UNIVERSIDAD PANAMERICANA CAMPUS GUADALAJARA

GRAPHIC SIMULATION

FINAL PROJECT

ANALYSIS REPORT OF A NEURAL NETWORK FOR THE CLASSIFICATION OF IMAGES

AUTHORS:

SARANIENKE CALDERÓN HOHMANN

ANA PAULA LUNA POSSO

LUIS HUMBERTO RUBIO VALENZUELA

MAY 30, 2023

Introduction

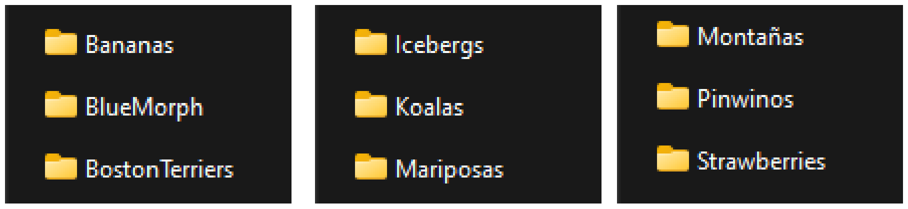
Artificial Intelligence is at its peak moment right now, and it is everywhere we look; it can even be said, with a certain degree of confidence, that we use it every single day. Continuing with the development of this trend, comes this project. The purpose of this project is to create an AI with the ability to distinguish images based on their content and adapt that to a website able to receive an image file as an input and classify the image into one of the possible groups.

Methodology

Datasets

Before we start describing our current methods, we are going to take some time to briefly explain a method we ultimately decided not to use. When we were commencing this project, we were planning to do manual feature extraction and then use this information to train a model based on simple linear regression. This approach is possible and very feasible, however, the features that we managed to extract were insufficient as they were only related to color and did not take into account important features of the image such as shapes, lines, contrast, perimeters, etc. That, added with the fact that we were going to use a simple method as a loss function, let us with little to no hope of achieving even above 50% accuracy in our model. That being said, let's explain the model what we currently use, with which we achieve an accuracy of 96%

Our dataset consists of a collection of 900 image, 100 of each category. These categories are the following:

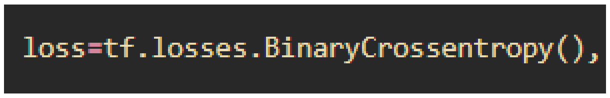


Bananas
BlueMorph
BostonTerriers
Icebergs
Koalas
Mariposas
Montañas
Pinwinos
Strawberries

All these images are 1280x960, in full color and with no added compression or effects.

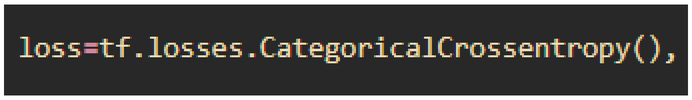
In order to introduce this images to the neural network, we need to make them easier to process, since 1,228,800 times every single neuron on the network is very time consuming and ineffective. That is why we create smaller versions of the images and then we apply a gaussian blur, and even after that, we apply 2D Convolution and Flatten the image at the end. That way we manage to obtain 64x64 versions of the images, which are effectively 0.3% of the work that needed to be done for the full res image. This means that we managed to make the process 300x faster by using this method.

Once this was done, we could start modifying the neural network. In its initial state, the neural network used binary crossentropy as the loss function, which allowed us to distinguish things in a binary manner, e.g. is a koala, or is not an koala, but it did not allow us to classify each image into its class, as it only had the capability to binarily distuinguish between two types of things.



```
loss=tf.losses.BinaryCrossentropy(),
```

For what we wanted to achieve, we needed something that allowed us to categorize between multiple groups, not only one. This is where categorical crossentropy comes in. With this we could start distinguishing between different categories and not only one.



```
loss=tf.losses.CategoricalCrossentropy(),
```

Optimizers

Optimizers for neural networks are very important when it comes to training deep learning models. Their main job is to adjust the parameters of the model in a smart way, so that the predicted outputs are as close as possible to the true targets. These optimizers make the process of updating the weights and biases in the neural network more efficient, which in turn improves how well the model performs. They use different optimization algorithms, like stochastic gradient descent (SGD), Adagrad, or RMSprop. These algorithms calculate the gradients of the loss function with respect to the model's parameters, and then decide how much and in which direction the parameters should be updated. They use mathematical techniques, such as gradient descent and adaptive learning rates, to find the best possible configuration that minimizes the overall error.

