By: Nasrullah Rami Hamad - 21110510

CNN + MLP Project – Deep Learning

Technical Report

Course: Deep Learning – Spring 2023/2024

Course instructor: Dr. Ala’a Al-Habashneh

**IMPORTANT: Remove the descriptions in the sections of this template and do not include them in your final report. These are meant to help you only.**

**PART-1: Implementation of an MLP from scratch**

Remember: This is this first project in the assignment, where you are not allowed to use packages (e.g., nn) to build your MLP.

# Design and implementation of MLP network from scratch

## Design

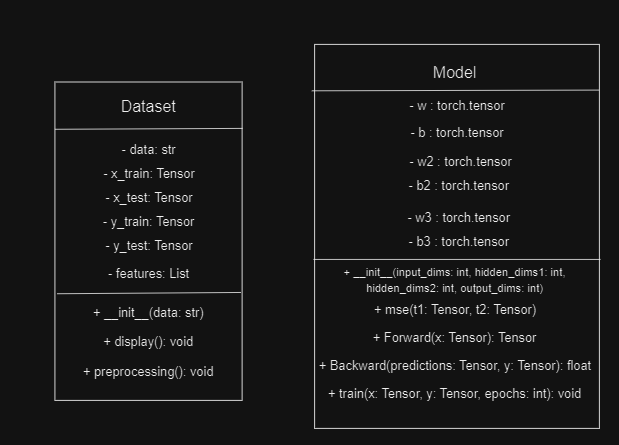
I worked on a regression problem that studies student performances in a school. The data was collected manually. And it consists of 5 independent variables and 1 dependent variable. Their datatypes are int, float, and object. The MLP I implemented consisted of the following components:

* Inputs and outputs: where I had 5 input features as input, and 1 output since the regression problem is going to generate one continuous value. (The student performance)
* Input, Hidden, Output layers: where the input layer takes the input features. The hidden layers are set between the input and output layers, and they’re responsible for learning the data patterns with the predicting process through nonlinear transformations inside the layers. The output feature is the final layer of the MLP structure and it grants us the predicted value.
* Activation function: I used the TanH activation function for the hidden layers since it generates values between -1 and 1 which would grant a faster training process for the model. And it is more stable than sigmoid.
* The Loss function: I used the MSE (Mean Squared Error) loss function, because it is commonly used in regression problems, and it gives an appropriate representation of the errors for the MLP implementation.
* Optimization function: Such as the gradient descent, optimization functions optimize and stabilize the work of the MLP model. It presents a more acceptable format of clear predictions and accurate insights. I used Adam as an optimization function due to the learning rate properties it owns.
* Random weight and bias initialization, I implemented random weights for each feature of the dataset, with a random initialization of bias. Each hidden layer was given random generations of weights. The weights go into the forward function inside the model.

In my implementation of an MLP, I used different numbers of hidden layers for five different combinations, to test how the model would work on different parameters. I used between 2 and 5 hidden layers. The combination with the highest amount of hidden layers gave me the best r2 score.

I implemented a **UML** **Diagram** for the MLP implementation that consisted of two classes, Dataset, which controlled the flow of the dataset, and its pre-processing before it is trained to the MLP model. The class is made from three functions/methods. The init function is the initialization function and it took one parameter which is the Data. The data parameter is going to take the link of the dataset. The display function displayed the dataset and its features and some of its rows using df.head(). The preprocessing() function processed the data and made it ready to be fit inside the MLP model in the next class.

The Model class consisted of five functions, each had necessary part in the implementation of the MLP model. First, the init method took 4 parameters, input dims, hidden dims1, hidden dims2, and output dims. Each parameter took input as the number of neurals in each layer. Noting that since there are two hidden layer parameters, this means that there are two hidden layers inside this MLP combination. The mse function controls the loss function process. It takes two tensors and calculates the mean squared error between them. The Forward function consists of the model equation, it takes the input as a tensor. The backward function controls the backpropagation process in the model. It takes the predictions and the actual values of the targets. The train function takes x and y and the number of epochs as an integer and it starts the modelling process of the MLP implementation.



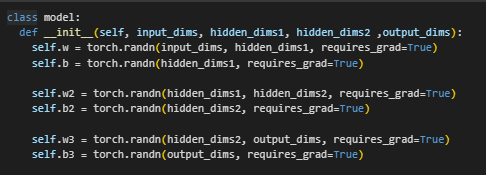
## Implementation

Inside the MLP implementation of the regression model, there was multiple classes, Dataset and Model, each with different functions, attributes, and modules. The OO design principle was also considered in each class.

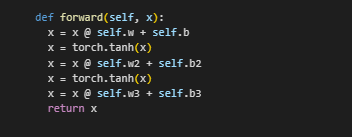
The class Dataset, combined three functions which were the init function and it took the path of the dataset as an object. The display function displayed the dataset with its features and showed the first few rows with df.head. The proprocessing function read the dataset, processes it by encoding and scaling it. Then finished by splitting the data to training and testing data. The attributes inside this class were the “data” which was the link of the dataset. “X\_train”, “X\_test”, “Y\_train”, “Y\_test”. Which hold the training and testing splits of the data as tensors. And the features attribute which takes the features of the dataset. This class held the start of the implementation of the MLP architecture for the regression problem. And it combined many principles of the Object Oriented design, such as Encapsulation and modularity. Since it focused on a single task which was managing the dataset and processing it so it can be fit and trained inside the model in the following class. It also held the attributes inside the class to provide interactions with the dataset within the model itself.

The class Model, combined five different functions with each having a single task required to finish the MLP implementation and build. The first function was the initialization function that took four parameters each with the number of neurons in each layer, starting from the input to the output layers. The MSE function took two tensors and calculated the mean squared error loss between them resulting in an error value. The forward function takes the X value and computed the modelling process based on its equation. The backward function calculates the backpropagation process inside the modelling step. It takes two parameters, the predicted y value and the actual Y value. This step is used to update the weights of the input and hidden layers, so they proceed in a better performing model on upcoming epochs and iterations. Gradient Descent is used to update the weights here. The train function controls the training of the model, by taking the features, the target variable and the number of epochs. The attributes inside this class consisted of the weights and bias variables, w, b, w2, b2, w3, b3. And same as the previous class, this class encapsulated the attributes and the operations for them within the class. And the class was tasked to operate and build and train the model so the code runs smoothly, correctly, and efficiently.

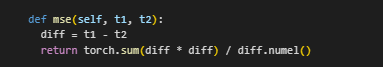
The usage of modules was considered in the MLP implementation, especially inside the Model class. Which combined four modules of torch’s nn module. Such as Linear transformation, activation functions, the loss function, forward, and backward propagation.



