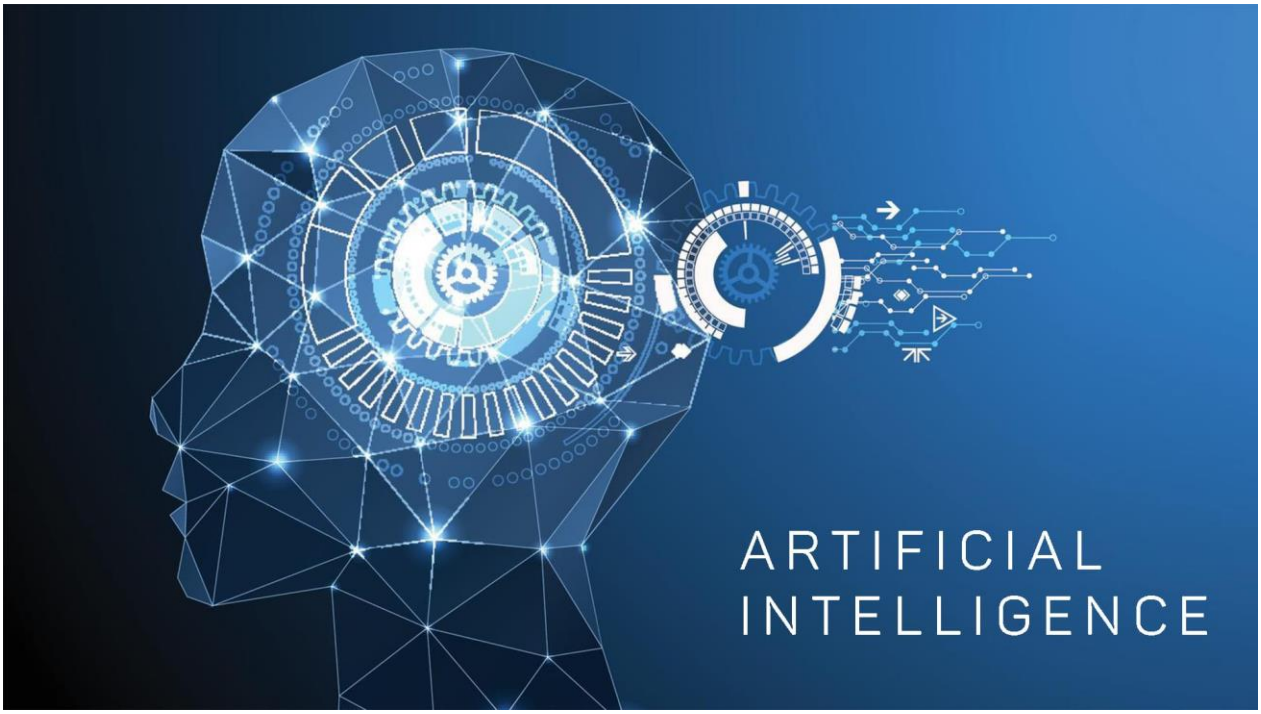
VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

PROJECT 03



Teacher: Chau Thanh Duc

Teaching Asistant: Ngo Dinh Hy, Phan Thi Phuong Uyen

Ho Chi Minh City, August, 2021

Group members table:

No.	Student's ID	Full name
1	19127478	Bui Huynh Trung Nam
2	19127622	Ngo Truong Tuyen
3	19127640	Hoang Huu Giap

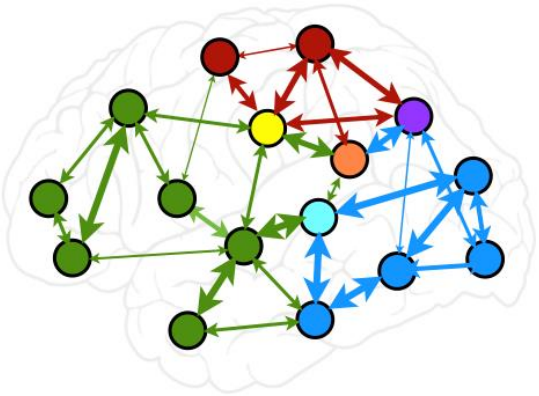
Assignment table:

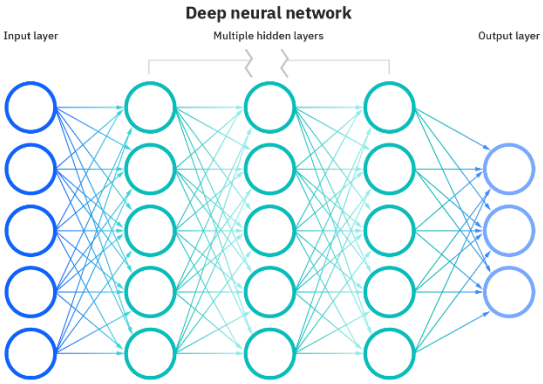
No.	Student's ID	Assignment	Progress
1	19127478	- Neural network, CNN. - Describe + explain model. - Activation function. - Weight initialization. - Advantages and disadvantages of the model. - Explain code: label_of_image, summarize_diagnostics and load_image functions. - Present the results and explain the reasons for that result.	33.3%
2	19127622	- Training set, validation set, test set. - Describe + explain model. - Loss function - Stochastic Gradient Descent. - Advantages and disadvantages of the model. - Explain code: create_parent_folder and define_model functions. - Sugest ideas for improvement.	33.3%
3	19127640	- Detect and prevent overfitting and underfitting problem. - Describe + explain model. - Metric: Accuracy, precision, recall, f1 score, … - Advantages and disadvantages of the model. - Explain code: classify_imgs, start_training and run_example functions. - Sugest ideas for improvement. - Run code.	33.3%

Table of contents

1. Neural Network:

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain (The biological brain is considered as a highly complex, nonlinear and parallel information-processing system), mimicking the way that biological neurons signal to one another. Neural network is A reasoning model based on the human brain, including billions of neurons and trillion connections between them.





Biological neural network	Artificial neural network
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.

Neural networks rely on training data to learn and improve their accuracy over time. However, once these learning algorithms are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity. Tasks in speech recognition or image recognition can take minutes versus hours when compared to the manual identification by human experts. One of the most well-known neural networks is Google's search algorithm.

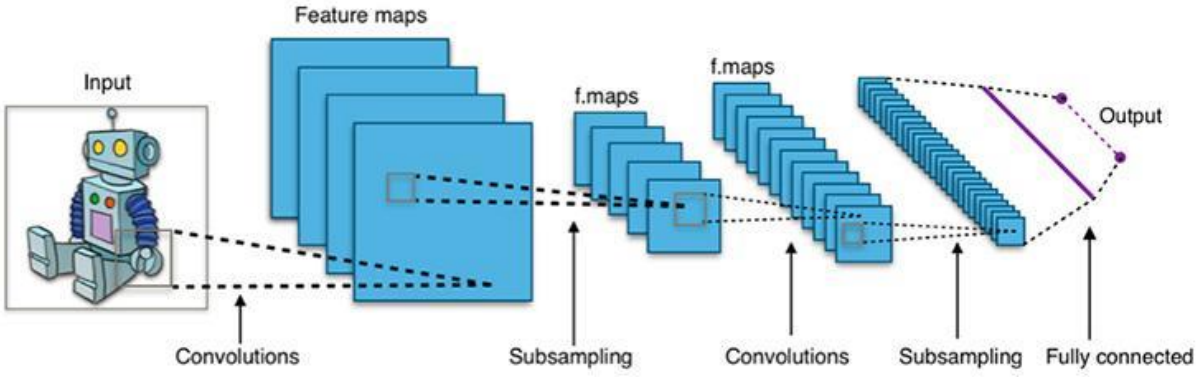
2. Convolutional Neural Network (CNN):

A convolutional neural network (CNN) is a type of artificial neural network (ANN) used in image recognition and processing that is specifically designed to process pixel data.

CNNs are powerful image processing, artificial intelligence (AI) that use deep learning to perform both generative and descriptive tasks, often using machine vison that includes image and video recognition, along with recommender systems and natural language processing (NLP).

Traditional neural networks are not ideal for image processing and must be fed images in reduced-resolution pieces. CNN have their "neurons" arranged more like those of the frontal lobe, the area responsible for processing visual stimuli in humans and other animals. The layers of neurons are arranged in such a way as to cover the entire visual field avoiding the piecemeal image processing problem of traditional neural networks.

