Deep Learning Assignment
2AMM10
Technische Universiteit Eindhoven

Panagiotis Banos (1622773)

1 Task 1-Recognize characters from alphabets from around the world

Figure 1: A sample of Omniglot dataset

For the first task of the assignment, we used the Omniglot dataset to build a model for character recognition. This dataset contains 20 images of 1600 handwritten characters from 50 different alphabets. The training dataset includes 18800 images in the shape of 28*28 of 893 handwritten characters from 29 different alphabets.

The query dataset contains 100 sets of 6 images in the size of (28,28) of handwritten characters, not included in the training data. Each of these 100 sets has 1 query image and 5 candidate images organized in two arrays with shapes (100,1,28,28) and (100,5,28,28), respectively. The goal of the first task is to identify the closest image from the set of candidates to the query image and compute the top-1 and top-3 performance of the model.

1.1 Problem formulation and proposed approach

The problem is finding a way to compare images that are not present during the model's training. What we need to solve is the problem of building a model that can learn to learn, which is called meta-learning. As we have several hundreds of classes with few samples for each of them, we used a few shot learning methods to maximize the model's learning, and one of the famous meta-learning models is Siamese Network. In Siamese Networks, two or more networks use shared weights to learn similarity and dissimilarity between images. We used a model architecture inspired by [1] to extract the embeddings in our model.

We used a balanced batch sampler to create a minibatch that contains N classes and M samples for each class. To make the model learn efficiently, we are going to use triplet loss. Using triplet loss, our model can map two similar images close to one another and far from the dissimilar sample image. So, we have the anchor image (the sample image), positive image (another variation of the anchor image), and negative image (a different image from above two similar images). The positive image helps the model learn the similarities between two images, and the negative image helps our model learn dissimilarities with the sample image. We defined a margin that can increase the distance between similar and dissimilar output vectors. By controlling this hyperparameter, we can map similar images close to one another and different images far from each other. So, our loss is defined as follows:

$$L_{triplet}(x_a, x_p, x_n) = max(0, m + \|f(x_a) - f(x_p)\|_2^2 - \|f(x_a) - f(x_n)\|_2^2) \quad (1)$$

We used informative triplet loss rather than random triplet loss. A random negative example is selected for each positive pair to create triplets in the random triplet loss. However, a negative example with less similarity with the positive pair will be selected in the informative triplet loss.

Our goal is to use these several hundred classes to create a model with the proper weights needed for image embeddings of a particular dimension and generalized well for unseen data. At first, the model learns a similarity function from the large-scale training dataset. Then, the model applies the similarity function for prediction. We used it to feed images in the query set and compare the query image with every candidate image of the test set. Then, we used the Cosine distance similarity measure because it works well with high dimensional spaces. We

used Cosine similarity to find the similarities between two images and find the candidate with the highest similarity score. Cosine similarity, or the cosine kernel, computes similarity as the normalized dot product of X and Y:

$$k(x,y) = \frac{xy^T}{||x|| \cdot ||y||} \tag{2}$$

The architecture of the model is based on an implementation of the methodology described in the research paper by Gregory Koch et al. [1] However, the model implemented here is simpler considering the number of neurons per layer than the one in the paper because the size of the images here is in a smaller shape (28,28) in comparison to the shape of the images in the paper which are (105,105). We have only included the layers before the moment where the embedding comparison is made in their report.

1.2 Table of results

We conducted several tests and benchmarking various configurations, such as changing the value of linear output (10, 20, or 50), changing the margin (1, 2, or 5), and using informative and random triplet loss with the model architecture built. These results are available in the figures and tables below. By running the model on the test set, we have the top-1 and top-3 performance as follows:

Accuracy (Top-1 performance)	Top-3 performance
94%	100%

Table 1: Top-1 and Top-3 performance of the model on the test set with **informative** triplet loss

