MACHINE LEARNING UNIVERSITÀ DELLA
SVIZZERA ITALIANA

Assignment 2

LIBERATI JACOPO

May 14, 2024

LINK TO THE ALL FOLDER ON GOOGLE DRIVE:
Here you'll find again the Report, the Code and the Saved Models.
https://drive.google.com/drive/folders/14G_SEzn2SQfFk2YF8MSR7GISwbbzGlLX?usp=share_link

In this assignment, you are asked to:

1. Implement a fully connected feed-forward neural network to classify images from the **Cats of the Wild** dataset.
2. Implement a convolutional neural network to classify images of **Cats of the Wild** dataset.
3. Implement transfer learning.

1 IMAGE CLASSIFICATION WITH FULLY CONNECTED FEED FORWARD NEURAL NETWORKS (FFNN)

Task 1 of this assignment requires to build a classifier for the provided dataset. In particular, a Feed Forward Neural Network will be created, and its potential will be evaluated.
The first block of code is aimed at loading, processing and displaying a dataset of images, present within the "Wild Cats Datasets" folder which in turn is made up of 6 sub-folders, each for each type of wild cat provided. The folder was previously saved on Google Drive, so that it could then be imported onto Google Collab for subsequent use.
Initially, the libraries necessary for image manipulation **(PIL)** and visualization **(Matplotlib)** were imported, as well as libraries for carrying out operations with Arrays **(NumPy)** and other specific ones for Machine Learning and Deep Learning operations such as **Scikit learn** and **PyTorch**. Consequently, the drive was also imported, so as to be able to access the previously loaded folder.
The code proceeds with loading the images *[def load_imgs (path, folders)]:*
Each image is labeled based on the subfolder from which it was loaded.
Additionally, images and assigned labels are stored in separate lists.
Subsequently, the function aimed at displaying a random sample of images is implemented, inserting them into a grid with their respective labels. In addition to this, the functions necessary for the creation of the dataset and for the subsequent saving and loading of the models are defined.

Subsequently, the specific requests of task 1 begin to be carried out.

In general, this block of code was designed in order to train an FFNN neural network model, to classify

the images of the different species.
Initially the images are normalized by dividing each pixel value by 255. This was done in order to reduce the pixel values to a range between 0 and 1, which was done to help the model converge more quickly during the subsequent training.
In addition to this, always following the sell assignment instructions, the previously defined image labels are binarized, using "LabelBinarizer". This operation converts the labels into a one hot encoding format, and this is also a fundamental step for calculating the loss in the classification process.
Again, as part of the data preparation process, the normalized images were converted into PyTorch tensors for use in the model and in addition to this, the 2D(224x224) images are flattened into 1D vectors, a necessary step as the dense layers of an FFNN accept inputs of this size.
Subsequently the Dataset is divided into three parts, one for training to which 60% of the data has been attributed, one for validation and one for the test, to which 20% of the data has been equally attributed. This is followed by conversion to PyTorch tensors.

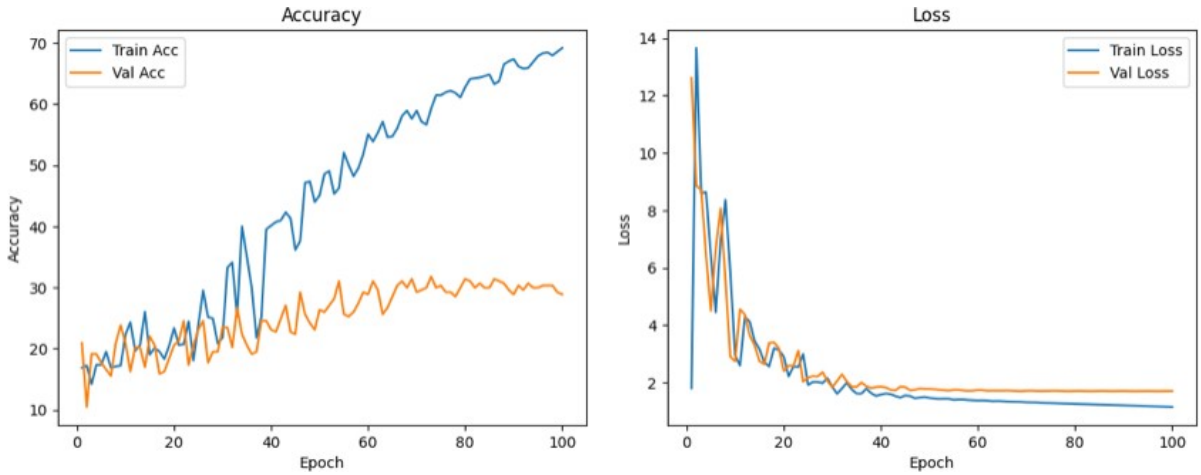
Consequently, the Feed Forward Neural Network (FFNN) was defined on the basis of the information provided.
The network, in particular, has two hidden layers with GELU (Gaussian Error Linear Units) activation and a linear output layer. The use of the **GELU** function was specifically requested, but can be considered a good choice due to its non-linear activation properties which can improve performance compared to a ReLu. From a structure point of view, *"input_dim"* is the size of the inputs (number of pixels per image), *"hidden_dim"* is the size of the neurons of the hidden layers and finally *"output_dim"* is the number of classes present.

```
class FFNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FFNN, self).__init__()
        self.layer1 = nn.Linear(input_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.output_layer = nn.Linear(hidden_dim, output_dim)
        self.gelu = nn.GELU()  # Define GELU activation once

    def forward(self, x):
        x = self.gelu(self.layer1(x))  # Apply GELU activation after the
first layer
        x = self.gelu(self.layer2(x))  # Apply GELU activation after the
second layer
        x = self.output_layer(x)  # No activation for the output layer
        return x
```

