UNIVERSITÀ DEGLI STUDI
DI MILANO

Algorithms for massive datasets

Prof: Dario Malchiodi

Student: Pramodh Chinthareddy (Data Science and Economics)

Matriculation Number: 936461

Table of Contents:

1. Turkish Lira Dataset:

This dataset is collected to develop applications for visually impaired people.

Content

- Each banknote category is divided into folders.
- Each folder contains 1000 images. 925 for training, 75 for validation.
- Some data augmentation techniques applied to images (increase brightness, decrease brightness, flip, add salt&pepper noise).
- Image Shape: (1280, 720, 3)

Below are a few snippets of the different types of labelled images:

TÜRKİYE CUMHURİYET MERKEZ BANKASI
50
ELLİ
TÜRK LİRASI

CUMHURİYET MERKEZ BANKASI
100
YÜZ
TÜRK LİRASI

CUMHURİYET MERKEZ BANKASI
200
İKİYÜZ
TÜRK LİRASI

2. Data Organizing:

Scalability in reading data: We won't load all images into main memory, but we use **tf.Data API** in order to read data in small batches. In order to do this, we won't use the train/test split, but we will randomly split the data: 80% for training and 20% for testing.

Further rescaling the data (RGB to Grayscale) is done in real-time while batch reading the images from disk. We took a batch size of 64 per worker.

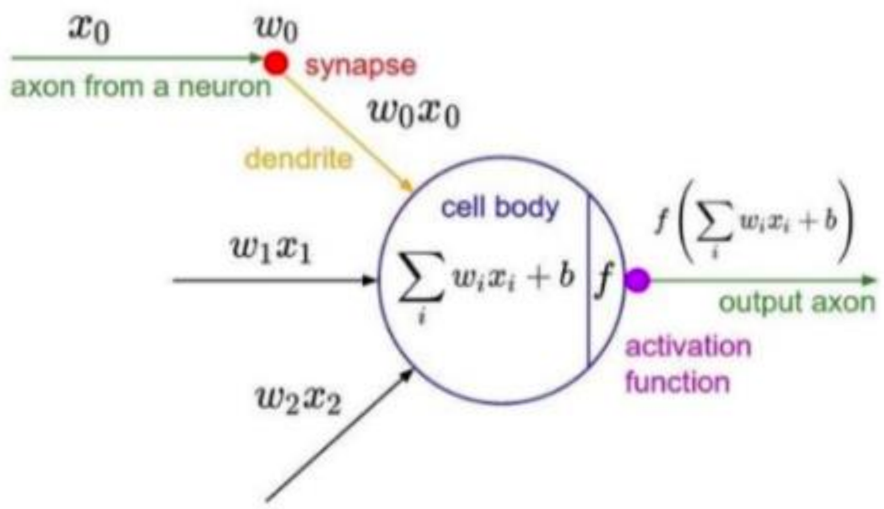
Where the **Global batch size = Per Worker Batch Size * Number of Workers**

3. Preprocessing:

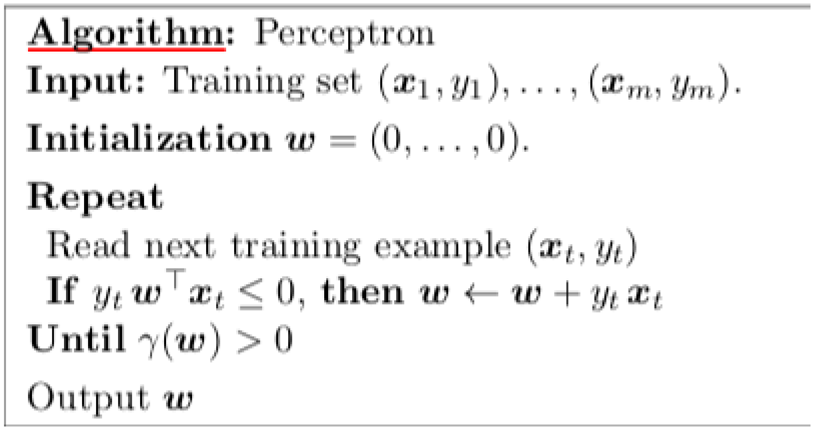
We resized each image to 64*64 (height*width) from 1280*720 to reduce the memory usage. We also converted the colored images to gray scale (in the range of [0,1]) suitable for a Neural network architecture.