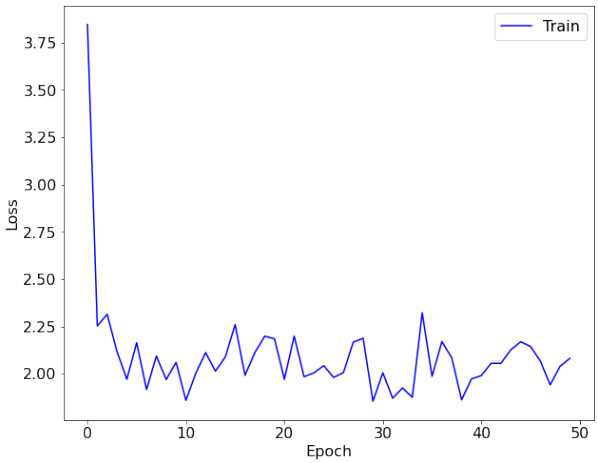

Figure 2: Loss for linear output 20 and margin 2 with informative triplet loss

1.3 Analysis and conclusions

As it can be seen in the Figure 2, we see the loss values for different epochs during the training of the model for linear output value of 20 and margin value of 2. When we compare this figure with Figure 3 and 4, we can conclude that the model works better by decreasing the linear output of the model but it does not make a big difference in the values of loss. However, changing the value of margin from 2 to 1 and 5 changed the values of loss according to the Figure 5 and 6. Also, we tried the model with the random triplet loss in comparison to the informative triplet loss we used before. As it can be seen in the Table 2, the top-1 performance of the model was decreased by 6% but it is also clear that the random triplet loss has better values of loss.

Accuracy (Top-1 performance)	Top-3 performance
88%	100%

Table 2: Top-1 and Top-3 performance of the model on the test set with **random** triplet loss

