to analyze what applications and architectures would produce the most accurate results. The number of hidden layers for each experiment is 4, each having a convolutional, pooling and batch normalization layer. The batch size takes values of (128,128,3). The activation function used is ReLu, and the optimization is Adam with a learning rate of 0.001. Dropout with a rate of 0.5 is also used within the models. Models are trained over 50 epochs, with steps per epoch being calculated as **S = 2 * ((length of training set) / 128**

The three experiments done are:

1. **Data Augmentation and Normalization with adaptive learning rate**
2. **Data Oversampling with adaptive learning rate**
3. **Data Augmentation and Standardization**

4.3. Experiment Results

1. Data Augmentation and Normalization with adaptive learning rate

The data used to train and test this model is augmented in order to increase the data size, and try to get a more even distribution of data. It is also normalized in order to scale the individual samples to have a unit norm between 0 and 1. The learning rate for this model changed from 0.001 to 0.0003 at Epoch 15, and to 0.0005 at Epoch 30. After augmenting the dataset with splits of testing: 15%, training: 65%, and validation: 20%, the data distribution for this experiment are displayed in the following table:

	Daisy	Dandelion	Rose	Sunflower	Tulip	Total
Training	1450	1350	1478	1378	1253	6909
Validation	460	420	466	435	390	2171
Testing	347	310	350	330	297	1634

Model Accuracy Metrics:

