Report

Biologically inspired artificial intelligence

Topic: Guessing geographical location based on landscape image

Authors :

Eustachy Lisiński
Miłosz Wojtanek

1. Short introduction presenting the project topic.

Using photographs of the terrain, the model learns to recognize the characteristics for each country's landscape in the selected group of countries, such as architecture, roads, signs, vegetation, terrain or soil.

2. Analysis of the task:

Convolutional Neural Networks (CNNs)

Description: CNN is most frequently chosen for image recognition. The model can be trained to classify images into countries or regions based on the learned patterns of architecture, roads, vegetation, and other landscape features.

- **Pros:**
    - Proven success in image classification and recognition tasks.
    - Ability to extract local features (edges, textures, shapes) effectively.
    - Pretrained models such as ResNet, VGG, or Inception can be fine-tuned, saving training time.
    - High accuracy when sufficient data is provided.
- **Cons:**
    - Requires a large, labeled dataset from multiple countries.
    - May struggle with generalizing to unseen landscapes or similar features between countries.
    - Lacks explicit spatial understanding of geography; purely focused on visual patterns.
    - May miss fine-grained details unless computationally expensive deeper networks are used.

Recurrent Neural Networks (RNNs)

- **Description**: primarily used for sequential data, can be applied to image recognition tasks that involve a sequence, such as video frames.

- **Pros**:

    - Well-suited for tasks where temporal or spatial sequences are involved.
    - Useful for tasks like video recognition or processing image sequences (e.g., **image captioning**).
- **Cons**:
    - Poor at capturing spatial hierarchies within a single image, which CNNs excel at.

Graph Neural Networks (GNNs)

- **Description**: operate on graph-structured data. In the context of image recognition, they can be used when image regions or objects are represented as nodes in a graph, with edges representing relationships between them.
- **Pros**:
    - Captures complex relationships and dependencies, such as object interactions.
    - Useful in applications like scene understanding.
- **Cons**:
    - May not perform as well as CNNs in raw image classification but excels in tasks that involve relational reasoning.

Long Short-Term Memory (LSTM) Networks

Description: designed to overcome the limitations of traditional RNNs. LSTMs are effective in tasks where context from earlier in the sequence is crucial to understanding later data.

Pros:

- Addresses the vanishing gradient problem seen in standard RNNs, making it more effective for longer sequences.
- Useful for image captioning or sequential tasks, especially where understanding long-term dependencies in temporal data is critical.
- Can be integrated with CNNs to handle video data or sequence-based image recognition.

Cons:

- Like RNNs, LSTMs are not ideal for tasks that require capturing spatial hierarchies in a single image.

- Computationally more intensive than simpler RNNs, especially for very long sequences.

2.B

Dataset description

The dataset of approximately 18,000 images is organized into folders based on countries, with each folder representing a class for classification. The dataset is divided into training, validation, and test sets with a ratio of 70%, 20%, and 10%, respectively. The pictures pixel values of the images in the dataset by dividing them by 255, converting the values from the range [0, 255] to [0, 1]. This helps improve the performance and stability of the machine learning model.

Dataset source

https://universe.roboflow.com/geoguessr-1st-dataset/geoguessr-ai-1/dataset/4

Dataset example

Picture form Canada

2.C

TensorFlow

- **Description:** open-source machine learning framework developed by Google. It is one of the most popular libraries for deep learning due to its flexibility, scalability, and comprehensive ecosystem.
- **Pros:**
  - **Scalability:** TensorFlow is highly scalable and supports distributed computing, which is useful for training large models on multiple GPUs or TPUs.
  - **Flexibility:** TensorFlow offers low-level operations for fine-grained control as well as high-level APIs (like Keras) for ease of use.
  - **Pretrained Models:** TensorFlow Hub provides access to many pretrained models, which can be fine-tuned for the landscape classification task.
  - **Visualization:** TensorFlow comes with TensorBoard, a visualization tool for tracking metrics, visualizing network architectures, and monitoring training progress.
  - **Large Community and Support:** TensorFlow has extensive documentation, tutorials, and a large community of users, making it easier to find support.
- **Cons:**
  - **Complexity:** The flexibility of TensorFlow can make it more complex to use for beginners compared to higher-level libraries.
  - **Verbose Syntax:** TensorFlow 1.x was notorious for its verbose syntax, but this has been improved with TensorFlow 2.x.

Keras (High-Level API)

- **Description:** open-source neural network library that provides a user-friendly, high-level API for building and training deep learning models.
- **Pros:**
  - **Ease of Use:** Keras provides a simple, easy-to-understand API that allows for quick prototyping and development of neural networks.
  - **Integration with TensorFlow:** Keras is tightly integrated with TensorFlow, giving access to TensorFlow's powerful features while retaining the simplicity of Keras.
  - **Modularity:** Keras is modular, meaning that you can easily add, remove, or customize layers, optimizers, loss functions, and other components.

    - **Support for Transfer Learning:** Keras supports pretrained models (e.g., ResNet, Inception), which can be easily fine-tuned for the landscape classification task.
    - **Good for Beginners:** The high-level abstraction makes it ideal for beginners or those who want to quickly experiment with different models.
- **Cons:**
    - **Less Flexibility than Raw TensorFlow:** While Keras is easier to use, it may lack the flexibility needed for highly customized models (this can be mitigated by using Keras inside TensorFlow).

PyTorch

- **Description:** popular open-source deep learning framework developed by Facebook's AI Research Lab. It is known for its dynamic computational graph and is popular in academic and research settings.
- **Pros:**
    - **Dynamic Graphs:** PyTorch's dynamic computation graph is more intuitive for developers, especially when debugging.
    - **Community:** PyTorch has a strong community, particularly in research, which means access to cutting-edge tools and ideas.
- **Cons:**
    - **Lesser Focus on Production:** PyTorch is often used in research but lacks TensorFlow's level of deployment support, although this is changing with PyTorch's growing ecosystem.
    - **Scalability:** TensorFlow's distributed training and serving capabilities are more mature, making it preferable for large-scale production systems.

Anaconda

Description: Anaconda is a free and open-source distribution of Python that simplifies package management and deployment. It is particularly useful for data science, machine learning, and deep learning workflows due to its large collection of pre-installed libraries and tools.