The code continues with the definition of the training function, through which it is possible to train the model, calculating the loss and updating the weights with *"optimized.step()"*.
Every *"eval_freq"* (defined equal to 1) epochs, the loss and accuracy are evaluated on the validation set. The accuracy is calculated through the *"compute accuracy"* function, which compares the model's predictions with the real labels assigned to the images.
Before starting the training and validation process, lists are defined, necessary for storing the training and validation losses and the training and validation accuracies for each epoch, so that they can then be subsequently recalled at the time of plotting.

Furthermore, the hyperparameters are defined as follows:
→ ***"input_dim"*** represents the number of input features and since the images have been flattened into 1D vectors, this value corresponds to the total number of pixels in the image.
→ ***"hidden_dim"*** defines the number of neurons in each of the two hidden layers that have been defined within the model. The value 64 is a common choice, aimed at not making the model too complex and causing overfitting. The objective was to balance the learning capacity of the model with computational efficiency as much as possible.
→ ***"output_dim"*** corresponds to the number of output classes, which are 6, corresponding to the number of classes present within the dataset.
→ ***n_epochs:*** represents the number of iterations on the training dataset. Too high a number would have led the model to learn the training data too well, potentially causing negative performance at validation due to overfitting. Likewise, even a too low number would have led to the model failing to learn, also in this case causing sub-optimal performance at the time of validation due to under fitting. This value allowed to conduct training and validation in an optimal way.
→ ***"eval_freq"*** defines the frequency with which the model is evaluated on the validation set and the choice to set this hyper parameter equal to 1 was dictated by the need for careful monitoring of the model's performance.
The **Cross-Entropy loss function** was chosen as it is suitable for multi-class classification problems, also allowing for a probabilistic interpretation of the results, through a comparison of the predicted probabilities with the true labels.
→ ***"optimizer"*** defines the algorithm that is used within the process and a common choice that has also been followed in this process is the Adam algorithm, which generally allows for stable and rapid convergence compared to other algorithms. The learning rate in this case too was set to a standard value, i.e. 0.01.
Consequently the *"plot accuracy"* and *"plot loss"* functions are defined, necessary for the graphic display of the results obtained during the epochs.



Regarding the results obtained based on this model, it can be noted that the training accuracy is initially very low and this is typical for the beginning of the training process, since the model still has to learn the characteristics present within the data. Therefore, the validation accuracy is also very low.
Progressively we notice an increase in training accuracy and also in validation accuracy, although with greater fluctuations. For example, evaluating the 30th epoch, the training accuracy is 30%, while the validation accuracy reaches 23% in the same period.

In the final part of the training, the training accuracy increases, reaching almost 70%, while the validation accuracy remains lower, partly signaling overfitting of the model, since the latter achieves excellent performance on the train set, but is less able to generalize on non-specific data seen at the time of validation.
However, this is in line with the fact that for the classification problem that needs to be solved within the assignment, a CNN can be considered a better choice than a FFNN. CNNs are specifically designed to process image data and provide superior performance by exploiting the spatial structure of images. Precisely for this reason, a CNN will consequently be implemented, whose performances will be compared with those just obtained.