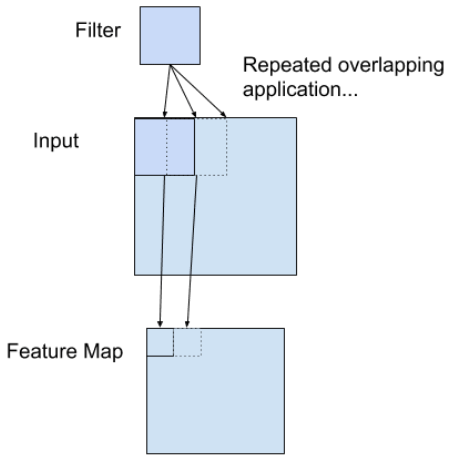
A CNN uses a system much like a multilayer perceptron that has been designed for reduced processing requirements. The layers of a CNN consist of an input layer, an output layer and a hidden layer that includes multiple convolutional layers, pooling layers, fully connected layers and normalization layers. The removal of limitations and increase in efficiency for image processing results in a system that is far more effective, simpler to trains limited for image processing and natural language processing.



2.1. Convolutional Layer:

Central to the convolutional neural network is the convolutional layer that gives the network its name. This layer performs an operation called a "*convolution*".

In the context of a convolutional neural network, a convolution is a linear operation that involves the multiplication of a set of weights with the input, much like a traditional neural network. Given that the technique was designed for two-dimensional input, the multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel.



The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. A dot product is the element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Because it results in a single value, the operation is often referred to as the "*scalar product*".

Using a filter smaller than the input is intentional as it allows the same filter (set of weights) to be multiplied by the input array multiple times at different points on the input. Specifically, the filter is applied systematically to each overlapping part or filter-sized patch of the input data, left to right, top to bottom.

The output from multiplying the filter with the input array one time is a single value. As the filter is applied multiple times to the input array, the result is a two-dimensional array of output values that represent a filtering of the input. As such, the two-dimensional output array from this operation is called a "feature map".

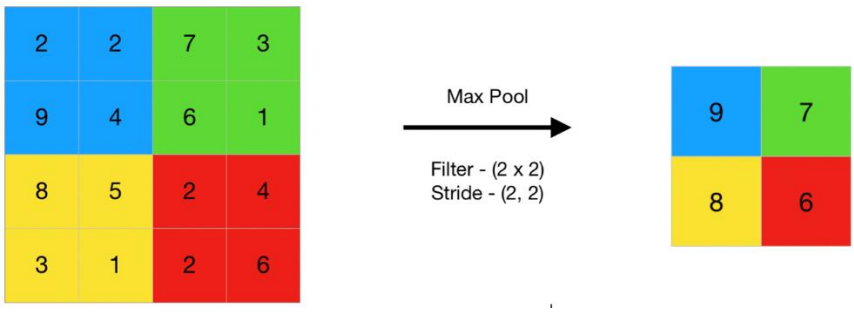
Once a feature map is created, we can pass each value in the feature map through a nonlinearity, such as a ReLU, much like we do for the outputs of a fully connected layer.

2.2. Pooling Layer:

It is common to periodically insert a Pooling layer in-between successive Convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially. Currently, there are 2 types of pooling layer: max pooling and average pooling.

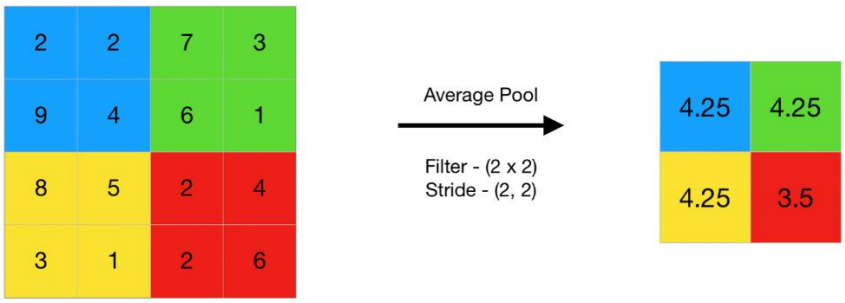
- **Max pooling:**

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.



- **Average pooling**

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.



2.3. Fully Connected Layer:

This layer is responsible for giving the results after the convolutional layer and pooling layer have received the transmitted image. At this point, we get the result that the model has read the information of the images and to link them and produce more output, we use the fully connected layer.

In addition, if the fully connected layer has image data, they will convert it to unsegmented quality. This is quite similar to a vote, which is then evaluated to select the highest quality image.

We flatten the output of the last convolution layer and connect every node of the current layer with the other node of the next layer. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks and work in a similar way. The last layer of our CNN will compute the class probability scores.

3. Training Set – Validation Set – Test Set:

According to the Ripley's book, page 354, "Pattern Recognition and Neural, Networks", 1996:

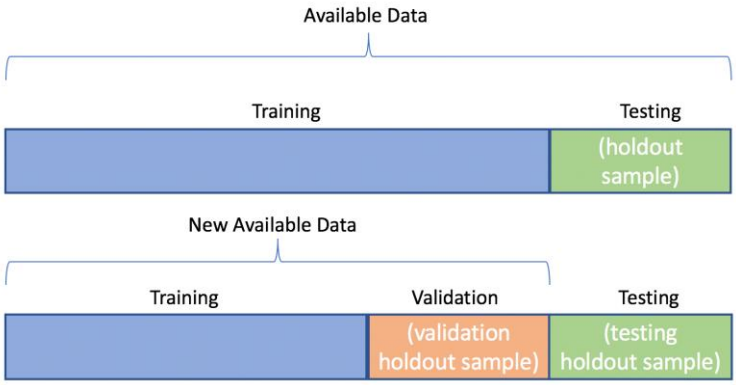
- **Training set:** A set of examples used for learning, that is to fit the parameters of the classifier.
- **Validation set:** A set of examples used to tune the parameters of a classifier, for example to choose the number of hidden units in a neural network.
- **Test set:** A set of examples used only to assess the performance of a fully-specified classifier.

Clear definitions:

- **Training set:** The training set is the set of data we analyze (train on) to design the rules in the model. A training set is also known as the in-sample data or training data.
- **Validation set:** The validation set is a set of data that we did not use when training our model that we use to assess how well these rules perform on new data. It is also a set

we use to tune parameters and input features for our model so that it gives us what we think is the best performance possible for new data.

- **Test set:** The test set is a set of data we did not use to train our model or use in the validation set to inform our choice of parameters/input features. We will use it as a final test once we have decided on our final model, to get the best possible estimate of how successful our model will be when used on entirely new data. A test set is also known as the out-of-sample data or test data…



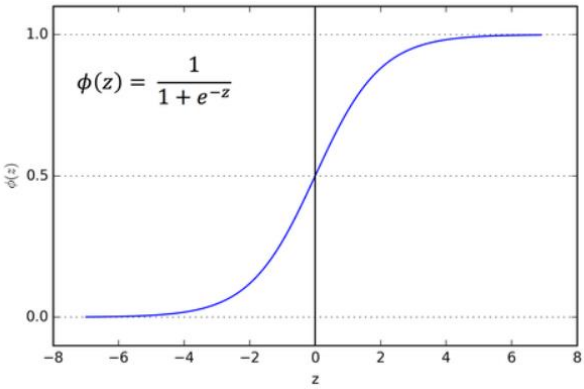
4. Activation Function:

An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. The activation functions can be basically divided into 2 types: linear and non-linear activation function.

4.1. Sigmoid (logistic activation function):

Formular: $\phi(z) = \frac{1}{1+e^{-z}}$

The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.



The function is differentiable. That means, we can find the slope of the sigmoid curve at any two points. The function is monotonic but function's derivative is not. The logistic sigmoid function can cause a neural network to get stuck at the training time.

4.2. Step:

Formular: $\phi(z) = \begin{cases} 1 \; if \; z \geq 0 \\ 0 \; if \; z < 0 \end{cases}$

This function is very simple, result is 1 if input ≥ 0, otherwise result will be 0, it is often used in perceptron.

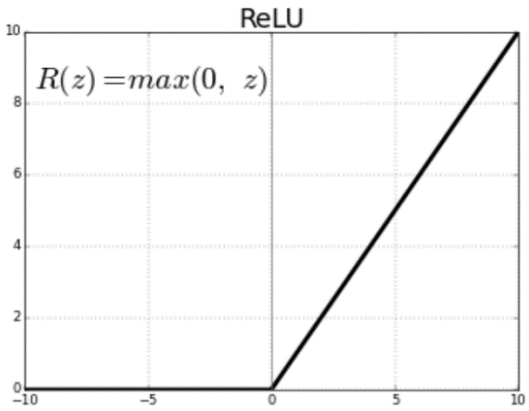
4.3. ReLU (Rectified Linear Unit):

The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning.

Formular: $\phi(z) = \max(0, z)$

The function and its derivative both are monotonic.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.



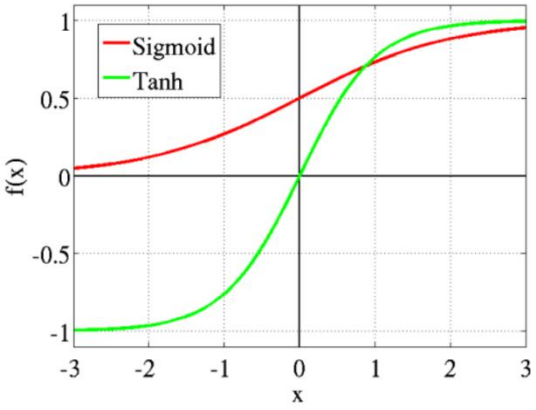
4.4. Tanh (hyperbolic tangent activation function):

Tanh is also like logistic sigmoid but better. The range of the Tanh function is from (-1 to 1). Tanh is also sigmoidal (s - shaped).

