Politechnika Śląska

Katedra Grafiki, Wizji Komputerowej i Systemów Cyfrowych

Academic year			Group	Section
2022/2023	**SSI**	**BIAI**	**GKiO1**	**1**
Supervisor:	mgr inż. Grzegorz Baron		**Classes: (day, hour):**	
Section members:	**Kamil Łuc** **Nikodem Polak**		**Tuesday**	
emails:	nikopol989@student.polsl.pl kamiluc137@student.polsl.pl		**9:45 AM**	

Report

Subject:

Big cats classification

1. Introduction

In the field of machine learning, a classifier is a model that learns to assign predefined categories or labels to input data based on their features. It analyzes patterns and characteristics in the data to make predictions or classifications. In our project, we focus on developing a classifier designed to classify big cats, such as lions, tigers, cheetahs, leopards, etc, using the PyTorch framework. By training our classifier on a diverse dataset of big cat images, we aim to create a model that can accurately identify different species of these majestic creatures.

2. Analysis of the task

For our project on big cat classification, we utilized a dataset obtained from Kaggle, specifically the "10 Big Cats of the Wild" dataset (available at this link: https://www.kaggle.com/datasets/gpiosenka/cats-in-the-wild-image-classification). The dataset consists of 2339 images in the training set, 50 images each in the test and validation sets, These images are in JPEG format with a resolution 224x224 pixels and three color channels (RGB). To tackle the classification task, we decided to use convolutional neural networks (CNNs) as our models of choice. CNNs are appropriate in image-related tasks, making them suitable for our big cat classification project. In our task, models are designed and implemented using the PyTorch framework in Python. To execute our code and train the models, we used the Google Colab environment. Google Colab provides a free cloud-based Jupyter notebook environment equipped with GPUs, allowing us to harness the computing power of GPUs to speed the training process. Charts are created with Excel software.

3. Internal and external specification of the software solution

Most of the code is written in python with PyTorch framework which is a high-level deep learning framework that provides a user-friendly interface for building and training neural networks, simplifying the development process. Code related to data loading and preparation is written with Pandas library.

```
df = pd.read_csv('/content/drive/MyDrive/BIAI/dataset/WILDCATS.CSV')
indices = df.iloc[:, 0].unique()
species = df.iloc[:, 2].unique()
species_map = {i: big_cat_class for i, big_cat_class in zip(indices, species)}
class_ids = df[df.iloc[:, 3] == 'train'].iloc[:, 0]

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.Resize(224),
        transforms.RandomRotation(10),
        transforms.ToTensor()
    ]),
    'valid' : transforms.Compose([
        transforms.Resize(224),
        transforms.ToTensor()
    ]),
    'test' : transforms.Compose([
        transforms.Resize(224),
    transforms.ToTensor()
    ])
}
```

Picture 3.1. *Prepare class indices and big cats species.*

This code loads data from a CSV file using Pandas library, creates a mapping between unique indices and big cat species based on the corresponding columns in the CSV file. It also defines data transformations for three different sets: training, validation and testing. These transformations include random horizontal flips, resizing the images to a size of 224x224 pixels, random rotations of up to 10 degrees and converting the images to tensors.

```
train_path = '/content/drive/MyDrive/BIAI/dataset/train/'
valid_path = '/content/drive/MyDrive/BIAI/dataset/valid/'
test_path = '/content/drive/MyDrive/BIAI/dataset/test/'

train_images = torchvision.datasets.ImageFolder(train_path, transform=data_transforms['train'])
test_images = torchvision.datasets.ImageFolder(test_path, transform=data_transforms['test'])
valid_images = torchvision.datasets.ImageFolder(valid_path, transform=data_transforms['valid'])
```

Picture 3.2. *Load input data to arrays with appropriate transformation.*

Using torchvision, code creates “ImageFolder” datasets for training, test and validation images. “ImageFolder” class assumes that the images are organized in subdirectories based on their class labels. The “transform” parameter is set to the respective data transformation. These datasets are used with PyTorch data loaders to efficiently load and iterate over the images during training, validation and testing phases (Picture 3.3).

```
batch_size = 64

train_loader = DataLoader(train_images, batch_size=batch_size, shuffle=True, num_workers=2)
test_loader = DataLoader(test_images, batch_size=1, shuffle=False, num_workers=2)
valid_loader = DataLoader(valid_images, batch_size=1, shuffle=False, num_workers=2)
```