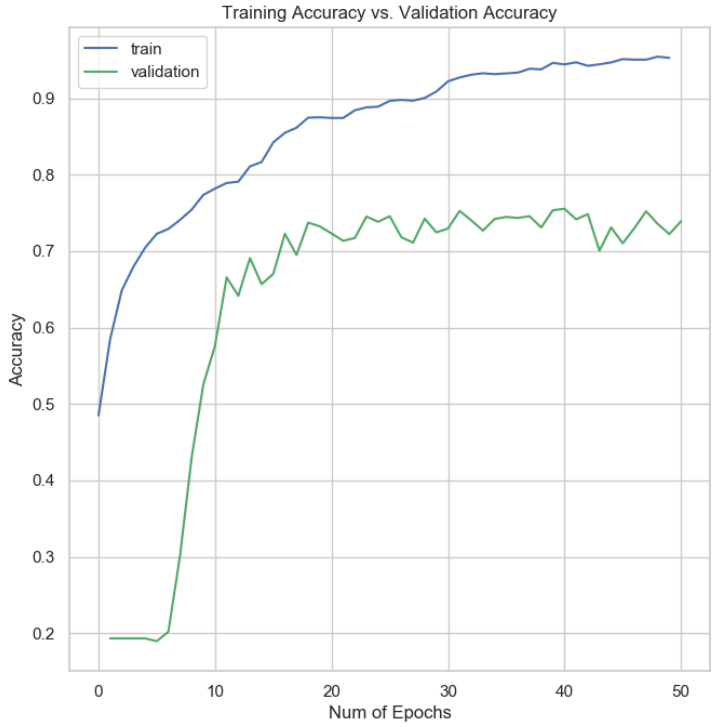


Fig. 12: Experiment 1 - Accuracy

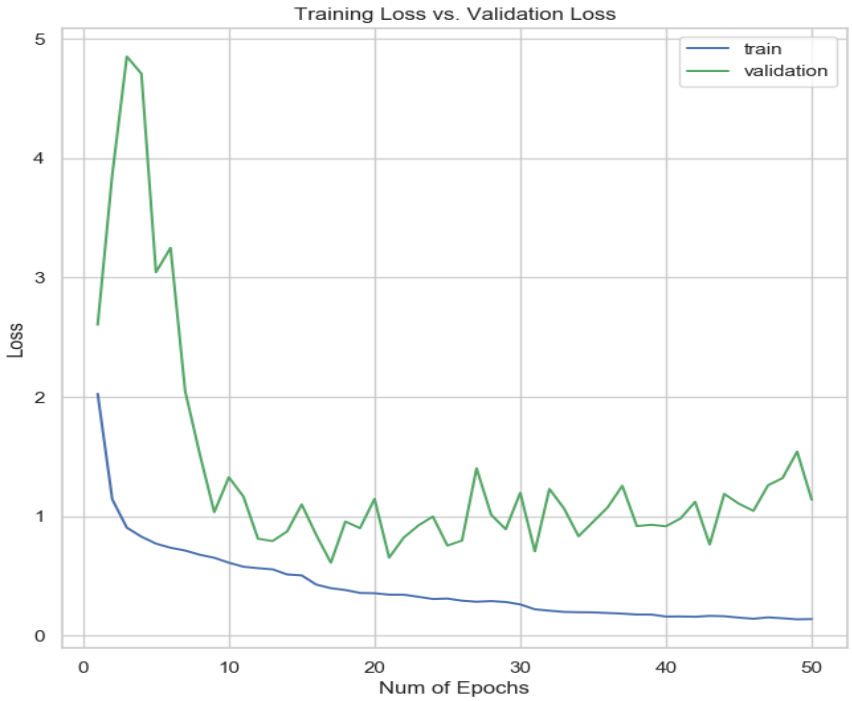


Fig. 13: Experiment 1 - Loss

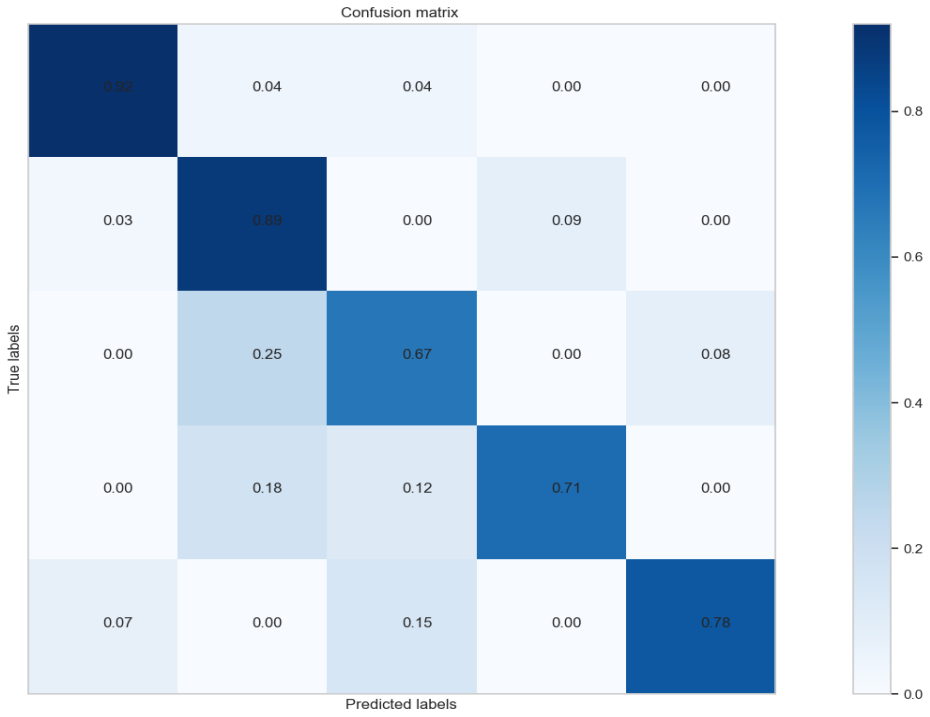


Fig. 14: Experiment 1 - Confusion Matrix

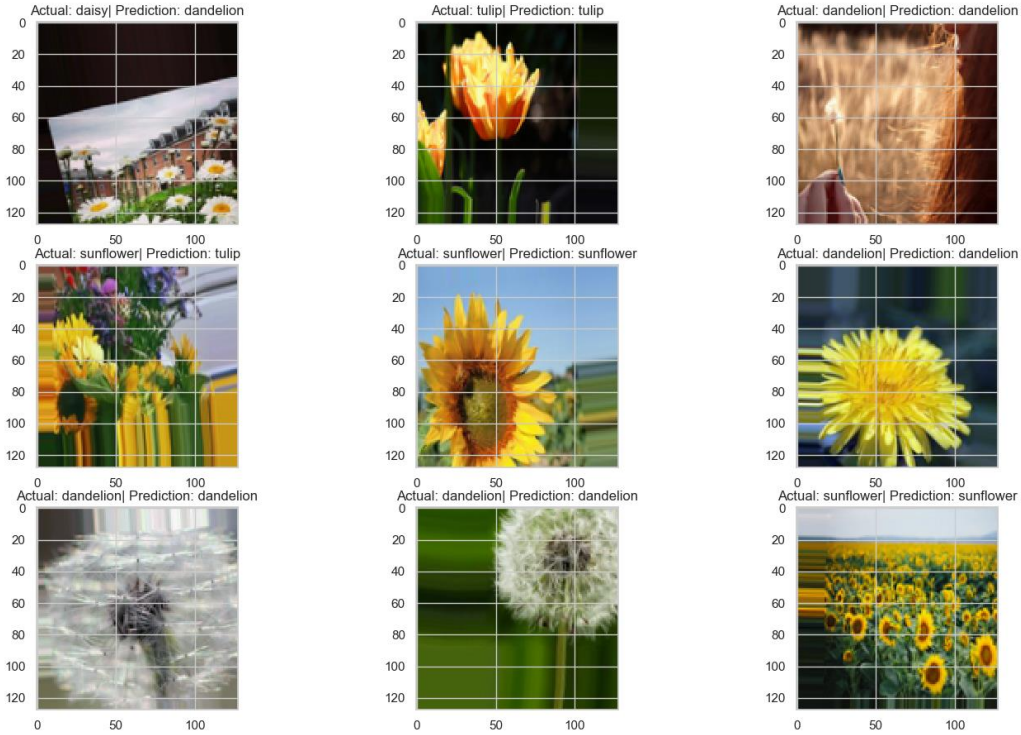


Fig. 15: Experiment 1 – Model Predictions

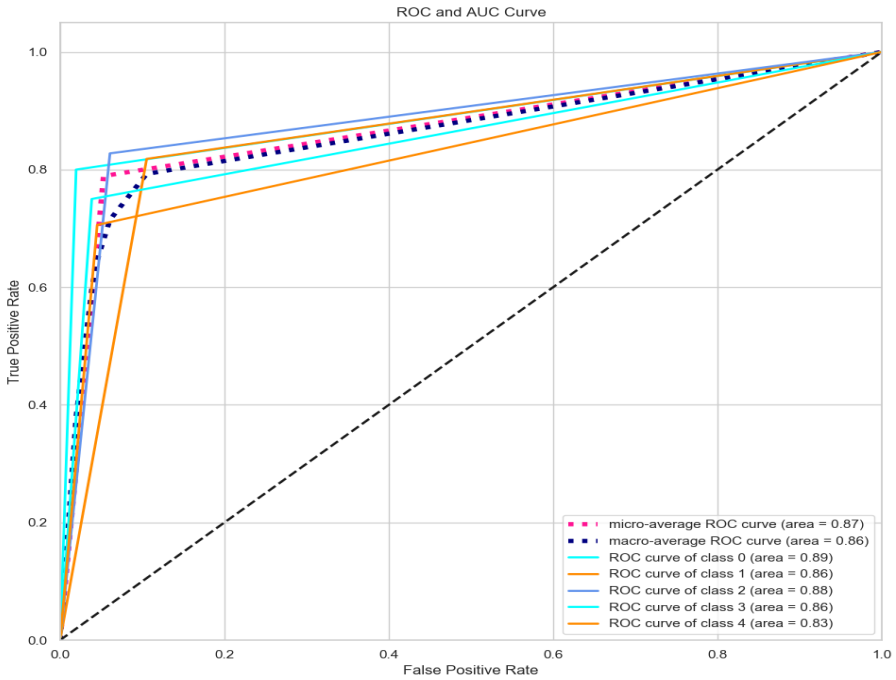