Pros:

- **Environment Management**: Anaconda makes it easy to create, manage, and switch between isolated environments, ensuring compatibility between different versions of libraries without conflicts.
- **Comprehensive Ecosystem**: Anaconda comes pre-installed with many popular libraries and frameworks for machine learning and deep learning, along with the ability to easily install others through its package manager, Conda.

- **Cross-Platform**: Works on Windows, macOS, and Linux.
- **Simplified Deployment**: Anaconda provides tools like Conda-Forge and Anaconda Cloud, simplifying the sharing and deployment of machine learning projects.

Cons:

- **Large Size**: Anaconda distributions tend to be large in size, which may be unnecessary for users only needing a few specific libraries.
- **Dependency Issues**: Sometimes, due to multiple pre-installed packages, dependency conflicts can arise when installing newer versions of specific libraries.

Matplotlib

Description: Matplotlib is a widely-used Python library for creating static, interactive, and animated visualizations. It is often used in data science and machine learning for plotting graphs, charts, and visualizing model performance.

3. Internal and external specification of the software solution.

3.1 Internal specification

Project is divided into 3 files/scripts:

- **TrainNew.py:** Creates a new AI model based on specified keras model in file and trains it uses prepared dataset.
- **TestLoaded.py:** Enables user to inquire about predicted class of input image
- **TrainLoaded.py:** Can be used to further train already created models.

Imports

- **import os**: For OS-level operations.
- **from keras.layers import ...**: Keras layers for building the neural network.
- **import tensorflow as tf**: Main TensorFlow module.
- **from tensorflow.keras import Sequential**: Sequential model class.
- **import matplotlib.pyplot as plt**: For plotting.

Data Handling

- **tf.keras.utils.image_dataset_from_directory(directory, shuffle=True)**: Load images from a directory and create a dataset - directory is the path to data, shuffle determines if data should be shuffled.
- **.map(lambda x, y: (x/255, y))**: Normalize images.

Argument Parsing

- **argparse.ArgumentParser(description='...')**: Create an argument parser.
- **parser.add_argument('--save_path', type=str, ...)**: Define --save_path argument for specifying where to save the model.
- **model_name = parser.parse_args()**: Parse command-line arguments.

Model Operations

- **model = tf.keras.models.load_model(model_name.save_path)**: Load a model from the specified path.
- **model.summary()**: Display model architecture.

Model Training

- **tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir='logs')**: Setup TensorBoard callback for logging.
- **hist = model.fit(train_data, epochs=30, validation_data=val_data, callbacks=[tensorboard_callback])**: Train the model.

Visualization

- **plt.plot(hist.history['loss'], ...)**: Plot training and validation loss.
- **plt.plot(hist.history['accuracy'], ...)**: Plot training and validation accuracy.
- **plt.show()**: Display plots.

Evaluation

- **test_loss, test_accuracy = model.evaluate(test_data)**: Evaluate model performance on test data.
- **print(f'Test Loss: {test_loss}')**: Print test loss.
- **print(f'Test Accuracy: {test_accuracy}')**: Print test accuracy.

Saving Model

- **model.save(model_name.save_path)**: Save the trained model to the specified path.

3.2 External specification

TrainNew.py:

To use the TrainLoaded.py script, ensure you have TensorFlow, Keras, and matplotlib installed. Navigate to the script's directory with cd path/to/your/script and run python TrainLoaded.py to train and save the model to the default path (models/imageclassifier.keras). To specify a custom save path, use python TrainLoaded.py --save_path path/to/save/model.keras.

TrainLoaded.py:

To use the TrainLoaded.py script, ensure TensorFlow, Keras, and matplotlib are installed. Navigate to the script's directory with cd path/to/your/script, then run python TrainLoaded.py to train and save the model to the default path (models/imageclassifier.keras). To specify a custom model path, use python TrainLoaded.py --model_name path/to/save/model.keras.

TestLoaded.py:

To use the TestLoaded.py script, ensure you have TensorFlow, Keras, matplotlib, numpy, and PIL installed. Navigate to the script's directory with cd path/to/your/script, then run python TestLoaded.py to classify an image using the default model and image path (models/imageclassifier.keras and Data/zdj/a.jpg). To specify a custom model or image path, use python TestLoaded.py --model_name path/to/model.keras --img_path path/to/image.jpg.

4. Experiments

Experimental background:

During the project, we went through a few main ideas that shaped the main networks which we then tested making minor changes between each test. Entire experimentation in this project can be divided into 4 distinct stages which represent our main attempts at solving targeted problem:

Early CNN

Our first attempts focused on creating a Convolutional network which stemmed from the fact, that problem proposed in project boils down to categorizing images, therefore this first conclusion was quite obvious. The dataset used at this stage was *Geoguessr ai 1 Dataset*.

As we began testing, attempts at adjusting our network started. At this stage we were using mostly quick training (5-10 epoch), our focus was mostly on following parameters:

- **Number of Convolutional and Dense layers**
  Our strategy was to start with a small network, then progress to a larger one while monitoring results. We tested various combinations – 2-7 CNN Layers, 1-5 Dense Layers. We concluded that 5 CNN Layers and 2 Dense Layers worked best.
- **Filter number and kernel size on each layer**
  Starting with kernel size, given small size of images (256x256) we opted for small kernels of 3x3. While we tested 5x5 and even 7x7 on singular layers, results were worse than all-layer 3x3 kernel.
  The number of filters was less obvious choice, we tried a vast combination from 8 to 512 filters using powers of 2 as values, concluding that filters combination of 8,16,32, 256 works best.
- **Basic setup of sequential model – optimizer, activation functions etc.**
  As we had little to no practical knowledge of using keras we went for an easy-to-use sequential model. Choosing *Adam* as quite popular and versatile optimizer and *relu* as activation function for conv. layers, as it is versatile and computationally efficient. For the final layer we used *softmax* function because our problem is a multi-class classification.
- **Computational optimalisation**
  Trying to hasten the training process we decided to use MaxPooling layers, downsampling the feature maps, reducing dimensions of the input.

We also set up visualization of accuracy and loss through matplotlib and enabled saving created models to file.

Late CNN

After concluding results from our earlier attempts, we began longer tests (10-20 epochs) from which we realized that we have large overfitting. Therefore, new focus was on combating overfitting through various means:

- **Regularization**

Combating overfitting problem, we added L2 regularization to penalize large weights and reduce overfitting. We tested L1 but as it did not improve overall performance we opted against it. Testing a variety of values from 0.0001 to 0.05 we found that 0.001 tends to work best.