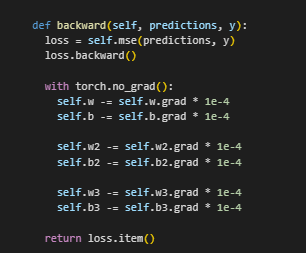
In this picture, the random initialization of the weights and biases inside the model is presented, we initialized w, w2, w3 as the weight matrices, and b, b2, b3 as bias vectors. We used requires\_grad to compute the gradient descent for them when backpropagating.



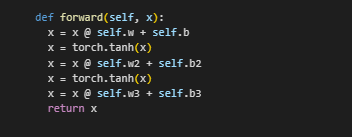
The activation function TanH between the hidden layers was used to further enhance the model and give values between 1 and -1 as outputs. It was used to use non-linearity inside the neural network and to get a faster processing speed.



This picture shows the function that carries the loss function (MSE) which was used to calculate the mean squared error between two tensors. The tensors being the predicted and actual target labels.



This picture shows the backpropagation part of the MLP implementation, using the backward function. We used loss.backward() to calculate the gradients with respect to the parameters using the Gradient descent optimization function.



This picture shows the forward function that takes the X parameter which is the input of the model. Then it uses dot product to multiply it with the weight matrix of the corresponding layer, then combines it with the bias vector of the corresponding layer. After each equation and going inside another layer, we use the tanh activation function for X. at the end of the last hidden layer, we will return X and get the final result. The equation being SUM(dot(X, w) + b).

# Training and testing of the MLP

## Dataset

The dataset I used talks about Student performances inside a school. It is a regression based problem, consisting of 10,000 instances with 6 features. 5 independents and 1 dependent as the following:

1. **Hours studied**. The first independent variable is an integer variable, it stores the hours that students studied for their exams.
2. **Previous scores**. Which stores any previous scores the students took in former exams in previous periods so they can be compared with in this year’s scores. It is an integer variable.
3. **Extracurricular Activities**. This variable is an object variable that talks about any activities the students are participating in the school.
4. **Sleep Hours**. This variable shows how many hours the students sleep in a day, it is an integer variable.
5. **Sample Question Papers Practiced**. Which shows how many mock exams each student practices before their exam. It is an integer variable.
6. **Performance Index**. Is the target variable, and it stores the performance of each student, It is a float variable.

The features that are going to be fit inside the model are going to be Hours studied, Previous Scores, Extracurricular Activities, Sleep hours, Sample question papers practiced. They are going to predict the Performance Index which is the target variable of the problem.

## Training and validation

The data inside the dataset was split between training, testing, and validation. I used two splitting steps, first to bring the data down to training and testing. And the second step to bring the training data down to train and validation data. First step had a test size of 0.2, while the second step had a test size of 0.25. So, the 10,000 instances of my dataset were split first between 8,000 instances for training and 2,000 for testing. Putting the testing data aside. We then used the 8,000 training instances and split them down to 6,000 training data and 2,000 Validation data.

By doing this three-way split, I ended up with 6,000 instances for training, and 2,000 for each testing and validation data. After doing the splitting. I tried different combinations of hyperparameters to test different outcomes and performances for the student performance MLP model. The hyperparameters I changed were the numbers of hidden layers, the learning rate, the initialization of the hidden weights, and the number of epochs. This table shows each hyperparameter and the best combination I found.

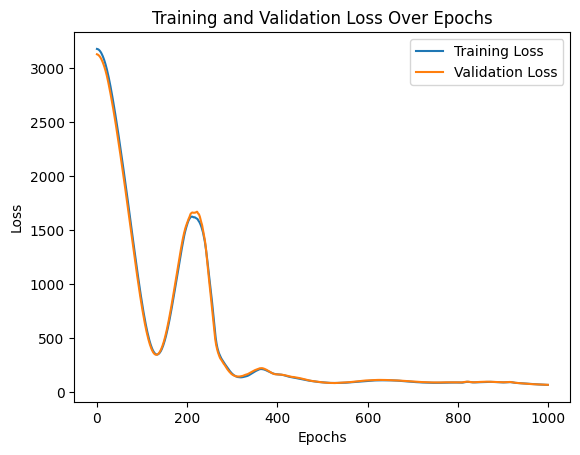
Show the learning curve (training and validation performance vs epochs) for the combination that gave you the best validation performance.

Table 1: Description of the hyperparameters considered and best value for each

|  |  |  |
| --- | --- | --- |
| **Hyper-parameter** | **Description** | **Value** |
| **# Of Hidden Dims** | The MLP model consists of Input, Hidden, and Output. This number controls the number of hidden layers within the model. This can be increased or decreased by increasing the amount of weight and bias initialization layers inside the MLP implementation, 4 means there are 4 weight and bias initializations. | 4 |
| **Learning Rate** | The learning rate controls the response of the model to the error when weights get updated inside the model’s training phase. A good learning rate can generate great results | 0.00001 |
| **Hidden Layer Initialization** | Controls the number of neurons inside each hidden layer. It Is recommended to have a higher neuron number for the layers at the beginning of the model for better performance | Hidden layer 1=60  Hidden layer 2=45  Hidden layer 3=30  Hidden layer 4=10 |
| **Epochs** | The number of iterations the model has to go through to obtain results. While updating the model’s weights each iteration for better results | 1000 |

Table 2: Combination of the hyperparameter values and corresponding performance achieved

|  |  |  |
| --- | --- | --- |
| **Combination of hyperparameter values** | **Training performance** | **Validation performance** |
| **2 / 0.0001 / 20 – 10 / 1000** | 0.8025 | 0.8036 |
| **3 / 0.00001 / 40 – 15 – 10 / 2000** | 0.9308 | 0.9337 |
| **2 / 0.001 / 35 – 15 / 2000** | 0.8537 | 0.8542 |
| **4 / 0.0001 / 50 – 40 – 30 – 10 / 1000** | 0.7804 | 0.7868 |
| **4 / 0.00001 / 60 – 45 – 30 – 10 / 1000** | 0.9592 | 0.9647 |



This learning curve was generated from the best combination which consisted of four hidden layers, a learning rate of 0.00001, with hidden layer initializations of 60, 45, 30, and 10, and the number of epochs or iterations was 1000. The training loss and validation loss started from nearly 3,300. Then went drastically down to 500 at the 150th epoch. Before going up on 1500 at the 200th Epoch. Then both losses went straight down below 500 at the 300th epoch. And went in a straight line constantly at loss = 90 until the end of the 1000 epochs.

## Testing

For the testing phase, I involved the same steps, but I worked on the Testing data which consists of 2,000 instances from the dataset. I managed to correct the code and the MLP implementation so it takes the testing data instead of the validation data. The testing phase is used to ensure that the model is reliable and correct. And to ensure that it has a good performance.