4. CNN and its Implementation:

Neural networks are parametric algorithms that train on a predetermined set of variables (unlike a k-NN or Tree Predictors). Neural Networks are linear classifiers that try to differentiate the different data points using Separating Hyperplanes (In linearly separable case).



Neural Networks train with help of a Convex loss function where the goal is to find the local minima and descent in the negative direction of the gradient proportional to the gradient (A commonly known technique called Gradient descent – which happens inside the Backpropagation algorithm).



```
Algorithm: Perceptron
Input: Training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)$.
Initialization $\boldsymbol{w} = (0, \ldots, 0)$.
Repeat
  Read next training example $(\boldsymbol{x}_t, y_t)$
  If $y_t \boldsymbol{w}^\top \boldsymbol{x}_t \leq 0$, then $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_t \boldsymbol{x}_t$
Until $\gamma(\boldsymbol{w}) > 0$
Output $\boldsymbol{w}$
```

The Perceptron algorithm finds a homogeneous separating hyperplane by running through the training examples one after the other. The current linear classifier is tested on each training example and, in case of misclassification, the associated hyperplane is adjusted.

If the algorithm terminates, then w is a separating hyperplane. Perceptron always terminates on linearly separable training sets.

The Perceptron algorithm accesses training data in a sequential fashion, processing each training example in linear time, making Perceptron very competent on large training data sets. It is also very good at dealing scenarios in which new training data are generated all the time.

```
**Parameters:** Class $\mathcal{H}$ of predictors, loss function $\ell$.
The algorithm outputs a default initial predictor $h_1$
For $t = 1, 2, \ldots$
    1. The next example $(\boldsymbol{x}_t, y_t)$ is observed
    2. The loss $\ell\big(h_t(\boldsymbol{x}_t), y_t\big)$ of the current predictor $h_t$ is computed
    3. The online learner updates $h_t$ generating a new predictor $h_{t+1}$
```

Model update h(t) --> h(t+1) is typically local.

```
**Projected OGD**
Parameters: $\eta > 0$, $U > 0$
Initialization: $\boldsymbol{w}_1 = \boldsymbol{0}$
For $t = 1, 2, \ldots$
    1. $\boldsymbol{w}'_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{t}} \nabla \ell_t(\boldsymbol{w}_t)$
    2. $\boldsymbol{w}_{t+1} = \operatorname*{argmin}_{\boldsymbol{w} : \|\boldsymbol{w}\| \le U} \big\| \boldsymbol{w} - \boldsymbol{w}'_{t+1} \big\|$
```

Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. They have a loss function (e.g. SoftMax) on the last (fully-connected) layer.

ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**.

A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.

Every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer**. We will stack these layers to form a full ConvNet **architecture**.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

In summary:

- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

Convolutional Layer:

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

ReLU layer:

ReLU is the abbreviation of **rectified linear unit**, which applies the non-saturating activation function

f(x) = max (0, x), It effectively removes negative values from an activation map by setting them to zero. t increases the **nonlinear properties** of the **decision function** and of the overall network without affecting the receptive fields of the convolution layer.

ReLU is often preferred to other functions because it trains the neural network several times faster without a significant penalty to **generalization** accuracy.

Pooling Layer:

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice).

Fully-connected layer:

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

Number of parameters in the network: Input * Output + Biases

Dropout:

Dropout as a Regularization method, helps in training a complex model without additional computational expense of training and maintaining multiple models, by randomly dropping out nodes during training.

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by **1 / (1 - rate)** such that the sum over all inputs is unchanged.

Adaptive Gradient Descent:

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. we use gradient descent to update the parameters of our model.

Ada delta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address two drawbacks:

- The continual decay of learning rates throughout training
- The need for a manually selected global learning rate
- Ada delta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.
- **Adam** optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments, Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

Batch Normalization:

A powerful technique that reshapes the distribution of activations (making them close to Normal Distribution), which improves the performance and stability of the network.

Batch Norm = (Activation – Mean Activation)/S.D

With Mean Activation close to 0 and Active Standard Deviation of the distribution close to 1.

5. Distributed Scaling:

We used Tensorflow's **MultiWorkerMirroredStrategy** (tf.distribute.Strategy) to achieve distributed synchronized training across multiple machines with multiple GPUs.

In tensorflow configure the TF_CONFIG file like

```
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': ["localhost:20000", "localhost:
20001"]
    },
    'task': {'type': 'worker', 'index': 0}
})
```

and replace "localhost:20000" and "localhost:20001" with the **IP addresses** of your workers.

