INT248

ADVANCED MACHINE LEARNING

Project report-

Image classification using Tensor Flow

Submitted to:

Md. Imran Hussain sir

By:

Robin

11814491

KMO72

Introduction

1.Image classification

Image classification is where a computer can analyse an image and identify the 'class' the image falls under. (Or a probability of the image being part of a 'class'.) A class is essentially a label, for instance, 'car', 'animal', 'building' and so on.

For example, you input an image of a sheep. Image classification is the process of the computer analysing the image and telling you it's a sheep. (Or the probability that it's a sheep.)

Early image classification relied on raw pixel data. This meant that computers would break down images into individual pixels. The problem is that two pictures of the same thing can look very different. They can have different backgrounds, angles, poses, etcetera. This made it quite the challenge for computers to correctly 'see' and categorise images.

Image classification has a few uses — and vast potential as it grows in reliability. Here are just a few examples of what makes it useful.

Self-driving cars use image classification to identify what's around them. I.e. trees, people, traffic lights and so on.

2. Tensor Flow

Tensor flow is an open-source library for numerical computation and large-scale machine learning that ease *Google Brain TensorFlow*, the process of acquiring data, training models, serving predictions, and refining future results.

Tensor flow bundles together Machine Learning and Deep Learning models and algorithms. It uses Python as a convenient front-end and runs it efficiently in optimized C++.

Tensor flow allows developers to create a graph of computations to perform. Each node in the graph represents a mathematical operation and each connection represents data. Hence, instead of dealing with low-details like figuring out proper ways to hitch the output of one function to the input of another, the developer can focus on the overall logic of the application.

A tensor is a vector/matrix of n-dimensions representing types of data. Values in a tensor hold identical data types with a known shape. This shape is the dimensionality of the matrix. A vector is a one-dimensional tensor; matrix a two-dimensional tensor. Obviously, a scalar is a zero dimensional tensor.

```
# a rank 0 tensor, i.e. a scalar with shape ():
42
# a rank 1 tensor, i.e. a vector with shape (3,):
[1, 2, 3]

# a rank 2 tensor, i.e. a matrix with shape (2, 3):
[[1, 2, 3], [3, 2, 1]]
# a rank 3 tensor with shape (2, 2, 2) :
[ [[3, 4], [1, 2]], [[3, 5], [8, 9]]]
```

Dataset used

The MNIST dataset is one of the most common datasets used for image classification and accessible from many different sources.

In fact, even Tensorflow and Keras allow us to import and download the MNIST dataset directly from their API.