The Testing phase started with loading, displaying, and preprocessing the data using the Dataset class. Which handled all the encoding, scaling, and missing values of the dataset. It also showed a overview of the dataset and its features and some of the instances. Then the initialization of the MLP model started taking place, so I initialized the weight matrices, bias vectors and the functions. Then I used the same functions as the validation set. The forward, MSE, backward, and training. The testing phase was evaluated on four different metrics. R2 score, MSE, MAE, and RMSE. As follows:

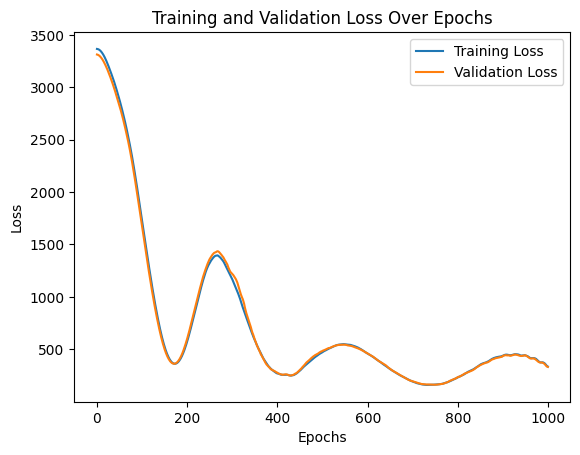
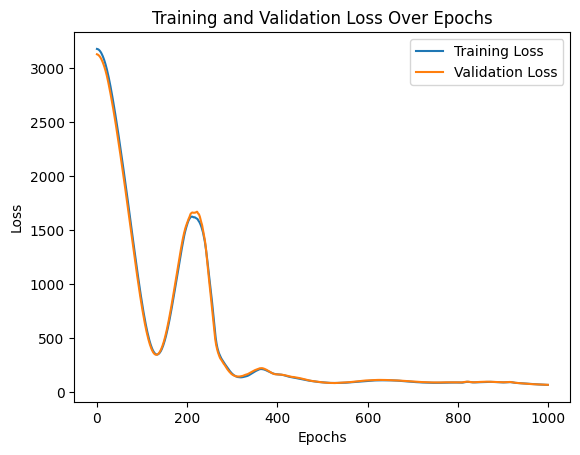
Table 3: Best values for the evaluation metrics on the test set.

|  |  |  |
| --- | --- | --- |
| **Evaluation metric** | **Description** | **Value obtained** |
| **R2** | shows how well the model’s predictions match the actual data based on the proportion of the variance | 0.93 |
| **MAE** | Shows the average distance between the predictions and the actual values without considering the direction | 3.91 |
| **MSE** | Shows the average of squared differences between the predictions and the actual values, and it is sensitive to outliers | 24.6 |
| **RMSE** | The square root of the MSE. Which shows the STD of the predicted errors | 4.96 |

To further validate the implementation of the MLP without using libraries (From scratch), I implemented a model that follows the same hyperparameter tuning I used in the best combination I had in the scratch method. To compare it with its performance. These are the hyperparameter I used in the implementation of the nn module MLP model.

* Number of hidden layers: 4
* Learning Rate: 0.00001
* Initialization of hidden layers: 60/ 45/ 30/ 10
* Epochs = 1000

After initializing an MLP model with these parameters, I plot a learning curve that will show how the errors are compared to the MLP scratch model.



The plot on the right is the implementation using the NN module, and it shows a nearly identical learning curve when compared with the MLP from scratch implementation, since both losses started at the 3300 Mark. Then came down at the 180th epoch, with an error equal to 500. Then went up a bit at the 300th epoch mark to a value of 1500. Then went down and nearly went into a straight line to the end of the 1000th epoch line.

This indicates a good validation of the model and its performance; it also shows that the model is good performing even if it was implemented in another way. Plus, it shows that the combinations I used are efficient when dealing with other MLP implementations.

## Evaluation of the developed MLP

The design of the MLP has been very efficient with the different combinations, by using the best hyperparameters, and evaluating the model’s metrics and accuracy. And by testing the model on a three-way split and performing the combinations on a torch module implementation model. We can say that the model was very reliable in gathering a good amount of good performance.

At first, I had many bad combinations or bad performances from the mode, which were the result of many bad hyperparameter matching. Since we’re working with only 10,000 instances, some hyperparameters might not be very efficient. So, I tried combining some of the best hyperparameter values I tried, to get the best result.

Some epochs had bad evaluations too due to a bad randomizing initializer for the weight metrices. Which was fixed by running the code again and getting better results. Since we’re using a random weight initializer, the performance for the model can be very random and challenging to find the best hyperparameter tuning for it.

So, there must be some enhancements in different parts of the model and the code itself. As the following:

* Since we’re dealing with deep neural networks, a 10,000-instance dataset may not give very accurate results, since we need to work with a larger amount of data for better results. We must find a bigger dataset to work with such data.
* We can use different hyperparameters inside the neural network which would give us more efficient and accurate results, such as Dropout layers, learning rate schedulers. Which would reduce the learning rate when the validation loss starts increasing. And many other hyperparameters
* We can use other activation functions such as ReLU. TanH is good for hidden layers, but ReLU may perform better because of its simplicity and effectiveness in working with neural networks. We can check other activation functions that work on smaller datasets too.
* Batch sizes can also be introduced into the implementation of the MLP model, since batches may give a better performing time for the model, and better performance overall. Batch normalizations stabilize the speed and the efficiency of the model by normalizing the layer inputs.

**PART-2: Developing a Deep-Learning system for a Computer Vision Application**

Remember: This is a different system that is separate from the previous project. In this system, you are allowed to use deep-learning frameworks/packages (e.g., nn).

# Problem statement

Many big cat variants in the wild are not being accurately detected by the zoologists and other hunters, which can cause a big trouble for some of the zoologists in the forests and many campers. So we wanted to make a good image classification CNN model that classifies the image and predicts which class the image is from. There are 10 classes inside the dataset of images. Which are as the following:

* **African Leopard**
* **Caracal**
* **Cheetah**
* **Clouded Leopard**
* **Jaguar**
* **Lions**
* **Ocelots**
* **Puma**
* **Snow Leopard**
* **Tigers**

The dataset itself consists of nearly 250 images inside each class. And the data is already split into a three-way split (train, test, validate). By getting a big accuracy for this model, we can ensure the safety of campers and individuals inside the forests and the wild by classifying the image correctly into its class.

# Research on the Neural Networks and architectures

## Neural Networks used for the problem

Convolutional Neural networks (CNNs) are commonly used in image classification due to their ability to visualize visual data such as images and videos. They are a form of computer vision that’s widely used in applications such as Object detection, image segmentation and classification. For my project, I’m going to use CNN for an image classification task that’s going to classify the images of cats into their suitable class.

CNNs are effective in image classification due to many features, such as the pooling layers and the convolutional layers. Since CNNs can capture the low-level features in the early layers, and then gradually improve the model into the deeper layers. By also sharing the weights across the entire image input, CNNs can recognize the patterns of such images which ensures the robustness of the model.