Fig. 16: Experiment 1 - ROC-AUC CURVE

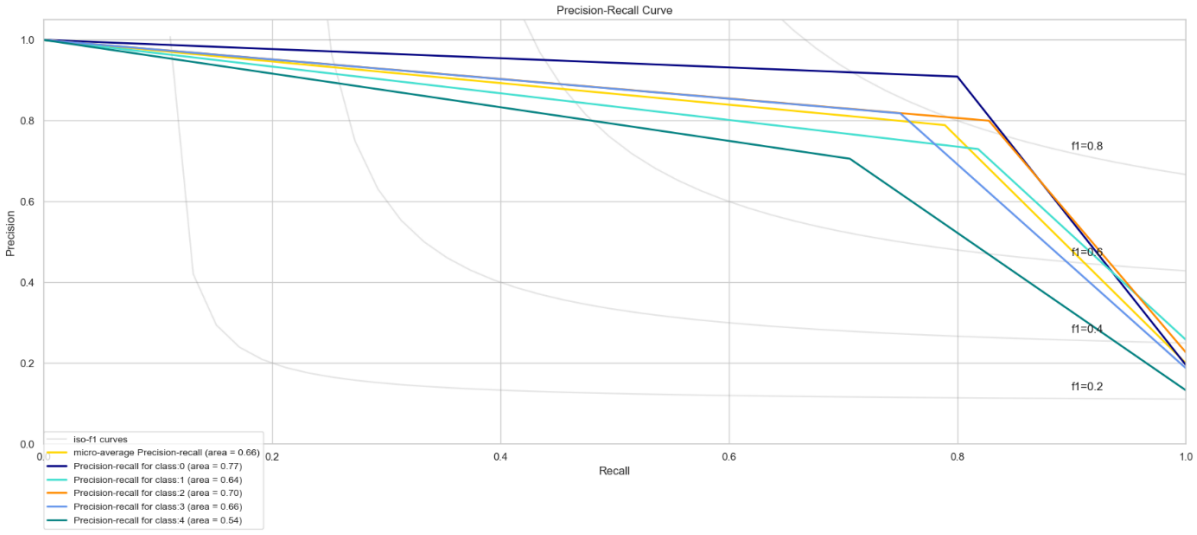


Fig. 17: Experiment 1 - Precision Recall Curve

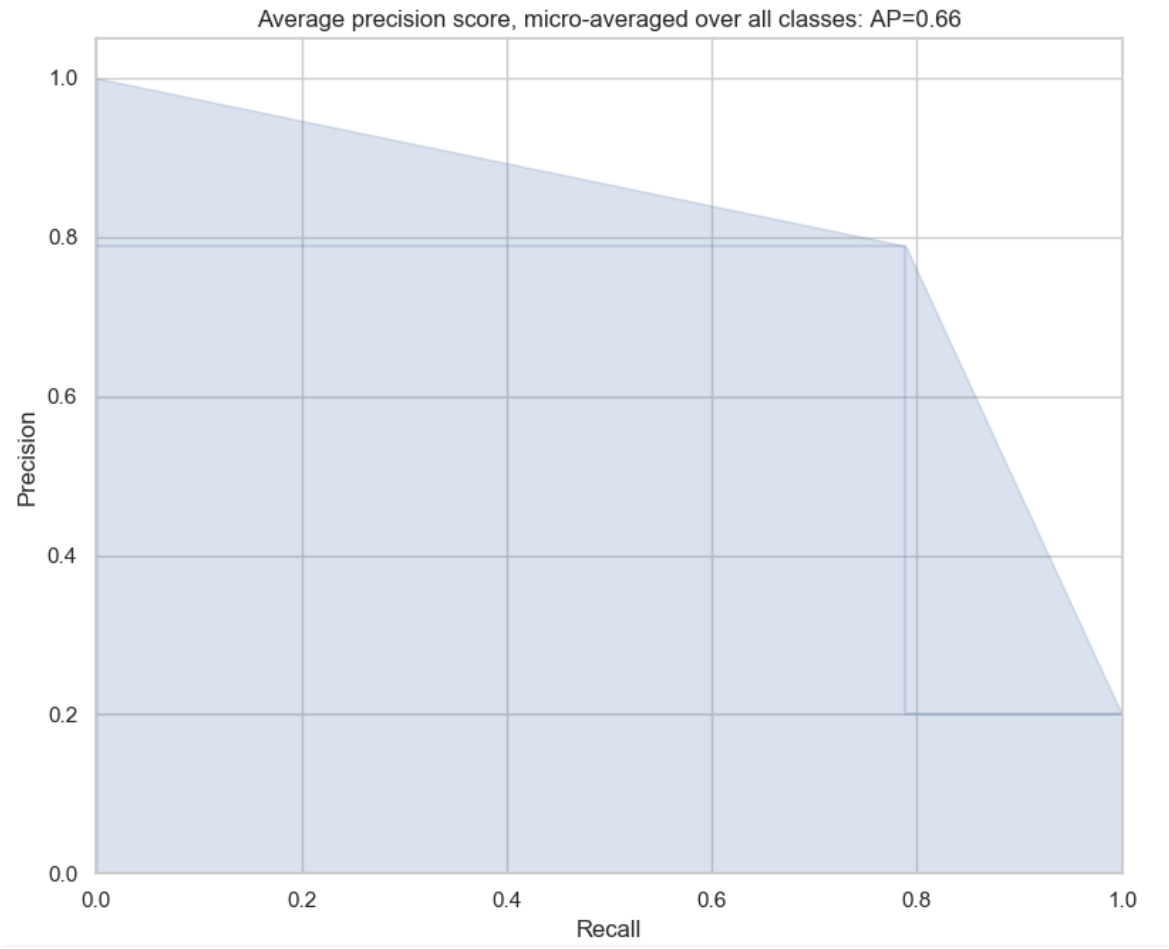


Fig. 18: Experiment 1 - Average Precision Score

2. Data Oversampling with Adaptive Learning Rate

The learning rate for this model changed from 0.001 to 0.0003 at Epoch 15, and to 0.0005 at Epoch 30. After applying oversampling to the dataset using SMOTE, the data distribution for this experiment are displayed in the following table:

	Daisy	Dandelion	Rose	Sunflower	Tulip	Total
Training	685	685	685	685	685	3425
Validation	211	211	211	211	211	1055
Testing	118	159	120	112	151	660

Model Accuracy Metrics:

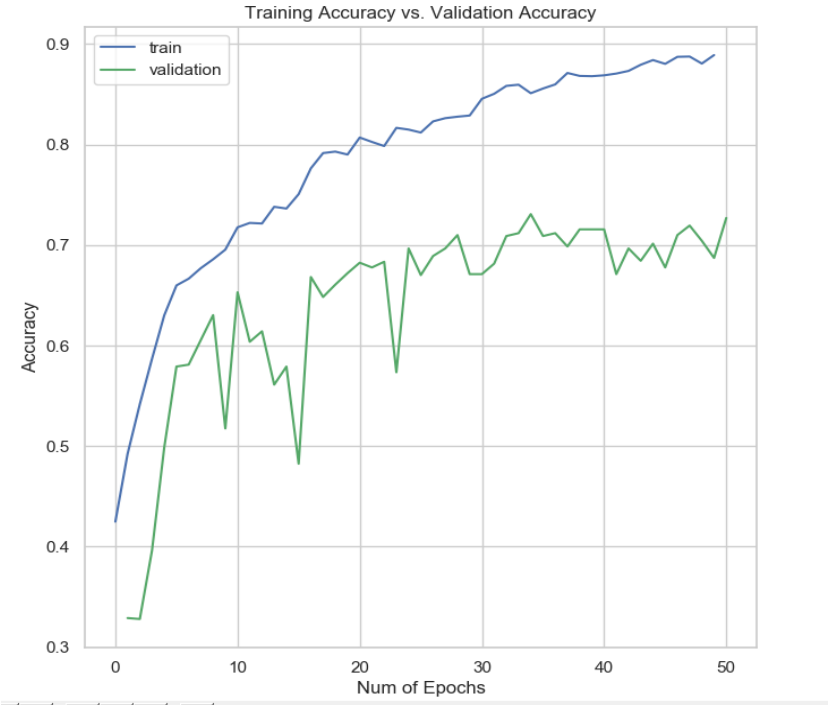


Fig. 19: Experiment 2 - Accuracy

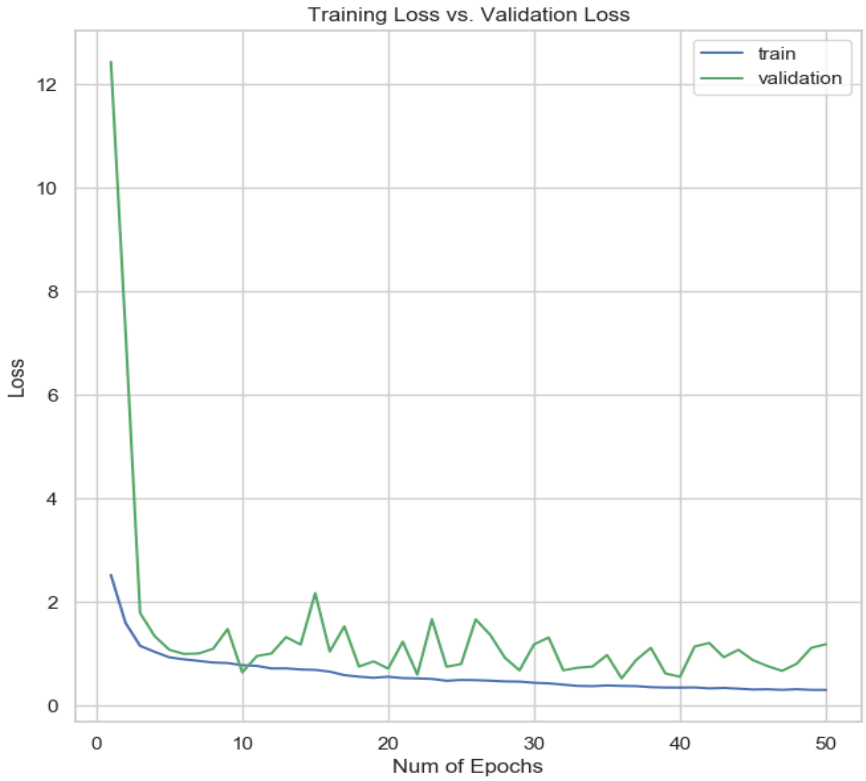


Fig. 20: Experiment 2 – Loss

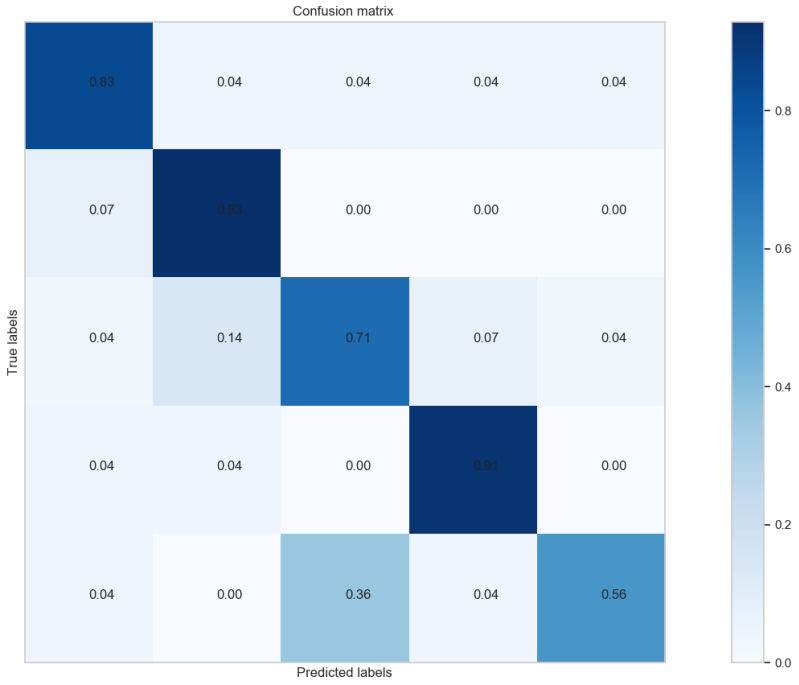


Fig. 21: Experiment 2 - Confusion Matrix

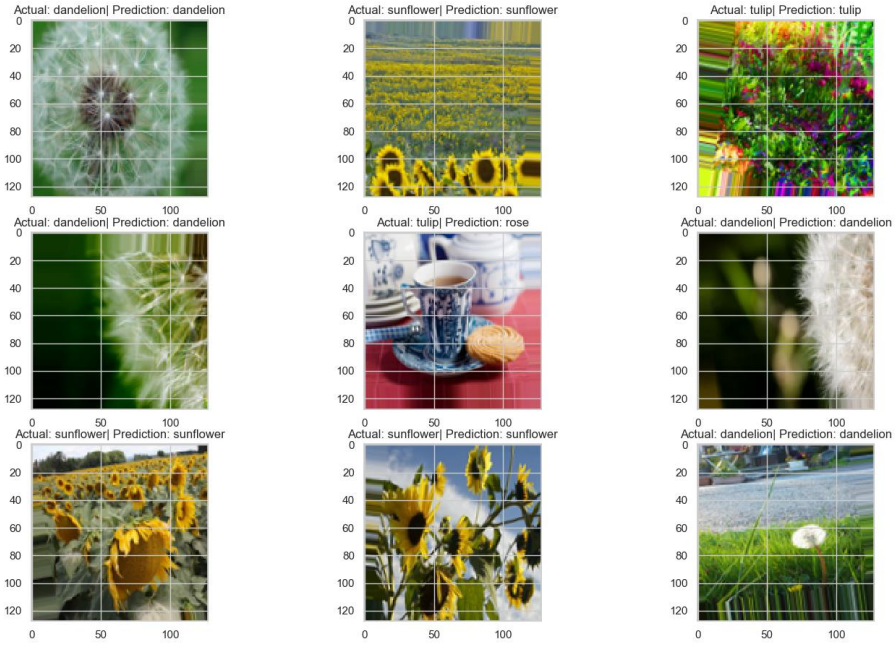


Fig. 22: Experiment 2 – Model Predictions

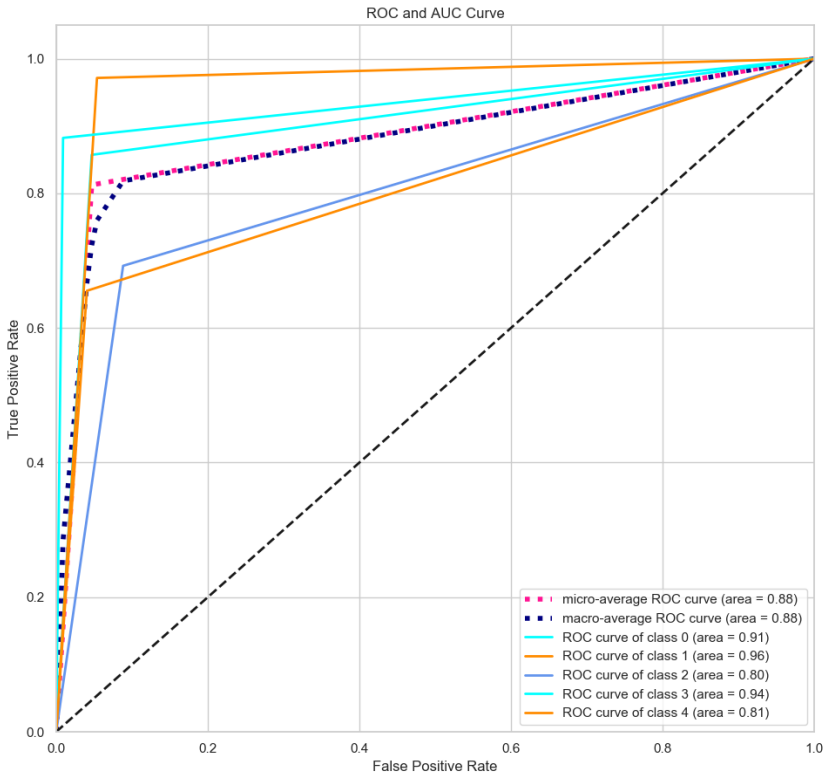


Fig. 23: Experiment 2 - ROC-AUC CURVE

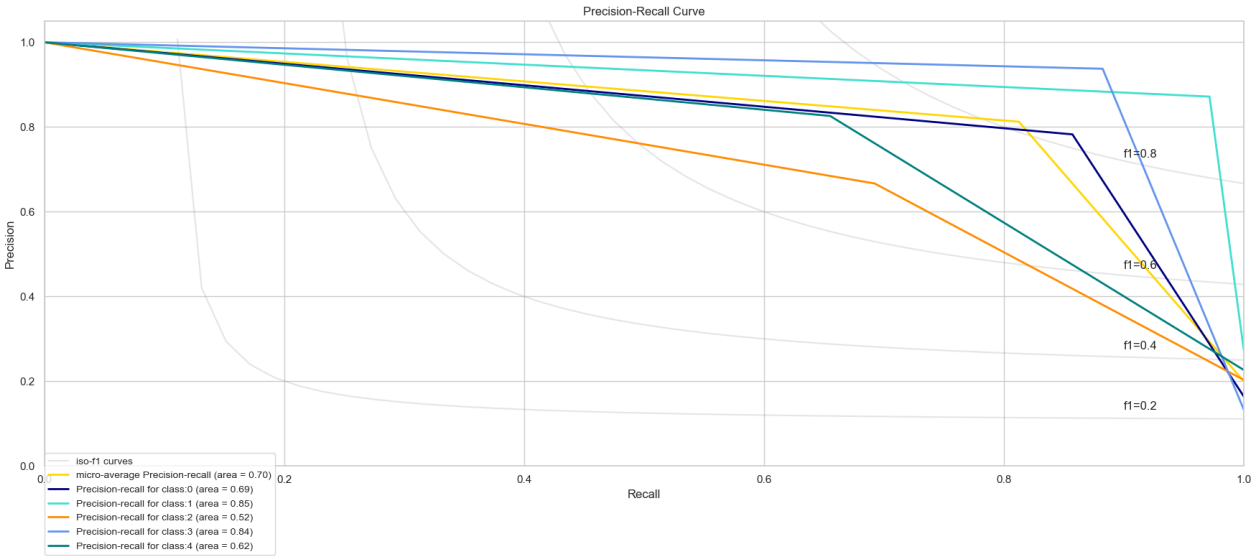


Fig. 24: Experiment 2 - Precision Recall Curve

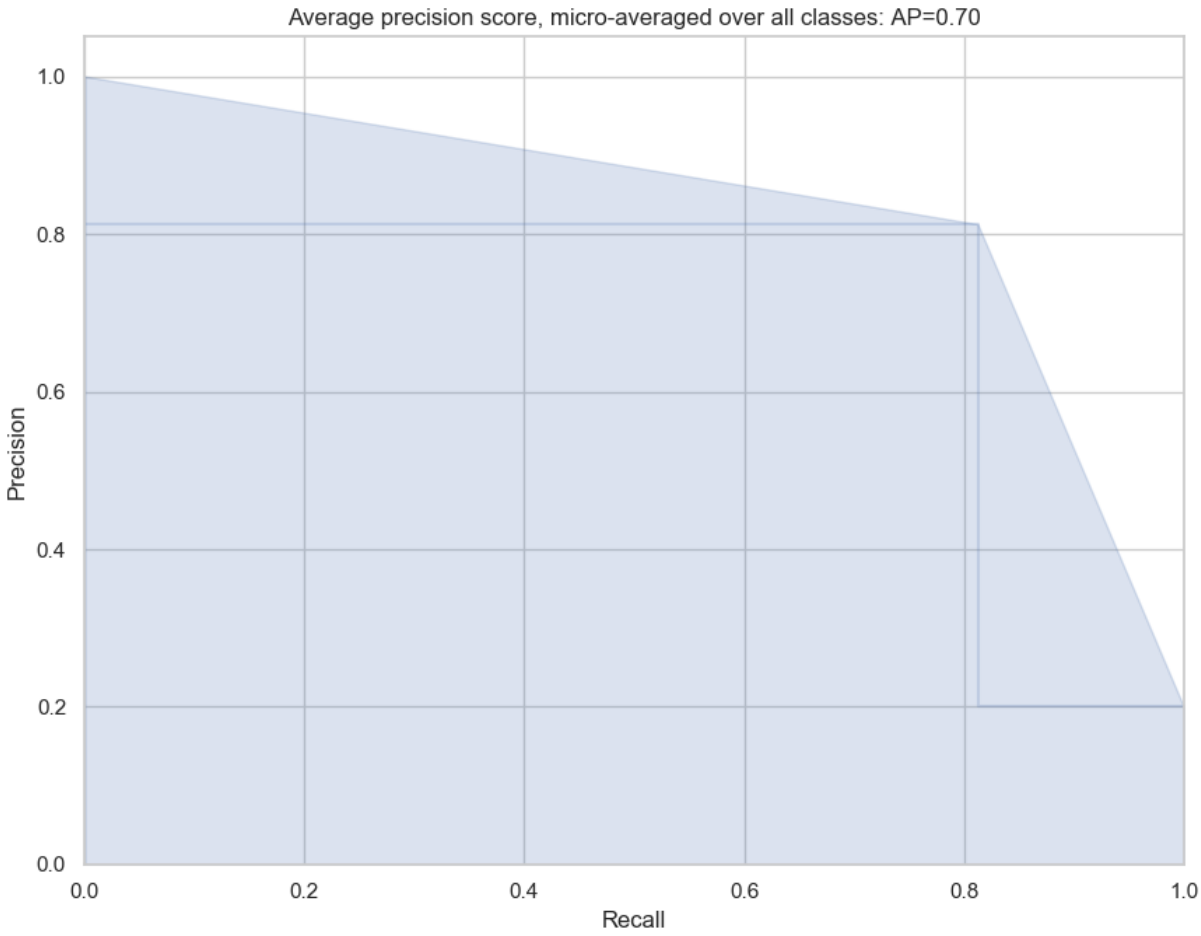


Fig. 25: Experiment 2 - Average Precision Score

3. Data Augmentation and Standardization

The data used to train and test this model is also augmented in order to increase the data size, and try to get a more even distribution of data. It is also standardized in order to shift the distribution of each attribute to have a mean of zero and a standard deviation of one. The learning rate for this model changed from 0.001 to 0.0003 at Epoch 15, and to 0.0005 at Epoch 30. After augmenting the dataset with splits of testing: 25%, training: 50%, and validation: 25%, the data distribution for this experiment are displayed in the following table:

	Daisy	Dandelion	Rose	Sunflower	Tulip	Total
Training	685	685	685	685	685	3425
Validation	211	211	211	211	211	1055
Testing	118	159	120	112	151	660

Model Accuracy Metrics:

Fig. 26: Experiment 3 – Accuracy

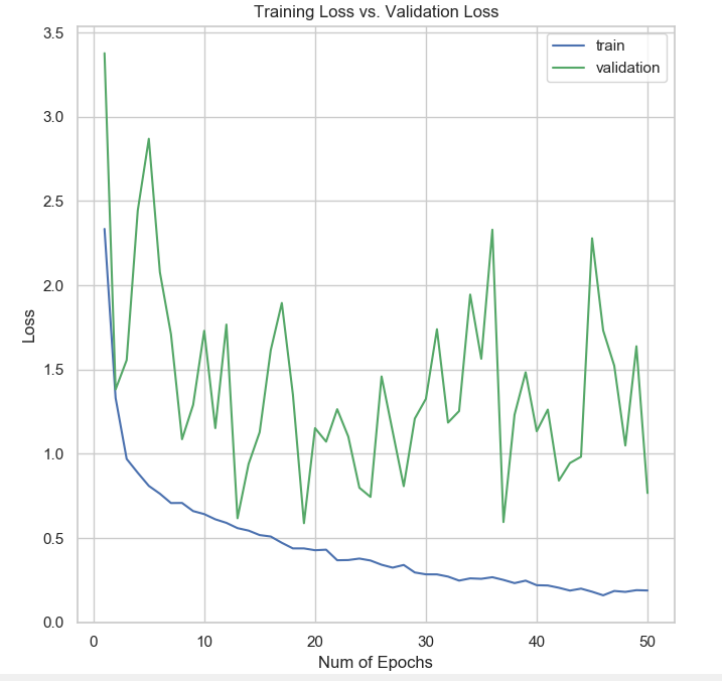


Fig. 27: Experiment 3 – Loss

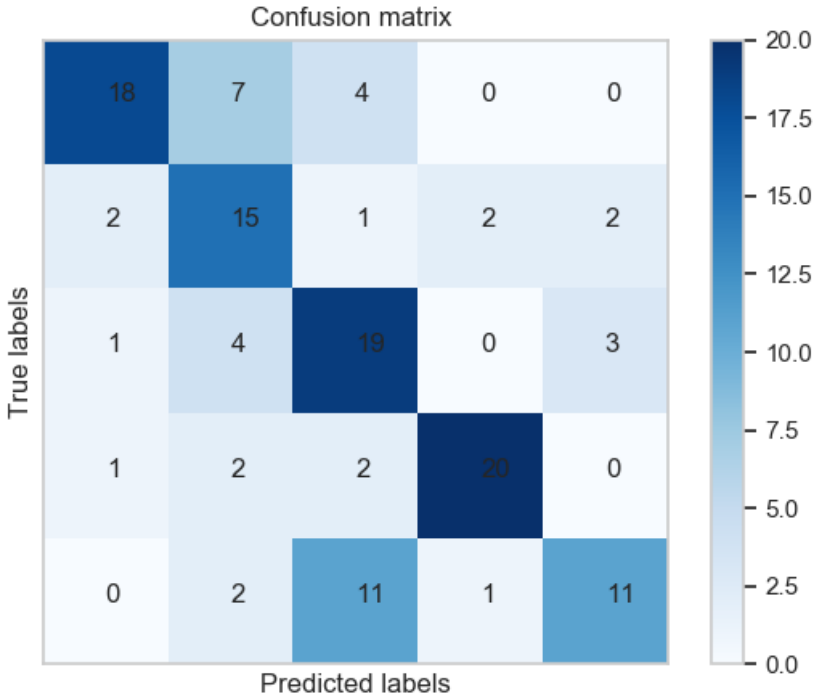


Fig. 28: Experiment 3 – Confusion Matrix

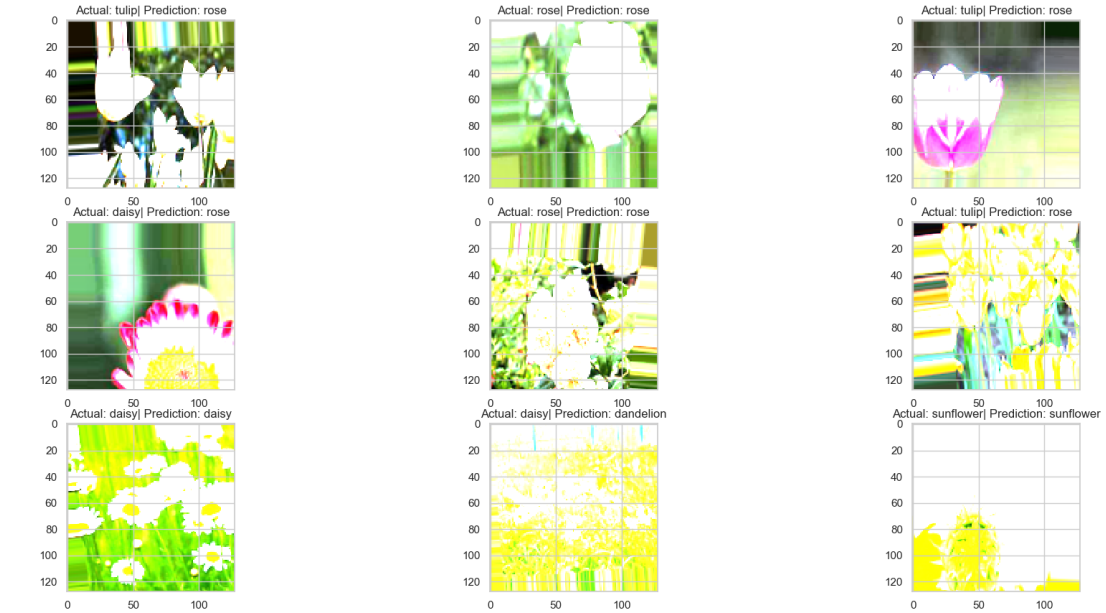


Fig. 29: Experiment 3 – Model Predictions

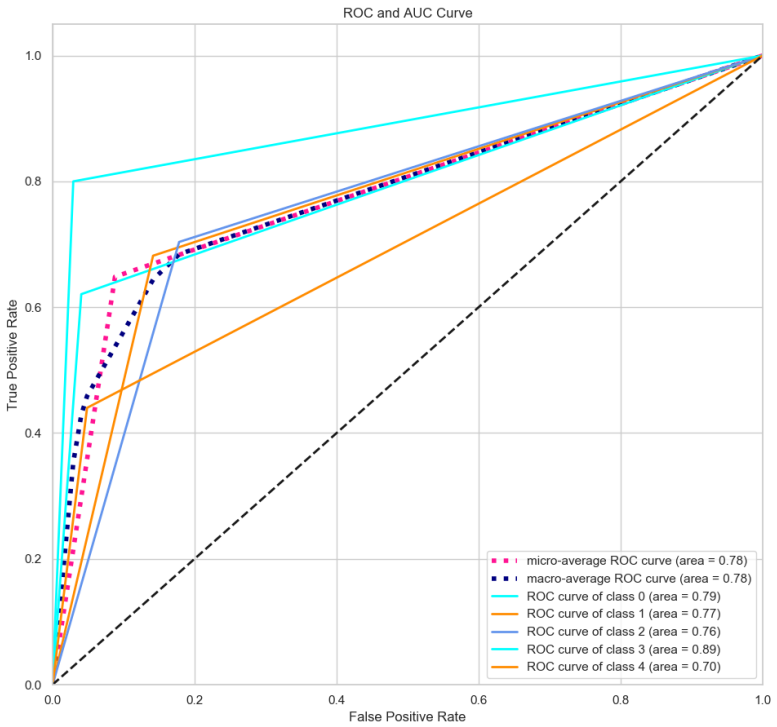


Fig. 30: Experiment 3 – ROC-AUC Curve

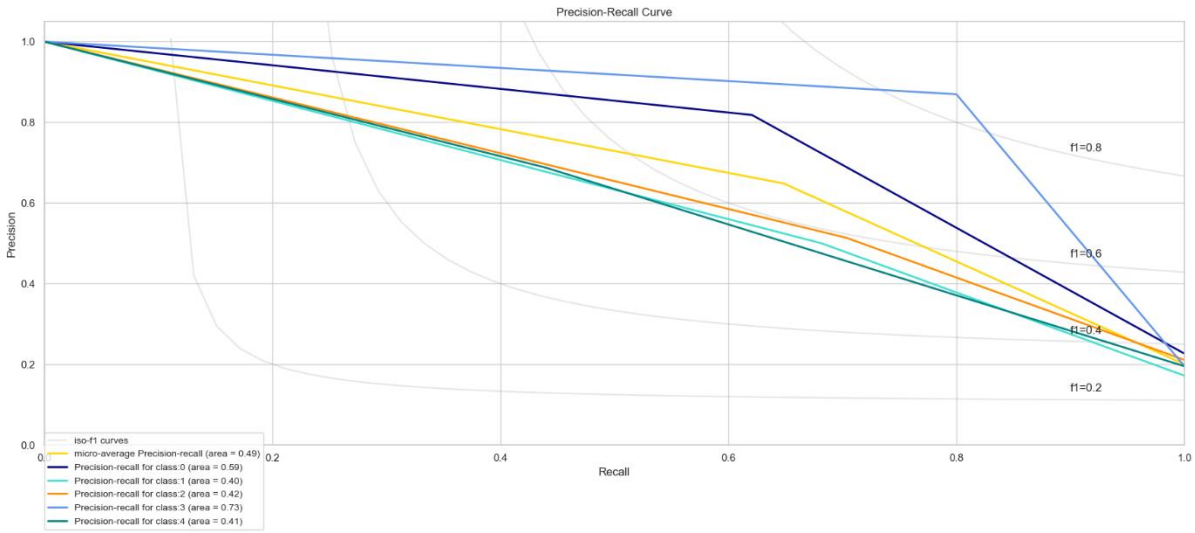


Fig. 31: Experiment 3 – Precision-Recall Curve

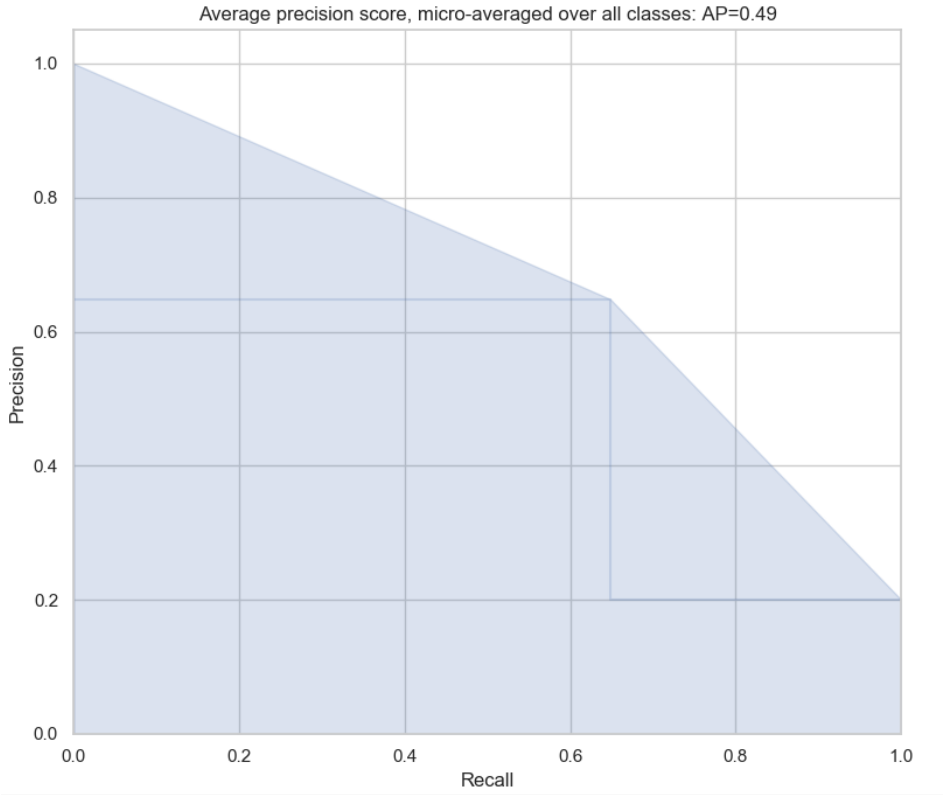


Fig. 32: Experiment 3 – Average Precision Score

5. Conclusion

Once the three models were trained, and compared against one another, the model with the highest accuracy applies data augmentation and normalization to the data, reaching a training accuracy of 95.30%, and validation accuracy of 73.90%. This is evident by analyzing the confusion matrix, ROC-AUC curve, and the precision-recall curve. That model also has the lowest training loss value but it does not have the lowest validation loss values. The model with the lowest validation loss values applies oversampling to the data. As all 3 models had the same optimizer, activation function, and number of layers, these loss and accuracy values allow us to make the conclusion that using augmentation and normalization on input data along with a adaptive learning rate will allow for more accurate models, then using augmentation and standardization, or oversampling with an adaptive learning rate. Although the model has good accuracy, there are several ways it can be further improved, especially the validation accuracy and loss values.

Potential Model Improvements:

a. **Change value of Adaptive Learning Rate**: As visible in the accuracy value plots, once the training process reaches near the 10^{th} epoch the validation accuracy begins a process of switching between increasing and decreasing. A possible fix would be to change the learning rate to an even lower value, and add additional epochs for the learning rate to change.
b. **Add/Remove Hyperparameters:** Hyperparameters are the parameters of a model that are not changed during the learning, such as number of hidden layers, activation function, optimizers, etc. Hyperparameter tuning is the optimization loop of a model to find the best fitting set of hyperparameters on a validation set of data. Some potential algorithms that can be used are grid search or random search. Additionally, addition of layers, dropout, different optimizations and activation functions could be implemented to see the difference in results.