- **Batch Normalization**
  Seeking optimalization we began using batch normalization. Normalizing outputs of convolutional layers and keeping better gradient sizes.

- **Dropout**
  As the last overfitting countermeasure, we added dropout to our dense layers forcing the network to be more redundant and generalized by disabling part of neurons. After research, most used values were between 0.3 and 0.5 (30%-50% neurons disabled), after testing we concluded that value 0.4 works best in our case.
- **Changing input data segregation.**
  After project presentation and consultation with teacher, we changed our dataset segregation. Earlier the dataset was pre-divided into training, validation, and test folders. Following advice after consultation we combined 3 folders into one and divided them in code. This resolved our overfitting problem.

During consultation we also were given second advice to try and test LSTM model as they are better at capturing patterns over long term.

LSTM

Following advice, we created LSTM model to familiarize ourselves with its capabilities. Creating model somewhat resembling our CNN model in layer distribution, replacing conv. layers with LSTM layers. Some of our considerations where:

- **Reshaping**
  To use LSTM Layers input must be in correct shape therefore we had to calculate what this shape should be.
- **Activation function**
  The activation function chosen for LSTM layers is tanh (hyperbolic tangent), it is the most recommended function for this type of layer.

After brief testing, we realized that LSTM is much slower to train and produced worse results than CNN even after 20 epochs, leading us to our final strategy.

Combined CNN+LSTM network

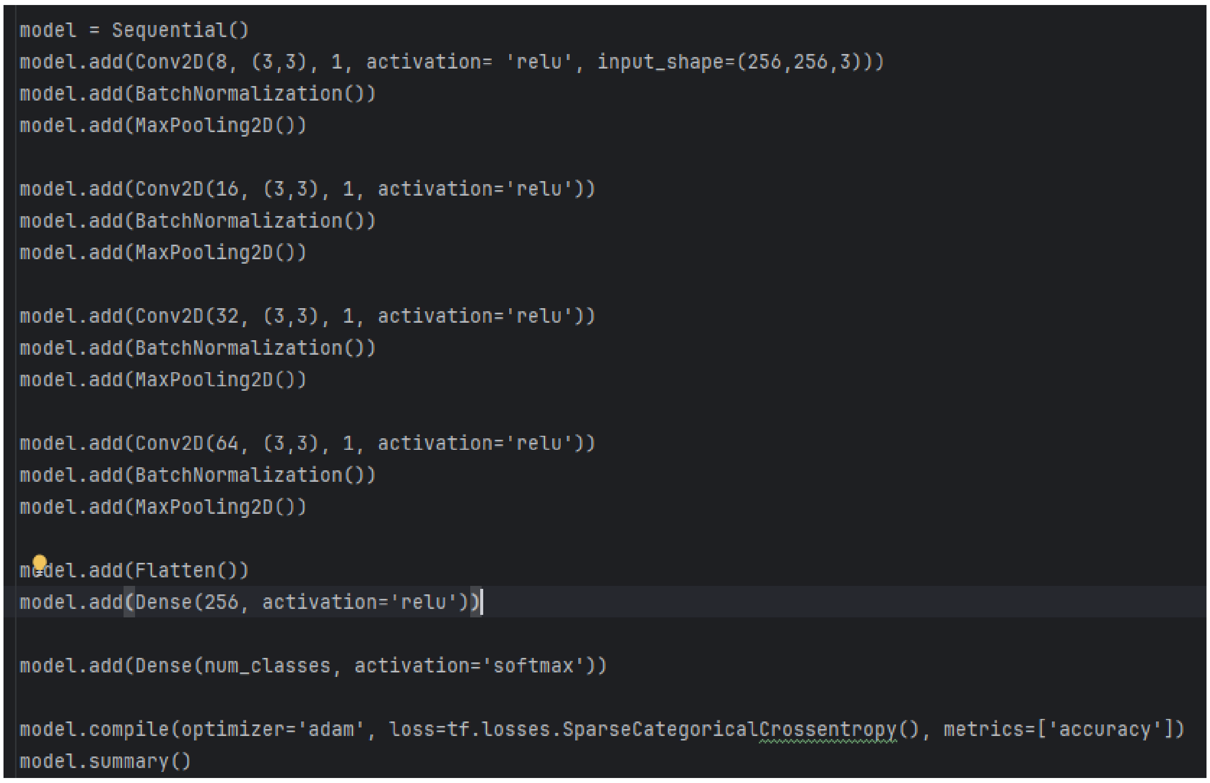
Our last approach was to combine CNN and LSM models by adding intermediate LSTM layers, between convolutional and dense layer. Trying to capture long-term patterns resulting from convolution. Tests conducted with this model were long (longest – 40 epochs), as it was very promising from start and resilient to overfitting. We also tested a total of 3 optimizers of which "Adam" produced better results than the rest. More in-depth analysis in the next section.

Presentation of experiments:

1.Early CNN

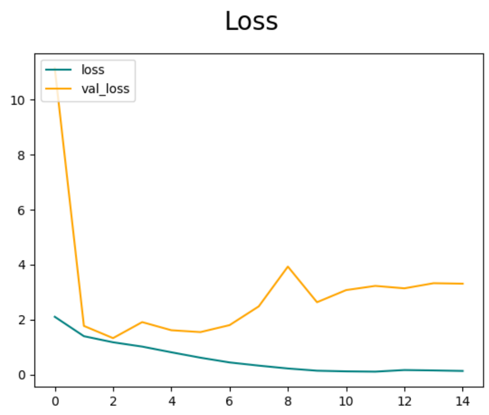
Training was performed during the first batch of models we created, using simple combination of convolutional and dense layers with added normalization.