Picture 3.3. *Create data loaders to pass input data to the neural network.*

“DataLoader” for the training set is configured with a batch size which is equal to the “batchSize” (64 in this case) variable, enabling it to load and process 32 images at a time. The “shuffle” parameter is set to “True”, which means that the training data will be shuffled randomly before each epoch. “num_workers” is set to 2, allowing data loading to be performed in parallel using multiple worker processes.

```
class BigCatClassifier(nn.Module):
  def __init__(self):
    super(BigCatClassifier, self).__init__()
    self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
    self.relu1 = nn.ReLU()
    self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
    self.relu2 = nn.ReLU()
    self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)

    self.flatten = nn.Flatten()

    self.fc1 = nn.Linear(32*56*56, 256)
    self.relu3 = nn.ReLU()
    self.dropout = nn.Dropout(0.5)

    self.fc2 = nn.Linear(256, 10)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.maxpool1(x)

    x = self.conv2(x)
    x = self.relu2(x)
    x = self.maxpool2(x)

    x = self.flatten(x)

    x = self.fc1(x)
    x = self.relu3(x)
    x = self.dropout(x)

    x = self.fc2(x)
    return x
```

Picture 3.4. *BigCatClassifier class.*

Architecture of this class consists of two convolutional layers followed by ReLu activation functions and max pooling layers. These layers are responsible for extracting features from the input images. After the convolutional layers, the feature maps are flattened to prepare them for the fully connected layer. The flattened features are then passed through two fully connected layers with ReLU activation applied after the first fully connected layer. A dropout layer is also applied after the ReLU activation of the first fully connected layer to help prevent overfitting. In the “forward” method, the input is passed through the defined layers in a sequential way, applying the necessary activation function and pooling operations. The final output of the model is obtained from the last fully connected layer, which represents the predicted class probabilities for the input.

```
class BigCatClassifier2(nn.Module):
    def __init__(self):
        super(BigCatClassifier2, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.maxpool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(64 * 28 * 28, 256)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)

        x = self.conv3(x)
        x = self.relu3(x)
        x = self.maxpool3(x)

        x = self.flatten(x)

        x = self.fc1(x)
        x = self.relu4(x)
        x = self.dropout(x)

        x = self.fc2(x)
        return x
```

Picture 3.5. *BigCatClassifier2 class.*

In this updated model, the main change is the addition of another convolutional layer and its associated activation and pooling operations. The additional convolutional layer increases the depth of the network, allowing for potentially better feature extraction.

```python
class BigCatClassifier3(nn.Module):
    def __init__(self):
        super(BigCatClassifier3, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()
        self.maxpool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu4 = nn.ReLU()
        self.maxpool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(128 * 14 * 14, 256)
        self.relu5 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)

        x = self.conv3(x)
        x = self.relu3(x)
        x = self.maxpool3(x)

        x = self.conv4(x)
        x = self.relu4(x)
        x = self.maxpool4(x)

        x = self.flatten(x)

        x = self.fc1(x)
        x = self.relu5(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Picture 3.6. *BigCatClassifier3 class.*

Third version of the classifier is extended by another convolutional layer.

```python
def train(log_file):

  for epoch in range(num_epochs):
    model.train()

    for batch_idx, (inputs, targets) in enumerate(train_loader):
      inputs, targets = inputs.to(device), targets.to(device)
      optimizer.zero_grad()
      outputs = model(inputs)
      loss = criterion(outputs, targets)
      loss.backward()
      optimizer.step()

    log_file.write(f'{epoch+1},{loss.item():.4f},')
    test_accuracy(log_file)
```

Picture 3.7. *train function.*

"train" function performs the training loop for the model. It iterates over each epoch, sets the model to training mode and iterates over each batch of inputs and targets. It calculates the loss, performs backpropagation and updates the model parameters. The function also writes the epoch number and current loss to a log file. At the end, the "test_accuracy" function is executed to evaluate the accuracy of the model.

```python
def test_accuracy(log_file):

  model.eval()
  test_correct = 0
  valid_correct = 0
  test_total = 0
  valid_total = 0

  with torch.no_grad():
      for images, labels in test_loader:
          images, labels = images.to(device), labels.to(device)
          outputs = model(images)
          _, predicted = outputs.max(1)
          test_total += labels.size(0)
          test_correct += predicted.eq(labels).sum().item()

  with torch.no_grad():
      for images, labels in valid_loader:
          images, labels = images.to(device), labels.to(device)
          outputs = model(images)
          _, predicted = outputs.max(1)
          valid_total += labels.size(0)
          valid_correct += predicted.eq(labels).sum().item()