After training and using the validation set as a metric to understand the model's learning and its ability to generalize to unseen data, the model is evaluated on the Test Set, calculating the loss and accuracy. The Validation Set is used to monitor the performance of the model during training, allowing to adjust hyperparameters to prevent overfitting and determine the best training break point (for example through early stopping). The Test Set, however, is used in this final phase to evaluate the performance of the model after training is completed, offering an unbiased estimate of the model's capacity, since 20% of the data used in this phase are completely separate from the training and validation previously done. In this test phase, the batch size was defined as 32, compromising between computational efficiency and accuracy of the gradient estimate. A larger Batch size could have guaranteed more stable and less noisy gradients, but also greater memory consumption during the process.

```
Test LossT1: 1.7458
Test AccuracyT1: 31.41%
Performance on the test set:
----------------------------
Test LossT1: 1.7458
Test AccuracyT1: 31.41%

Estimate of Classification Accuracy on New and Unseen Images:
-------------------------------------------------------------
Based on the performance observed on the test set and assuming similar data distribution,
we can expect the model to achieve an accuracy of around 31.41% on new and unseen images.
```

Test Loss represents the loss calculated on the test set. Lower loss generally implies better model performance. Regarding the accuracy of the model on the test set, an accuracy of 31.41% is obtained, indicating that the model is able to correctly classify approximately a third of the images present within the test set. This accuracy is relatively low, suggesting that the model is not performing well in the classification task entrusted to it.
Assuming that the test set data are representative of the data, it can be assumed that the performance of the model on new data will be similar to that observed on the test set. The results indicate that the FFNN model is not sufficiently performant for the image classification task and a restructuring towards a CNN could provide improvements.

2 IMAGE CLASSIFICATION WITH CONVOLUTIONAL NEURAL NETWORKS (CNN)

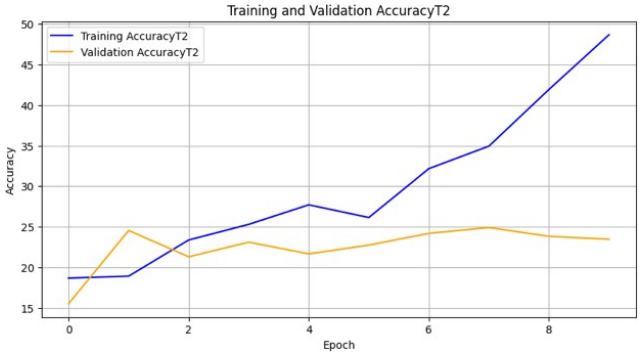
In task 2 a new model is initially implemented for the classification of images present within the same dataset. Several elements of the code architecture are similar, as can be seen in the first part, similarly reported with respect to the previous block of code to show how data preparation is performed in the same way, through normalization, label conversion and data splitting in train, test and validation set, with the consequent creation of PyTorch tensors.
The convolutional neural network is defined in the ***"CustomCNN"*** module and also in this case the model follows the indications given within the assignment.
As regards the three **convolutional layers**, the following have been defined:
→ ***in_channels:*** number of channels in the input.
→ ***out channels:*** number of feature maps produced by the convolutional layer. Each feature map is the result of a convolution between a filter and the image imputed.
→ ***kernel_size*** indicates the spatial dimensions of the filter; during the convolution operation this filter is applied to the input to extract the local features.
→ ***stride:*** specifies the number of pixels the filter moves when applying the convolution.
→ ***padding:*** operation that adds padding values around the edges of the input, such that the output has the same dimensions as the original input. This is useful for preserving image dimensions during convolution.
In the architecture provided, after each convolutional layer there is a **pooling layer**, responsible for progressively reducing the spatial dimensions of the representation, thus reducing the number of parameters and operations, helping to make the model more efficient and limiting the risk of distortion. ***Max Pooling*** was used, through which the maximum value is returned within a pooling window, thus helping to preserve the most significant characteristics.
Among the specified parameters we have:
→ ***kernel_size*** which specifies the size of the pooling window.
→ ***stride:*** which specifies the pace with which the pooling window moves on the input. at the end of the network fully connected layers are used, which combine the features extracted from the convolutional feature maps.
Consequently, the **fully connected layers** serve to recombine and reduce the features extracted from the convolutional and pooling layers, in order to create the final classification output. In total they were inserted into four levels, with a progressive reduction in the dimensions of the vector created at the level of the first fully connected layer. It comes with the 4 fully connected layer which reduces the vector to 6 neurons, which correspond to the 6 output classes.
Consequently the *"forward"* function defines the forward passage of the data through the neural network, defining how the input data is transformed to produce the final output

```
class CustomCNN(nn.Module):
    def __init__(self):
        super(CustomCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,
kernel_size=3, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, stride=1, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 28 * 28, 512)
```

```
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 6)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 28 * 28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

The loss function (Cross Entropy Loss) and the optimizer (Adam) are the same ones used in the previous task. This is followed by the training cycle which performs training and validation for the specified number of epochs, calculating train loss and train accuracy, validation loss and validation accuracy, similarly to the previous point. At the end of each epoch, as seen for the FFNN, the losses and accuracies are stored for the purpose of the subsequent graphical representation.



After the training cycle, similarly to what was done in task 1, the results of the train accuracy and validation accuracy and the results of the train loss and validation loss for each epoch are represented.
In this case the number of epochs is deliberately lower than in the previous case since it was decided to limit the number of epochs to 10. This is due to the fact that CNNs are generally more complex than FFNNs, as CNNs are designed to extract automatically spatial features through convolutional and pooling layers. This can lead to a greater ability to adapt to the training data and therefore to a greater risk of overfitting, especially when the model is too complex compared to the size and complexity of the dataset. This is precisely what occurs within this model, where we notice how the train loss drops rapidly while the validation loss begins to increase, clearly showing overfitting of the model. The train accuracy also increases to approximately 50% in the space of just 10 epochs. On the contrary, the validation accuracy remains below 25%.
The main responsibility is to be attributed to the fully connected layers, in which we go from 64x28x28 to 512, then to 256, 128 and finally 6, an approach that involves the introduction of many parameters, which are excessive for the dataset we are dealing with. As a result, the model is too complex, leading to over-fitting to train data, but poor generalization to unseen data.
This model has potential in image processing, but at present it does not guarantee excellent performance, as can also be verified on the basis of the results obtained from the test set, with a very low Test Accuracy, equal to 19.13%.