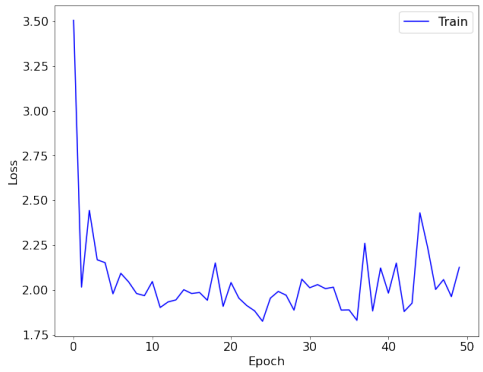

Figure 3: Loss for linear output 10 and margin 2

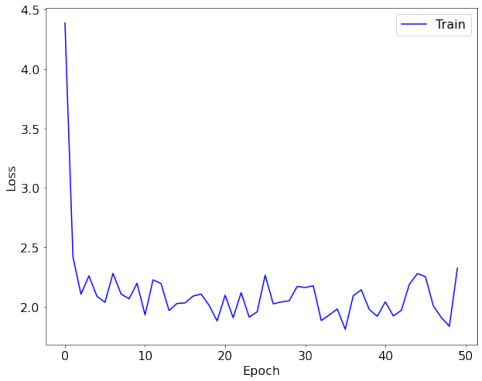

Figure 4: Loss for linear output 50 and margin 2

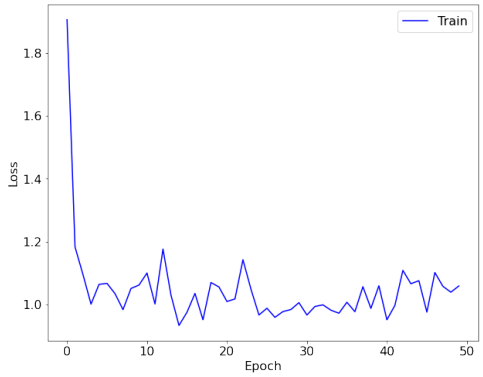

Figure 5: Loss for linear output 20 and margin 1

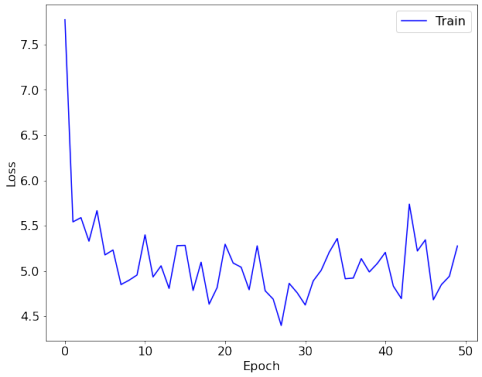

Figure 6: Loss for linear output 20 and margin 5

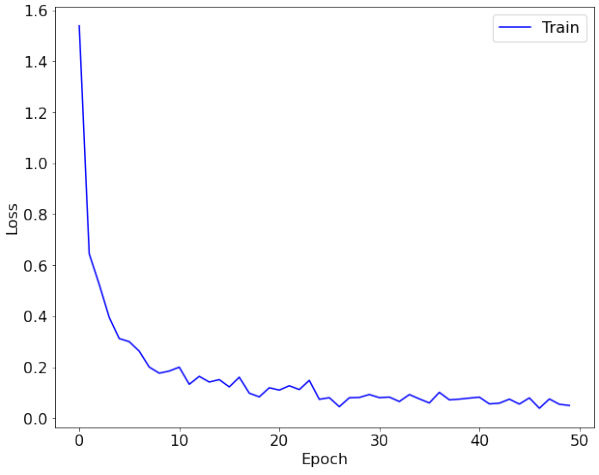

Figure 7: Loss for linear output 20 and margin 2 with the random triplet loss

1.4 Implementation

The implementation can be found in the Jupyter Notebook named **few-shot-learning.ipynb**.

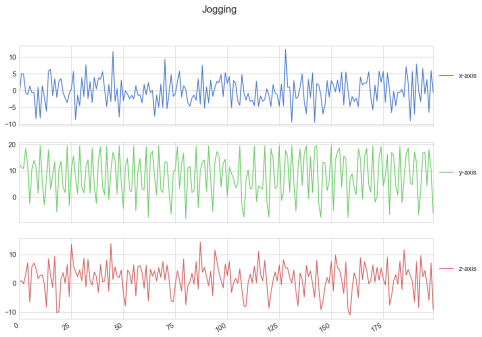

Figure 8: HAR dataset

2 Task 2-Human activity recognition from accelerometer data

The goal of the second task is to develop a model to recognize the activity from accelerometer data. We are using the dataset including data generated for 30 participants who are engaged in one of the six different activities defined (Walking, walking upstairs, walking downstairs, sitting, standing and laying). The model should be able to predict the movement of a person according to the sensors data. There are 3 sensors including total acceleration, body acceleration and body gyroscope and we have 3 components X, Y, Z for each of sensors. There are 1000 measurements in total.

2.1 Problem formulation and proposed approach

This problem can be characterized as a multivariate time-series classification problem where the proposed model has to take into account both data from the timestamp of the current input as well as data from previous timestamps. To that end, we will be using a Recurrent Neural Network architecture that contains a stack of two bidirectional LSTM layers followed by two fully connected layers. The reasoning behind using LSTMs is that due to their construction, they allow us to retain longer memory which in our case is going to be very important since our data comes in sequences of 1000 time steps, and learning temporal data without suffering from vanishing gradients is going to be very difficult. Stacking two LSTMs enables us to learn a temporal hierarchy of features where the second LSTM learns an ever increasingly complex feature map of the input from the previous LSTM and then outputs the hidden vector to the fully connected part of the model where a full representation of the input sample will be learned.

We will be splitting the dataset into mini-Batches and pass them to the model. Each mini-Batch will contain a sequence of 1000 timestamps and 9 features. The matrix will be shaped like this $[100, 1000, 9]$. Essentially the LSTM will be working on the batches of 100 samples where each sample includes 1000 data points so that each data point is described by 9 features. The layer will produce a weight matrix that has been shaped by every individual time moment t_i, of a *sequence*$_j$ where $j \in [0, 100]$, $i \in [0, 1000]$ while taking into account all the previous moments from t_0 to t_{i-1} and which are tracked by the hidden state and the cell states of the LSTM. *Note: (This inherent shortcoming of the LSTMs, the fact that during the first few timestamps of the sequence previous knowledge is very weak and thus accuracy is low, we make an effort to fix by adding a second LSTM after the first one).* By implementing a bidirectional network, we will have one layer of the LSTM learning forwards by following the sequence of the data and another layer learning backward following the opposite direction. After the input has gone through every neuron in LSTMs and the cell state matrix has its final values, we will forward the hidden state to the fully connected layer. At this point, the hidden state will be equal to