Therefore, I will start with the following two lines to import TensorFlow and MNIST dataset under the Keras API.

```
Import tensorflow as tf

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

The MNIST database contains 60,000 training images and 10,000 testing images taken from American Census Bureau employees and American high school student.

Therefore, in the second line, I have separated these two groups as train and test and also separated the labels and the images. x_train and x_test parts contain greyscale RGB codes (from 0 to 255) while y_train and y_test parts contain labels from 0 to 9 which represents which number they actually are.

To visualize these numbers, we can get help from matplotlib.

You will get (60000, 28, 28). As you might have guessed 60000 represents the number of images in the train dataset and (28, 28) represents the size of the image: 28 x 28 pixel.

Proposed Archiecture

Convolutional Neural Networks

The main structural feature of Regular Nets is that all the neurons are connected to each other. For example, when we have images with 28 by 28 pixels in greyscale, we will end up having 784 (28 x 28 x 1) neurons in a layer that seems manageable. However, most images have way more pixels and they are not grey-scaled. Therefore, assuming that we have a set of color images in 4K Ultra HD, we will have 26,542,080 (4096 x 2160 x 3) different neurons connected to each other in the first layer which is not really manageable.

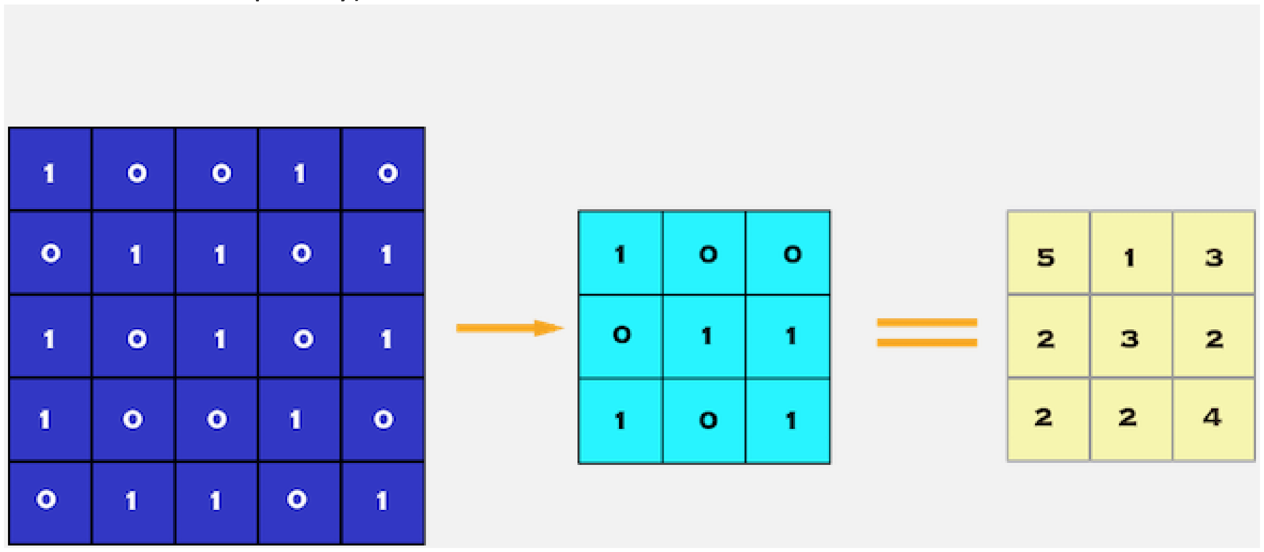
Therefore, we can say that Regular Nets are not scalable for image classification. However, especially when it comes to images, there seems to be little correlation or relation between two individual pixels unless they are close to each other. This leads to the idea of Convolutional Layers and Pooling Layers.

Layers in a CNN

We are capable of using many different layers in a convolutional neural network. However, convolution, pooling, and fully connected layers are the most important ones. Therefore, I will quickly introduce these layers before implementing them.

Convolutional Layers

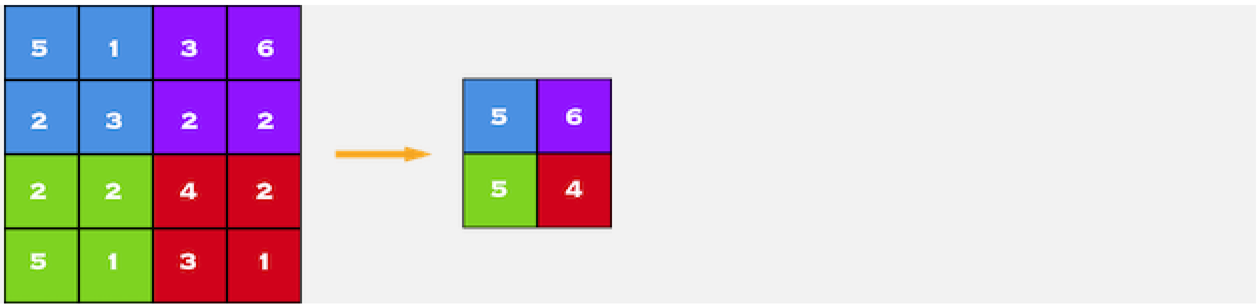
The convolutional layer is the very first layer where we extract features from the images in our datasets. Due to the fact that pixels are only related to the adjacent and close pixels, convolution allows us to preserve the relationship between different parts of an image. Convolution is basically filtering the image with a smaller pixel filter to decrease the size of the image without losing the relationship between pixels. When we apply convolution to 5x5 image by using a 3x3 filter with 1x1 stride (1-pixel shift at each step). We will end up having a 3x3 output (64% decrease in complexity).