  test_accuracy = 100 * test_correct / test_total
  valid_accuracy = 100 * valid_correct / valid_total
  log_file.write(f'{test_accuracy},{valid_accuracy}\n')
```

Picture 3.8. *test_accuracy function.*

"test_accuracy" function evaluates the model's accuracy on the test and validation datasets. It calculates the number of correct predictions and the total number of examples for each dataset. The accuracy is then calculated as a percentage and written to a log file.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("GPU" if torch.cuda.is_available() else "CPU")

#----------------------------------------------------
lr = 0.001
model = BigCatClassifier3().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr)
num_epochs = 150
results_folder = "/content/drive/MyDrive/BIAI/results/ThirdNeuralNetwork"
model_name = "nn.CrossEntropyLoss-optim.Adam"
#----------------------------------------------------
result_file = results_folder + "/" + model_name + "-" + str(batch_size) + "-" + str(lr) + ".result"
model_file = results_folder + "/" + model_name + "-" + str(batch_size) + "-" + str(lr) + ".pth"

with open(result_file, 'w') as file:
  file.write(f"batch_size: {batch_size}\n\n")
  file.write(f"learning_rate: {lr}\n\n")
  file.write(f"model: {str(model)} \n\n")

  file.write(f"begin training\n\n")
  file.write(f"epoch,loss,test accuracy,valid accuracy\n")
  train(file)
  file.close()

torch.save(model.state_dict(), model_file)
```

Picture 3.9. *Perform auto training.*

This code snippet sets the device based on the availability of CUDA. After various variables initialization the training loop is executed and the results are written to the result file. Finally, the trained model's state dictionary is saved to a model file.

```
model.eval()

class_labels = ['AFRICAN LEOPARD', 'CARACAL', 'CHEETAH', 'CLOUDED LEOPARD', 'JAGUAR', 'LIONS', 'OCELOT',
'PUMA', 'SNOW LEOPARD', 'TIGER']

test_correct = 0
valid correct = 0
test_total = 0
valid_total = 0

real_labels = []
predicted_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = outputs.max(1)
        test_total += labels.size(0)
        test_correct += predicted.eq(labels).sum().item()
        real_labels.extend(labels.cpu().numpy())
        predicted_labels.extend(predicted.cpu().numpy())

    conf_matrix = confusion_matrix(real_labels, predicted_labels)
    plt.figure(figsize=(8, 6))
    plt.imshow(conf_matrix, cmap=plt.cm.Blues)

    plt.colorbar()
    classes = np.arange(len(class_labels))
    plt.xticks(classes, class_labels, rotation=90)
    plt.yticks(classes, class_labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('Valid Labels')
    plt.title('Confusion Matrix')
    plt.show()
    plt.close()
```

Picture 3.10. *Generate confusion matrix.*

Above code is responsible for demonstrating the evaluation and visualization of the model's performance using a confusion matrix providing insights into the model's predictions and potential misclassifications.