$$h_t = o_t \bullet \tanh(C_t) \tag{3}$$

C_t denotes the cell's output, and o_t the output of the LSTMs output gate.
Finally h_t contains a matrix $W \in R^{H*Sequences}$, H is the number of neurons in the LSTM. Each timestamp of the sequence will now have a specific weight matrix. Continuing the data will now enter two fully connected layers, with the first one having as input an equal number of neurons to the output of the bidirectional LSTM and double the output. After exiting the first dense layer, the input will pass through a ReLU followed by a dropout where 20% of the node outputs will be nullified. Following, another fully connected layer will reduce the output to that of the number of classes.

To calculate the loss we are using a categorical cross entropy where the losses are averaged for the samples of each mini-batch. The loss is calculated as such:

$$loss = \frac{\sum_{i=1}^{N} loss(i, class[i])}{\sum_{i=1}^{N} weight(class[i])} \tag{4}$$

$N \in [1,6]$. With respect to the activation function, a softmax is used s.t. the probability distribution of every activity(class) is calculated for each sequence in the minibatches.

2.2 Table of results

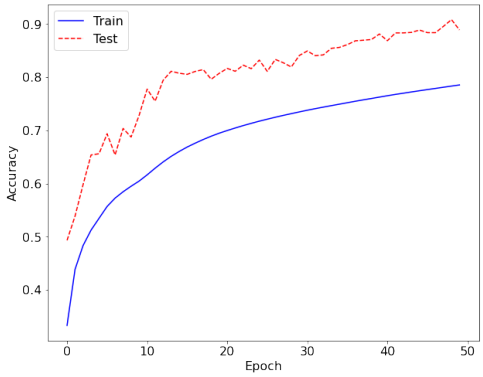

Figure 9: Accuracy for Non-Bidirectional LSTM model.

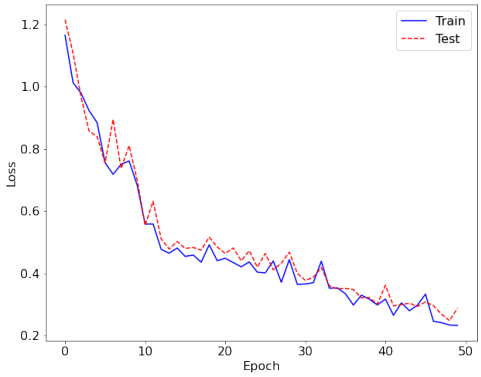

Figure 10: Loss for Non-Bidirectional LSTM model.

With the architecture of the model established, we conducted several tests, benchmarking various configurations, such as making the LSTMs bi-directional vs. normal, altering the number of cells in the LSTM, and the number of neurons in the dense layer. These results are available in the figures and tables below.

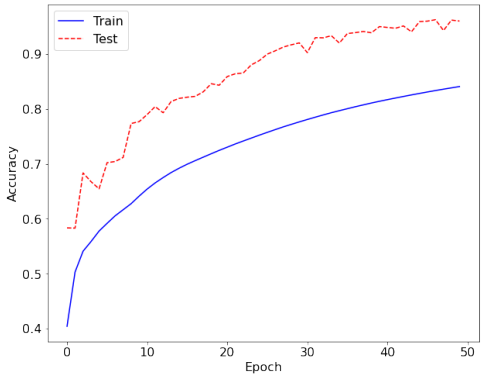

Figure 11: Accuracy for Bidirectional LSTM model with LSTM output of 128*2 and fully connected layers of 256 neurons

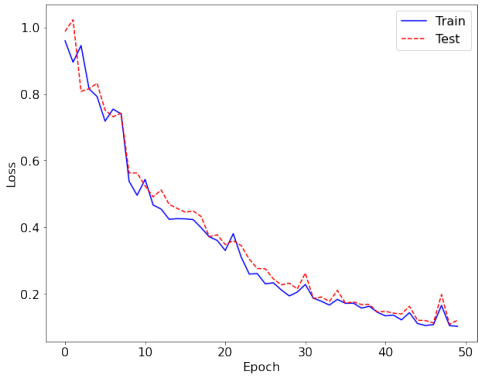

Figure 12: Loss for Bidirectional LSTM model with LSTM output of 128*2 and fully connected layers of 256 neurons.