```
Test LossT2: 2.7923
Test AccuracyT2: 19.13%
Performance on the test set:
----------------------------
Test LossT2: 2.7923
Test AccuracyT2: 19.13%

Estimate of Classification Accuracy on New and Unseen Images:
-------------------------------------------------------------
Based on the performance observed on the test set and assuming similar data distribution,
we can expect the model to achieve an accuracy of around 19.13% on new and unseen images.
```

However, the model was deliberately kept in this state, in order to show how image manipulation allows obtaining optimal results, without modifying the model.

Before proceeding with the transformations, a comparison is performed between the two identified models, the FFNN of task 1 and the CNN of task 2. The first model was found to be not excellent in problem processing, but the problems were not resolved by the second model, not so much due to the inadequacy of the type of model, but due to the choices made from the point of view of the parameters used.
The comparison between the performances of the two models is made both in terms of final accuracy on the train set and validation set, and in terms of average accuracy on the train set and validation set, in such a way as to provide a vision of the average performances during the training.
A t-test is also performed to compare the validation accuracies of the two models, in particular the difference between the means of the two distributions is calculated taking into consideration the validation accuracies, in order to determine whether the difference is significant or not. The P-Value used is 0.05, indicating that if the value obtained is less than 0.05 the difference in the performance of the two models is statistically significant, otherwise it is concluded that there is no significant difference.
In the case in question the conclusion is that there is no significant difference between the models; one would expect a better performance from the CNN, but as already discussed this is limited by choices that have been made within the structuring of the model itself.

