Sapienza University of Rome

Neural Networks Final Project
Classifying HEM vs ALL cells

Jean-Pierre Richa

June 20, 2019

1 Introduction

Neural networks (NNs) have reshaped the way computers can handle data and understand its content. Ever since they were first introduced, NNs were used in a wide variety of applications, such as *Natural Language Processing (NLP), Computer Vision (CV) and many others*. Convolutional Neural Networks (CNNs), inspired by multi-layer perceptrons and convolutional layers, have proven to be more accurate and robust than other networks architectures when used to handle tasks related to Computer Vision and are being widely used in this area of research.

The target behind this project is to train a CNN to classify HEM (Human Epidermal Melanocyte) cells, which are normal cells, vs ALL (Acute lymphoblastic leukaemia, malignant) cells, which is a type of cancer that affects the white blood cells.

To realize this task, a CNN had to be built from scratch, trying to achieve the highest classification accuracy. This is a CodaLab global competition, but was used only for the university course project. The dataset used for the training was given by CodaLab on their website. The images were transformed into tfrecords, which makes it easier to shuffle the dataset, reduce its size hence to reduce the training complexity, then the images read from the tfrecords were fed into the network.

The Project was developed in python and trained locally on a MacBook pro.

2 Dataset Preparation and Augmentation

The dataset given by CodaLab contains 3 folds, each containing a HEM folder and an ALL folder, where the images were named according to their ID containing whether the image belongs to an ALL, or HEM cell. In order to easily work with the images, they were first read and included into a list of lists, which contained each image path and its corresponding label's binary class. Since the original dataset was not large enough to train the network and have good predictions, the data had to be augmented. In order to augment the dataset, the images had to be handled in a way that reserves the important features, so rotating the image was a good approach to use for the augmentation step, where the images were rotated by 90 degrees. This approach was followed, because CodaLab team had already pre-processed the images, so there weren't much things that can be done on the final dataset given by them. Another problem was the sparsity of the HEM dataset, having approximately half the number of images with respect to the ALL cells set. Some images from the ALL images were neglected during the training in order to have a balanced dataset. After the augmentation step, the final dataset contained more images than the original one, which will also help in preventing the network from overfitting, hence to be able to train and generalize to other unseen samples. After having the final list of images and their corresponding labels, it was split into 2 lists, where the first contained 80% of the original one that was later used for training, and the second one contained the other 20%, that was used for testing. using the 2 final lists, the images where then converted into tfrecords format, so the dataset size can be reduced and data manipulation would be easier, while resulting in a faster training with less computational and time complexity. Following the above mentioned steps, the final dataset contained 13,833 images, instead of originally 10,670 images, with 11,056 images for the training set and 2,764 for the test set.

3 Network Training

Several trainings were done, testing different hyper-parameters and architectures.

The network was trained using different architectures and hyper-parameters until the best configuration was found. The training was done on a macbook pro without GPU acceleration.

After each epoch, the updated weights were used to get the accuracy of the prediction made on the training set and the test set, achieving an accuracy of 87.1% Fig **1** and 84.3% Fig **3** respectively.

Two different loss functions were used to train the network and calculate the loss:

- **tf.nn.l2_loss with weight decay:** here the L2 loss function was used for regularization and its mathematical representation can be seen **below**

$$L2LossFunction = \sum_{i=1}^{N}(y_i true - y_i predict)^2 \tag{1}$$

- **tf.nn.sparse_softmax_cross_entropy_with_logits:** the Cross Entropy loss function, which was used for calculating the loss on the output layer, then its gradient was calculated and back propagated through the network to update the weights. Its mathematical representation can be seen **below**.

$$H(y, \hat{y}) = -\sum_{i=1}^{N}(y_i \ log \ \hat{y_i}) \tag{2}$$

A summary that included the accuracy of both sets was created and updated through the training, which can be visualized using Tensorboard.

4 Network Implementation

The accuracy was tested on several architectures and hyper-parameter, until the final most accurate ones were chosen to train the model for the classification.