Accuracy	Loss	LSTM Size	Hidden Size	LSTM stack size	time/epoch
91.7	0.202	64	256	2	3.4s
93.6	0.164	128	256	2	7.3s
94.2	0.121	128	512	2	15s

Table 3: Performance of non-Bidirectional LSTM model

2.3 Analysis and conclusions

We understand that this model is not that efficient with regards to the number of epochs required for it to reach an acceptable amount of accuracy for the test set. Additionally, even though we thought that having a bidirectional network would greatly benefit the model's accuracy, we did not expect such a drastic increase difference, at least for some of the configurations. Checking Table 3 first, we can see the highest accuracy is achieved for the non-Bidirectional model for a network that has 2 stacked LSTMs of 128 nodes each. This comes at the cost of 15

Accuracy	Loss	LSTM Size	Hidden Size	LSTM stack size	time/epoch
95.4	0.110	64	256	2	6.5s
97.1	0.082	128	256	2	14s
96.7	0.081	128	512	2	34s

Table 4: Performance of Bidirectional LSTM model

seconds per epoch. At the same time, we can achieve even better performance by using a Bidirectional model with 64 nodes in the LSTM for almost a third of the training time.

Moving on to the number of LSTM layers, we settled to include two of them in a stacked configuration. As we mentioned in section 2.1, feeding the results of an LSTM to another LSTM module shows improvement to the model's performance. We tested models that included an even higher number of LSTMs, but the drop in accuracy was so drastic, it was seemed counterproductive even to consider them for further tests.

On all tests, we notice that the performance of the model will actually continue to improve should the model keep on training for more epochs. We decided to terminate in 50 epochs because even thought the results are improving, they are doing so with diminishing returns.

Looking forward, we had ideas to include a few convolutional layers before the LSTMs as proposed by Ordonez and Roggen[2], but we had trouble implementing them. Thus we opted to focus on optimizing the performance of the current model.

2.4 Implementation

The implementation can be found in the Jupyter Notebook named **biLSTM.ipynb**.

References

[1] Gregory Koch, Richard Zeme, Ruslan Salakhutdinov, *Siamese Neural Networks for One-shot Image Recognition*, ICML deep learning workshop, vol. 2, 2015.

[2] Francisco Javier Ordóñez, Daniel Roggen *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*

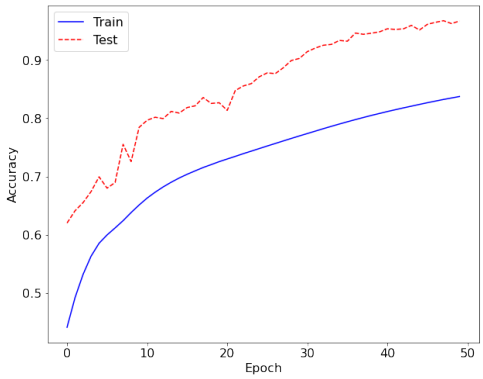

Figure 13: Accuracy for Bidirectional LSTM model with LSTM output of 128*2 and fully connected layers of 512 neurons.

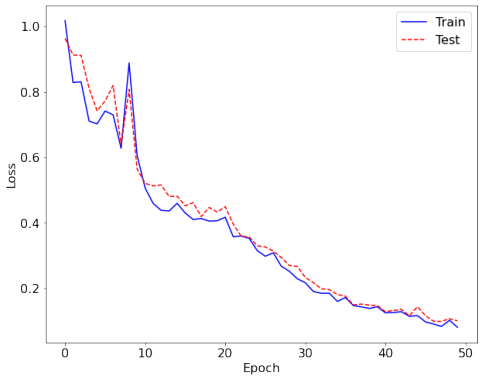

Figure 14: Loss for Bidirectional LSTM model with LSTM output of 128*2 and fully connected layers of 512 neurons.