```
COMPARISON BETWEEN THE TWO MODELS
T1 - Final Train Accuracy: 53.49%, Final Validation Accuracy: 28.52%
T2 - Final Train Accuracy: 44.10%, Final Validation Accuracy: 24.55%
T1 - Mean Train Accuracy: 33.92%, Mean Validation Accuracy: 23.65%
T2 - Mean Train Accuracy: 28.67%, Mean Validation Accuracy: 23.97%
T-statistic: -0.22141525582663732, P-value: 0.8251872118602646
The differences in the validation performances between the 2 models are NOT statistically significant.
```

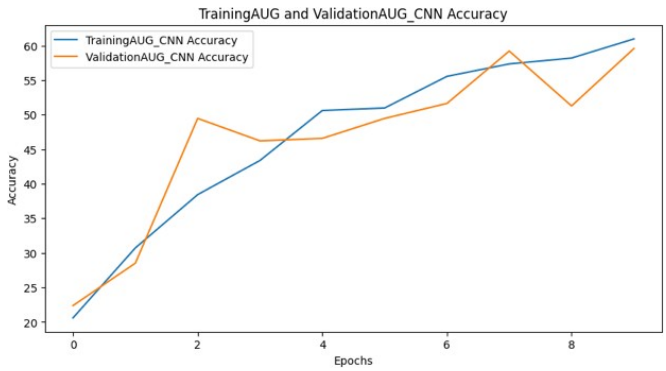
In the next part of the code, the functions previously used for training, validation and testing of the two models (the FFNN and the CNN), are implemented again.
Part of this code has simply been reported compared to what was done above (some parts could have been removed to lighten the code, but they were kept for greater clarity given the educational purpose of the project). The additional part is the one to the transformations that are applied to the data, in order to improve the performance of the model. The transformations were designed in such a way as to improve the performance of the CNN, but the same transformed dataset was then also used for the FFNN, to see the impact of the transformations on this model too.
Furthermore, the code is defined in such a way that the transformations are applied only to the train set while keeping validation set and test set unchanged, since the objective always remains to maximize the performance of the model on real data.
The transformation pipeline is as follows:

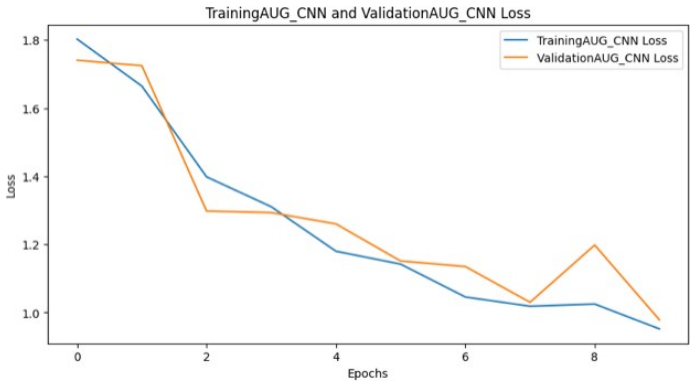
```
train_transforms = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(20),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
```

```
    transforms.ToTensor(),
    transforms.Normalize(mean=[129.73103399 / 255.0, 115.74991322 / 255.0,
93.65555466 / 255.0],
                         std=[62.75922075 / 255.0, 58.79315587 / 255.0,
59.74380684 / 255.0])
])
```

→ ***RandomHorizontalFlip*** **(Random Horizontal Flip):** this transformation randomly flips images horizontally with a probability of 50%. This means that some of the images are horizontally flipped, simulating the effect of viewing an image from a different angle.
→ ***RandomVerticalFlip*** **(Random Vertical Flip):** similar to the horizontal reflection transformation, this transformation randomly flips images vertically with a probability of 50%. This helps the model recognize patterns even when the images are oriented differently.
→ ***RandomRotation*** **(Random Rotation):** this transformation randomly rotates images within a specific angle, which in this case is set to 20 degrees. This allows the model to learn the variability of object positions in the images.
→ ***RandomResizedCrop*** **(Random Crop and Resize):** this transformation performs a random crop of the image followed by resizing the resulting size. The cropped area is randomly chosen but with a minimum area of 80% of the original image and a maximum of 100% of the original image. This allows the model to focus on different parts of the image during training.
→ ***ToTensor*** **(Convert to Tensor):** this transformation converts the image into a tensor, which is the required input format the models.
→ ***Normalize*** **(Normalization):** this transformation normalizes the values of the image tensor using the specified means and standard deviations. This helps stabilize the training process by reducing variation in the pixel values of the images. The values used for normalization depend on the characteristics of the dataset. Instead of standard values, it was preferred to carry out normalization by calculating the mean and standard deviation of the pixels of the entire training dataset, using these values for the normalization of all the images in the dataset.

After applying the transformations, both models are trained again and similarly to before are then evaluated on the basis of the test set. As can be seen in the graphs, the same transformations lead to better performance in the CNN, but not in the FFNN. There are several reasons why transformations may favor a CNN over a FFNN, including the CNN structure being designed to capture spatial patterns in images through the use of rotational filters. As a result, transformations such as flips, rotations, and clippings can help the CNN learn better than they do for an FFNN. Furthermore, CNNs have an intrinsic ability to learn spatial features from images through convolutional layers and pooling. Through transformations it is possible to increase the complexity and diversity of the patterns that a CNN can learn, thus improving its performance.



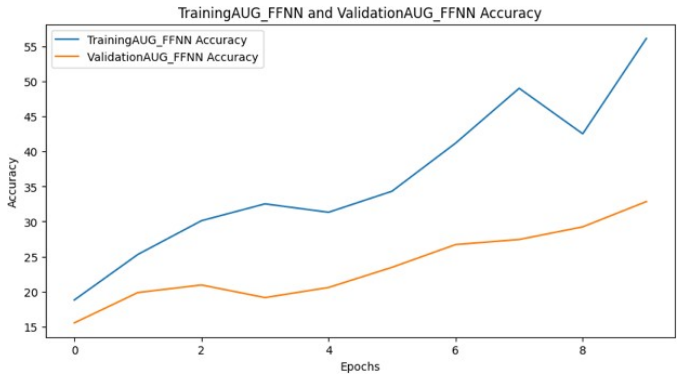


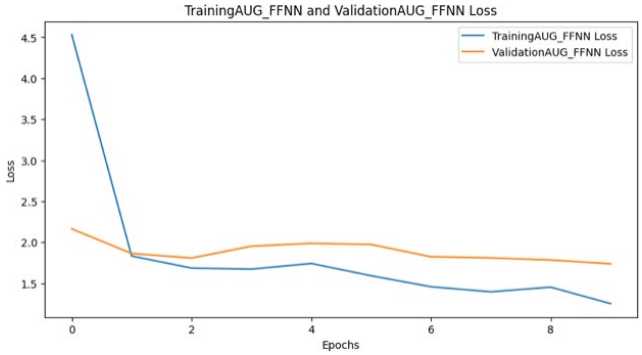