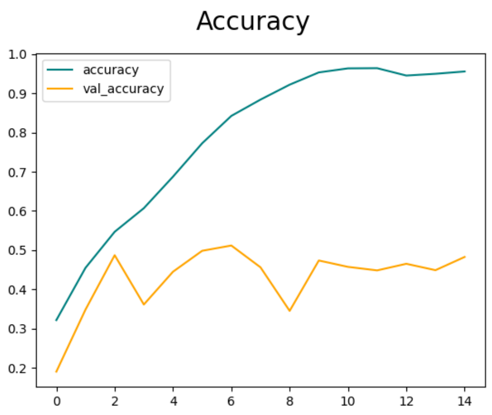
Code:

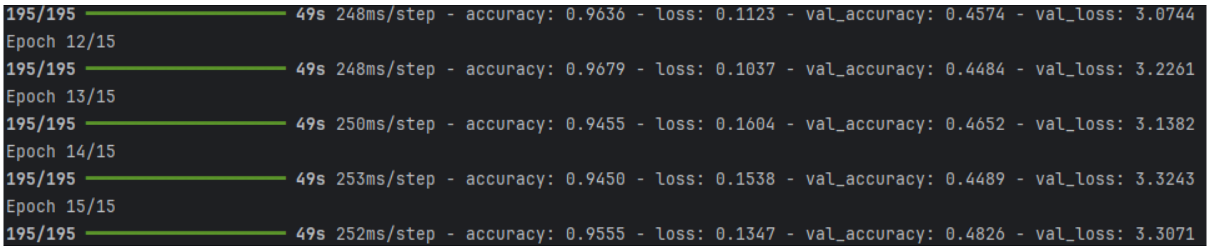


```
model = Sequential()
model.add(Conv2D(8, (3,3), 1, activation= 'relu', input_shape=(256,256,3)))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D(16, (3,3), 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D(32, (3,3), 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D(64, (3,3), 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Flatten())
model.add(Dense(256, activation='relu'))

model.add(Dense(num_classes, activation='softmax'))

model.compile(optimizer='adam', loss=tf.losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
model.summary()
```

Results:







```
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 248ms/step - accuracy: 0.9636 - loss: 0.1123 - val_accuracy: 0.4574 - val_loss: 3.0744
Epoch 12/15
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 248ms/step - accuracy: 0.9679 - loss: 0.1037 - val_accuracy: 0.4484 - val_loss: 3.2261
Epoch 13/15
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 250ms/step - accuracy: 0.9455 - loss: 0.1604 - val_accuracy: 0.4652 - val_loss: 3.1382
Epoch 14/15
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 253ms/step - accuracy: 0.9450 - loss: 0.1538 - val_accuracy: 0.4489 - val_loss: 3.3243
Epoch 15/15
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 252ms/step - accuracy: 0.9555 - loss: 0.1347 - val_accuracy: 0.4826 - val_loss: 3.3071
```

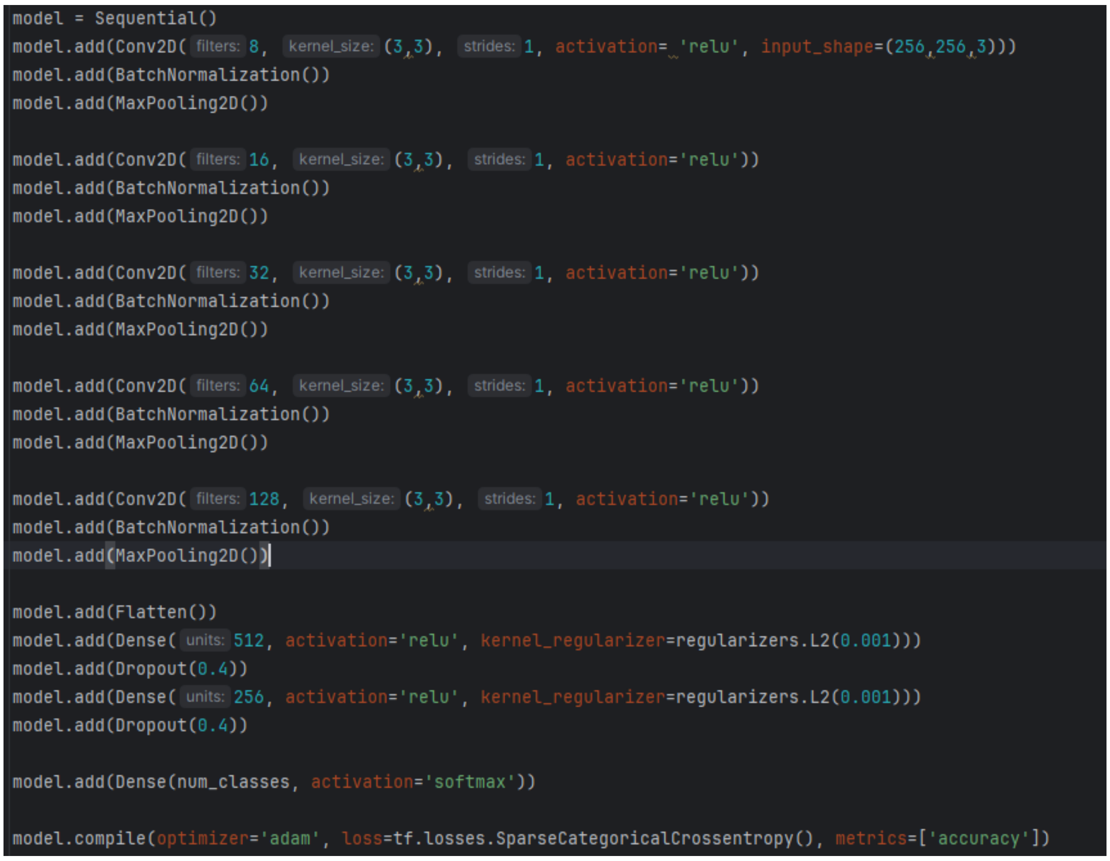
Analisys:

Looking at training graphs one major problem becomes apparent – stagnation of validation accuracy/loss compared to equivalent train values. The most probable cause of this problem is overfitting. Otherwise, training shows stable growth, no noticeable oscillations or other problems. Our focus will be on combating overfitting.