Formular: $\phi(z) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

The function is differentiable. The function is monotonic while its derivative is not monotonic. The Tanh function is mainly used classification between two classes.



5. Loss function:

A loss function takes a theoretical proposition to a practical one. Building a highly accurate predictor requires constant iteration of the problem through questioning, modeling the problem with the chosen approach and testing.

The only criteria by which a statistical model is scrutinized is its performance - how accurate the model's decisions are. This calls for a way to measure how far a particular iteration of the model is from the actual values. This is where loss functions come into play.

Loss functions measure how far an estimated value is from its true value. A loss function maps decisions to their associated costs. Loss functions are not fixed, they change depending on the task in hand and the goal to be met.

5.1. Loss functions for regression:

Regression involves predicting a specific value that is continuous in nature. Estimating the price of a house or predicting stock prices are examples of regression because one works towards building a model that would predict a real-valued quantity.

5.1.1. Mean Absolute Error(MAE):

Mean Absolute Error (also called L1 loss) is one of the most simple yet robust loss functions used for regression models.

For a data point x_i and its predicted value y_i; n being the total number of data points in the dataset, the mean absolute error is defined as:

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

As the name suggests, MAE takes the average sum of the absolute differences between the actual and the predicted values.

5.1.2. Mean Squared Error (MSE):

For a data point y_i and its predicted value $\hat{y}_i$, where n is the total number of data points in the dataset, the mean squared error is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Mean Squared Error (also called L2 loss) is almost every data scientist's preference when it comes to loss functions for regression. This is because most variables can be modeled into a Gaussian distribution. Mean Squared Error is the average of the squared differences between the actual and the predicted values.

5.1.3. Mean Bias Error (MBE):

For y_i is the true value, $\hat{y}_i$ is the predicted value and n is the total number of data points in the dataset. The formula of Mean Bias Error is:

$$\text{MBE} = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)}{n}$$

Mean Bias Error is used to calculate the average bias in the model. Bias, in a nutshell, is overestimating or underestimating a parameter. Corrective measures can be taken to reduce the bias post-evaluating the model using MBE.

5.2. Loss functions for classification:

Classification problems involve predicting a discrete class output. It involves dividing the dataset into different and unique classes based on different parameters so that a new and unseen record can be put into one of the classes.

Binary Cross Entropy Loss

This is the most common loss function used for classification problems that have two classes. The word "*entropy*", seemingly out-of-place, has a statistical interpretation.

Entropy is the measure of randomness in the information being processed, and cross entropy is a measure of the difference of the randomness between two random variables.

If the divergence of the predicted probability from the actual label increases, the cross-entropy loss increases. Going by this, predicting a probability of .011 when the actual observation label is 1 would result in a high loss value. In an ideal situation, a "perfect" model would have a log loss of 0. Looking at the loss function would make things even clearer.

$$J = -\sum_{i=1}^{N} y_i . \log\big(h_\theta(x_i)\big) + (1 - y_i) . \log\big(1 - h_\theta(x_i)\big)$$

Where y_i is the true label and $h_\theta(x_i)$ is the predicted value post hypothesis. Since binary classification means the classes take either 0 or 1 , if $y_i = 0$, that term ceases to exist and if y_i = 1, the $(1 - y_i)$ term becomes 0.

6. Metric:

6.1. Accuracy:

Accuracy is defined as the percentage of correct predictions for the test data. It can be calculated easily by dividing the number of correct predictions by the number of total predictions.

$$accuracy = \frac{correct\ predictions}{all\ predictions}$$

In this project, we use binary classification so accuracy can also be calculated in terms of positives and negatives:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP = True Positives TN = True Negatives FP = False Positives and FN = False Negatives.

6.2. Precision:

In the simplest terms, precision is the ratio between the True Positives and all the Positives. Precision is defined as the fraction of relevant examples (true positives) among all of the examples which were predicted to belong in a certain class.

$$precision = \frac{TP}{TP + FP}$$

6.3. Recall:

Recall is defined as the fraction of examples which were predicted to belong to a class with respect to all of the examples that truly belong in the class. For this metric one will see how many positive samples are in fact correctly determined. This metric is used to evaluate a model when incorrect prediction of an actual positive sample is very dangerous. As the prediction of ill patients.

$$recall = \frac{TP}{TP + FN}$$

6.4. F1-score:

However, only Precision or only Recall do not evaluate the quality of the model. If we use only model Precision, model only makes predictions for the point where it is most certain precision=1 then. However we can not say this model is good if we use recall only , if the model predicts all points are positive. Then recall = 1, but we can not say that this is a good model. When F1-score used, F1-score is the harmonic mean of precision and recall. F1-score is mathematically:

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

6.5. Confusion matrix:

A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix. The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives you insight not only into the errors being made by your classifier but more importantly the types of errors that are being made. It is this breakdown that overcomes the limitation of using classification accuracy alone. In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa). The confusion matrix is an N×N matrix used to evaluate the performance of a classification model, where N is the number of target classes. For the binary classification problem, we will have a 2 × 2 matrix as shown below with 4 values:

	Negative	Positive
Negative	True Negative	False Positive
Positive	False Negative	False Positive

7. Weigh Initialization:

The nodes in neural networks are composed of parameters referred to as weights used to calculate a weighted sum of the inputs.

Neural network models are fit using an optimization algorithm called stochastic gradient descent (will explain below) that incrementally changes the network weights to minimize a loss function, hopefully resulting in a set of weights for the mode that is capable of making useful predictions.

This optimization algorithm requires a starting point in the space of possible weight values from which to begin the optimization process. Weight initialization is a procedure to set the weights of a neural network to small random values that define the starting point for the optimization (learning or training) of the neural network model.

Each time, a neural network is initialized with a different set of weights, resulting in a different starting point for the optimization process, and potentially resulting in a different final set of weights with different performance characteristics (this is also purpose of training in CNN).

Historically, weight initialization follows simple heuristics, such as:

- Small random values in the range [-0.3, 0.3]
- Small random values in the range [0, 1]
- Small random values in the range [-1, 1]

Nevertheless, more tailored approaches have been developed over the last decade that have become the defacto standard given they may result in a slightly more effective optimization (model training) process.

7.1. Weight initialization for Sigmoid And Tanh activation function:

The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or Tanh activation function is called "*glorot*" or "*xavier*" initialization.

Xavier Weight Initialization

Formular: $weight = U\left[\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$

The Xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range $\frac{-1}{\sqrt{n}}$ and $\frac{1}{\sqrt{n}}$, where n is the number of inputs to the node. Moreover, the normalized Xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range $\frac{-\sqrt{6}}{\sqrt{n+m}}$ and $\frac{\sqrt{6}}{\sqrt{n+m}}$, where n us the number of inputs to the node (e.g. number of nodes in the previous layer) and m is the number of outputs from the layer (e.g. number of nodes in the current layer).

$$weight = U\left[\frac{-\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}}\right]$$

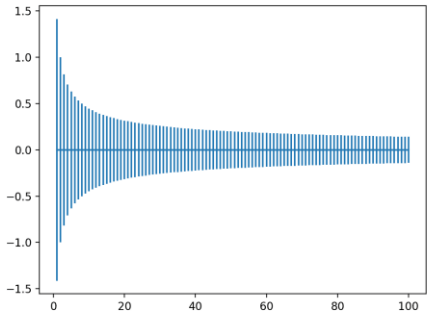
7.2. Weight initialization for ReLU activation function:

The Xavier weight initialization was found to have problems when used to initialize networks that use the rectified linear (ReLU) activation function.

The current standard approach for initialization of the weights of neural network layers and nodes that use the rectified linear (ReLU) activation function is called "*he*" initialization.

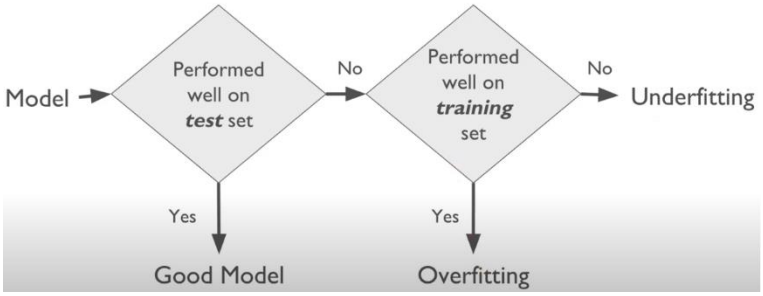
He Weight Initialization

Formular: $weight = G\left(0.0, \sqrt{\frac{2}{n}}\right)$



The he initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the node.

