Turkish lira classification

Nicolò Verardo

Università degli Studi di Milano, Milano, Italia
nicolo.verardo@studenti.unimi.it
nicoloverardo.surge.sh

Abstract. In this work we address a multi-class classification task on a dataset of images representing banknotes of the Turkish currency. Although the dataset is made available offline, we will address the requirement of implementing a solution that is scalable and works in a distributed scenario. We will train a Convolutional Neural Network in order to achieve the goal of correctly classify each banknote with its value.

Keywords: image · multiclass · classification · scalable · distributed-training

1 Introduction

Today, we live in an era where every day we generate lots of data of any kind: video, images, texts (just to cite a few), where the abundance of information has posed some interesting problems regarding how to store and analyze this huge amount of data that are continuously generated. However, this has also allowed some fields, like image recognition, to gain significant improvements in their predictive power thanks to the fact that we can use powerful algorithms like neural networks and feed them with a huge quantity of training samples. Nevertheless, many scenarios cannot be handled by commodity hardware anymore and they require different solutions: this is were distributed computing comes to the rescue. In this work we will train a Convolutional Neural Network in order to recognize banknotes but we will put emphasis on the scalability problem, so that the solution presented will be able to deal with a much larger dataset. The scalability issue will be addressed in a specific section.

2 The dataset

The "Turkish Lira Banknote Dataset" is a dataset publicly available on Kaggle containing 6000 images of banknotes of the Turkish currencies belonging to the classes 5, 10, 20, 50, 100 and 200. This dataset was collected to develop applications for visually impaired people. Images are divided in sub-folders according to their label; each folder contains 1000 images of shape 1280x720 with 3 color channels. Some data augmentation techniques like flipping, noise addition,

brightness increase or decrease, have already been applied to the images. The dataset originally includes also two text files containing the filenames of the images belonging to the training set and the test set. Moreover, the dataset classes are perfectly balanced: thus, there is no need to apply any of the techniques used to fight imbalancement in the labels, like oversampling/undersampling.

Fig. 1. Original image of a banknote of class 10

Fig. 2. Data augmented (flipped) image of the same banknote

3 Algorithms

3.1 Neural Networks

An artificial neural network is a model of computation inspired by the structure of neural networks of the brain. It receives an input, changes its internal state (activation) according to that input, and produces an output depending on the input and activation. A neural network can be described as a directed graph whose nodes correspond to neurons and edges correspond to links between them. Each neuron receives as input a weighted sum of the outputs of the neurons connected to its incoming edges.
Neural networks are structured in layers, which are sets of nodes. The first layer is called *input layer*, those in the middle of the network are called *hidden layers* and their nodes *hidden nodes* (or *hidden units*), since we do not directly observe them during training; the final one is called *output layer*. A neural network that has more than one hidden layer is called *deep neural network*.
There are several activation functions. Among the most used ones we can find the Sigmoid, the Softmax and the ReLU.

Although neural networks have proven to be very powerful in many scenarios, they are still subject to some problems.
Bad starting values for the weights, for instance, may lead to bad performance of the network. If we set starting values that are exactly zero, then the algorithm never moves; on the contrary, using too large initial values for the weights may lead to poor solutions.
Neural networks may also suffer from overfitting: using too many weights for our model is one of the causes.

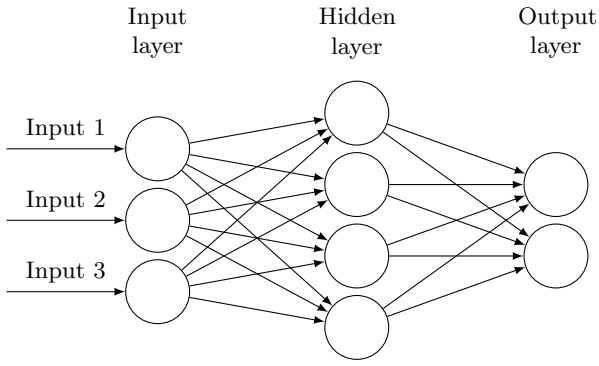

Fig. 3. A neural network with a single hidden layer

3.2 Convolutional Neural Network

A Convolutional Neural Network is a deep learning algorithm that has recently gained a lot of attention because of its suitability into the computer vision field because the pre-processing required in a CNN is much lower as compared to other classification algorithms; in fact, it provides a reduction in the number of parameters involved and the reusability of weights.