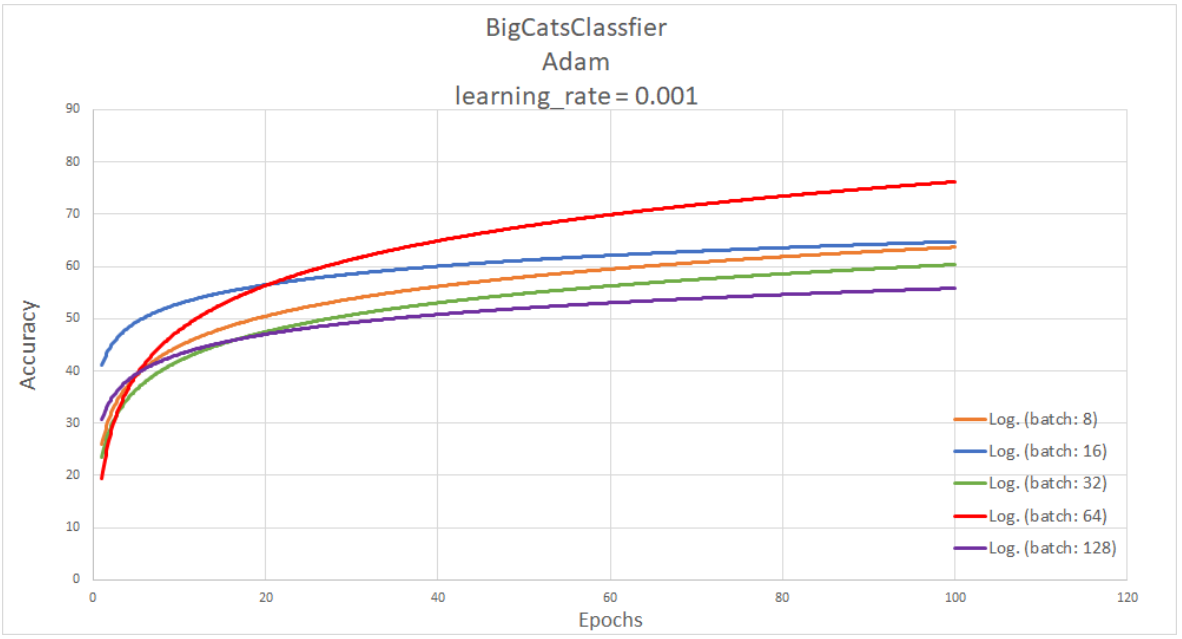
4. Experiments

4.1 Experimental background

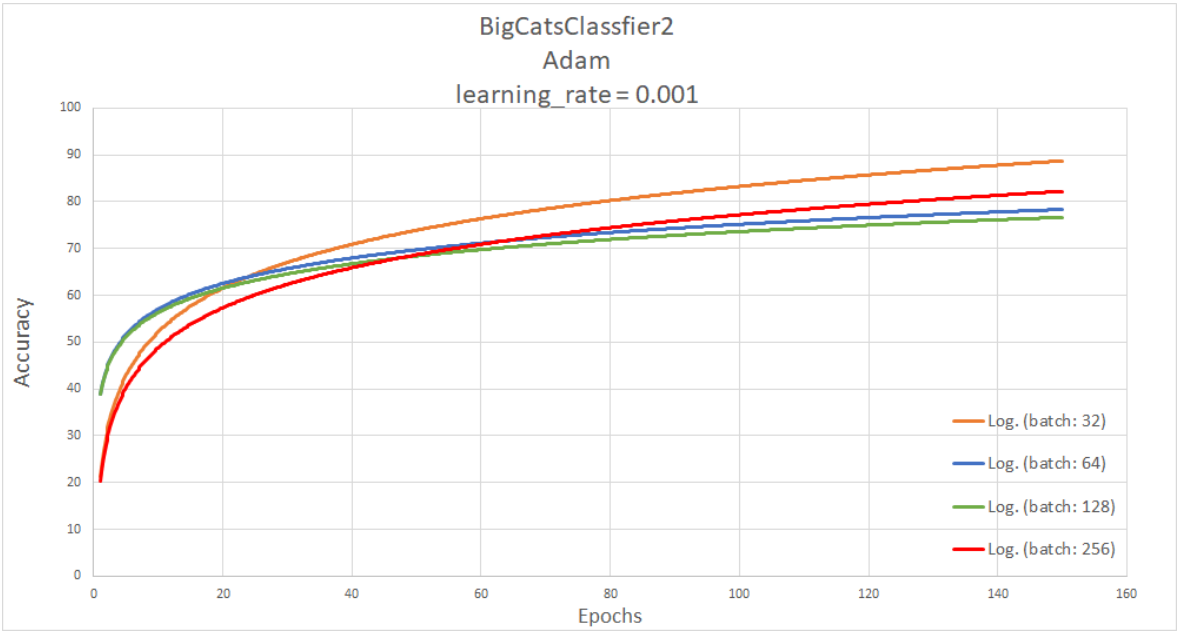
In our experiments, we utilized three different variations of the "BigCatClassifier" class, each differing in the number of convolutional layers. The purpose of these experiments was to investigate the impact of varying model architectures on the classification performance of big cats. Additionally, during the training process we explored the effects of using two different optimizers, namely ADAM and SGD momentum. Optimizers purpose is to adjust the model's parameters based on the computed gradients to minimize the loss function and improve the model's accuracy in making predictions. To measure the performance of our models, we used CrossEntropyLoss criterion. Cross-entropy loss is commonly used in multi-class classification tasks, including image classification. It computes the loss by comparing the predicted probability distribution with the true label distribution. The criterion aims to minimize the difference between these distributions, resulting in more accurate and confident predictions. During the training process, we experimented the most with modifying two additional values: "batch_size" and "learining_rate". The "batch_size" refers to the number of samples processed in one iteration, while "learning_rate" determines the step size at which the model parameters are updated during optimization. These values were adjusted to observe their effects on training convergence and overall model performance.