8. Describe how to detect and prevent overfitting and underfitting problem:



Overfitting refers to the scenario where a machine learning model can't generalize or fit well on unseen dataset. A clear sign of machine learning overfitting is if its error on the testing or validation dataset is much greater than the error on training dataset. As a result, overfitting may fail to fit additional data, and this may affect the accuracy of predicting future observations. Overfitting happens when a model learns the detail and noise in the training dataset to the extent that it negatively impacts the performance of the model on a new dataset. This means that the noise or random fluctuations in the training dataset is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new datasets and negatively impact the model's ability to generalize. The opposite of overfitting is underfitting. Underfitting refers to a model that can neither model the training dataset nor generalize to new dataset. An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training dataset. Underfitting is often not discussed as it is easy to detect given a good performance metric.

Overfitting and underfitting are the two biggest causes of poor performance of machine learning algorithms or models.

8.1. How to detect overfitting and underfitting problem:

A key challenge of detecting any kind of fit (be it underfitting or best fit or overfitting), is almost impossible before you test the data. It can help address the inherent characteristics of overfitting, which is the inability to generalize a dataset. The data can therefore be separated into different subsets to make it easy for training and testing. The data is split into two main parts, in example, a test dataset and a training dataset. Splitting technique may vary based on the type of dataset and one can use any splitting technique.

If our model does much better on the training dataset than on the test dataset, then we're likely overfitting. For example, our model performed with a 99% accuracy on the training dataset

but only 50-55% accuracy on the test dataset. It is overfitting the model and did not performed well on unseen dataset.

If our model does much better on the test dataset than on the training dataset, then we are likely underfitting.

And if our model does well on both the training and test datasets then we have the best fit. For example, our model performed 90% accuracy on the training dataset and performs 88% - 92% accuracy on the test dataset. It is the best fit model. Another simple way to detect this is by using cross-validation. This attempts to examine the trained model with a new data set to check its predictive accuracy. Given a dataset, some portion of this is held back (say 30%) while the rest is used in training the model. Once the model has been trained the reserved data is then used to check the accuracy of the model compared to the accuracy of derived from the data used in training. A significant variance in these two flags overfitting.

8.2. How to prevent overfitting and underfitting problem:

The remedy for underfitting, is to move on and try alternate machine learning algorithms. Nevertheless, it does provide a good contrast to the problem of overfitting.

To prevent overfitting, there are various ways and a few of them are shown below.

8.2.1. Cross-validation:

Cross-validation is a powerful preventative measure against overfitting. Use your initial training data to generate multiple mini train-test splits. Use these splits to tune your model. In standard k-fold cross-validation, we partition the data into k subsets, called folds. Then, we iteratively train the algorithm on k-1 folds while using the remaining fold as the test set (called the "*holdout fold*"). Cross-validation allows you to tune hyperparameters with only your original training dataset. This allows you to keep your test dataset as a truly unseen dataset for selecting your final model.

8.2.2. Train with more data:

It won't work every time, but training with more data can help algorithms detect the signal better. As the user feeds more training data into the model, it will be unable to overfit all the samples and will be forced to generalize to obtain results. Users should continually collect more data as a way of increasing the accuracy of the model. However, this method is considered expensive, and, therefore, users should ensure that the data being used is relevant and clean

8.2.3. Data augmentation:

An alternative to training with more data is data augmentation, which is less expensive compared to the former. If you are unable to continually collect more data, you can make the available data sets appear diverse. Data augmentation makes a data sample look slightly different every time it is processed by the model. The process makes each data set appear unique to the model and prevents the model from learning the characteristics of the data sets.

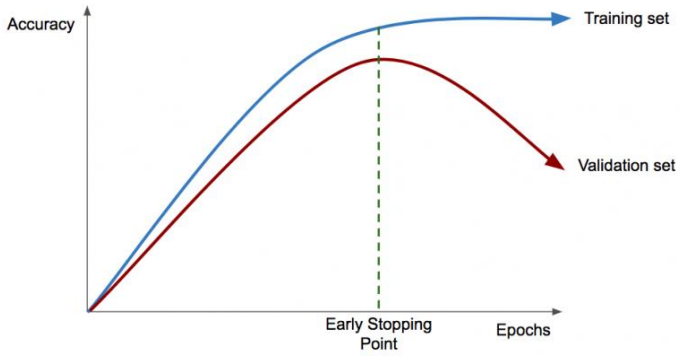
8.2.4. Reduce Complexity or Data Simplification:

Overfitting can occur due to the complexity of a model, such that, even with large volumes of data, the model still manages to overfit the training dataset. The data simplification method is used to reduce overfitting by decreasing the complexity of the model to make it simple enough that it does not overfit. Some of the actions that can be implemented include pruning a decision tree, reducing the number of parameters in a Neural Networks, and using dropout on a Neural Networks. Simplifying the model can also make the model lighter and run faster.

8.2.5. Validation Sets and Early Stopping:

This first technique used requires the data set to be broken down into three sub sets, the training, validation and test set. The training set contains the data that is used to train the model (as expected), while the validation and test sets contain data that is unseen to the model that is used to evaluate its effectiveness. Evaluation on the validation set takes place at regular intervals during the training process, while a test set evaluation typically takes places on the fully trained model, when wishing to compare accuracy of different models. A typical split of the dataset would be 80% for the training set, and 10% each for the validation and test sets (in our project we split dataset to 75% for training set and 25% for validation set).

Early stopping is a simple, but effective, method to prevent overfitting. The effectiveness of the model is evaluated on the accuracy from the validation set, rather than the training set. When the validation accuracy begins flatlining, or even decreasing, the training process stops, with the model corresponding to the highest validation accuracy taken as the final model. This point is known as the early stopping point and can be clearly seen in figure below. Today, this technique is mostly used in deep learning



9. Stochastic Gradient Descent (SGD):

Gradient Descent is a convex function whose output is the partial derivative of a set of parameters of its inputs. Starting from an initial value, Gradient Descent is run iteratively to find the optimal values of the parameters to find the minimum possible value of the given cost function. Typically, there are three types of Gradient Descent: Batch Gradient Descent, Stochastic Gradient Descent and Mini-batch Gradient Descent. In this project we use Stochastic Gradient Descent so that we will go deeper into it.

The word "*stochastic*" means a system or a process that is linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima is reached. Hence, it becomes computationally very expensive to

perform. This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

SGD algorithm

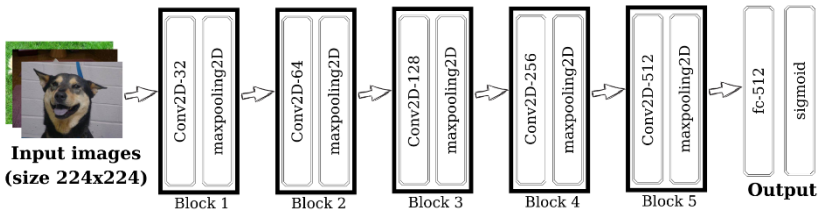
So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.

$$for\ i\ in\ range\,(m):$$
$$\theta_j = \theta_j - \alpha\,(\hat{y}^i - y^i)\,x_j^i$$

One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

In this project we make an addition to classic SGD algorithm, called momentum which almost always works better and faster than Stochastic Gradient Descent. It is method which helps accelerate gradients vectors in the right directions, thus leading to faster converging. It is one of the most popular optimization algorithms and many state-of-the-art models are trained using it.

10. Explain Model Used:



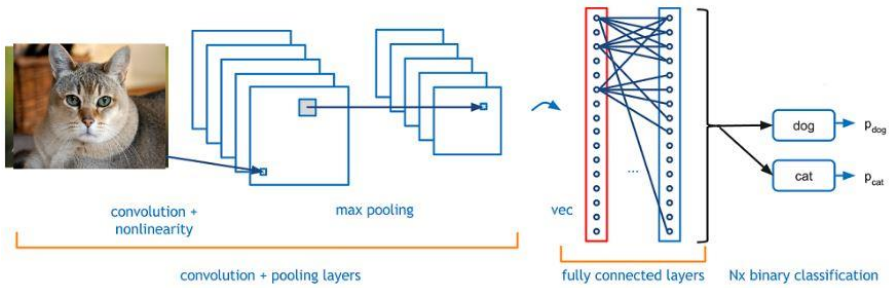
Five block VGG model

The picture above is the five block VGG model that we use in this project.

In Keras, there are 2 ways to build a model: Sequential model and Function API. In this project, we use Sequential to build our model because Sequential model is simplest type of model, a linear stock of layers. If we need to build arbitrary graphs of layers, Keras functional API can do that for us.

In training set and validation set, we use data from Kaggle with 25 000 pictures. We will hold back 25% (ratio = 0.25) of the images into the validation dataset and remaining part is training dataset (all thing will operate randomly).