The sparsity and parameter sharing of CNN can also reduce the number of parameters when using the CNNs. This can create a strong and robust model without the need of a big number of parameters. Max pooling is also used to decrease the dimensions of features inside the dataset. This helps preserve important features and get rid of bad or useless features.

## Modern architectures

The CNN has many modern architectures that can be used for the image classification task, some architectures are better than others and some can be worse. The architectures I chose were VGG16, ResNet18, MobileNetV2, and DenseNet121.

* VGG16: VGG is a CNN architecture that’s introduced by the GG company at the university of Oxford. It contains 16 layers, each containing learnable weights. They consist of 13 convolutional layers and 3 fully connected layers. The model uses a 3x3 filtering method throughout the neural network. And applies max-pooling layers to reduce the spatiality. The three fully connected layers at the end can include a softmax layer for a classification problem. The model is known for its consistency and depth. Since it uses the 3x3 filters for the convolution layers. And it is used to learn deep and complex data features.

<https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/>

* ResNet18: It is a CNN architecture from the ResNet family, which introduced the idea of the residual learning in neural networks. It contains 18 layers and 8 residual blocks within. They also contain 2 convolutional layers. The idea of residual blocks is that they map the inputs to their output in a block. They work as a connection shortcut inside the neural network. ResNet18 also uses batch normalization to get a better stabilized performance and a faster training process. It is known for its efficiency in image classification tasks since it is shallower than the other ResNet variations. It also helps with the vanishing gradient problem that some neural networks face. Because the residual blocks can help gradients perform the backward propagation more effectively.

[https://www.productteacher.com/quick-product-tips/resnet18-and-resnet50](https://www.productteacher.com/quick-product-tips/resnet18-and-resnet50%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20)

* MobileNetV2: This CNN architecture is designed for mobile applications hence the name. It is a next version of the original MobileNet which introduced residual inversion. It performs a splitting on the convolution layers, into depth-wise and point-wise convolutional layers, to reduce the computational cost, it also uses the ReLU6 activation function inside hidden layers to prevent low performance inside the computations on mobile devices so they perform more smoothly and efficiently. It is known for its flexibility and efficiency for low performing devices, making it fast and light for mobile devices.

<https://www.analyticsvidhya.com/blog/2023/12/what-is-mobilenetv2/>

* DenseNet121: It is a CNN architecture from the densenet family, that is known for connecting layers with other layers in a feed forward way, it uses dense connections which typically mean that each layer takes the output of the previous layer and trains itself on it which can allow the usage of parameter sharing and other features. It uses bottleneck layers to reduce the dimensionality of the data. It also uses batch normalization same as the resnet18 model. Which enables a faster training process and a more stable performance. The DenseNet can reuse the features and allow for an improved gradient flow inside the neural network. Which results in a better parameter efficiency when compared to other models. It is known for outperforming the traditional models in image classification problems.

<https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a>

## Modern architectures comparison

Table 4: Modern architectures used to solve the problem.

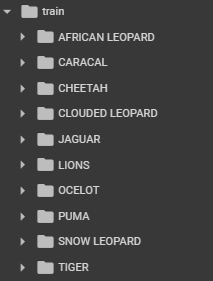
|  |  |  |  |
| --- | --- | --- | --- |
| **Architecture** | **Description and number/types of layers** | **Advantages** | **Disadvantages** |
| **VGG16** | CNN architecture from the VGG family. Consists of 16 layers (13 convolutional layers, 3 fully connected layers) | Simple, deep, can be used in transfer learning, layers have learnable weights | High computational cost, due to number of parameters. It also has many useless features that may affect the model |
| **ResNet18** | CNN architecture from the resnet family. Consists of 18 layers (8 residual blocks, 2 convolutional layers) | Solves the vanishing gradient problem, and is efficient and robust with image classification problems | It has a shallow depth which may not include complex features on the more challenging datasets. We must use more complex versions of resnet for larger datasets |
| **MobileNetV2** | CNN architecture from the mobilenet family, it consists of 53 layers. Which can be depth-wise or point-wise layers. | Used for mobile devices, optimized use, light, and fast. | It performs low since it focuses more on optimization and efficiency. And it can be complex in its implementation |
| **DenseNet121** | CNN architecture from the densenet family. it consists of 121 layers, hence its name. Dense layers, transition layers, and convolutional layer and a fully connected layer | Uses dense layers for a better optimization since it uses a feed forward method. It supports reusing of features. | It has a low memory usage since it is used on limited resources. It can be computationally complex to add more hidden layers in the model. And it has a challenge in choosing the best parameters. |

# Models’ development and training

## Dataset

The dataset I used for the image classification problem was split into a three-way split from the beginning of its implementation (train, test, validation), each split consisted of 10 classes of images. Each class has 250 different images with cats. The type of the images is jpg images. And their resolution is 224X224 which is considered the standard resolution for all images for image classification. The 10 classes of cats are Tiger, Puma, Snow leopard, Ocelot, Lion, Jaguar, Cheetah, Clouded leopard, Caracal, and African leopard. The training set consisted of 250 images for each class. While the testing and validation set consisted of 5 images for each of the 10 classes.





After importing the dataset, It was shown that it had a three-way split already implemented, train, test, and validation. I used the transformers package to normalize, resize, grayscale, and converting the images to tensors, with the compose function

* Resize: First, I managed to resize all the images to 224X224, which is the standard input size for the images that will be fit inside the CNN architectures
* Grayscale: This converts the image to grayscale, then reveals three identical channels for it, so they can be trained on RGB images.
* Tensor: This step ensures that all the images are converted to tensors before normalizing them.
* Normalize: This step normalizes all the image tensors with the mean and standard deviation values.

The dataset and its splitting were stored in a zipped file, so I used the !unzip method to unzip the zipped file and contain the folders inside the google colab application I used. The dataset was cleaned from its implementation, so it was ready to do all the resizing and normalizing processes it needed. Without the need to clean it or pre-process the image files.

I used three Data loaders to load the images from their folders and used batch normalization with shuffling on for the training split only and off for the other two splits.

## Training and validation

The image classification problem consisted of finding the right class for the image of the cats I had, the training process for the VGG16, ResNet18, MobileNetV2, and DenseNet121 was identical and all the CNN architectures mentioned above had four different combinations each, with different hyperparameters each combination. I used the GPU in training the data since it was very computationally expensive on the CPU. As mentioned above, the dataset was split into a three-way split (training, testing, and validation). The training process started with initializing the modelling parameter based on what model I’m going to use in the combination I’m trying. For each model, we must specify the number of layers, the in features and the out features. The in-features and the number of layers might change from model to another. But the out-features stays the same since all the models are working on the dataset with the 10 classes. In VGG16, I specified the number of layers as 6, and the in features as 4096 and the out features as 10 which are equal to the dataset classes I have. Then I specified the loss function which is the Cross Entropy loss, which is very suitable and usable in classification problems. Then I used Adam as an optimizer and selected the hyperparameters momentum and learning rate. I didn’t use the dropout layer in this combination, but I used it in other combinations. I ran some of the combinations on 5 epochs and some on 10 epochs. Which resulted in different results and prediction accuracy. I calculated the time of the modelling process and used the GPU In it too, since it was computationally expensive to run on the CPU.