```
Test LossAUG_CNN: 0.9736
Test AccuracyAUG_CNN: 59.21%
Performance on the test set:
----------------------------
Test LossAUG_CNN: 0.9736
Test AccuracyAUG_CNN: 59.21%

Estimate of Classification Accuracy on New and Unseen Images:
------------------------------------------------------------
Based on the performance observed on the test set and assuming similar data distribution,
we can expect the model to achieve an accuracy of around 59.21% on new and unseen images.
```

Observing what is obtained after applying the transformations, we notice how the performance of the CNN has significantly improved. The accuracy of the model during training and validation increases significantly over the 10 epochs. In particular, the trend is similar over the train and validation accuracy and this implies that the model is able to generalize better even with respect to data not seen during training. This indicates that the transformations are helping the model fit the input data more effectively. This trend that is noted in the area of accuracy is also reflected in a decreasing loss, indicating that the model is learning more efficiently. Furthermore the model shows convergence, as the accuracies and losses stabilize over the epochs, demonstrating how the model is really producing coherent and reliable predictions. The accuracy on the test set goes from 19.13% in the case of an untransformed dataset (performed in Part 3) to 59.21% in the case of a transformed dataset, signaling a very clear improvement in the model's performance and its ability to generalize on unseen data. Also, the test loss goes from 2.7923 (Part 3) to 0.9736.

The outputs of the same operations carried out on the FFNN are also shown below, although it is noted that the transformations did not significantly modify the results obtained previously. In this case 10 epochs were used compared to the 100 used in task 1, so it is possible to argue that the model was not given enough time to adapt as done previously. However, with a higher number of epochs, a convergence of the train accuracy to 100% and a train loss very close to 0 was verified, while the validation loss continued to grow and the validation accuracy continued to decrease, showing clear signs of overfitting of the model.