4.2.1 ADAM optimizer

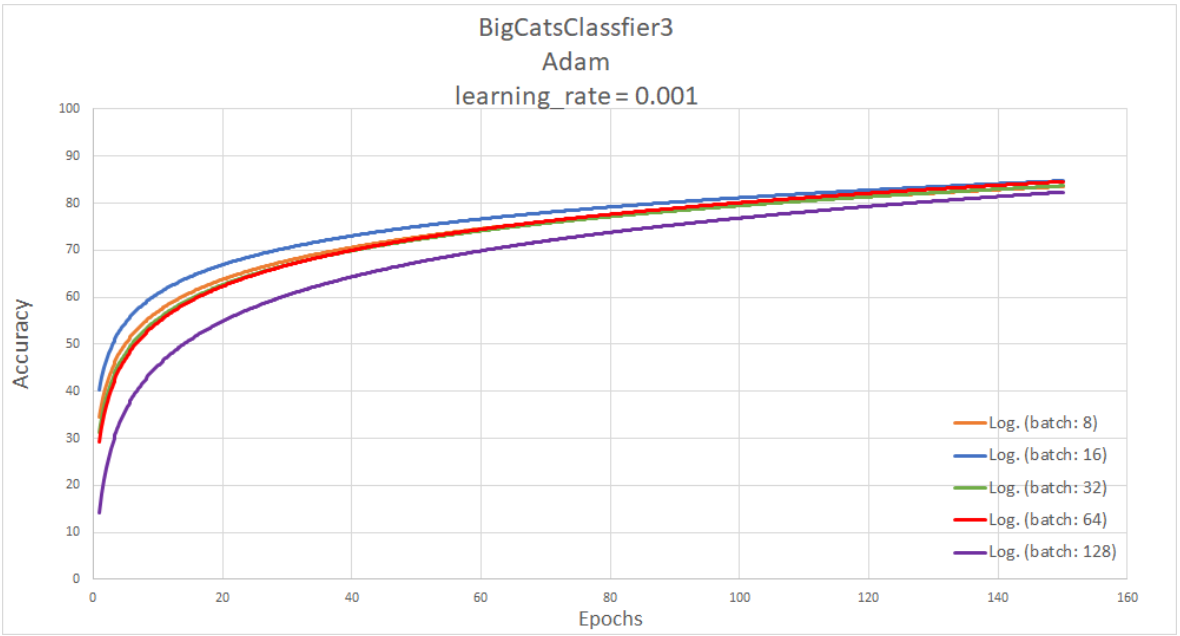
In the case of ADAM optimizer, we analyzed the performance of three neural networks: BigCatClassifier, BigCatClassifier2 and BigCatClassifier3. The optimal learning rate we discovered for each network was 0.001. It is worth noting that learning rate larger than 0.01 (0.1, 1 for example) resulted in lack of progress in network training, on the other hand, learning rate smaller than 0.0001 resulted in extension of learning process and reduced efficiency. During our experiments with ADAM optimizer we manipulated the batch size parameter, ranging from 8 to 256.