In our work we tested the performance of different optimizers and found that some of them fitted the needs of our neural network, and some did not. In this part we are going to discuss the key differences between those optimizers that fit our model and show the overall performance of the neural network using those optimizers.

The first optimizer we used is *Stochastic Gradient Descent* (SGD), to understand SGD first we need to know what Gradient Descent is. GD aims to minimize the difference between predicted outputs and true runs. It works by iteratively adjusting the model's parameters in the direction of the steepest descent of the loss function by using the whole dataset each time. The goal is to find the optimal parameter values that result in the lowest possible error. In the other hand we have SGD, which fixes this by changing how much of the dataset is introduced each time. With Stochastic Gradient Descent we use mini-batches of random elements of the dataset, this allows the training to be faster as it skips local minimums more often than GD.

The second optimizer that fitted our model is ADAM. ADAM is the result of a lot of research in the optimizers field and it all started with the learning rate. Learning rates are how “big” are the changes between steps when training a neural network. Up until the days of SGD, learning rates were a fixed value that determined either if the learning process was going to be accurate and last longer, or imprecise but faster. Adaptive Rate Optimizers came in to change this, and Adagrad appeared. Adagrad offered an adaptive learning rate, but it came with a cost: the sum of squares of the gradients increased monotonically, which caused the learning rate to eventually in some cases, tend to zero.

$$\underbrace{G_{t,ii} = \nabla^2_{\theta_{1,i}} J(\theta_{1,i}) + \nabla^2_{\theta_{2,i}} J(\theta_{2,i}) + \cdots + \nabla^2_{\theta_{t,i}} J(\theta_{t,i})}_{\text{Monotonically increasing}}$$

$$\theta_{t+1,i} = \theta_{t,i} - \underbrace{\frac{\eta}{\sqrt{G_{t,ii} + \epsilon}}}_{\text{tends to 0}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

Adadelta came to the rescue and added a new gamma parameter that acted as a weight, which reduces the effects of the G parameters by an exponential factor, so the learning rate does not tend to zero anymore.

$$G_{t,ii} = \gamma G_{t-1,ii} + (1-\gamma) \nabla^2_{\theta_{t,i}} J(\theta_{t,i})$$

$$\theta_{t+1,i} = \theta_{t,i} - \underbrace{\frac{\eta}{\sqrt{E\left[G_{t,ii}\right] + \epsilon}}}_{\text{doesn't tend to 0}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$$