We choose this model because it's quite easy and accuracy is very good (>90%).



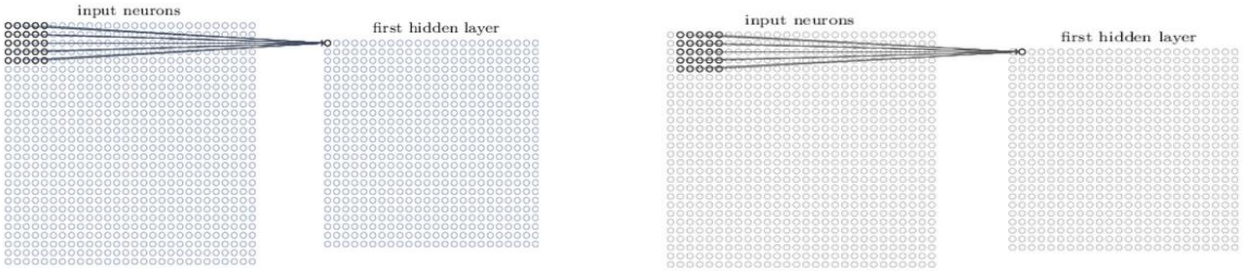
$$\text{Input} \xrightarrow{create} feature\ maps \xrightarrow{reduce\ size} pooling\ unit \xrightarrow{flatten} vector \xrightarrow{determine\ input} god\ or\ cat$$

Overview: In this model, we will use 5 blocks means for each block we will create some feature maps (explained above), after that we will use pooling function for each feature map to reduce size of it. Then we flatten it become a vector, so we can get result from this vector.

Detail:

- **Block**

We use a slide window (3x3) to move through pixels of image, after moving all we can get a 2D-dimensional array (or matrix 2D). For 1st block we use 32 filters – this figure was experimented many times and apply in practice, so we choose 32 to get started. For the blocks of 2nd to 5th this number becomes 64, 128, 256, 512 respectively. Because filters in each block increase => feature map of block 1 < feature map of block 2 <…< feature map of block 5. After finish this process, we get a lot of feature maps which will use to classify image.



(Illustration for slide window)

In training process, CNN learns automatically value through filters base on our model. In classification, CNN will try to find the optimal parameters for the corresponding filters in the order raw pixels > edges > shapes > facial > high-level features.

- **Convolutional Layer**

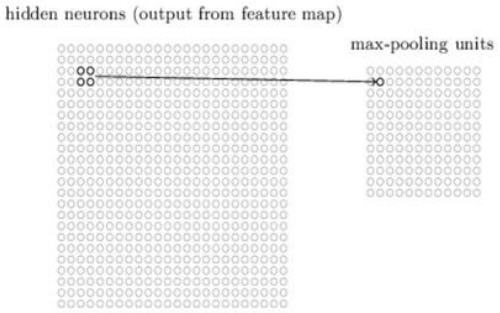
We all know that filter just go through pixels of input and obviously, we need convolutional layer to create feature map (explain in convolutional layer part). We will combine all knowledge explained above: activation function used is ReLU → define how to sum weight and input → cater to update weight through training. For beginning, corresponding to ReLU, we use He initialization (He Uniform) to create beginning weight => accuracy will increase because after each input, weight will be updated =>feature maps become more and more accuracy.

In short, a convolutional layer consists of different feature maps. Each feature map helps to detect some feature in the image. The biggest benefit of shared weights is to minimize the number of parameters in the CNN network.

- **Pooling Layer**

The pooling layer is often used right after the convolutional layer to simplify the output information to reduce the number of neurons, so with 5 blocks model, we will operate like this:

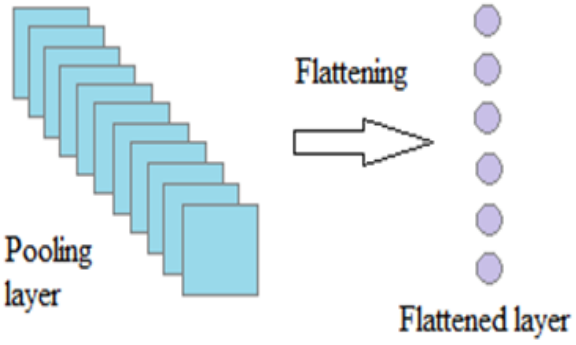
$$\begin{cases} Block\ 1\ \left(\dfrac{Convolutional\ with\ 32\ filter}{\text{Max} - Pooling}\right) \rightarrow feature\ maps \\ Block\ 2\ \left(\dfrac{Convolutional\ with\ 64\ filter}{\text{Max} - Pooling}\right) \rightarrow feature\ maps \\ Block\ 3\ \left(\dfrac{Convolutional\ with\ 128\ filter}{\text{Max} - Pooling}\right) \rightarrow feature\ maps \\ Block\ 4\ \left(\dfrac{Convolutional\ with\ 256\ filter}{\text{Max} - Pooling}\right) \rightarrow feature\ maps \\ Block\ 5\ \left(\dfrac{Convolutional\ with\ 512\ filter}{\text{Max} - Pooling}\right) \rightarrow feature\ maps \end{cases}$$



- **Fully Connected Layer**

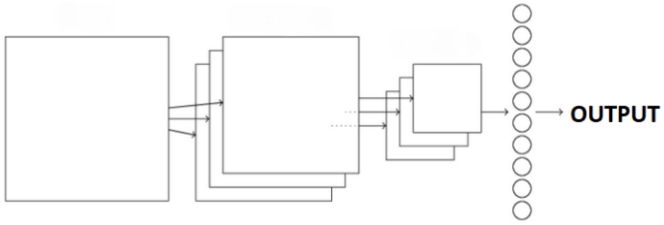
We use flatten to take all pooled feature maps into a single vector as the input for the fully connected layers as shown in the picture:

We don't directly flatten the input image into a single vector because of doing that will only keep the pixel values of the image, but not the spatial structure. In another word, it will lose how each pixel is spatially connected to one around it. But with convolution, we get many feature maps, each of which represents a specific feature of the image. Thus, each node in the flattened vector will represent a specific detail of the input image.



The input to the fully connected layer is the output from the final Pooling or Convolutional Layer, which is flattened and then fed into the fully connected layer. We can get probabilities of the input being in a particular class (classification).

Finally, we will calculate the model accuracy and loss on the test and training dataset at the end of each epoch. After calculating, we will draw line plots which provide learning curves that we can use to get an idea of whether the model is overfitting, underfitting, or has a good fit. (metric is accuracy, loss function is binary cossentropy.



11. Explain Code:

11.1. label_of_image (img):

Argument: img is a file name.

Purpose: determine the image name is dog or cat.

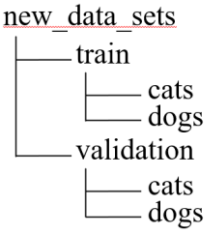
Use split method to divide string into a list of 3 elements by '.' (because img has format as dog/cat.number.jpg), and use [-3] to get 1st element. Return 1 if it is cat, otherwise return 0.

11.2. create_parent_folder():

Purpose: This function creates directories which use to store images after they are classified in classify_imgs() function.

We can create directories in Python using the makedirs() function and use a loop to create the "dogs/" and "cats/" subdirectories for both the "train/" and "validation/" directories.

The directory structure will be created as picture.



11.3. classify_imgs():

Purpose: This function is used to create a training folder and validation folder (each folder also has subfolder of dogs and cats) from the source folder we downloaded from Kaggle. We need to create these folders in order to make it easy for Keras to understand and differentiate the animal category of each image.

The ratio is 0.25, the meaning of this number is from 25000 images from dataset, this function will put about 25000 * 0.25 = 6250 images to the validation set and 25000 * 0.75 = 18750 images to the trainning set. We depend on the label or the name of every images in the source folder (train folder) to know it's a dog or a cat. From that we can copy the images from the source folder to the destination exactly by using shutil library. We also use tqdm library to help us look easier when Jupyter notebook running.

11.4. define_model:

Purpose: define model used (5 blocks).

Firstly, we need to initial models Sequential.

The first block: Create Convolutional Layers: Conv2D is convolution used to get features from images with parameters:

- filter: number of filter of convolution. We choose filters = 32 in this block because in most CNN architectures, a common practice is to start with 32 feature detectors and increase to 64, 128, 256, 512 for the blocks of 2^{nd} to 5^{th}.
- kernel_size: size of the search window on the image (kernel_size = (3,3))
- activation: in this block we choose the "relu" function because the ReLU is the most used activation function in the world right now.
- kernel_initializer: The neural network needs to start with some weights and then iteratively update them to better values. The term kernel_initializer is a fancy term for which statistical distribution or function to use for initialising the weights. We use "he_uniform" because corresponding to ReLU, we use he initialization.
- padding: can be "valid" or "same". With same it means padding = 1.
- input_shape: the shape of input images on which we apply feature detectors through convolution. The problem is that images in the dataset have different formats and image size. So, we need to convert images into the same format and fixed size. We will process the image to be (224, 224, 3). Here, 3 is the number of channels for a colored image, (224, 224) is the image dimension for each channel, which is enough for good accuracy. We choose (224, 224, 3) because we train the model on GPU.