Picture 4.2.1.1. *BigCatClassifier accuracy.*



Picture 4.2.1.2. *BigCatClassifier2 accuracy.*

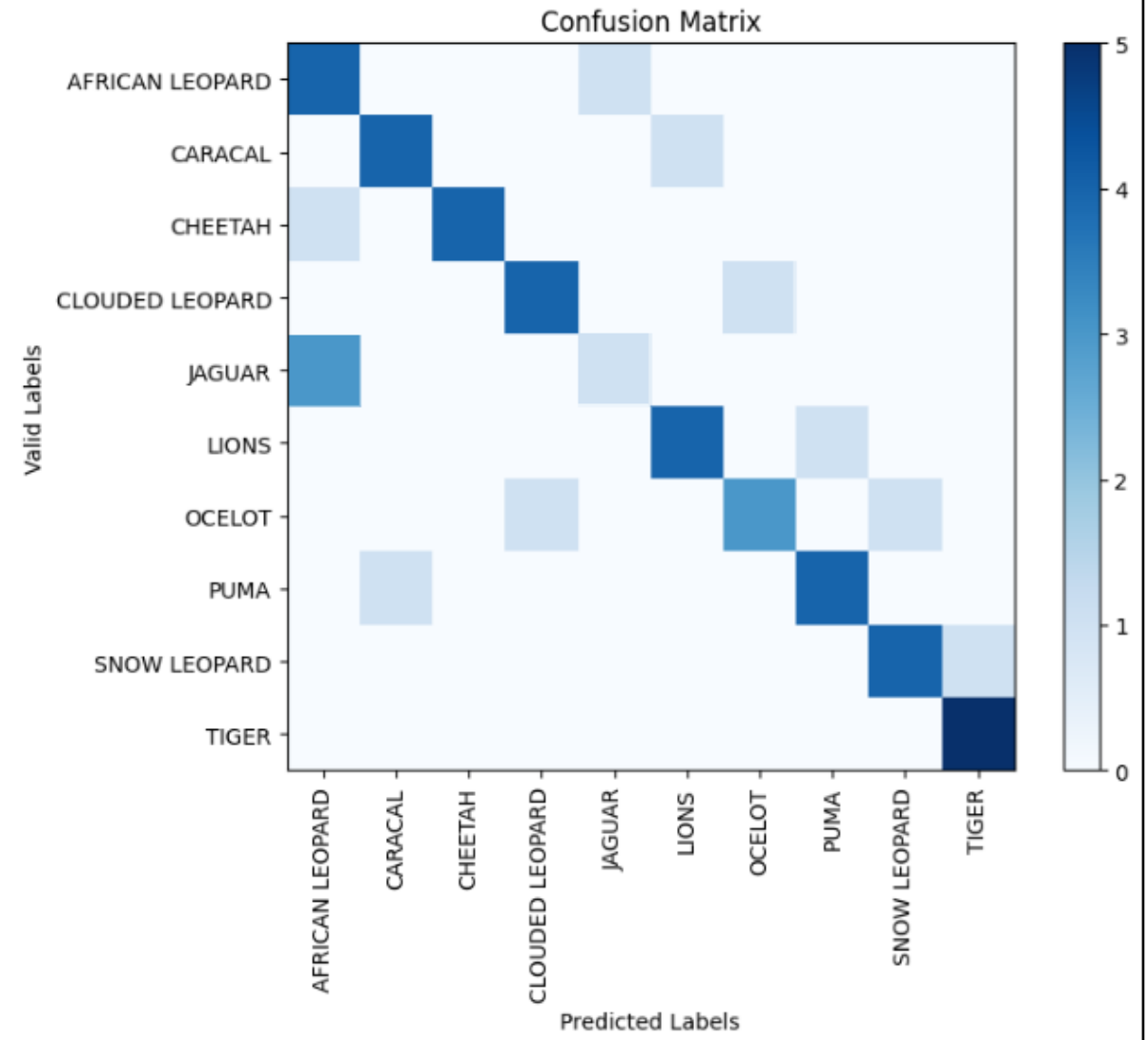


Picture 4.2.1.3. *BigCatClassifier3 accuracy.*

Based on the analysis of the accuracy graphs, we can draw partial conclusions regarding the optimal batch size for each network:

1. For BigCatClassifier, the best-performing batch size was 64.
2. BigCatClassifier2 achieved the highest accuracy with a batch size of 32.
3. BigCatClassifier3 demonstrated its best performance does not really depend on batch size.

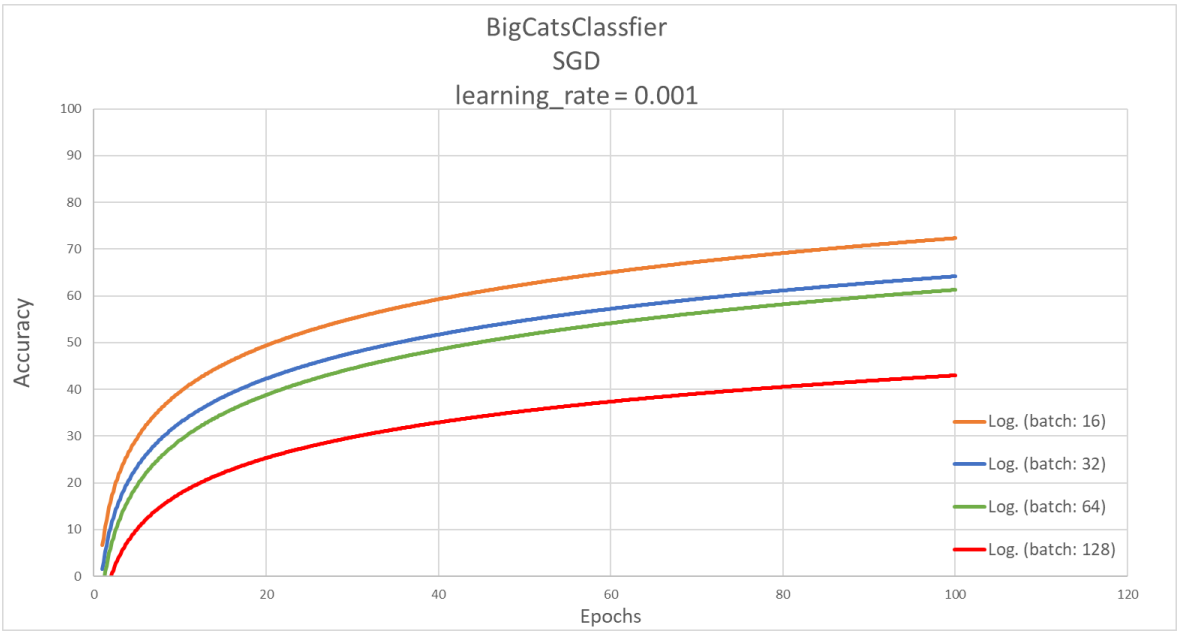
Furthermore, we observed that BigCatClassifier2 outperformed other networks, achieving an impressive accuracy of 90%. In comparison BigCatClassifier achieved an accuracy of 75% and BigCatClassifier3 attained 85% accuracy.