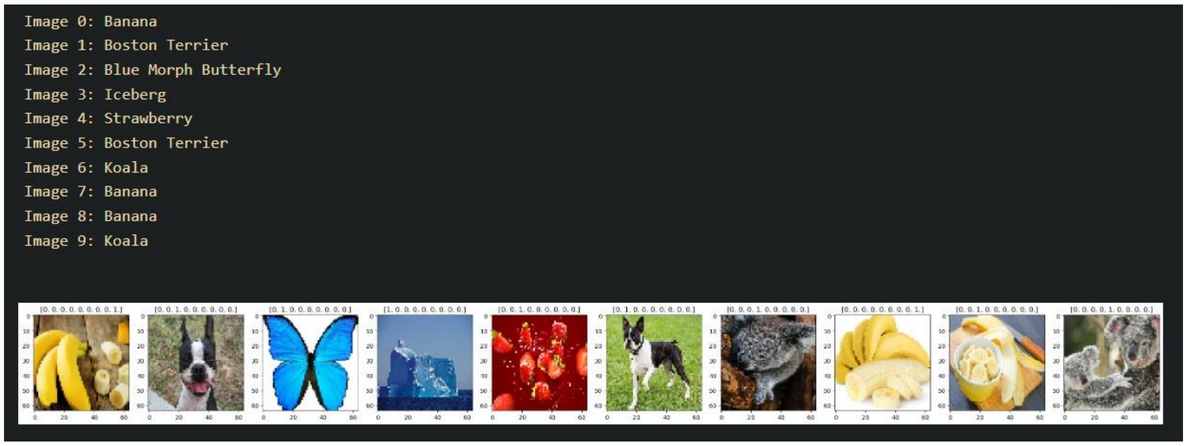
The problem with this approach is that the learning rate can get so big that it starts skipping over minimums in some cases, which is not a desired outcome, so something had to be done with Adadelta. This is when momentum made its way to the adaptive rate algorithms, and the birth of the ADAM optimizer. ADAM is the combination of Adadelta with the use of momentum. What it does is it starts changing its pace according to how similar the inputs are to what it knows, when it accumulates momentum it goes faster, as there is not much risk of being wrong, and when it finds something different from what it knows, it resets the momentum to be more "careful". All this work that created ADAM was very worth it, since ADAM is now one of the most rounded algorithms that can be used in a lot of different kinds of problems.

Lastly, we used NADAM, which is a modified version of ADAM, it uses the Nesterov's Accelerated Gradient algorithm, which predicts the future gradient position using its momentum and applies those changes to the parameters. This allows to have a better estimation of the gradient and leads to faster convergence. However, although convergence often happens quicker with NADAM, it does not mean that it is superior to ADAM in all cases, as ADAM is still more precise, which means that ADAM is a better fit to complex problems that require more time to learn.

Since our problem is very simple, we sticked to NADAM and obtained better accuracy, that will be shown in the graphics further ahead.

Results

After many tests with different optimizers, learning rates and epochs (which are going to be discussed in a further section), we successfully managed to classify all the images onto their respective categories.



```
Image 0: Banana
Image 1: Boston Terrier
Image 2: Blue Morph Butterfly
Image 3: Iceberg
Image 4: Strawberry
Image 5: Boston Terrier
Image 6: Koala
Image 7: Banana
Image 8: Banana
Image 9: Koala
```

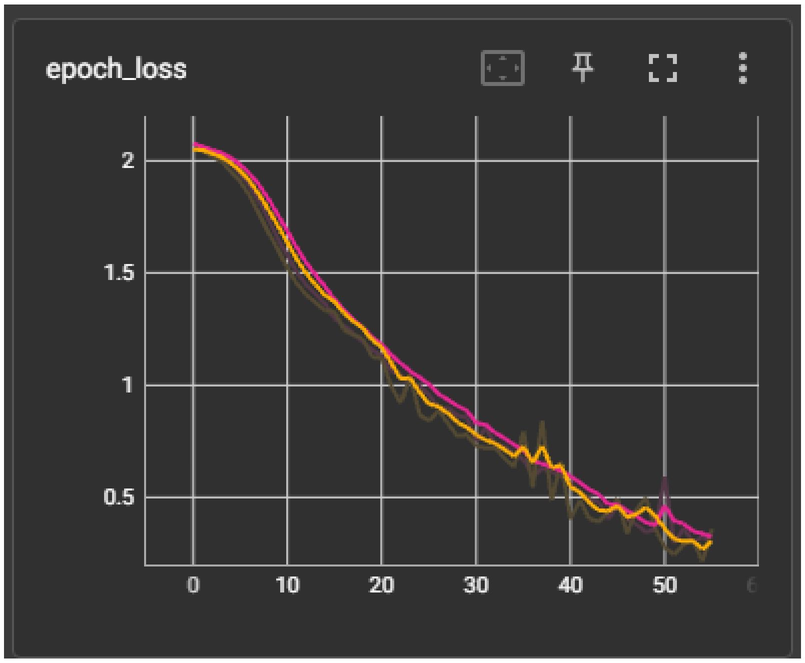
As it can be seen, it properly identifies between all the 9 categories of images. Now lets discuss the setup regarding optimizers, learning rates and epochs.