Convolution of 5 x 5 pixel image with 3 x 3 pixel filter (stride = 1 x 1 pixel)

Pooling Layer

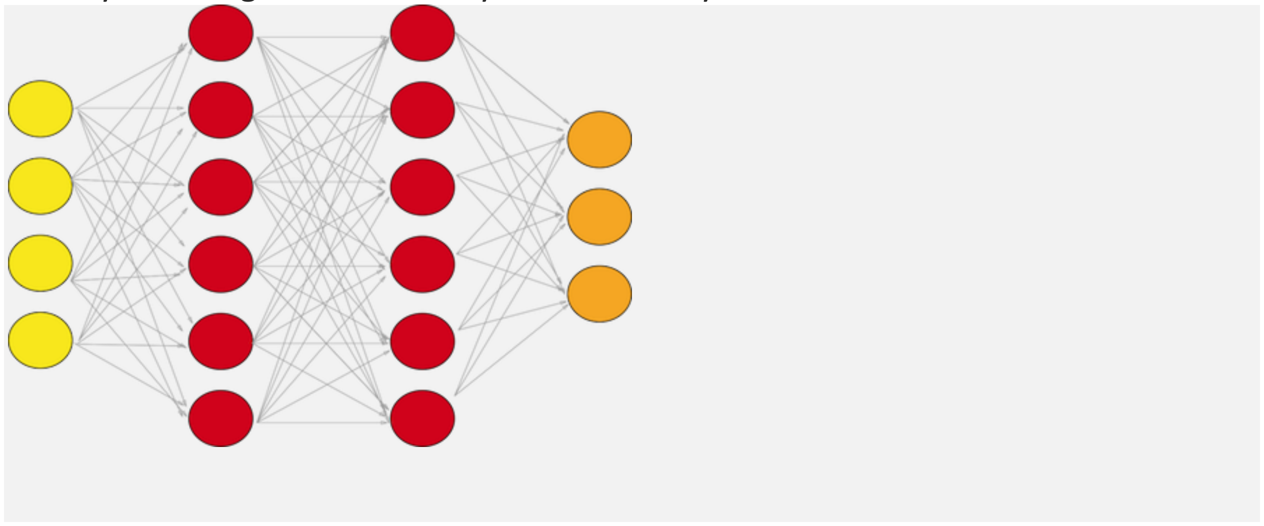
When constructing CNNs, it is common to insert pooling layers after each convolution layer to reduce the spatial size of the representation to reduce the parameter counts which reduces the computational complexity. In addition, pooling layers also helps with the overfitting problem. Basically, we select a pooling size to reduce the amount of the parameters by selecting the maximum, average, or sum values inside these pixels. Max Pooling, one of the most common pooling techniques, may be demonstrated as follows:



Max Pooling by 2 x 2

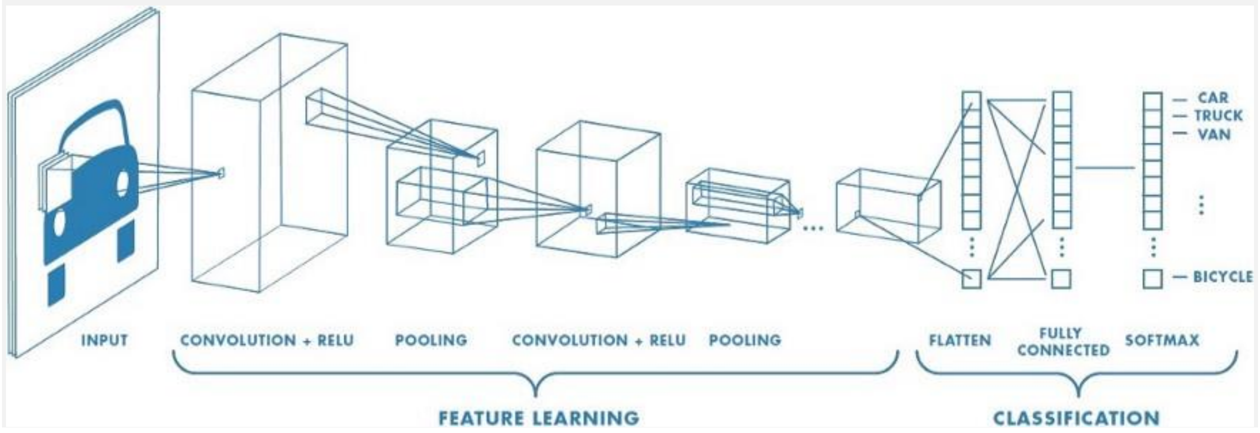
A Set of Fully Connected Layers

A fully connected network is our Regular Net where each parameter is linked to one another to determine the true relation and effect of each parameter on the labels. Since our time-space complexity is vastly reduced thanks to convolution and pooling layers, we can construct a fully connected network in the end to classify our images. A set of fully-connected layers look like this:



A fully connected layer with two hidden layers

Now that you have some idea about the individual layers that we will use, I think it is time to share an overview look of a complete convolutional neural network.

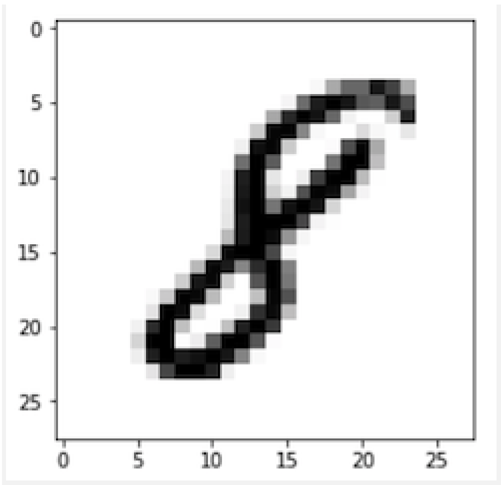


And now that you have an idea about how to build a convolutional neural network that you can build for image classification, we can get the most cliche dataset for classification: the MNIST dataset, which stands for Modified National Institute of Standards and Technology database. It is a large database of handwritten digits that is commonly used for training various image processing systems.

Experiment Analysis and results

Downloading the MNIST Dataset

The MNIST dataset is one of the most common datasets used for image classification and accessible from many different sources. In fact, even Tensor flow and Keras allow us to import and download the MNIST dataset directly from their API. Therefore, I will start with the following two lines to import TensorFlow and MNIST dataset under the Keras API.



A visualization of the sample image at index 7777

Reshaping and Normalizing the Images

To be able to use the dataset in Keras API, we need 4-dims NumPy arrays. However, as we see above, our array is 3-dims. In addition, we must normalize our data as it is always required in neural network models. We can achieve this by dividing the RGB codes to 255.

Reshaping:

```
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)
```

Normalizing

```
x_train /= 255
x_test /= 255
```

Building the Convolutional Neural Network

We will build our model by using high-level Keras API which uses either TensorFlow or Theano on the backend. I would like to mention that there are several high-level TensorFlow APIs such as Layers, Keras, and Estimators which helps us create neural networks with high-level knowledge. However, this may lead to confusion since they all vary in their implementation structure. Therefore, if you see completely different codes for the same neural network although they all use TensorFlow, this is why. I will use the most straightforward API which is

Keras. Therefore, I will import the Sequential Model from Keras and add Conv2D, MaxPooling, Flatten, Dropout, and Dense layers. I have already talked about Conv2D, Maxpooling, and Dense layers. In addition, Dropout layers fight with the overfitting by disregarding some of the neurons while training while Flatten layers flatten 2D arrays to 1D arrays before building the fully connected layers.

```
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten,
MaxPooling2D
# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation=tf.nn.relu))
model.add(Dropout(0.2))
model.add(Dense(10,activation=tf.nn.softmax))
```

Compiling and Fitting the Model

With the above code, we created a non-optimized empty CNN. Now it is time to set an optimizer with a given loss function that uses a metric. Then, we can fit the model by using our train data. We will use the following code for these tasks:

```
Model.compile(optimizer='adam',
loss="sparse_categorical_crossentropy",metrics=["accuracy"])

Model.fit(x=x_train,y=y_train,epochs=10)
```

Evaluating the Model

Finally, you may evaluate the trained model with x_test and y_test using one line of code:

```
model.evaluate(x_test, y_test)
```

The results are pretty good for 10 epochs and for such a simple model.

Conclusion and future Scope

CNN is a powerful algorithm for image processing. These algorithms are currently the best algorithms we have for the automated processing of images. Many companies use these algorithms to do things like identifying the objects in an image.

Images contain data of RGB combination. Matplotlib can be used to import an image into memory from a file. The computer doesn’t see an image, all it sees is an array of numbers. Color images are stored in 3-dimensional arrays. The first two dimensions correspond to the height and width of the image (the number of pixels). The last dimension corresponds to the red, green, and blue colors present in each pixel.

Convolutional Neural Networks, or CNNs, were designed **to map image data to an output variable**. They have proven so effective that they are the go-to method for any type of prediction problem involving image data as an input.
This data sets used both and training and testing purpose using CNN. It provides the accuracy rate **98%**. Images used in the training purpose are small and Grayscale images. ... The future enhancement will focus on classifying the colored images of large size and its very useful for image segmentation process

References

The dataset MNIST was taken from official tensorflow website, that is – www.tensorflow.org/datasets.

TensorFlow Datasets is a collection of datasets ready to use, with TensorFlow or other Python ML frameworks, such as Jax. All datasets are exposed as tf.data.Datasets , enabling easy-to-use and high-performance input pipelines

Also I have taken references from online websites like:

www.towardsdatascience.org

www.analyticsvidhya.com

THANK YOU