2.Late CNN

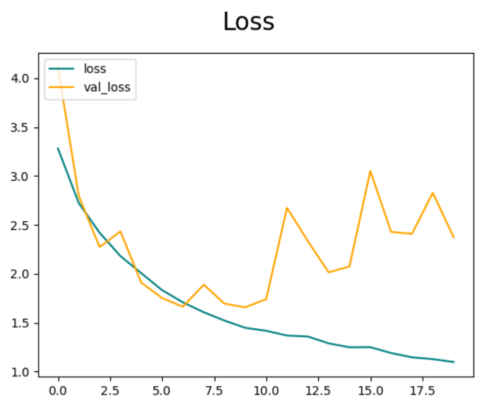
The last training we conducted before mid-project consultation. Number of best working layer combination was found and overfitting countermeasures such as regularization and dropout were used

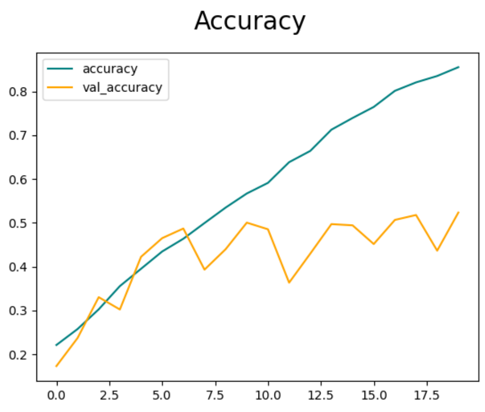
Code:

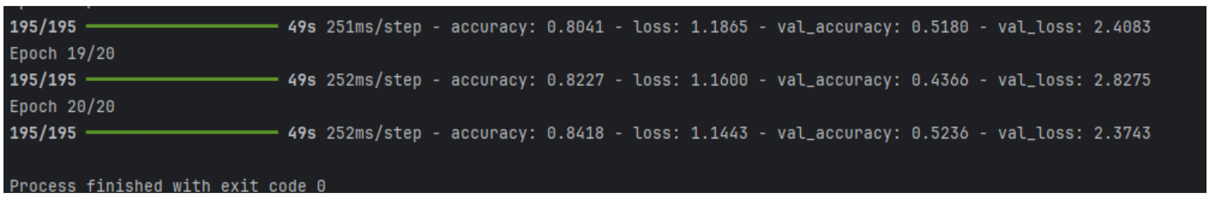


```
model = Sequential()
model.add(Conv2D( filters: 8,  kernel_size: (3,3),  strides: 1, activation= 'relu', input_shape=(256,256,3)))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 16,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 32,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 64,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 128,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Flatten())
model.add(Dense( units: 512, activation='relu', kernel_regularizer=regularizers.L2(0.001)))
model.add(Dropout(0.4))
model.add(Dense( units: 256, activation='relu', kernel_regularizer=regularizers.L2(0.001)))
model.add(Dropout(0.4))

model.add(Dense(num_classes, activation='softmax'))

model.compile(optimizer='adam', loss=tf.losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
```

Results:







```
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 251ms/step - accuracy: 0.8041 - loss: 1.1865 - val_accuracy: 0.5180 - val_loss: 2.4083
Epoch 19/20
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 252ms/step - accuracy: 0.8227 - loss: 1.1600 - val_accuracy: 0.4366 - val_loss: 2.8275
Epoch 20/20
195/195 ━━━━━━━━━━━━━━━━━━━━ 49s 252ms/step - accuracy: 0.8418 - loss: 1.1443 - val_accuracy: 0.5236 - val_loss: 2.3743

Process finished with exit code 0
```

Analisys:

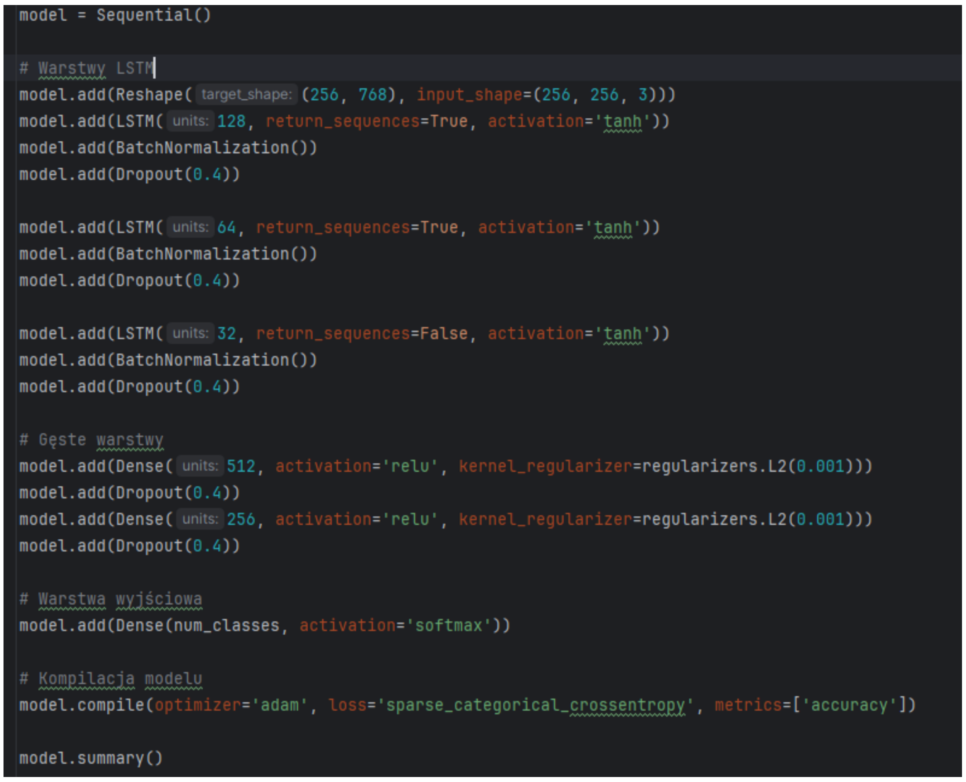
Compared to earlier attempts we can see a slight improvement in values, but the overall base problem of overfitting is the same, meaning the problem lies somewhere else. Also worth noting is decreased growth stability and learning curve being more linear than logarithmic which suggests training is slower and did not reach its peak. We may need to increase the number of epochs.

After consulting these results with the teacher, a suggestion was made to change dataset division method as it is the probable cause of growth stagnation.

3.LSTM

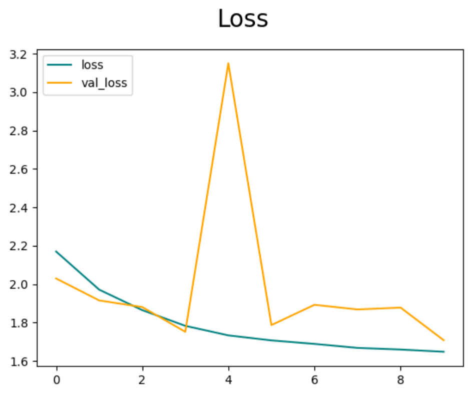
During consultation it has been suggested that we use LSTM network. Following this advice we first tried to create LSTM model somewhat equivalent to our CNN model.