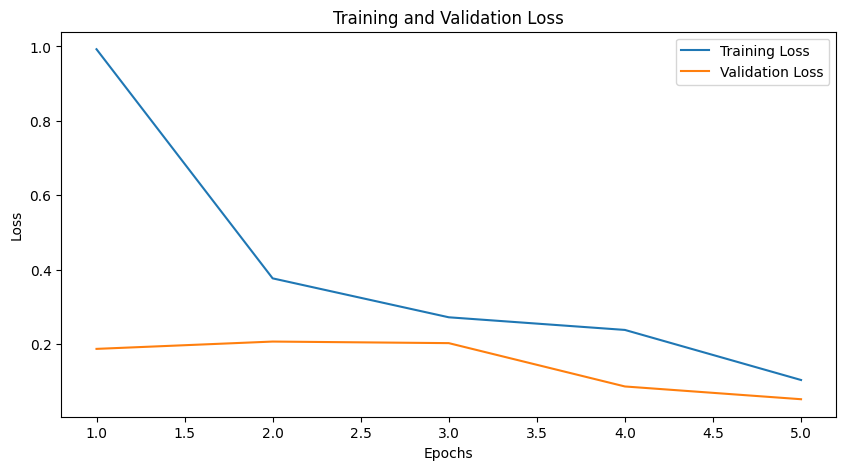
To evaluate the performance of the combinations with different CNN architectures, I used the training and validation accuracy, and the training and validation loss. I also plotted a learning curve showing both the training loss and the validation loss with the number of epochs used, in each of the 16 combinations for the 4 different CNN architecture models. The hyperparameters that I changed in each of the combinations were, Dropout, Momentum, Learning rate, epochs, and the weight decay.

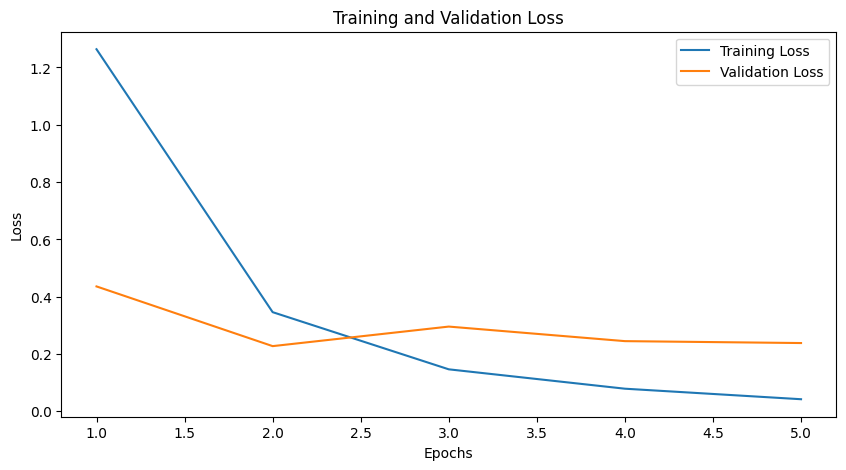
Table 5: Description of the hyperparameters considered and best value for each

|  |  |  |
| --- | --- | --- |
| **Hyper-parameter** | **Description (VGG16)** | **Value** |
| **Dropout Layer** | Regularization technique to avoid overfitting, a value of 0.5 means that half the neurons of the model would be turned off and the training process would happen on the other half | 1 (0.5) |
| **Momentum** | Optimization technique that accelerates the gradient descent by using the past gradients to help with the update rule. It helps the model to reach convergence faster. | Not used |
| **Learning Rate** | Hyperparameter that helps control the step size each iteration while going to the minimum of the cross entropy or the loss function used. An optimal LR value is needed to avoid instability or slow convergence process | 0.0001 |
| **Weight Decay** | Regularization technique to avoid overfitting by penalizing the large weights. This helps keep the weights low so that models can generalize better and perform on unseen data. | Not used |
| **Hyper-parameter** | **Description (ResNet18)** | Value |
| **Dropout Layer** | Regularization technique to avoid overfitting, a value of 0.5 means that half the neurons of the model would be turned off and the training process would happen on the other half | 1 (0.5) |
| **Momentum** | Optimization technique that accelerates the gradient descent by using the past gradients to help with the update rule. It helps the model to reach convergence faster. | Not used |
| **Learning Rate** | Hyperparameter that helps control the step size each iteration while going to the minimum of the cross entropy or the loss function used. An optimal LR value is needed to avoid instability or slow convergence process | 0.0001 |
| **Weight Decay** | Regularization technique to avoid overfitting by penalizing the large weights. This helps keep the weights low so that models can generalize better and perform on unseen data. | 0.0001 |
| **Hyper-parameter** | **Description (MobileNetV2)** | Value |
| **Dropout Layer** | Regularization technique to avoid overfitting, a value of 0.5 means that half the neurons of the model would be turned off and the training process would happen on the other half | Not used |
| **Momentum** | Optimization technique that accelerates the gradient descent by using the past gradients to help with the update rule. It helps the model to reach convergence faster. | Not used |
| **Learning Rate** | Hyperparameter that helps control the step size each iteration while going to the minimum of the cross entropy or the loss function used. An optimal LR value is needed to avoid instability or slow convergence process | 0.0001 |
| **Weight Decay** | Regularization technique to avoid overfitting by penalizing the large weights. This helps keep the weights low so that models can generalize better and perform on unseen data. | 0.01 |
| **Hyper-parameter** | **Description (DenseNet121)** | Value |
| **Dropout Layer** | Regularization technique to avoid overfitting, a value of 0.5 means that half the neurons of the model would be turned off and the training process would happen on the other half | 3 (0.3) (0.7) (0.5) |
| **Momentum** | Optimization technique that accelerates the gradient descent by using the past gradients to help with the update rule. It helps the model to reach convergence faster. | Not used |
| **Learning Rate** | Hyperparameter that helps control the step size each iteration while going to the minimum of the cross entropy or the loss function used. An optimal LR value is needed to avoid instability or slow convergence process | 0.0001 |
| **Weight Decay** | Regularization technique to avoid overfitting by penalizing the large weights. This helps keep the weights low so that models can generalize better and perform on unseen data. | 0.0001 |