There are two components of TF_CONFIG: cluster and task.

- cluster is the same for all workers and provides information about the training cluster, which is a dict consisting of different types of jobs such as worker. In multi-worker training with MultiWorkerMirroredStrategy, there is usually one worker that takes on a little more responsibility like saving checkpoint and writing summary file for TensorBoard in addition to what a regular worker does. Such a worker is referred to as the chief worker, and it is customary that the worker with index 0 is appointed as the chief worker (in fact this is how tf.distribute.Strategy is implemented).
- task provides information of the current task and is different on each worker. It specifies the type and index of that worker.

MultiWorkerMirroredStrategy provides multiple implementations via the CollectiveCommunication parameter.

With the integration of tf.distribute.Strategy API into tf.keras, the only change you will make to distribute the training to multiple-workers is enclosing the model building and model.compile() call inside strategy.scope(). The distribution strategy's scope dictates how and where the variables are created, and in the case of MultiWorkerMirroredStrategy, the variables created are MirroredVariables, and they are replicated on each of the workers.

Better performance with the tf.data API:

GPUs and TPUs can radically reduce the time required to execute a single training step. Achieving peak performance requires an efficient input pipeline that delivers data for the next step before the current step has finished. The tf.data API helps to build flexible and efficient input pipelines.

We implemented **prefetch** and **caching** of portions of the dataset in order to improve performance.

Prefetching overlaps the preprocessing and model execution of a training step. While the model is executing training step s, the input pipeline is reading the data for step s+1. Doing so reduces the step time to the maximum (as opposed to the sum) of the training and the time it takes to extract the data.

The number of elements to prefetch should be equal to (or possibly greater than) the number of batches consumed by a single training step. You could either manually tune this value, or set it to tf.data.experimental.AUTOTUNE which will prompt the tf.data runtime to tune the value dynamically at runtime.

The tf.data.Dataset.cache transformation can cache a dataset, either in memory or on local storage. This will save some operations (like file opening and data reading) from being executed during each epoch.

When you cache a dataset, the transformations before the cache one (like the file opening and data reading) are executed only during the first epoch. The next epochs will reuse the data cached by the cache transformation.

Distributed Input:

We now distribute the input across multiple devices (tf.distribute.Strategy.experimental_distribute_dataset). The tf.distribute APIs provide an easy way for users to scale their training from a single machine to multiple machines. When scaling their model, users also have to distribute their input across multiple devices. tf.distribute provides APIs using which you can automatically distribute your input across devices.

6. Description of Experiments:

We trained 4 different models (VGG network (https://d2l.ai/chapter_convolutional-modern/vgg.html), VGG network with Batch Normalization, A Simpler network, Simpler Network with Batch Normalization)

In all the experiments, we used Adaptive optimization for learning the weights and biases, ReLu activation for all the layers except the final Classification layer where we used Softmax

SoftMax activation function for Classification because it creates an exponential probability distribution in the range of (0,1) making it easy for classification based on the activation with highest probability.

Loss function used for all the experiments: Categorical Cross Entropy, which makes the labels mutually exclusive for each training data point making it suitable for our **Single-label Classification**

On Test Set:

Our VGG network achieved an accuracy of 96.25%

VGG with Batch Normalization achieved an accuracy of 92.00%

Our Simple network achieved an accuracy of 96.83%

Simple network with Batch Normalization achieved an accuracy of **97.33%**

7. Final Comments:

While we used Adaptive gradients as optimizer in all the networks, Stochastic gradient descent (SGD) takes longer training time.

We conclude that a Simpler network with Batch Normalization trains faster (in 12 epochs) and also achieves the better accuracy for this dataset. That is the labels are predicted more accurately when images are normalized and fed to a Simple network without using any Regularization method.

8. References:

[1] https://www.tensorflow.org/tutorials/distribute/multi_worker_with_keras

[2] https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/MultiWorkerMirroredStrategy

[3] https://www.tensorflow.org/guide/data_performance

[4] https://www.tensorflow.org/tutorials/distribute/input

[5] http://cs231n.github.io/convolutional-networks/

[6] https://en.wikipedia.org/wiki/Convolutional_neural_network

[7] http://cesa-bianchi.di.unimi.it/MSA/Notes/linear.pdf

[8] http://cesa-bianchi.di.unimi.it/MSA/Notes/online.pdf

[9] https://www.kaggle.com/baltacifatih/turkish-lira-banknote-dataset