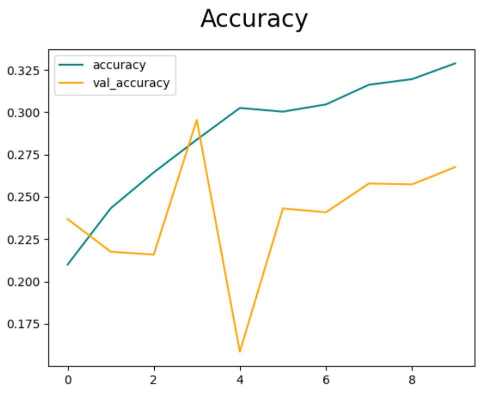
Code:

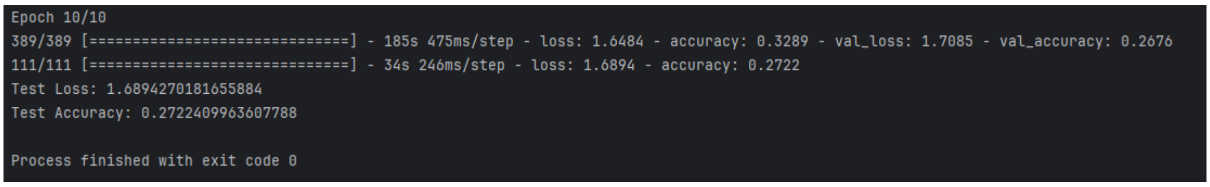


```
model = Sequential()

# Warstwy LSTM
model.add(Reshape( target_shape: (256, 768), input_shape=(256, 256, 3)))
model.add(LSTM( units: 128, return_sequences=True, activation='tanh'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(LSTM( units: 64, return_sequences=True, activation='tanh'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(LSTM( units: 32, return_sequences=False, activation='tanh'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

# Gęste warstwy
model.add(Dense( units: 512, activation='relu', kernel_regularizer=regularizers.L2(0.001)))
model.add(Dropout(0.4))
model.add(Dense( units: 256, activation='relu', kernel_regularizer=regularizers.L2(0.001)))
model.add(Dropout(0.4))

# Warstwa wyjściowa
model.add(Dense(num_classes, activation='softmax'))

# Kompilacja modelu
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.summary()
```

Results:







```
Epoch 10/10
389/389 [==============================] - 185s 475ms/step - loss: 1.6484 - accuracy: 0.3289 - val_loss: 1.7085 - val_accuracy: 0.2676
111/111 [==============================] - 34s 246ms/step - loss: 1.6894 - accuracy: 0.2722
Test Loss: 1.6894270181655884
Test Accuracy: 0.2722409963607788

Process finished with exit code 0
```

Analisys:

As this was only a short run to test feasibility of this approach, we only trained model for 10 epochs. But comparing results to that of CNN models at same time scale LSTM seems to perform worse. Huge loss at 4th epoch is probably caused by compounding LSTM learn cycle and dropout. Based on these results we decided to discontinue mainly LSTM network in Favour of CNN+LSTM solution.

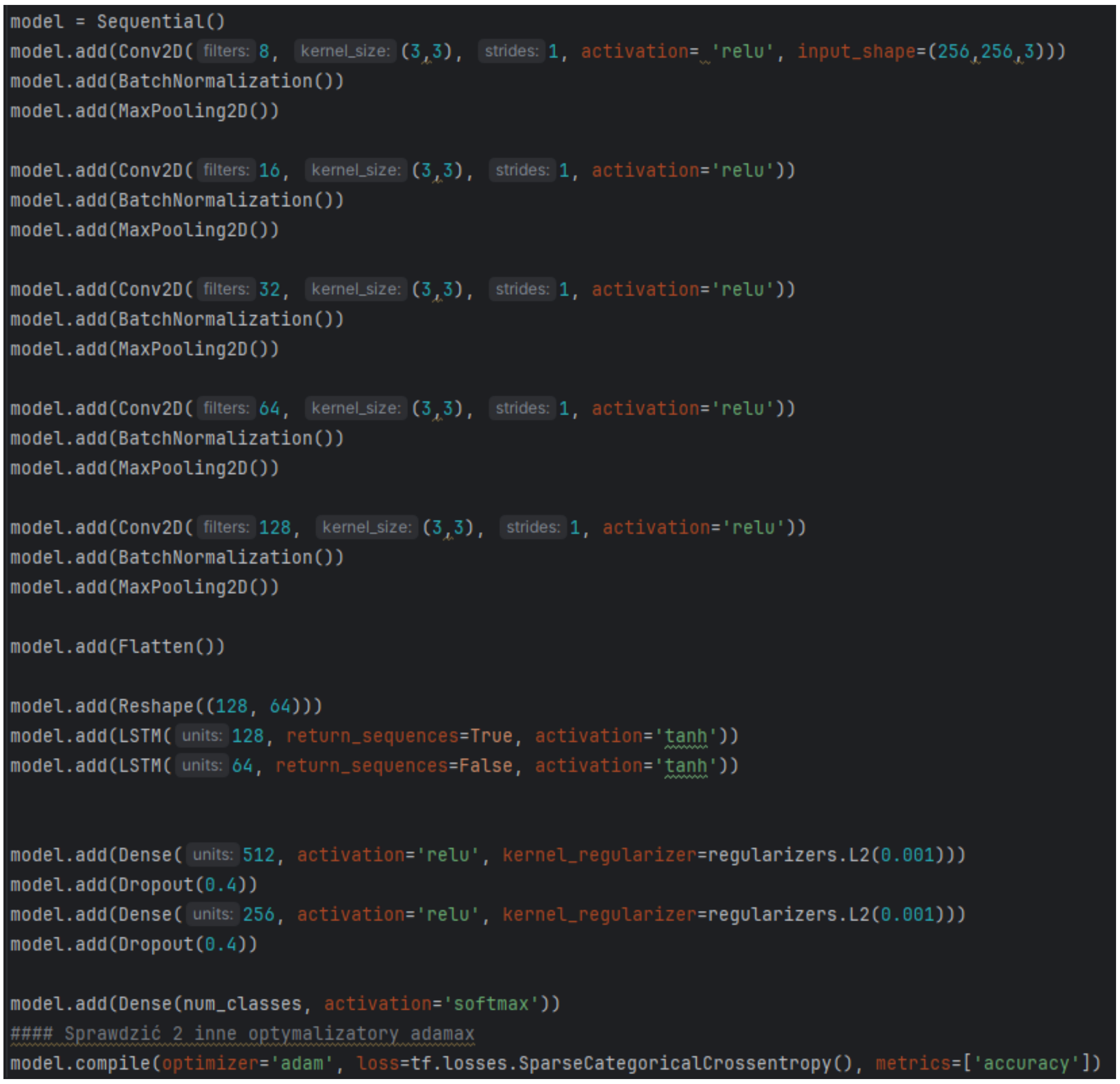
4.Combined CNN+LSTM network

Experiments from our last approach.