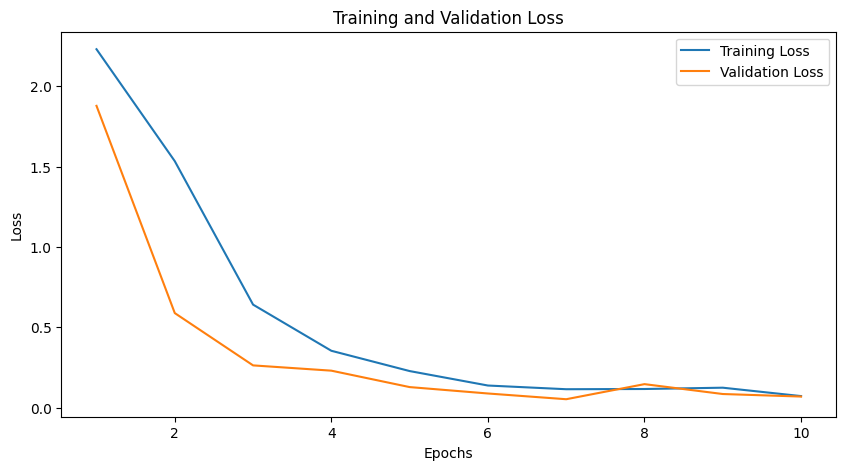
Table 6: Combination of the hyperparameter values and corresponding performance achieved

|  |  |  |
| --- | --- | --- |
| **Combination of hyperparameter values (VGG) (Dense / Momentum / LR / WD)** | **Training performance** | **Validation performance** |
| **Not used / 0.9 / 0.001 / Not used** | 0.52 | 0.48 |
| **1 (0.5) / Not used / 0.0001 / Not used** | 0.96 | 0.98 |
| **1 (0.5) / 0.5 / 0.00001 / Not used** | 0.92 | 0.94 |
| **2 (0.3) (0.7) / 0.9 / 0.0001 / Not used** | 0.94 | 0.90 |
| **Combination of hyperparameter values (ResNet) (Dense / Momentum / LR / WD)** | **Training performance** | **Validation performance** |
| **1 (0.5 / Not used / 0.001 / 0.001** | 0.82 | 0.80 |
| **1 (0.5) / Not used / 0.0001 / 0.0001** | 0.99 | 0.90 |
| **1 (0.5) / 0.9 / 0.00001 / 0.0005** | 0.81 | 0.86 |
| **1 (0.5) / 0.9 / 0.0001 / 0.001** | 0.25 | 0.28 |
| **Combination of hyperparameter values (MobileNet) (Dense / Momentum / LR / WD)** | Training performance | Validation performance |
| **Not used / Not used / 0.0001 / 0.01** | 0.98 | 0.96 |
| **1 (0.5) / Not used / 0.0001 / 0.0001** | 0.97 | 0.96 |
| **2 (0.5) (0.5) / Not used / 0.001 / 0.001** | 0.75 | 0.80 |
| **2 (0.3) (0.7) / 0.5 / 0.001 / 0.01** | 0.75 | 0.66 |
| **Combination of hyperparameter values (DenseNet) (Dense / Momentum / LR / WD)** | Training performance | Validation performance |
| **1 (0.5) / Not used / 0.001 / 0.001** | 0.83 | 0.88 |
| **2 (0.5) (0.5) / Not used / 0.001 / 0.001** | 0.80 | 0.68 |
| **3 (0.3) (0.7) (0.5) / Not used / 0.0001 / 0.0001** | 0.92 | 0.90 |
| **Not used / Not used / 0.00001 / 0.00001** | 0.90 | 0.90 |









# Models’ testing and evaluation

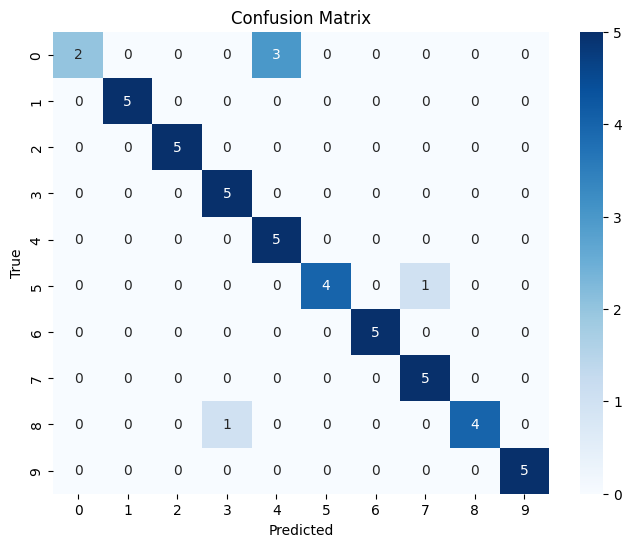
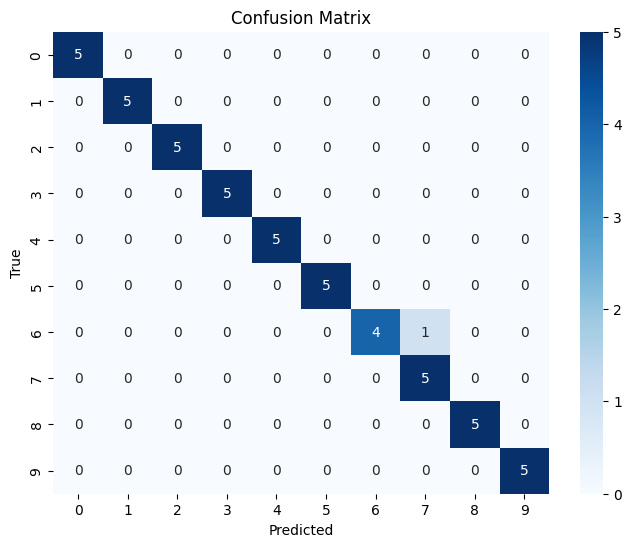
## Testing