After that, we use maxpooling2D function for 1^{st} block to reduce size of feature map, we will use max pooling with size 2x2 (2x2 → 1 node)

The blocks of 2^{nd} to 5^{th} are similar to 1^{st} block, but the number of filters is different (64, 128, 256 and 512 respectively).

After finished 5 blocks, we flatten all data to single vector.

After converted an image into a one-dimensional vector. We build a classifier using this vector as the input layer. First, create a hidden layer. *output_dim* is the number of nodes in the hidden layer. As a common practice, we choose 512 to start with, ReLU as the activation function and kernel_initializer is "he_uniform". After that we need to add an output layer. For binary classification, output_dim is 1, and the activation function is Sigmoid.

With all layers added, let's compile the CNN by choosing an SGD algorithm, a loss function, and performance metrics. We use "*binary cross entropy*" for binary classification because the outcome of this project is classify dog and cat, if it's a cat then return 0, otherwise return 1 if it's a dog, and we don't use the "*categorical_cross entropy*" because it's used for multiple classification problem. The optimizer is Stochastic Gradient Descent optimizer (SGD) with the learning rate is 10^{-3} and momentum is 0.9.

11.5. summarize_diagnostics(history):

Argument: history is information to draw graph (we get this variable after running start_training function).

Purpose: draw graph of loss and validation loss.

We will draw 2 graphs → we use pylot.subplot(211) to draw 2 graphs in same picture, 211 means picture will have 2 rows, 1 column and this is 1st graph. Similar to 1st graph, we will use this method again to draw 2nd graph but at this time, argument will be 212.

Use pylot.title to name the graph (name will be passed through argument).

Use pylot.plot to draw a graph, each graph we will draw 2 lines (loss and val_loss properties of history) argument of this method:

- 1st argument: information about x and y axis (we already provide history to do this thing).
- 2nd argument: color of line.
- 3rd argument: label of current line.

We do the same thing to graph 2. We use 1st element of list of commandline argument passed to the Python program, after that we split it and get last element.

We will save file with name = filename + =_plot.png by method pylot.save.

Finally we close pylot by pylot.close().

11.6. start_training():

Purpose: Training our model

Firstly, we need to define our model by using define_model function. We perform image augmentation, such as rotating, flipping, or shearing to increase the number of images. It splits training images into batches, and each batch will be applied random image transformation on a random selection of images, to create many more diverse images so that our model would never see twice the exact same picture. This helps prevent overfitting and helps the model generalize better. In Keras this can be done via the *keras.preprocessing.image.ImageDataGenerator* class.

The agument we used in the trainning set is:

- rescale is a value by which we will multiply the data before any other processing. Our original images consist in RGB coefficients in the 0 - 255, but such values would be too high for our models to process (given a typical learning rate), so we target values between 0 and 1 instead by scaling with a 1/255 factor.
- width_shift and height_shift are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- horizontal_flip is for randomly flipping half of the images horizontally -relevant when there are no assumptions of horizontal assymetry (e.g. real-world pictures).
- there are more options we can do with our image like fill_mode, zoom_range…, but in this project we just some options.

Unlike the trainning set, the validation set we don't need to do much. After that, we will use *flow_from_directory(directory)* method from Keras to load images and apply augmentation. This is why we structured the data folders in a specific way by using the *classify_imgs()* function so that the class of each image can be identified from its folder name. This is a generator that will read pictures found in subfolers of *"C:/Users/ADMIN/Desktop/AI/Project3/new_data_sets/train/"* or *"C:/Users/ADMIN/Desktop/AI/Project3/new_data_sets/validation/"*, (you can change folder if your computer is different) and indefinitely generate batches of augmented image data.

```
# apply image augmentation on train set by resizing all images to 224x224 and creating batches of 15 images.
training_set = train_datagen.flow_from_directory('C:/Users/ADMIN/Desktop/AI/Project3/new_data_sets/train/',
                                                 target_size = (IMG_SIZE, IMG_SIZE),
                                                 batch_size = batch_sizes,
                                                 class_mode = 'binary')

# apply image augmentation on train set by resizing all images to 224x224 and creating batches of 15 images.
validation_set = valid_datagen.flow_from_directory('C:/Users/ADMIN/Desktop/AI/Project3/new_data_sets/validation/',
                                           target_size = (IMG_SIZE, IMG_SIZE),
                                           batch_size = batch_sizes,
                                           class_mode = 'binary')
```

- target_size is 224x224, so that all images will be resized to 224x224.
- batch_size is a term used in machine learning and refers to the number of training examples utilized in one iteration. In this project we define batch_size equal to 64.
- class_mode is "binary" because we use binary_crossentropy loss, we need binary labels.

We can now use these generators to train our model by applying model.fit_generator. It's a deep learning libraries which can be used to train our machine learning and deep learning models. We use this library instead of *model.fit* because .fit is used when the entire training dataset can fit into the memory and no data augmentation is applied. On the other hands, *.fit_generator* is used when either we have a huge dataset to fit into our memory or when data augmentation needs to be applied. All of the agument we use in the *.fit_generator* library is:

```
history = model.fit_generator(training_set,
          steps_per_epoch = len(training_set),
          epochs = 50,
          validation_data = validation_set,
          validation_steps = len(validation_set))
loss,accuracy = model.evaluate_generator(validation_set, steps=len(validation_set))
print('> %.3f' % (accuracy * 100.0))
model.save('model_v1.h5')  # always save weights after training or during training
```

- object (trainning_set) : the Keras Object model.
- steps_per_epoch: it specifies the total number of steps taken from the generator as soon as one epoch is finished and next epoch has started. We can calculate the value of steps_per_epoch as the total number of samples in your dataset divided by the batch size. But in this project we assign steps_per_epoch equal to length of the the training_set.
- epochs (50): an integer and number of epochs we want to train our model for.
- validation_data: an be either: an inputs and targets list, a generator or an inputs, targets, and sample_weights list which can be used to evaluate the loss and metrics for any model after any epoch has ended. Our group use the generator (validation_set).
- validation_steps: only if the validation_data is a generator then only this argument can be used. It specifies the total number of steps taken from the generator before it is stopped at every epoch and its value is calculated as the length of the validation_set.

After trainning our model, we need to know how much accuracy is it, so we apply the *model.evaluate_generator* which require more than 2 argument but we only use 2 most important argument is the generator (validation_set) and the total number of steps (batches of samples) to yield from "*generator*" before stopping. Optional for "*Sequence*" is if unspecified, we will use the "*len(generator)*" as a number of steps. We also use the steps = length of validation_set. Finally, and the most important thing to save our time is we need to save our model weights after training or during training for future use.

11.7. load_image(filename):

Argument: filename is name of image.

Purpose: load and format image for predict (classify).

Use load_img to load image with filename and format it to 200x200.

Use img_to_array to convert image to array to use reshape.

Use reshape to format image to a single sample with 3 channel (use for predict) and return it.

```
def load_image(filename):
    # load the image
    img = load_img(filename, target_size=(200, 200))
    # convert to array
    img = img_to_array(img)
    # reshape into a single sample with 3 channels
    img = img.reshape(1, 200, 200, 3)
    # center pixel dat
#     img = img.astype('float32')
#     img = img - [123.68, 116.779, 103.939]
    return img
```

11.8. run_example():

Purpose: classify an image is dog or cat.

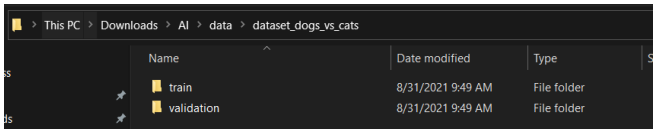
This function is used to predict one image, it's only can predict is it a cat or a dog, no other type of animal. Because we train the model on the dogs and cats dataset and we also use binary classification. We don't need to train the model every time we want to predict an image. By saving the weights in .h5 file, we can load it and predict very fast. If the result is 1, it's a dog. Otherwise, the result is 0 if it's a cat.