4.1 CNN Architecture

The created network consists of 5 convolutional (conv) layers, 5 maxpooling layers, and two fully connected (dense) layers. The size of the input images is 256x256x3 (image width, image height, and RGB channels respectively). The images are fed into the first conv layer, 32 feature detectors are then applied, with a size of 5x5, after that a maxpooling layer is applied on the first conv output, which results in a 128x128x64 (image width, image height, and number of filter, or features detectors, respectively) tensor. The input for the second conv layer is the first conv layer's output, which is 128x128x64, in the second conv layer, the same steps are applied, which results in a 64x64x128 tensor, that is fed to the third conv layer and so on, till the fifth conv layer, which will give as an output an 8x8x512 tensor. The output from the last convolutional layer will then be flattened and fed to the first dense layer, which has 1024 nodes, and is connected to the second dense layer, which also has 1024 nodes. The output from the second dense layer will be the input to the output layer which will give out the binary class for each image.

4.2 Network Hyper-parameters

After testing different architectures and hyper-parameters, the following gave the best results for this task:

4.2.1 Learning Rate

The learning rate tells the optimizer being used how far to move the weights in the direction opposite to the gradient of the error that is calculated at the output layer (i.e., how fast should the network forget about the its old beliefs and use the new ones instead). In this network a learning rate of 0.001 was used, this small learning rate was chosen, so the weights update don't overshoot the minimum.

4.2.2 Number of Epochs

The network was trained for 14 epochs, because the training was done on a regular laptop without GPU acceleration. The training can be done for more epochs without losing the pre-trained model thanks to the possibility of loading the previously saved weights from the checkpoints that I was saving.

4.2.3 Batch Size

A very well known trade-off between computational complexity, memory limitations and training accuracy, limits the options for choosing the batch size that should be used during the training process. Due to the limited resources on a regular computer and high computational complexity of the training, a batch size of 256 was chosen to train the network, which resulted in good accuracy on the training and test sets, but of course a smaller batch would improve the accuracy by a big factor.

4.2.4 Activation Functions

Due to the high non linearity in the images, the choice of the activation function should be first to break the linearity of the model. Another thing to consider during the training is the vanishing gradient, which can occur due to the propagated gradient calculation resulting in a very small gradient that eventually stops the network from learning due to the small gradient that tells the network not to update its weights. This problem can be solved by using the RE(ctified) L(inear) U(nit) activation function which maps the negative inputs to zero, having a gradient of zero and positive inputs to their same value, to which the gradient is one, preventing the gradient from vanishing after many gradient calculations.

4.2.5 weights initialization

One of the most important variables to take into consideration while training a neural network are the weights, since the training process and execution time highly depends on their initialization. If the weights were initialized with a high value, or a low value, we might face an exploding, or vanishing gradient, especially when dealing with large architectures, so they should be initialized in a manner that keeps the variance in a specific range, which will prevent us from facing these issues. For these reasons, xavier weights initialization [1] was used in this implementation, which initializes the weights with a small range variance on the different network layers.

4.2.6 dropout

A very important issue that should be accounted for is the overfitting of the model, which means that the model performs very well on the training set, but is not able to generalize to new unseen data. This can be prevented using dropout [3], which prunes nodes randomly from the network layers, keeping them from memorizing the values of the training set, driving the weights to be updated until the model is able to perform well on the testing set also.

4.2.7 Weight decay

In order to further prevent overfitting and be able to generalize more to unseen data, regularization [2] was performed using a weight decay of 0.0005, which penalizes large weights and keeps them from growing too much and this consequently prevents the update from overshooting the local minimum in the loss function. The weights are penalized with respect to their values (the larger the weights, the greater the penalization), if the weight is equal to 0 then we end up with the original L2 loss function and the weights will not get penalized. This approach keeps the weights' values small, which keeps the model from listening to particular weights and hence reduces the bias of the model. This regularization decreases the likelihood of network overfitting, which raises the accuracy of the predictions.

4.3 Implementation

The project was implemented in Python using Tensorflow. Below is the list of Python files implemented for the project with a brief description about each one.

- **create_TfRecords.py:** Generating the tfrecord files used for the training and evaluation of the network.
- **augmentSet.py:** A script used by createTfRecords.py that takes the split sets (training and testing) and returns 2 augmented final sets that will be converted to tfrecords.
- **read_TfRecords.py:** Used during the training to read the records and feed the images and labels in batches to the placeholders using a serialized example.
- **train.py:** Includes the network architecture, variables, graph, session and training section of the network.
- **config.py:** Contains all of the network hyper-parameters and other common variables.