The structure of a CNN consists of stacking convolutional and pooling layers with dense ones. In the convolutional layer, a convolution multiplies a matrix of features with a filter matrix (or *kernel*) and sums up the multiplication values. Then the convolution slides over to the next feature and repeats the same process until all the features have been covered.

The stride represents the number of feature shifts over the input matrix. Padding is needed when the filter does not perfectly fit the feature matrix. We have two possibilities: we may add zeros to the features such that the filter will fit, or we may drop the features where the filter did not fit (*valid padding*). The *pooling layer* decreases the computational power required to process the data by reducing the number of parameters when there are too many features. Although there are different kinds of pooling, we used *max pooling*, that takes the largest element from the rectified feature map.

Flattening simply converts the last convolutional layer into a one-dimensional layer that we can then connect to dense one(s). We may repeat this process by stacking by stacking convolutional and pooling layers.

At the end, the flattened output is fed to one or more fully-connected layer(s). In the last layer the network handles classification and outputs the predicted

class since the last fully connected layer computes the probability that the input belongs to each class by exploiting an activation function (the softmax in our case) that provides the probability distribution.

4 Experiments and results

The experiments were run on a free Google Colab instance running Ubuntu that provided a quad-core Intel Xeon E5-2650 v3 that run at 2,30 Mhz and 12GB of RAM. Moreover, it provided also a NVIDIA Tesla K80 as GPU that we used to train the Convolutional Neural Network using `keras` with `Tensorflow` as backend.