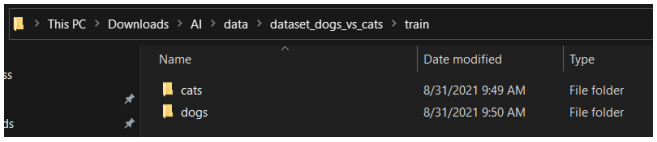
12. Result:

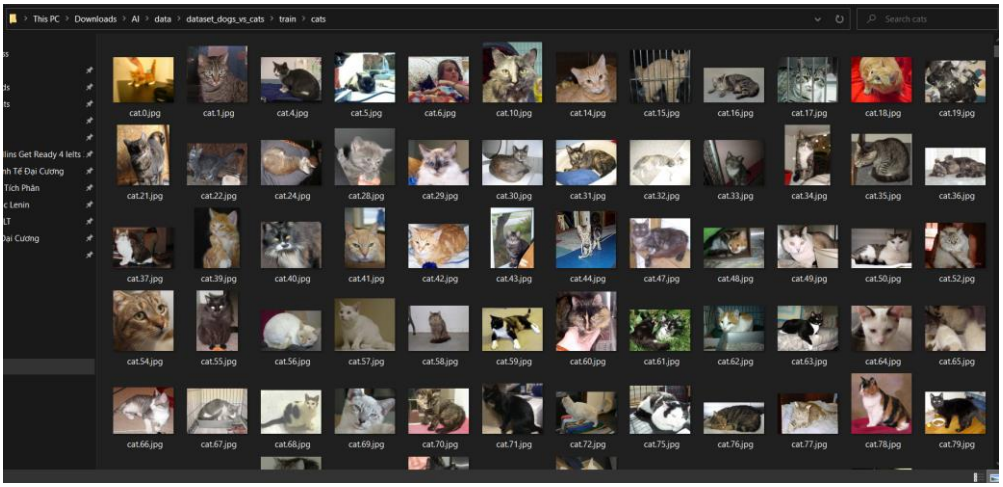
12.1. Create Training and Validation Set:

My empty folder is *C:\Users\Admin\Downloads\AI\data\dataset_dogs_vs_cats* (will explain in Run Code). After run code, we will have 2 subfolder which are train and validation. In each folder we have 2 subfolder are dogs and cats which hold images.

```
classify_imgs()
100%|██████████| 25000/25000 [00:27<00:00, 898.65it/s]
```







12.2. Training to get model_v1.h5 file and png file:

Because input is 25 000 images, it's very big, so we divide it into 50 epochs => we will pick one by one epoch to train.

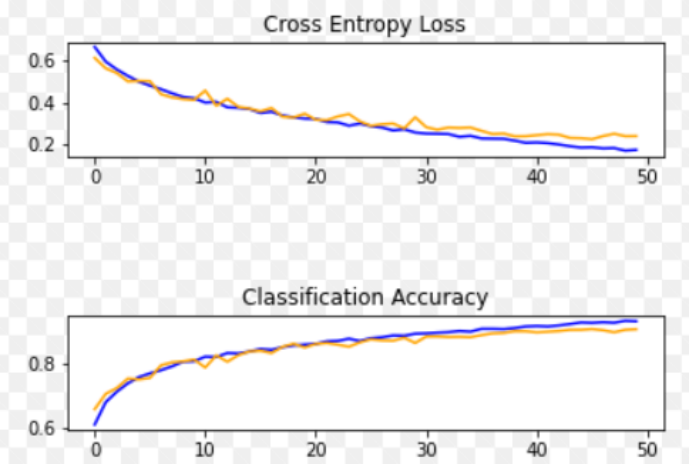
As in explain mode, we get 25% of images to make validation test => about 6211 images, the rest is for training test.

```
Found 18789 images belonging to 2 classes.
Found 6211 images belonging to 2 classes.
```

We can get loss, accuracy, validation loss and validation accuracy. Get total accuracy is 90.565% and get a model_v1.h5 file (use to classify images).

```
Epoch 1/50
294/294 [==============================] - 320s 1s/step - loss: 0.6678 - accuracy: 0.6095 - val_loss: 0.6150 - val_accuracy:
0.6572
Epoch 2/50
294/294 [==============================] - 205s 697ms/step - loss: 0.5971 - accuracy: 0.6790 - val_loss: 0.5651 - val_accurac
y: 0.7047
Epoch 3/50
294/294 [==============================] - 208s 707ms/step - loss: 0.5595 - accuracy: 0.7114 - val_loss: 0.5422 - val_accurac
y: 0.7227
Epoch 4/50
294/294 [==============================] - 211s 716ms/step - loss: 0.5285 - accuracy: 0.7375 - val_loss: 0.5023 - val_accurac
```

And after run summarize_diagnostic function, we get a picture:



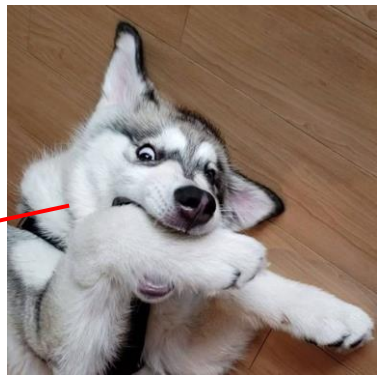
12.3. Predict an image:

Finally, we can predict whether an image is cat or dog.

```
def run_example():
    # load the image
    img = load_image('C:/Users/Admin/Downloads/AI/Cho1.jpg')
    # load model
    model = load_model('C:/Users/Admin/Five_Block.h5')
    # predict the class
    result = model.predict(img)
    if result[0] == 1:
        print("dog")
    else:
        print("cat")


# entry point, run the example
run_example()
```

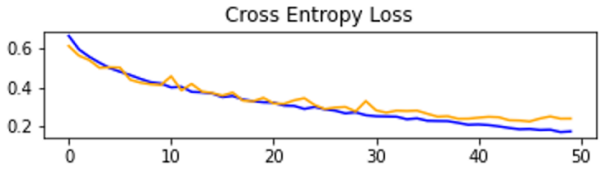
```
dog
```

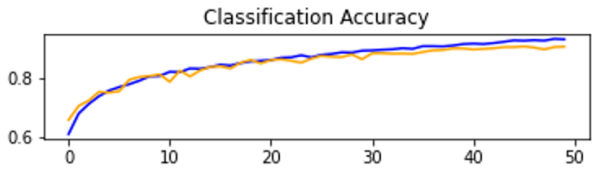


13. Advantages and Disadvantage of 5 blocks model:

Advantages

- Our model automatically detects the important features without any human supervision by using maxpooling layers.
- When make prediction we don't need to train our model again, we just need to load our model which is already saved in .h5 file. The prediction is pretty fast.
- We are working on 25.000 images of dogs and cats, and our model is good to model with nonlinear data with large number of inputs.
- The accuracy of our model is > 90%. To our group, with just 5 blocks, that accuracy is good.
- For this model, we don't need more time to training than other models
- The data may produce output even with incomplete information.
- Less complex, easy to design and maintain.
- Highly responsive to noisy data.





Two figures above show a line plot for the loss and another for the accuracy of the model on both the train (blue) and validation (orange) datasets. From 2 figures, we can see that overfitting has been reduced.

Disadvantages

- Our model is so simple, that maybe not perform well when it makes a prediction.
- Our model is significantly slower due to an operation such as max pooling.
- Although our model has only five layers, the training process still takes a lot of time because the computer doesn't consist of a good GPU.
- Our model requires a large dataset (25000 images) to process and train the neural network, if we use a smaller dataset, its performance is not good enough to due with several images.

- With the same image under different light, angle and shade, and the image maybe contain some degree of tilt or rotation then model usually have difficulty in classifying the image so that its prediction is false.
- Our model depend a lot on the training data. This may leads to the problem of overfitting and generalization. The mode relies more on the training data and may be tuned to the data.

14. Suggest ideas for improvement:

Increase image size: We change model with the image size is 200x200, the accuracy is > 80%. When we change the image size to 224x224, the accuracy is better > 90%. So that our group think that, by increasing the image size, the accuracy is more and more good.

Adding more convolution layers or adding more dense layers. Use of L1 and L2 regularization (also known as "weight decay"). Fine-tuning one more convolutional block (alongside greater regularization).

Using VGG-16 with 16 layers that at the time it was developed, achieved top results on the ImageNet photo classification challenge. The model is comprised of two main parts, the feature extractor part of the model that is made up of VGG blocks, and the classifier part of the model that is made up of fully connected layers and the output layer. We can use the feature extraction part of the model and add a new classifier part of the model that is tailored to the dogs and cats dataset. Specifically, we can hold the weights of all of the convolutional layers fixed during training, and only train new fully connected layers that will learn to interpret the features extracted from the model and make a binary classification. We don’t call the library of VGG-16 model, we try to create a model with all layers like VGG-16 model. But our computer is out of memory so that we can’t train on this model to see the efficient of this model.

Dropout regularization is a computationally cheap way to regularize a deep neural network. Dropout works by probabilistically removing, or “dropping out,” inputs to a layer, which may be input variables in the data sample or activations from a previous layer. It has the effect of simulating a large number of networks with very different network structures and, in turn, making nodes in the network generally more robust to the inputs. By applying dropout, the accuracy is increase but it’s just a small change.