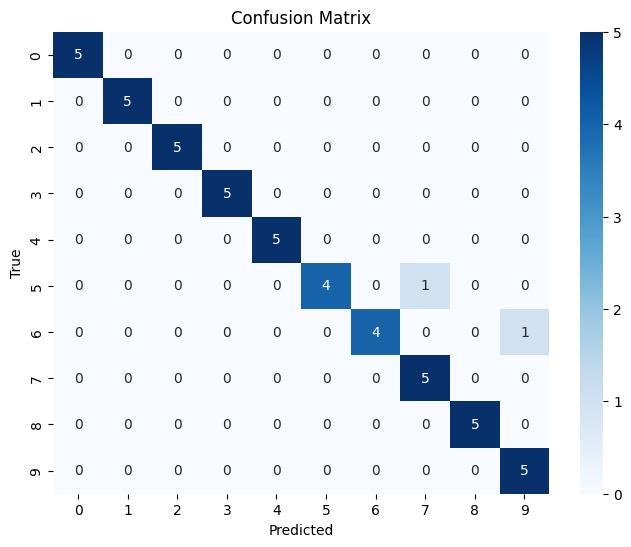
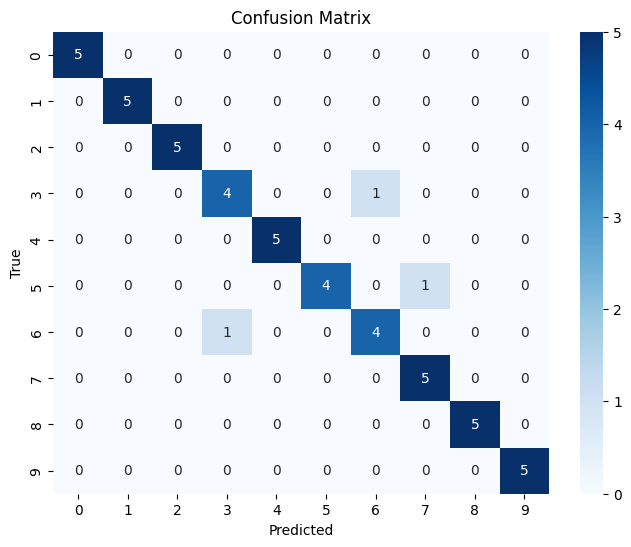
For the testing phase, instead of working on the training split and the validation split of the model. I will work on the testing data. Which consist of 5 images for each of the 10 classes. I managed to correct the code and change the validation set into the testing set. The testing process is used to further validate the process of all the four models to get a comprehensive review of the model’s performance on different images.

The testing process started the same way as the training and validation process in the previous step. With the initialization of the model and its parameters, the in-features and out-features. Which stayed the same based on the model. The loss function is the cross-entropy loss. And Adam was used as the optimizer with different hyperparameters. I took the best performing combinations from each model, and performed the testing process to validate their performance even further. I replaced the validation loader with the testing loader to perform the testing process. And to evaluate it, I added a confusion matrix.

Table 7: Best values for the evaluation metrics on the test set.

|  |  |  |
| --- | --- | --- |
| **Evaluation metric** | **Description (VGG)** | **Value obtained** |
| **Accuracy** | Ratio of correct predictions to the total instances | 0.94 |
| **Precision** | Ratio of correct positive predictions to the total positive predictions | 0.95 |
| **Recall** | Ratio of correct positive predictions to the total positive instances | 0.94 |
| **F1 Score** | Harmonic mean (balance) of precision and recall, useful when we care about total false instances | 0.94 |
| **Evaluation metric** | **Description (ResNet)** | Value obtained |
| **Accuracy** | Ratio of correct predictions to the total instances | 0.98 |
| **Precision** | Ratio of correct positive predictions to the total positive predictions | 0.98 |
| **Recall** | Ratio of correct positive predictions to the total positive instances | 0.98 |
| **F1 Score** | Harmonic mean (balance) of precision and recall, useful when we care about total false instances | 0.98 |
| **Evaluation metric** | **Description (MobileNetV2)** | Value obtained |
| **Accuracy** | Ratio of correct predictions to the total instances | 0.96 |
| **Precision** | Ratio of correct positive predictions to the total positive predictions | 0.97 |
| **Recall** | Ratio of correct positive predictions to the total positive instances | 0.96 |
| **F1 Score** | Harmonic mean (balance) of precision and recall, useful when we care about total false instances | 0.96 |
| **Evaluation metric** | **Description (DenseNet)** | Value obtained |
| **Accuracy** | Ratio of correct predictions to the total instances | 0.94 |
| **Precision** | Ratio of correct positive predictions to the total positive predictions | 0.94 |
| **Recall** | Ratio of correct positive predictions to the total positive instances | 0.94 |
| **F1 Score** | Harmonic mean (balance) of precision and recall, useful when we care about total false instances | 0.94 |





The confusion matrices showed the relationship between the actual and the predicted values, and it contained a meter ranging from 0 to 5 based on whether the prediction is true or false. It can be seen that all the matrices had a diagonal line with the correct predictions from each class. This means that our models performed strongly with the image classification task of cats. The numbers ranging from 0 to 9 are the numbers of classes (types of cats) in the dataset.

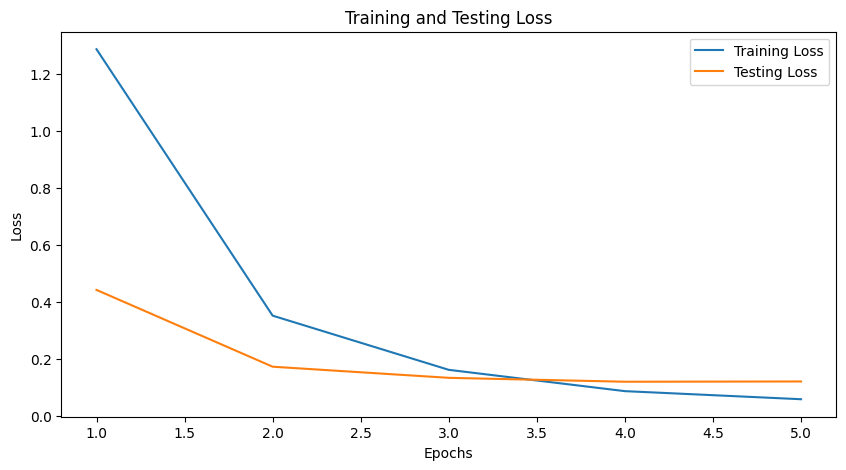
## Over/under-fitting assessment

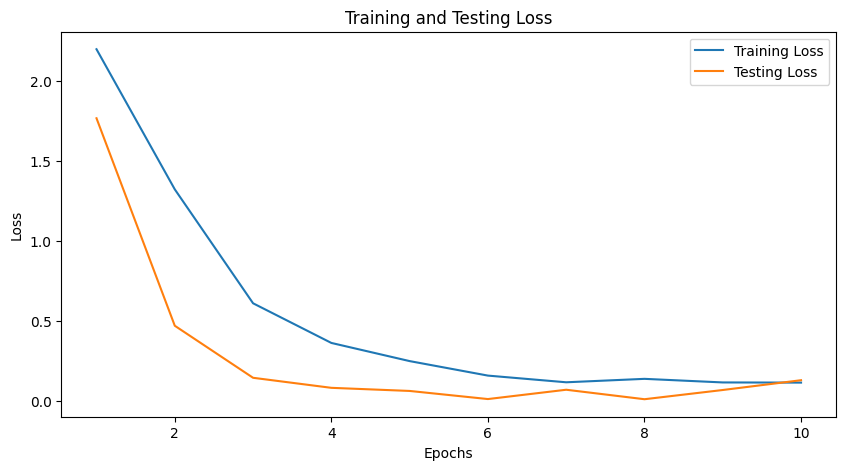
By looking at the CNN architectures’ testing process and results, it can be seen that they performed really well, with values over 90% for the four metrics, accuracy, precision, recall, and f1score. Since there isn’t a gap between the performance metrics, that means that the results are balanced. For the VGG16, ResNet18, MobileNetV2, and DenseNet121. And because the gap between the training and testing metrics isn’t big, which can conclude that the results when comparing both training and testing processes were balanced.