Although the dataset provided two lists with filenames for a pre-set split, we needed to make our own random 80%-20% training-test split in order to exploit the `tf.Data` API. The Convolutional Neural Network architecture was built using the famous VGG blocks, that have been proved to be powerful for image classification. One of our VGG blocks is composed by a first `Conv2D` layer, with 32 filters, a kernel size of 3×3 and padding `same`: this means that the input will be evenly padded to the left/right or up/down, so that the output has the same height/width dimension as the input. This layer is followed by a `ReLU` activation function. Then, we stack another same pair of layers, a `Conv2D` and `ReLU` activation, but the convolutional layer does not apply padding. The VGG block finishes with a pooling layer that uses *max pooling* with size 2×2. We stack three of this VGG blocks all together with increasing filters, so that the second block will have 64 filters and the last will have 128 filters. We then flatten the output of the last block and connect it to a fully connected layer with 128 neurons, followed by a `ReLU` activation function. Then, we apply dropout in order to reduce overfitting where possible, since we dedided to drop half of the inputs coming into the dropout layer. Finally, we connect it to an output layer that has a `softmax` activation function (since we are dealing with a multi-class classification problem). We used `adam` as optimizer, `categorical_crossentropy` as loss, 15 epochs and a batch size of 128. We also implemented early stopping in order to reduce overfitting, even though the conditions for it to be triggered were not met and thus the network ran for the whole 15 epochs.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 64, 32)        896
_________________________________________________________________
activation (Activation)      (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 62, 62, 32)        9248
_________________________________________________________________
activation_1 (Activation)    (None, 62, 62, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 31, 31, 64)        18496
_________________________________________________________________
activation_2 (Activation)    (None, 31, 31, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 29, 29, 64)        36928
_________________________________________________________________
activation_3 (Activation)    (None, 29, 29, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 128)       73856
_________________________________________________________________
activation_4 (Activation)    (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 12, 12, 128)       147584
_________________________________________________________________
activation_5 (Activation)    (None, 12, 12, 128)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 4608)              0
_________________________________________________________________
dense (Dense)                (None, 128)               589952
_________________________________________________________________
activation_6 (Activation)    (None, 128)               0
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 774
=================================================================
Total params: 877,734
Trainable params: 877,734
Non-trainable params: 0
```

Fig. 4. Our model

As previously mentioned, some data augmentation techniques were already applied to the images: thus, we only decided to reshape the pictures to a size of 64×64 and scale the RGB values in order to have them in the range $[0, 1]$, which is convenient for a neural network. Moreover, we decided not to convert images to greyscale because banknotes of different values have different colors: thus, RGB values may make an important feature that we will not want to discard.

After 15 epochs, we achieved a training accuracy of 98.16% and a test accuracy of 97.58%. Training took approximately 25 minutes.

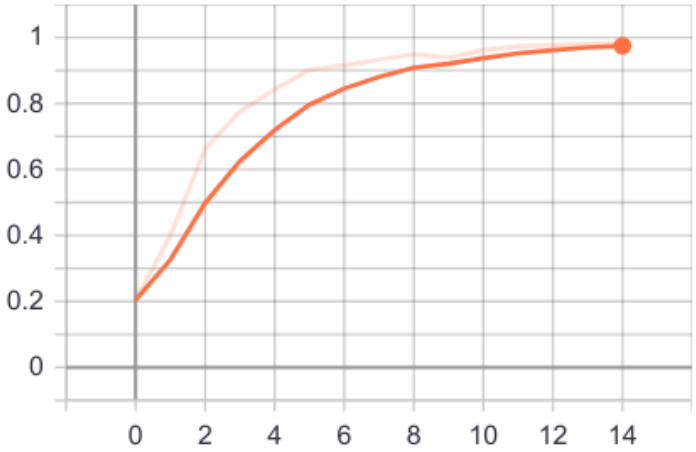

Fig. 5. Training accuracy over the epochs

5 Considerations on scalability

The dataset provided contained just 6000 images and its size was about 3.5Gb: thus, a single consumer hardware could have been able to process and analyze it. However, the main requirement for this project was that the pipeline we where going to build should have been scalable in the case of much larger datasets. We followed the "KISS" principle and for this reason we achieved this goal using only the `keras` API on the `Tensorflow` library with few modifications to the code that could have originally worked in a single machine scenario. The images are loaded through the `tf.Data` API by creating a `tf.Dataset` object. This allow us to introduce the a solution to the scalability of the input: in fact, using the `image_dataset_from_directory` function, we will not load all images into main memory, but we will read them in batches (with size 128) while training the network. It is important to mention that this function needs to point to a directory containing the images, but while we have downloaded them locally,

one could also store them in a distributed file system like HDFS or in a Google Cloud Bucket and then pass the remote path as argument: by doing this, the problem of local storage memory is solved.

We did not want to stop to the scalability of the inputs: we wanted also to allow for distributed training. In order to do this, we decided to rely on the `MultiWorkerMirroredStrategy` included into the experimental `tf.distribute` module. As its name suggests, the `MultiWorkerMirroredStrategy` allows us to distribute training (synchronously) across multiple workers, each with potentially multiple GPUs [4]. This strategy will automatically use all the devices in a single machine, but if we are going to use multiple workers, we need to configure the `TF_CONFIG` file specifying the cluster configuration: this means that, in that JSON file, we will just need to set the IP address of each worker and it's role (that is, *chief* or *worker*) [5]. Therefore, one could spawn a cluster of Google Cloud Compute Instances with GPU using a pre-made template, and then just put their IP address in the `TF_CONFIG` file: then, one could easily train the network on multiple machines in the cloud.
Finally, it is worth mentioning that if one wants to use multiple machines, the input needs to be distributed across each worker: we have also covered this by passing the `tf.Dataset` object into the `experimental_distribute_dataset` function so that it can be automatically distributed across all the devices.

Lastly, we also implemented prefetch and caching of portions of the dataset in order to improve the performance and exploit parallelism by specifing the `num_parallel_calls` argument [6].

References

1. Trevor Hastie, Robert Tibshirani, Jerome Friedman. The Elements of Statistical Learning. Springer Series in Statistics, 2017.
2. Shalev-Shwartz, Ben-David - Understanding Machine Learning. Cambridge University Press, 2014.
3. Wenpeng Yin, Katharina Kann, Mo Yu, Hinrich Schütze. Comparative Study of CNN and RNN for Natural Language Processing. 2017.
4. https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/MultiWorkerMirroredStrategy
5. https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras
6. https://www.tensorflow.org/guide/data_performance

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.