```
Test LossAUG_FFNN: 1.7208
Test AccuracyAUG_FFNN: 30.69%
Performance on the test set:
----------------------------
Test LossAUG_FFNN: 1.7208
Test AccuracyAUG_FFNN: 30.69%

Estimate of Classification Accuracy on New and Unseen Images:
-------------------------------------------------------------
Based on the performance observed on the test set and assuming similar data distribution,
we can expect the model to achieve an accuracy of around 30.69% on new and unseen images.
```

Consequently, in the bonus part of task 2 there are two blocks of code that perform a **grid search** to optimize the hyper parameters of the CNN that was defined previously. Two different approaches were used, one with **split train-validation (not specifically required**) and one with **hold out cross validation using KFold**.
In the first approach the data is divided into a training and validation set, a part of the data is reserved for validation and the other for training, resulting in results that may depend on the randomness of the data division.
In the second approach, hold out cross validation with KFold is used to divide the data into multiple folds. Consequently, a cross-validation is performed, training the model on one set of folds and evaluating it on another. This allows to obtain a more robust evaluation of the model, reducing the impact of randomness in the division of the data. In this approach K represents the number of folds and is set equal to 5. The final result is the average of the model's performance over multiple folds.
This is different from the first approach, in which the model evaluation occurs only once on a single validation set.
Despite some differences introduced in the approaches, the results are similar, with the only difference that in the first case the optimal batch size is 32 while in the second case 64.

FIRST APPROACH:

```
Best Accuracy: 0.5235
Best Params: {'learning_rate': 0.001, 'batch_size': 32, 'num_epochs': 20}
```

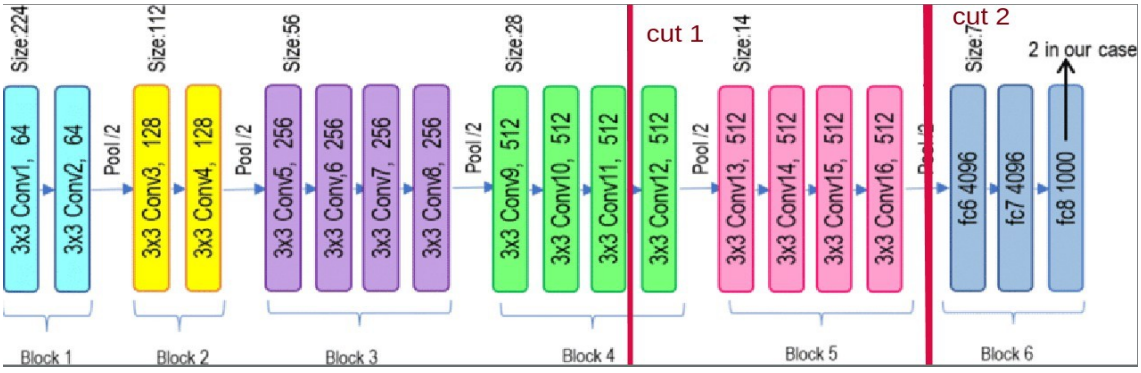
SECOND APPROACH:

```
Best Accuracy: 0.5367
Best Params: {'learning_rate': 0.001, 'batch_size': 64, 'num_epochs': 20}
```

3 TRANSFER LEARNING

This task involves loading the VGG19 model from PyTorch, applying transfer learning, and experimenting with different model cuts. The VGG19 architecture have 19 layers grouped into 5 blocks, comprising 16 convolutional layers followed by 3 fully connected layers. The provided code snippet sets *param.requires_grad = False* for the pre-trained VGG19 model's parameters.Can you explain the purpose of this step in the context of transfer learning and fine-tuning? Will the weights of the pre-trained VGG19 model be updated during transfer learning training?

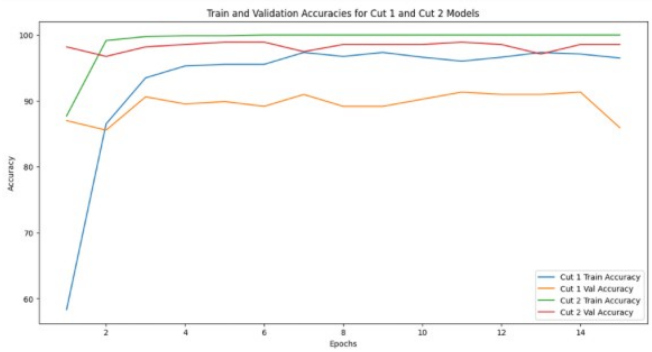
Within the proposed code, *param.equires_grad = False* has been set for the parameters of the pre-trained model and this is a fundamental step in the context of transfer learning.
In particular, this step freezes the parameters of the pre-trained model, and this implies that these parameters will not be updated during training, remaining fixed and acting as feature extractors. The motivation for this decision is to exploit the knowledge of the pre-trained model, which was trained on a large dataset to recognize a wide range of characteristics, among which it is possible to find essential characteristics that are also useful within the new classification problem. As a result, the hyperparameter tuning process focuses on updating the parameters that are not frozen, to adapt the model specifically to the new dataset.



Initially the code provided was completed, inserting the required cuts.
The "CuttedModifiedVGG19" class loads the pre-trained model and allows to keep only a part of the convolutional layers.
The first cut is said to be made using the value ***"cut_value=25",*** which is the value used to cut the previously imported VGG19 model up to the 25th position which corresponds to the end of the 11th conciliation, i.e. the end of the 4th block. After cutting, fully connected layers are added to adapt the model to the classification of the 6 classes of wild cats present in the starting dataset.
Similarly, the second cut was also made, using the value ***"cut_value=36",*** which cuts the VGG19 model at the 36th position, corresponding to the end of all convolutional levels, therefore to the end of block 5. Also in this case, being using the same function, fully connected layers are added after cutting for model fitting. The cuts are made in the convolutional part of the neural network, ensuring that only the features extracted up to that point are retained and removing the rest of the network which is subsequently trained.