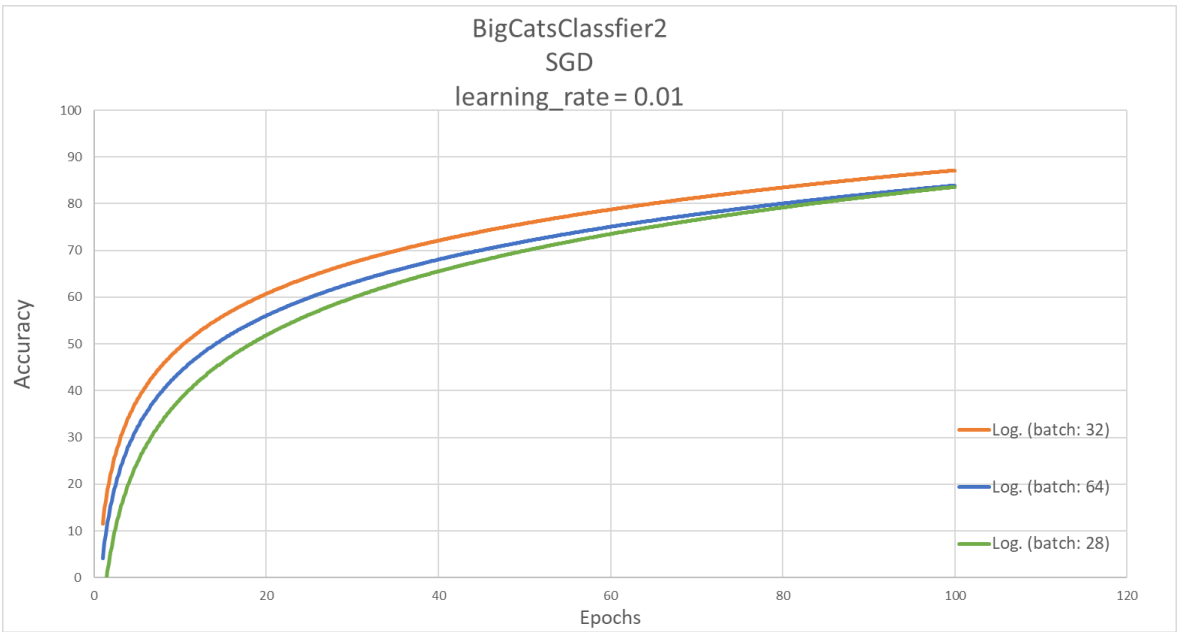
Picture 4.2.1.4. *BigCatClassifier2 confusion matrix.*

4.2.2 SGD optimizer

In the case of the SGD optimizer, we analyzed the performance of three neural networks: BigCatClassifier, BigCatClassifier2. The optimal learning rate for the first network was 0.001 and for the second network was 0.01. Other values of the learning rate turned out to be less effective. During our experiments with SGD optimizer we manipulated the batch size parameter, ranging from 16 to 128 for BigCatsClassfier and from 32 to 128 for BigCatClassfier2.