Image data augmentation is a technique that we used in this project. It can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset. Training deep learning neural network models on more data can result in more skillful models, and the augmentation techniques can create variations of the images that can improve the ability of the fit models to generalize what they have learned to new images. Data augmentation can also act as a regularization technique, adding noise to the training data, and encouraging the model to learn the same features, invariant to their position in the input. Small changes to the input photos of dogs and cats might be useful for this problem, such as small shifts and horizontal flips. These augmentations can be specified as arguments to the ImageDataGenerator used for the training dataset. The augmentations should not be used for the test dataset, as we wish to evaluate the performance of the model on the unmodified photographs. This requires that we have a separate ImageDataGenerator instance for the train and test dataset, then iterators for the train and test sets created from the respective data generators.

15. Run code:

15.1. 19127478_19127622_19127640_Code.ipynb file:

Firstly, you need to create an empty folder (because we do not know you want to place data for train in where) and a train folder with 25000 picture from Kaggle. (for example, we will create a folder whose name is dataset_dogs_vs_cats).

Empty folder: *C:\Users\Admin\Downloads\AI\data\dataset_dogs_vs_cats*

25000 images folder: *C:\Users\Admin\Downloads\AI\train*

You need to run code from beginning to ending for sure that code will run without error.

Before run code, you need to change link in somewhere below:

```
def create_parent_folder():
    # Parent Directory path
    parent_dir = 'C:/Users/Admin/Downloads/AI/data/dataset_dogs_vs_cats/'
    # Directory
    childs_dir = ['train/', 'validation/']
    labels_dir = ['dogs/', 'cats/']
    for directory in childs_dir:
        for label in labels_dir:
            path = parent_dir + directory + label
            os.makedirs(path)
    return parent_dir
```

Replace by link to empty folder

```
def classify_imgs():
    parent_dir = create_parent_folder()
    # copy image to new specified folder
    src_directory = 'C:/Users/Admin/Downloads/AI/train/'
    val_ratio = 0.25
    for file in tqdm(os.listdir(src_directory)):
        src = src_directory + file
        dst_dir = 'train/'
        if random.random() < val_ratio:
            dst_dir = 'validation/'
        label = label_of_image(file)
        if label == 1:
            dst = parent_dir + dst_dir + 'cats/'  + file
            shutil.copyfile(src, dst)
        elif label == 0:
            dst = parent_dir + dst_dir + 'dogs/'  + file
            shutil.copyfile(src, dst)
```

Replace by 25000 images folder

```
def start_training():
    model = define_model()
    # create an object of ImageDataGenerator, for augmenting train set
    train_datagen = ImageDataGenerator(rescale=1.0/255.0,
            width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
    # create an object of ImageDataGenerator, for augmenting validation set
    valid_datagen = ImageDataGenerator(rescale=1.0/255.0)


    # apply image augmentation on train set by resizing all images to 224x224 and creating batches of 15 images.
    training_set = train_datagen.flow_from_directory('C:/Users/ADMIN/Desktop/AI/Project3/new_data_sets/train/',
                                                     target_size = (IMG_SIZE, IMG_SIZE),
                                                     batch_size = batch_sizes,
                                                     class_mode = 'binary')

    # apply image augmentation on train set by resizing all images to 224x224 and creating batches of 15 images.
    validation_set = valid_datagen.flow_from_directory('C:/Users/ADMIN/Desktop/AI/Project3/new_data_sets/validation/',
                                                       target_size = (IMG_SIZE, IMG_SIZE),
                                                       batch_size = batch_sizes,
                                                       class_mode = 'binary')
```

Replace by link to subfolder of empty folder (after run code from beginning, empty folder will have 2 subfolders which are train and validation)

```
def run_example():
    # load the image
    img = load_image('C:/Users/Admin/Downloads/AI/Cho1.jpg')
    # load model
    model = load_model('C:/Users/Admin/model_v1.h5')
    # predict the class
```

Replace by link of image that you want to classify.

Replace by link of model v1.h5 file.

Summarize, after running all code, we will have:

- 2 subfolders of empty folder which also 2 subfolders cat and dog contain images.
- Model_v1.h5 file.
- File _plot.png file.
- Result classification.

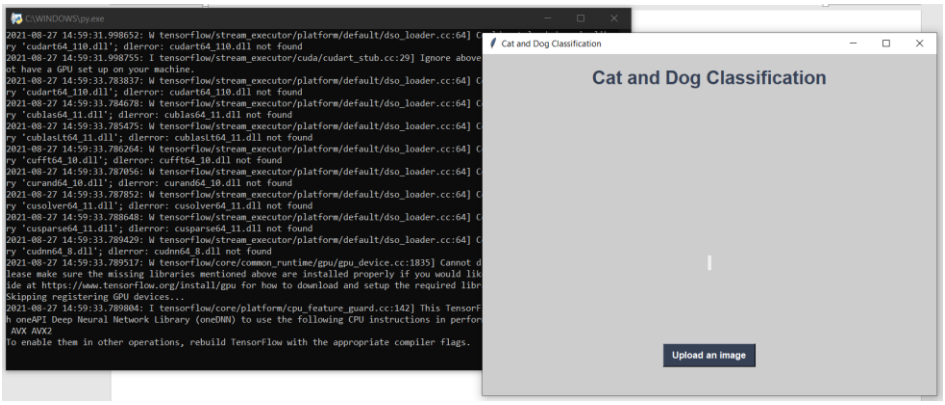
15.2. gui.py file:

To run this file, you will have to run code in *19127478_19127622_19127640_Code.ipynb* file first to get model_v1.h5 file.

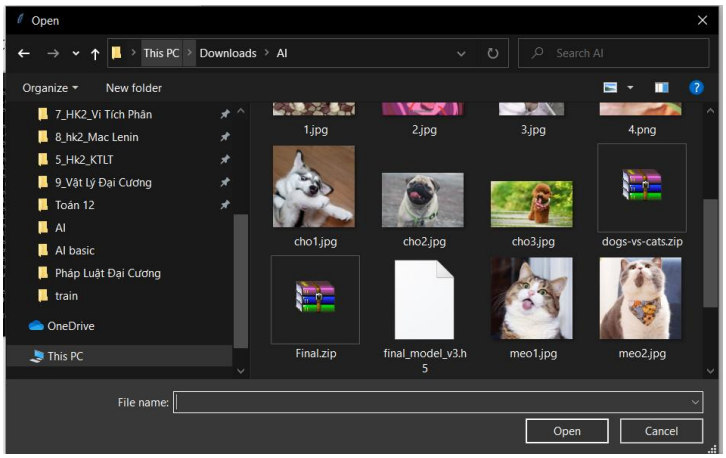
After that, open gui.py file and replace link to model_v1.h5 file in your computer.

```
10  from numpy.lib.type_check import imag
11  model = load_model('C:/Users/Admin/model_v1.h5')
12  #dictionary to label all traffic signs class.
13  #initialise GUI
```

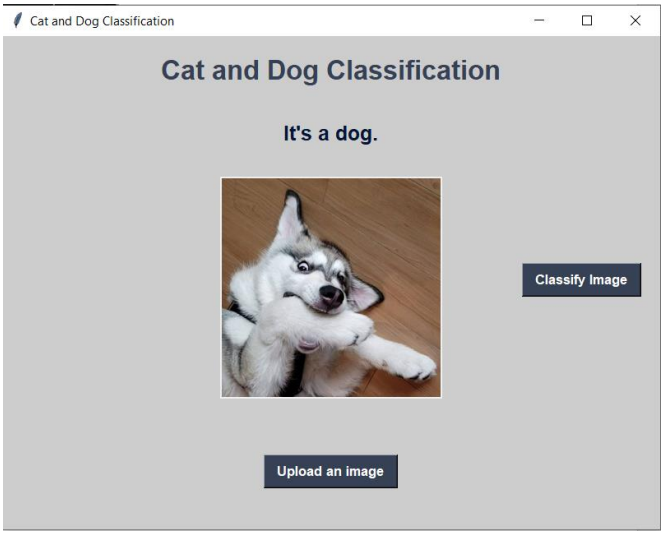
Now you just need to run gui.py file by click on it and a page will open for you:



Click upload an image button to choose image need to predict:



Click classify image to get a result.



16. Youtube video URL:

Demo Dog and Cat Classification

17. References:

[1] Chau Thanh Duc. *Introduction to Artificial Intelligence* lectures *(2021)*. Facalty of Information Technology, VNU-HCMUS.

[2] Nguyen Ngoc Thao, Nguyen Hai Minh. *Introduction to Artificial Intelligence* materials *(2021)*. Facalty of Information Technology, VNU-HCMUS.

[3] Beginner-friendly Project- Cat and Dog classification using CNN

[4] Build a Convnet for Cat-vs.-Dog Classification

[5] How to Classify Photos of Dogs and Cats (with 97% accuracy)

[6] Cats vs Dogs Classification (with 98.7% Accuracy) using CNN Keras – Deep Learning Project for Beginners

[7] "Cats vs Dogs" dataset

[8] Weight Initialization for Deep Learning Neural Networks

[9] Getting acquainted with Keras

[10] CNN algorithm - Convolution neural network

[11] Convolutional neural network

[12] Information about CNN architecture