```
VGG(
 (features): Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1) : ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3) : ReLU(inplace=True)
```

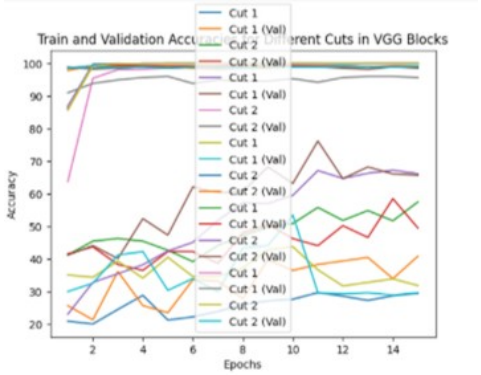
```
  (4) : MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8) : ReLU(inplace=True)
  (9) : MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17) : ReLU(inplace=True)
  (18) : MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace=True)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26) : ReLU(inplace=True)
  (27) : MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace=True)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace=True)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35) : ReLU(inplace=True)
  (36) : MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
 )
 (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
 (classifier): Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2) : Dropout(p=0.5, inplace=False)
  (3) : Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5) : Dropout(p=0.5, inplace=False)
  (6) : Linear(in_features=4096, out_features=1000, bias=True)
 )
)
```




```
Evaluating Cut 1 Model
Cut 1 Test Loss: 1.0109, Test Accuracy: 87.00%
Evaluating Cut 2 Model
Cut 2 Test Loss: 0.0947, Test Accuracy: 98.56%
```

Here are proposed the training and evaluation results of the **Cut 1** and **Cut 2** models with the consequent evaluation on the test set. Despite very good performances in both cases, with a test accuracy in the first case equal to 87% and a test accuracy in the second case equal to 98.56%, there is still a discrepancy between the performances of the two models, that can be seen also on the basis of the graphic representations offers.
Cut Model 1 shows a train accuracy which increases until it reaches 100% in the fifteenth epoch. In the same period, the validation accuracy fluctuates mainly around 89%, with a value of 87% in the first epoch and a value of 85.92% in the last epoch. This implies a tendency of the model to overfit, because it reaches a maximum train accuracy without stably increasing the validation accuracy.
Cut Model 2, on the other hand, starts with a train accuracy of 87.71%, also reaching an accuracy of 100% at the fifteenth epoch, all accompanied by a progressive decrease in train loss. However, in this case we note how the validation accuracy remains high and stable, showing a better generalization capacity compared to Cut1. The Cut 2 model includes more convolutional layers and has a greater ability to learn complex data representations, which explains the superior performance. Cut 1 model ends at layer 25 and may not be as capable of extracting detailed features as the cut 2 model. The greater feature extraction capacity allows the model to have a more accurate representation of the data, improving not only in the training phase, but also in generalization on test data.

In the bonus point of task 3 further cuts are carried out, and the results follow the logic of what was explained previously. it is required to carry out other cuts in order to consequently evaluate the performance of the model. Different cuts were chosen at different levels of the model, most made within the convolutional layer. The cuts, as explained previously, allow you to set some parameters and concentrate training only on the subsequent ones. The first convolutional layers tend to extract low-level features such as lines and edges, while the deeper layers extract high-level features. as can be seen from the results, through a deeper cut it is possible to obtain a better performance of the model.



JACOPO LIBERATI
ASSIGNMENT 2 – MACHINE LEARNING
ACADEMIC YEAR 2023-2024