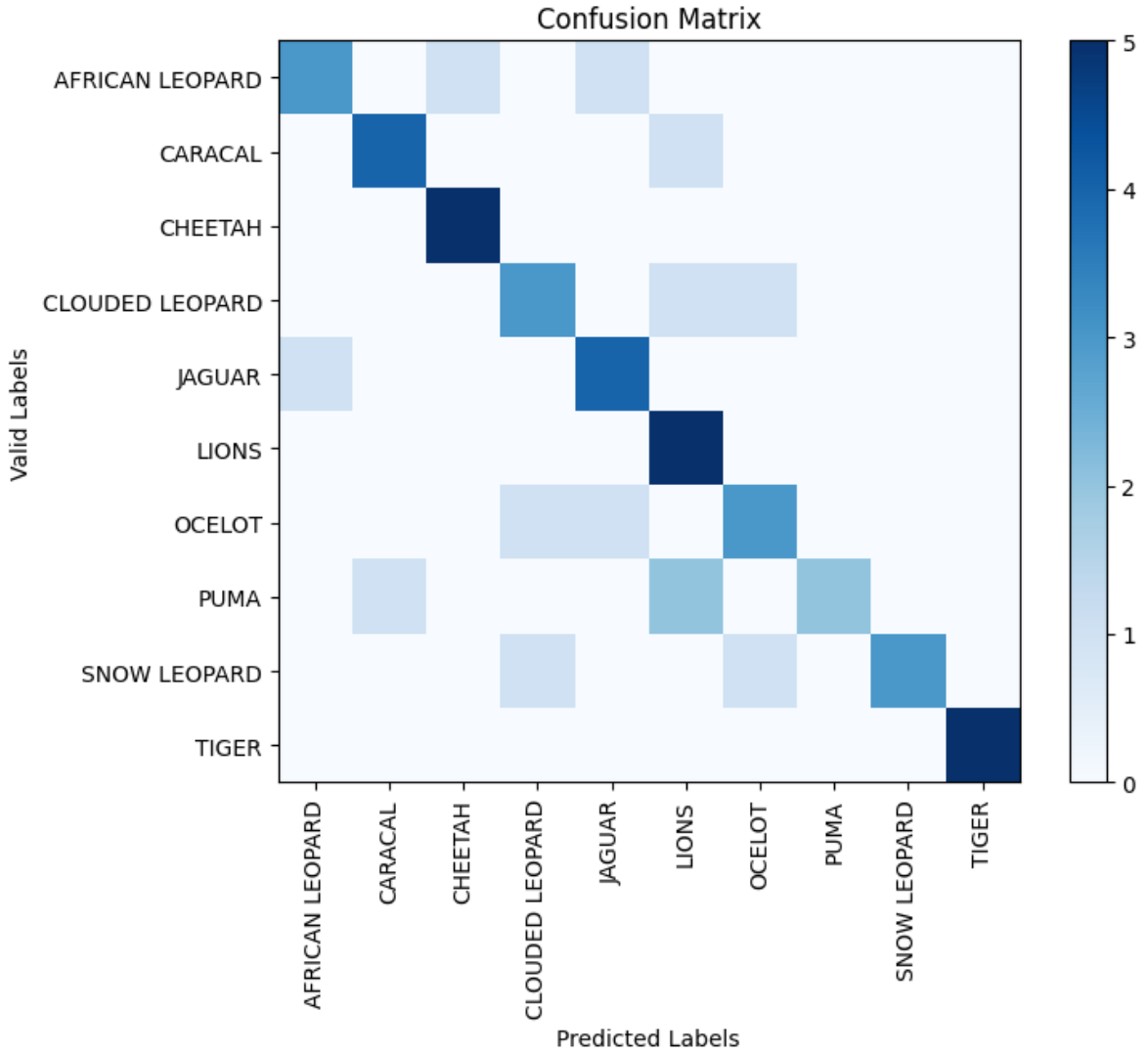
Picture 4.2.1.1. *BigCatClassifier accuracy with SGD optimizer.*



Picture 4.2.1.1. *BigCatClassifier2 accuracy with SGD optimizer.*

By analyzing these two graphs we can come to a following conclusions:

1. BigCatClassifier2 performed better than BigCatClassfier archiving 88%.
2. For BigCatClassifier2 the best performing batch size was 32.
3. For BigCatClassifier the best performing batch size was 16.
4. Best For BigCatClassifier accuracy was 74%.



Picture 4.2.1.4. *BigCatClassifier2 with SGD optimizer confusion matrix.*

5. Summary

The best-performing model in our project was the "BigCatClassifier2", trained with the ADAM optimizer with learning rate of 0.001 and a batch size of 32. The second-best model was the one trained with SGD using a learning rate of 0.01 and batch size of 32. Based on these results, we can conclude that the optimal batch size for our task was 32.

It is worth noticing that for the "BigCatClassifier3" model, with a learning rate of 0.001 and a ADAM optimizer, the batch size had minimal impact on results. However, for the SGD optimizer with a learning rate of 0.001, the batch size played a significant role, with an accuracy of 40% for a batch size of 128 and 70% for a batch size of 16. In most cases, reducing the batch size resulted in increased network accuracy

Based on the confusion matrices, it can be concluded that the most confused big cats in case of “BigCatClassifier2” with ADAM optimizer were the Jaguar and the African Leopard. This indicates that distinguishing between these two species can be challenging, even for the trained models.

Picture 5.1. *Comparison between Jaguar and African Leopard.*

In the case of “BigCatClassifier2” with SGD optimizer, the most confused big cats were Puma and Lion. This is because while the males of these species are easy to distinguish, the females of these species seem very difficult to tell apart.

Picture 5.1. *Comparison between female Puma and Lion.*

Unfortunately, due to time constraints and free Google Colab version limitations, we were unable to test the "BigCatClassifier3" model with SGD. However, the use of Google Colab and its GPU capabilities still greatly facilitated the execution of the project, allowing us to train our models efficiently. PyTorch, being a user-friendly framework, provided a smooth development experience for building and training neural networks.

In conclusion, while time constraints prevented us from fully exploring all aspects of the project, we successfully utilized GoogleColab, PyTorch and our expertise in parameter tuning to develop and analyze effective neural network models for classifying images.

6. References

1. https://pytorch.org/tutorials/
2. https://www.youtube.com/watch?v=Jy4wM2X21u0&ab_channel=AladdinPersson
3. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939
4. https://colab.research.google.com/?utm_source=scs-index
5. https://www.kaggle.com/datasets/gpiosenka/cats-in-the-wild-image-classification
6. https://www.3blue1brown.com/topics/neural-networks

7. Link to files

https://github.com/KamiLuc/BigCatClassifier