Neural Network Image Classification

Image classification is the process of categorizing and labeling groups of pixels or vectors within an image based on specific rules. This article will demonstrate a walk of a real-world example of using neural nets. The purpose is to classify images of rock, paper, and scissors using a convolutional neural net written in TensorFlow and Keras.

Todays methods regarding Image classification in machine learning focuses on Convolutional Neural Networks. They are are complex feed forward neural networks. CNNs are used for image classification and recognition because of its high accuracy. It was proposed by computer scientist Yann LeCun in the late 90s, when he was inspired from the human visual perception of recognizing things. The CNN follows a hierarchical model which works on building a network, like a funnel, and finally gives out a fully-connected layer where all the neurons are connected to each other and the output is processed.

TensorFlow offers prepared datasets that can be accessed via the TensorFlow builder. For this project, we chose the 'rock_paper_scissors' dataset. The dataset contains 2892 images with 300 heights on 300 widths on (0~255) color channels. The classes are represented via integers (0 for rock, 1 for paper, 2 for scissors). 2520 of the images will be used for training and the rest (372) will be used for testing and validation. All the images are artificially generated images of rock, paper, and scissors. Hence, they're not real images but the model will utilize them as though they were.

The first step in processing the data is meant for any sort of image classification task. Converting the format of the TensorFlow dataset into a NumPy format so that is a more universal and easier to work with instead of working with multidimensional images. Looking at the documentation for the datasets within TensorFlow shows how to iterate over the dataset and preprocess it before feeding it to the model.

The images are RGB and thus contain color. Images with color contain more data than is needed so it is best to minimize the number of things the network has to learn (complexity wise). Converting the images to a greyscale will do the trick. How? Reshaping the images to 300 x 300 x 1 is letting Kera's know these are just grayscale images. Scaling every value to be between 0 and 1 is a good common practice that helps with classification problems. The network can still learn if values between 0 to 255 were inserted, but ultimately, in most cases, it's going to decrease performance.

After preprocessing the images, removing the color, and converting them into NumPy arrays, the training can begin. Creating the first neural network for this task using the Keras library. The sequential() function helps build a new model and takes as an input the layers of the neural network.

One of the layers that will be used in this paper is Flatten(), which will turn the 300x300 images into a single column of data. Each other layer, dense() will output a dimensional vector according to the first variable, attempting at scoring the best features and pulling the most valuable data.

For each layer, an activation function is needed. In this paper, we chose to use ReLu. The main reason why *ReLu* is used is that it is simple, fast, and empirically seems to work well. Early papers observed that training a deep network with ReLu tended to converge much more quickly and reliably than training a deep network with sigmoid activation.

Finally, the output layer must consist of the same number of classes that the dataset has. For that, the activation function chosen was '*soft-max*' which is a generalization of the logistic function to multiple dimensions. It is used in multinomial logistic regression and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

For the loss function, the *Adam optimizer* is used, which is a replacement optimization algorithm for stochastic gradient descent for training deep learning models. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. The loss function itself is *Sparse Categorical Cross entropy*.

On the first attempt accuracy score for training was high and exceeded 85% accuracy. However, on the testing, the model performance was very poor and barely reached 50% accuracy. The conclusion was that the model was overfitting the training data and not generalizing enough.

The big disconnect here is that using a fully connected layer for these 300 by 300 images and so that's a total of 90 thousand single pixels that were connecting to the next layer of the network. Too much importance was put in those little single pixels so what's happening here is overfitting to the training data and the neural network not learning good patterns.

This is what is leading to the use of convolutional neural networks, to get a more sense of generalizable features within the rock-paper-scissors dataset.

Creating a CNN model requires a sequential model via Keras. This time the first layer will be AveragePooling2D() because the dataset consists of 2D images and it is a better practice to reduce the size of the image and create an average of the pixels in each block. The first variable inserted in the function is the size of the grid that is passing over the image, the second variable is the number of steps at each iteration.

Additional two Conv2D() layers convolution smaller layer are inserted. After that, another very common thing in convolutional neural nets is to do some max pooling. An additional layer called MaxPool2D(), which performs like the AveragePooling() but extracts the max pixel at each grid. Reducing the size of the data makes it so the individual pixels don't have as much weight anymore. Input space is a smaller parameter size so that the model should generalize more.

One of the main issues with the rock-paper-scissors training dataset is that it doesn't contain many pictures and the ability to generalize is not great. A possible fix for that will be using the Dropout() function.

What dropout will do is that after the process is done with the max pool 2D layer, it is going to cut out 50% of the connections so as the model trains. 50% of the connections are going to be dropped out and ultimately the epochs will be more effective because the data will be more "interesting". Since the connections are dropped randomly, each of the individual connections will have to generalize more and be able to handle more variation.

For the last layer, instead of going from the entire remaining data to three classes, the last layer is going to try and capture some higher-level information and boil it down to the three better than just passing all the outputs of the convolution and the dropout and everything straight to the classification layer.

Adding another dense layer after the flatten() layer made the accuracy score jump higher than our previous attempts. 70 percent accuracy.

The many attempts at reaching higher accuracy forced the conclusion that a tool for testing various combinations of layers and input must exist. And so it does, Keras Tuner is an easy-to-use, scalable hyperparameter optimization framework that solves the pain points of hyperparameter search. Easily configure your search space with a define-by-run syntax, then leverage one of the available search algorithms to find the best hyperparameter values for your models.

The Keras Tuner allows the user to insert an X parameter inside the layers and allow the machine to do all the Trial-Error attempts instead of him. It may take the computer a while to test out all the different combinations that the user inserted, but at the end the built-in function, RandomSearch will output the model that performed best.

At last, with the attempts of the Keras Tuner, the model reached 80% accuracy on the test set. This is a significant jump in accuracy in what appears to be "a shorter time".

To arrive at the 80% accuracy we had to experiment with different layers in different orders. Our worst result was arriving at 45% accuracy of the test samples. Each time we connected an additional layer we got closer to improving the results. MaxPooling() and AveragePooling() were the key layers to make the model perform better.

At some standards, 80% accuracy is not the best, and probably with additional try errors, it is possible reaching higher results. However, given the limited dataset, the results are satisfying for we managed to generalize the model with a small database.

An additional step that was taken in order to confirm the performance of the CNN is to compare the results with MLP and Soft-max predictive algorithms. The Softmax scored 0.5544 accuracy on the training set and only 0.3683 on the testing set. In comparison, MLP results were 0.771 on the training set and 0.5753 on the testing set.

	Logistic Regression (Soft-max)	**MLP**	**CNN**	**CNN-Tuner**
Train accuracy	0.5544	0.771	1	1
Test accuracy	0.3683	0.5753	0.771	0.8065

In conclusion, convoluted neural networks appear to be efficient and smart in comparison to other machine-learning algorithms. The main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision. Little dependence on preprocessing, decreasing the needs of human effort developing its functionalities. From this experience, we gathered that CNN might overfit quickly and thus require additional effort in generalizing.