Compiler configuration

All configurations were targeted to obtain an accuracy value higher than 0.90 and avoid overtraining the model.

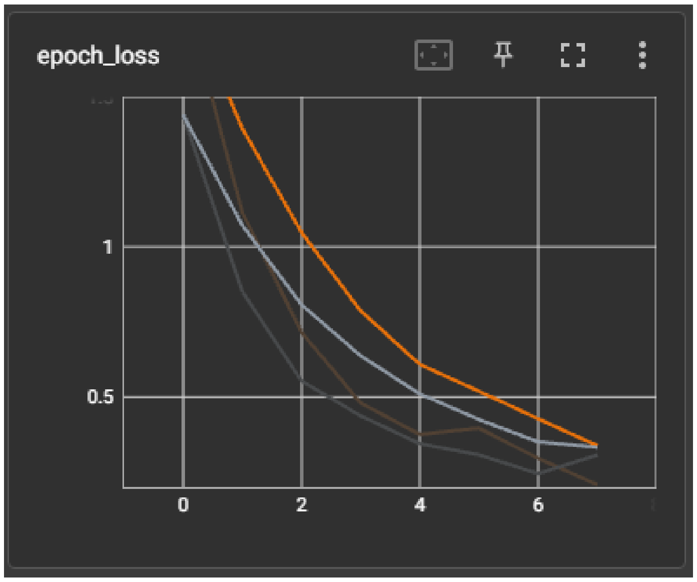
SGD

- Learning rate: 0.03
- Epochs: 56



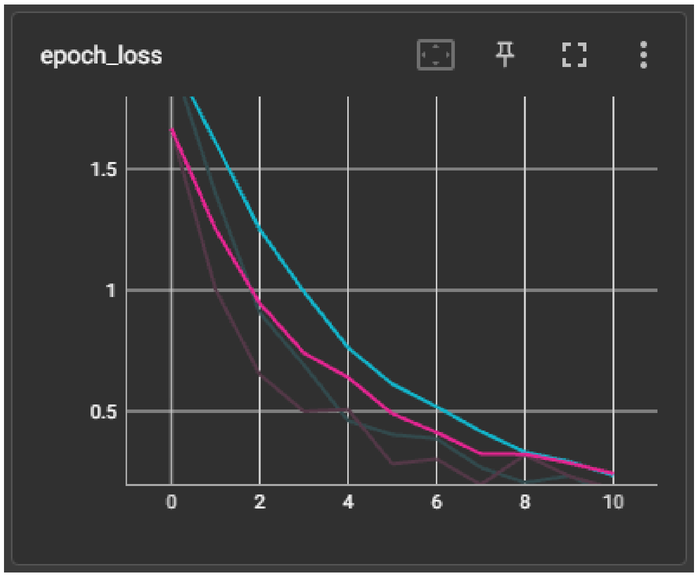
ADAM

- Learning rate: 0.05
- Epochs: 8



NADAM

- Learning rate: 0.01
- Epochs: 12



Reference list

- [CodeEmporium]. (2020, February 10). *Optimizers - EXPLAINED!* [Video]. YouTube. https://www.youtube.com/watch?v=mdKjMPmcWjY
- Lavika, G., Akshina, J., & Shray, M. (2022). Design and implementation of a crop recommendation system using nature-inspired intelligence for Rajasthan, India. Deep Learning for Sustainable Agriculture, Chapter 4, 109-128. https://doi.org/10.1016/B978-0-323-85214-2.00005-7