Things are different for the validation and training process, based on the four architectures, in VGG16, the validation set was overfitting in the best combination, because the training accuracy was given 0.64, while the validation accuracy was given 0.90. This can mean that the VGG16 had an overfitting problem at the beginning of the epochs. This can be solved by using regularization techniques to penalize the model for its overfitting problem. In ResNet18, both validation and training had an underfitting problem at the beginning of the modeling process, having a training accuracy of 19% and a validation accuracy of 48%, then the results became balanced for the most part at the end of the epoch run, having the training accuracy at 92% and the validation accuracy at 88%. The underfitting problem was fixed inside the ResNet18 by parameter sharing and by increasing the epochs inside the model. In MobileNetV2, the results were balanced with a training accuracy of 97% and a validation accuracy of 96%. Ending with the DenseNet121 model, where the models performed badly for both the training and validation datasets, having an underfitting problem which was fixed later by the dense layers, architecture and the feed-forward technique that the DenseNet uses. The results for the training and validation datasets were balanced in the end having a training accuracy of 92% and a validation accuracy of 91%. Generally, the models performed greatly on the three splits of the dataset. Any underfitting and overfitting problems were solved inside the model by either the technique that the CNN architecture itself uses, and by increasing the epoch count so the models learn more effectively and efficiently.

## Results analysis

After analyzing all the CNN architectures and checking their results for any overfitting or underfitting problems, we can now compare the performance of the models based on the four metrics Accuracy, precision, recall, and f1-score. Based on the table above in 6.1, the model that performed the best was ResNet18, it gave us the highest values for the metrics, 98% for accuracy 98% for precision, 98% for recall and 98% for f1-score. The worst performing model out of the four, was a tie between VGG16 and DenseNet121 with their metrics tied at 94%. But VGG16 had a higher performance on the Precision 94% to 95%. It can mean that the higher performing model out of the two was the VGG16, and the worst performing model out of the four was DenseNet121. But the DenseNet121 performed exceptionally faster than the VGG16. So in terms of usage, the difference between their performance can mean that DenseNet121 performed better.





The plot above is the ResNet18 learning curve, and the below is the DenseNet121 learning curve.

## Effectiveness assessment

Evaluating the effectiveness of the CNN architectures will be based on the memory, the computational efficiency, the time and size of the model. The CNN architectures are VGG16, Resnet18, MobileNetV2, and DenseNet121.

VGG16: This architecture can perform good accuracy in predictions, but it Is outperformed by other models, because of its high computational cost. It has a simple architecture which can be easy to understand. But it can be affected negatively by the high memory cost, since it can limit its usage on devices with limited power.

ResNet18: Compared to VGG16, ResNet18 can perform faster as a model, due to a reduction in the number of parameters. It can be used for deeper networks since it can mitigate the effect of the vanishing gradient. But it can be more complex than VGG16 due to the usage of residual networks inside its architecture.

MobileNetV2: This CNN architecture performed extremely efficient compared to the previous two architectures, it was fast due to its very small model size. It is very effective and can be used for mobile applications. It uses depth-wise layers which can be hard to implement, but the end result is very welcoming. Since it reduces the cost significantly. Since it is designed for mobile and low-resource environments. It may not be welcoming to huge datasets and high-resource environments.

DenseNet121: This architecture is considered the most complex out of the four models, due to the dense connections which require careful implementing. The model is efficient and can produce accurate predictions, and it can be used in a wide selection of different applications. It is faster than VGG16, but is slower than MobileNetV2.

## Interface development

The interface I implemented supports a prediction system for the cat dataset image classification problem. It takes the image from the folder inside Google Colab. Then the user presses on the “Predict” button, which performs the prediction process with the model I chose for the implementation (MobileNetV2). Then it displays the image, with a title above it that mentions the predicted class’s number. Since there are 10 classes inside the dataset, the number of the prediction ranges between 0 and 9.

The interface’s computational power is low, so its effective with low-power devices. But since it takes the models from the CNN architecture. It can be computationally expensive to implement with those models. But if GPU is used, it will be efficient and extremely useful in classifying the class of the cats in a short period of time (5 secs maximum).

To use the interface, the user first runs the models above (in the implemented interface, only MobileNetV2 is used). And runs the code, the interface will show a dropdown menu with all the images that he can choose from. Then the user presses Predict, and the image will be displayed along with the predicted class.





## Evaluation of models

The current implemented learning solution (The interface) meets the basic end-user requirements effectively. But it lacks some improvements and features in terms of speed, memory efficiency, user interaction and functionality. Some improvements for the future can combine

* Model optimization. The four CNN architectures vary in their speed and computational power and cost. Having VGG16 be significantly slow, while other models such as DenseNet1212 and MobileNetV2 significantly faster. We can optimize the VGG16 model to make it more efficient and effective to use with the interface.
* Expanded library of features. The dropdown menu and selection of images are great ways to implement the interface system, but it can be improved by adding more features to it, such as a drag-and-drop feature which implements a drag-and-drop system when selecting the image, we need to classify and predict the class for it. We can also add a prediction meter to show the user how accurate the prediction to the class was.
* Cross-platform deployment. The MobileNetV2 CNN architecture was used due to its high efficiency with low-resource devices. Since MobileNetV2 was specifically implemented for mobile-based applications. A cross-platform update to the interface can benefit the users extremely. We can implement a webpage-based interface that can allow PC and laptop users access to the interface’s prediction system. Which can increase the usage level for the interface.

The current status of the learning solution is that it provides a real-time image classification system for the users, using models such as VGG16, ResNet18, MobileNetV2, and DenseNet121. Some models can result in slower learning times due to the number of parameters for them. I chose mobilenetv2 as the implemented model for the interface because it is a good deployment for devices with limited memory. And since the dataset isn’t large (only 2500 images), it is suited for this model. But in terms of speed and memory, optimizing the models can enhance the interface’s processing power significantly.

# References

You need to write the references here using Harvard style of referencing.

* Pablo Ruiz (2018). Understanding and visualizing DenseNets. *Medium*. [online] 18 Oct. Available at: <https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a>.
* Boesch, G. (2021). *VGG Very Deep Convolutional Networks (VGGNet) - What Kou Need to Know*. [online] viso.ai. Available at: <https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/#:~:text=Vision%20Enterprise%20Platform->.
* Product Teacher. (n.d.). *ResNet18 & ResNet50 in Computer Vision*. [online] Available at: <https://www.productteacher.com/quick-product-tips/resnet18-and-resnet50>.
* Sharma, N. (2023). *What is MobileNetV2? Features, Architecture, Application and More*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2023/12/what-is-mobilenetv2/.

‌