c. **Improve Data:** There are several changes that could be made to the data in order to improve results. Addition of more data will allow for a higher variance during training and testing, which will allow the performance to get better. This could be with new data being added, or augmenting existing data on a greater scale. Modifying the feature selection that is being used will allow the creation of new views of the data to be used by the models. Finding a more efficient resampling method will also be useful, in order to ensure every class of data is being represented equally, while also contributing to a better performing model.
d. **Improve Performance with Ensembles:** We can combine the predictions of multiple well performing models that we have created. This will allow for an increase in performance from multiple good models, as opposed to going through trial and error of high tuning a single model. The mean or mode can be taken from models that are trained on the same algorithms to combine predictions.

6. References

[1] Chollet, Francois. "*Deep Learning with Python"*. Shelter Island, NY, Manning Publications Co, 2018.

[2] D. Gupta and Dishashree, "*Fundamentals of Deep Learning - Activation Functions and their use*," Analytics Vidhya, 23-Oct-2017. https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them.

[3] Michael A. Nielsen. *"Neural Networks and Deep Learning"*. Determination Press, 2015.

[4] García, S., Ramírez-Gallego, S., Luengo, J. "*Big data preprocessing: methods and prospects*". *Big Data Analytics.* 2016. https://bdataanalytics.biomedcentral.com/articles/10.1186/s41044-016-0014-0#citeas

[5] Wang, Shoujin & Liu, Wei & Wu, Jia & Cao, Longbing & Meng, Qinxue & Kennedy, Paul. (2016). Training deep neural networks on imbalanced data sets. https://www.researchgate.net/publication/309778930_Training_deep_neural_networks_on_imbalanced_data_sets

[6] Rashcka, Sebastian. "*Python Machine Learning"*. Packt Publishing, 2015. https://www.javiercancela.com/pymle-equations.pdf

[7] Kathuria, Ayoosh. "*Intro to optimization in deep learning: Gradient Descent*". 2018. https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/