Adam

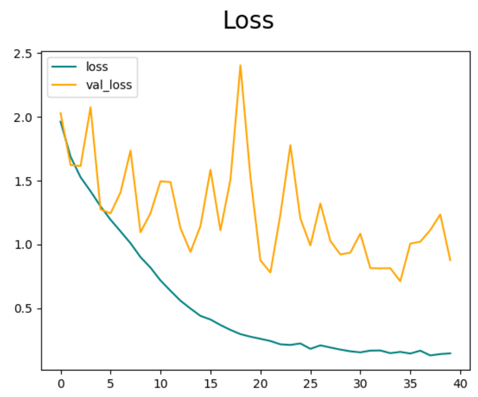
This training was conducted on our latest model combining LSTM and CNN, using Adam optimizer, we will compare results with same models with other optimizers.

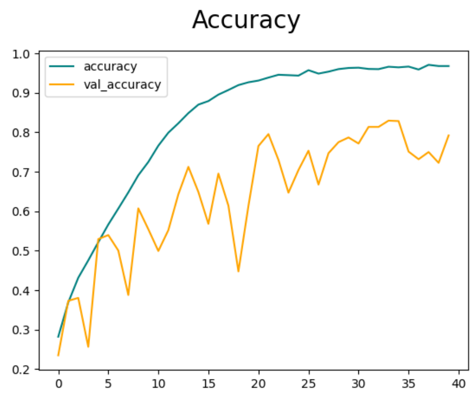
Code:

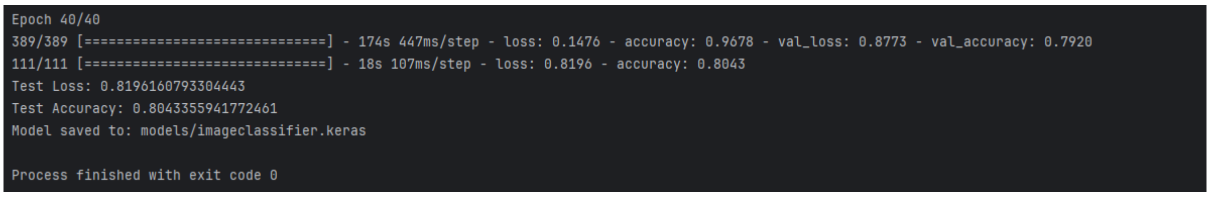


```
model = Sequential()
model.add(Conv2D( filters: 8,  kernel_size: (3,3),  strides: 1, activation= 'relu', input_shape=(256,256,3)))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 16,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 32,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 64,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Conv2D( filters: 128,  kernel_size: (3,3),  strides: 1, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())

model.add(Flatten())

model.add(Reshape((128, 64)))
model.add(LSTM( units: 128, return_sequences=True, activation='tanh'))
model.add(LSTM( units: 64, return_sequences=False, activation='tanh'))


model.add(Dense( units: 512, activation='relu', kernel_regularizer=regularizers.L2(0.001)))
model.add(Dropout(0.4))
model.add(Dense( units: 256, activation='relu', kernel_regularizer=regularizers.L2(0.001)))
model.add(Dropout(0.4))

model.add(Dense(num_classes, activation='softmax'))
#### Sprawdzić 2 inne optymalizatory adamax
model.compile(optimizer='adam', loss=tf.losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
```

Results:







```
Epoch 40/40
389/389 [==============================] - 174s 447ms/step - loss: 0.1476 - accuracy: 0.9678 - val_loss: 0.8773 - val_accuracy: 0.7920
111/111 [==============================] - 18s 107ms/step - loss: 0.8196 - accuracy: 0.8043
Test Loss: 0.8196160793304443
Test Accuracy: 0.8043355941772461
Model saved to: models/imageclassifier.keras

Process finished with exit code 0
```

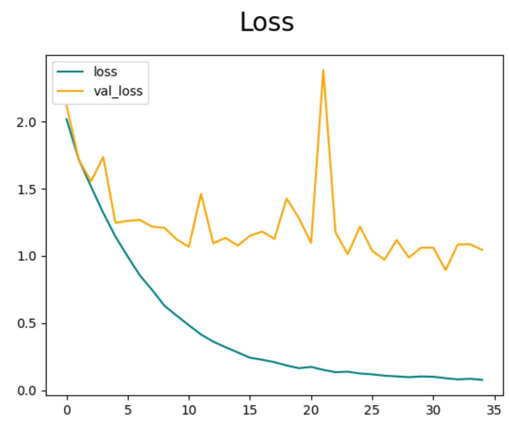
Analisys:

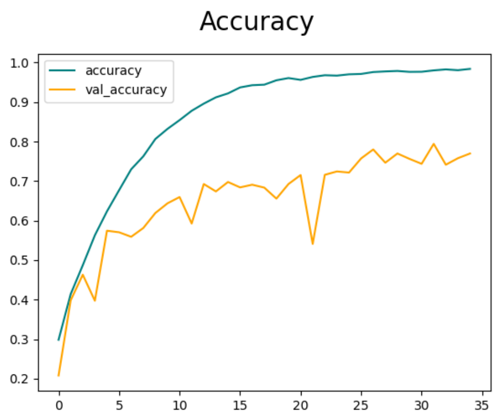
Results show continuous growth of train accuracy, and somewhat unstable growth of validation accuracy stagnating at 34 epoch. Overall end results are quite good and by far better than earlier models, combination of LSTM and dropout results in large oscillation of loss/accuracy. Longer training probably wouldn't produce better results as it did not improve during the last 6 epochs.