5 Results

The network was trained for 14 epochs and it was enough to see how the accuracy got better with each epoch passed. When the network was trained for more epochs, it was overfitting and this can be fixed by reducing the batch size, increasing the input image size, and training for more epochs. In other trainings it was also obvious that the smaller the input images were, the highest was the likelihood for model overfitting, when the input image size was increased, the accuracy of the model also increased and overfitting took more epochs to occur.

After each epoch the accuracy of the model was saved using the summary collector, this way the accuracy can be checked using Tensorboard, which also gives the chance to extract the plot that is easily visualized **below**.

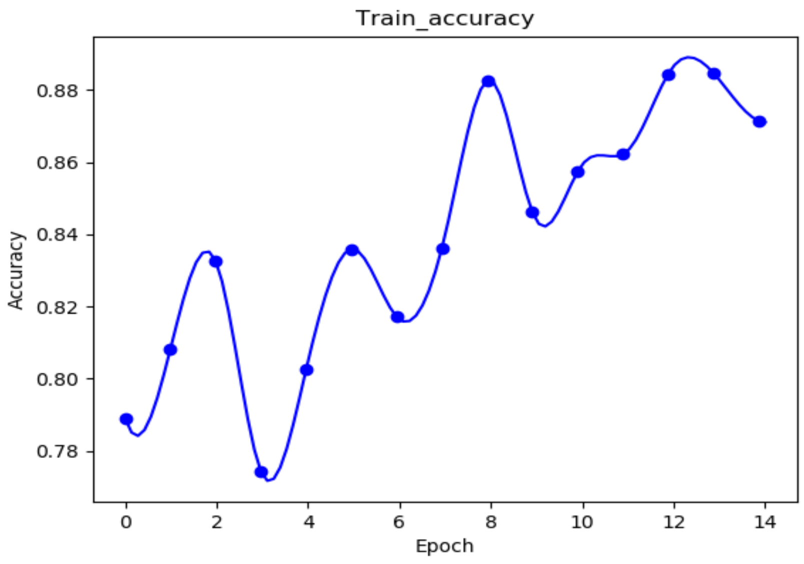

Figure 1: Train acc = 0.871, epochs = 14

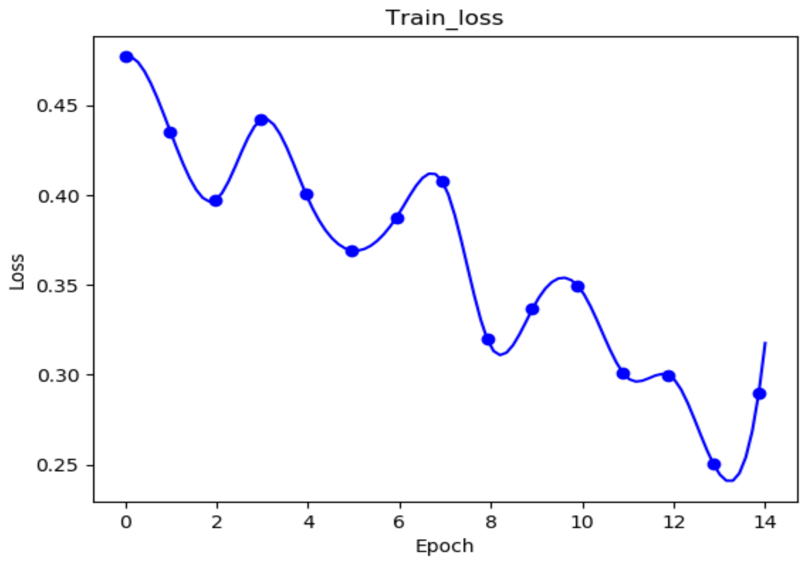

Figure 2: Train loss = 0.24, epochs = 14

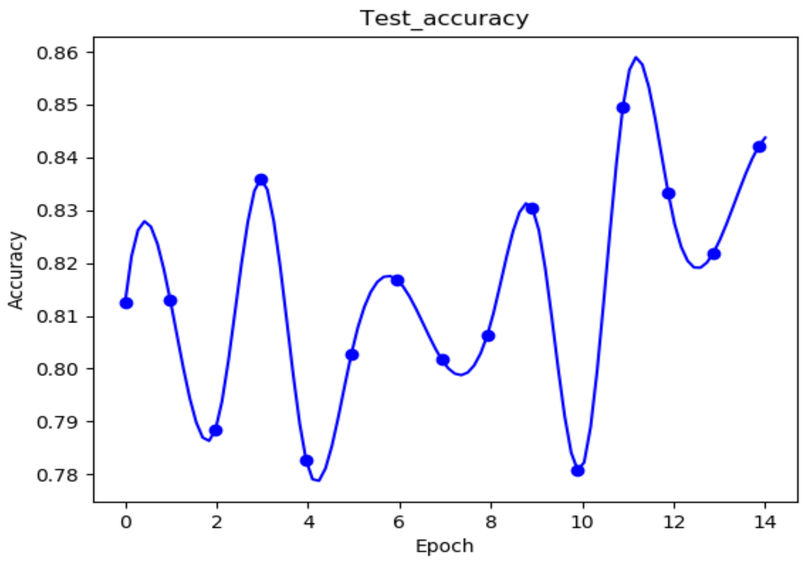

Figure 3: Test acc = 0.843, epochs = 14

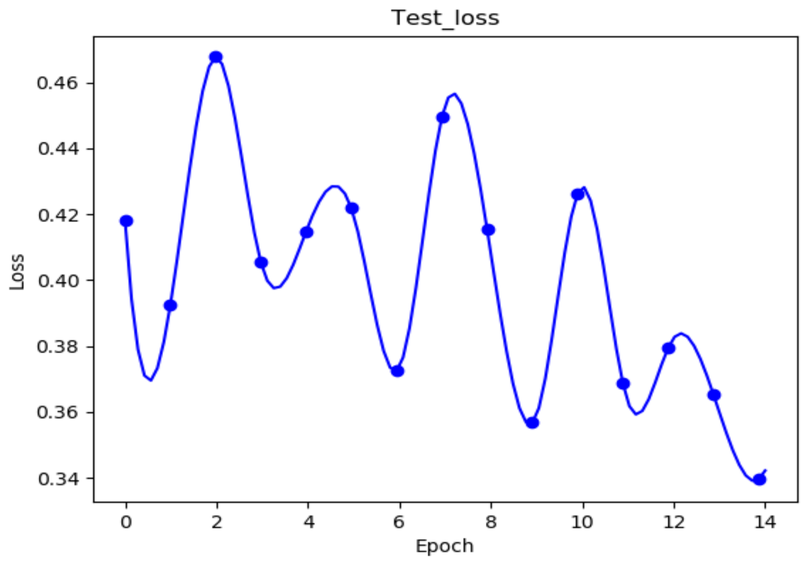

Figure 4: Test loss = 0.34, epochs = 14

Hyperparameters							
Image size	Epochs	LR	Weight decay	Train acc	Test acc	Train loss	Test loss
32	22	0.001	0.0005	0.56	0.53	0.69	0.61
64	40	0.001	0.0005	1.0	0.82	0.006	0.99
256	**14**	**0.001**	**0.0005**	**87.1%**	**84.3%**	**0.24**	**0.34**

Table 1: Few training tests showing how increasing the size of the input image, increased the accuracy of the model and reduced overfitting

6 Conclusion

As already shown in the report, the accuracy achieved on the training set was 86.5% and 85% on the testing set, which is a good accuracy given the large batch size. This accuracy also shows that the steps taken to prevent overfitting worked as were supposed to, knowing that it is a hard task to distinguish between the 2 classes. The training set was not enough, so it is also shown from the results that the augmentation played a very big role in providing enough samples for the network to train and generalize to other sets.

7 Future work

The improvement in the accuracy over the epochs was great and a better accuracy can be achieved by reducing the batch size and increasing the number of epochs. Another important step that I would like to perform is including new dataset taken from different sources, so I can expand the dataset even more, and achieve a better accuracy.

References

[1] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS 10). Society for Artificial Intelligence and Statistics.* 2010.

[2] Twan van Laarhoven. "L2 Regularization versus Batch and Weight Normalization". In: *CoRR* abs/1706.05350 (2017). arXiv: `1706.05350`. URL: `http://arxiv.org/abs/1706.05350`.

[3] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958. URL: `http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`.