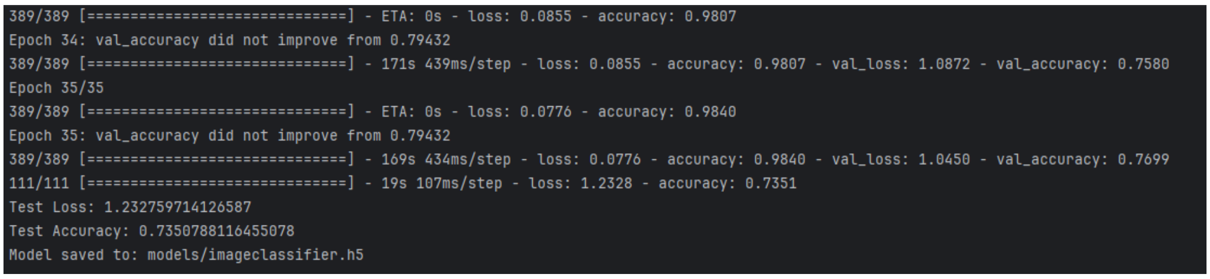
AdaMax

Same model, only difference is use of Adamax optimizer.

Results:







```
389/389 [==============================] - ETA: 0s - loss: 0.0855 - accuracy: 0.9807
Epoch 34: val_accuracy did not improve from 0.79432
389/389 [==============================] - 171s 439ms/step - loss: 0.0855 - accuracy: 0.9807 - val_loss: 1.0872 - val_accuracy: 0.7580
Epoch 35/35
389/389 [==============================] - ETA: 0s - loss: 0.0776 - accuracy: 0.9840
Epoch 35: val_accuracy did not improve from 0.79432
389/389 [==============================] - 169s 434ms/step - loss: 0.0776 - accuracy: 0.9840 - val_loss: 1.0450 - val_accuracy: 0.7699
111/111 [==============================] - 19s 107ms/step - loss: 1.2328 - accuracy: 0.7351
Test Loss: 1.232759714126587
Test Accuracy: 0.7350788116455078
Model saved to: models/imageclassifier.h5
```

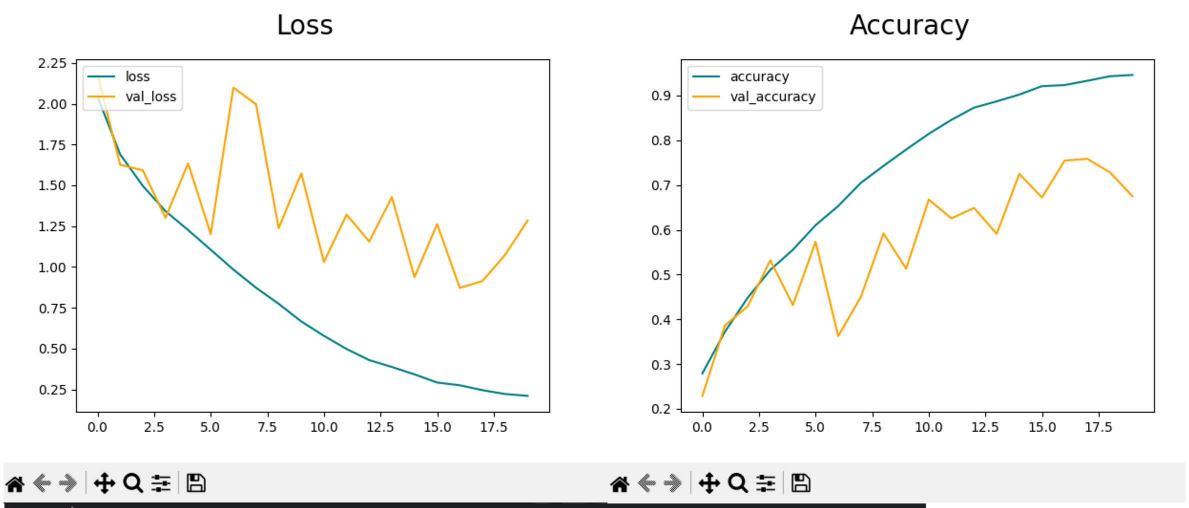
Compare:

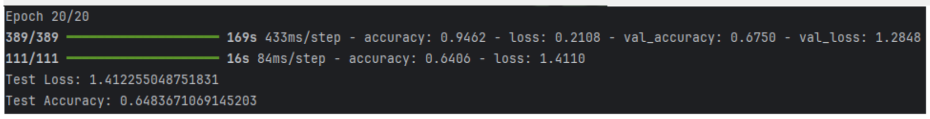
Compared to baseline of Adam, model produced much more stable growth but with big downside of stagnating much quicker, making slow progress after 12th epoch, finally producing 73% test accuracy compared to 80% when using Adam.

AdaDelta

Last version of our final model, using AdaDelta as optimizer.

Results:





```
Epoch 20/20
389/389 ━━━━━━━━━━━━━━━━━━━━ 169s 433ms/step - accuracy: 0.9462 - loss: 0.2108 - val_accuracy: 0.6750 - val_loss: 1.2848
111/111 ━━━━━━━━━━━━━━━━━━━━ 16s 84ms/step - accuracy: 0.6406 - loss: 1.4110
Test Loss: 1.412255048751831
Test Accuracy: 0.6483671069145203
```

learnig rate = 1.0

Compare:

Testing with final optimizer AdaDelta produced somewhat mid results compared to both previous models, being less stable than AdaMax and more than Adam. In terms of values, it was also in-between other versions. Final result is lowest, but training only for 20 epochs probably did not reach its full capacity.

5. Summary, overall conclusions, possible improvements, future work etc.

Project can be considered successful as proof of concept. Created models can, with quite good accuracy (around 80%), guess images country of origin, but being trained mostly on data from a total of 8 countries, it may be lacking when used on bigger variation of data. Next step for future development is therefore obvious: test model on more diverse dataset and see how well it work, then make modifications according to results.

6. References

Sites with information:
StackOverflow
AI Chat:
ChatGPT

Videos introducing the topic:

https://www.youtube.com/watch?v=jztwpsIzEGc
https://www.youtube.com/watch?v=5Ym-dOS9ssA&list=PLhhyoLH6IjfxVOdVC1P1L5z5azs0XjMsb

Repository used as example

https://github.com/nicknochnack/ImageClassification/blob/main/Getting%20Started.ipynb

7. Repository of project:

https://github.com/ElMilos/GeoGuesser-Biai