CS-433 Machine Learning - Resource-Efficient Machine Learning Algorithm Design for On-Implant Neurological Symptom Detection

Authors: Eugenie Chabenat, Sara Djambazovska, Sepideh Mamooler
Supervisors: Mahsa Shoaran, Bingzhao Zhu
Integrated Neurotechnologies Laboratory, EPFL, Switzerland

***Abstract*—Recurrent neural networks, such as long short-term memory (LSTM) and gated recurrent units, have revolutionized the analysis of time-series medical data. However, although modern Machine Learning tools generally obtain a high accuracy in neural signal classification, their deployment on neural interface implants is severely limited due to rigid power, area, and latency constraints for these tiny implants. In the following project, we focus on developing an efficient Machine Learning model to process neural data in real time, with low power consumption, small on-chip area and fast inference.**

I. Introduction

The aim of this paper is to implement a state-of-the-art Deep Learning model while maintaining a resource-efficient implementation. We will work on electrophysiological recordings, namely ECoGs recordings of movements of the five fingers, from three distinct subjects. Further on, we will present and justify the chosen architecture for our ML model, and evaluate the performance based on the correlation coefficient between predictions and labels. In the perspective of developing a model adapted for efficient hardware implementation, we will then use respectively Binarization, Pruning and Trained Quantization for reducing the model memory size. Binarization will allow us to consider all arithmetic operations as bit-wise operations, in order to improve power efficiency. Pruning removes redundant connections, which allows to further compress the model and thus reducing storage and energy consumption. Finally, Trained Quantization and Weight Sharing between several connections of the same layer using k-means clustering, improves further the compression rate.

II. Data Description

The dataset is a subset of the data available for the BCI Competition IV, which had the goal of validating Signal Processing and Classification Methods for Brain-Computer Interfaces (BCI). We will be working on the dataset 4, composed of ECoG recordings of single-trials of spontaneous brain activity. These ECoGs signals were recorded while the subjects performed individual finger flexions while wearing a data glove (Figure 1). The parameters for ECoGs signals are the following : 48-64 ECoGs channels (0.3-200Hz filter bandwidth), 1kHz sampling rate, 5 classes corresponding to the 5 fingers and 3 subjects, 150 finger flexing trials per subject. The dataset labels consist of six classes - 0 as the rest state and 1 to 5 representing individual finger movements.

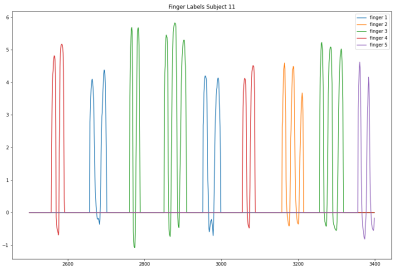


Figure 1: Subject 11 - Superimposed labels for all fingers, no two fingers are moving simultaneously

III. Data Pre-Processing

we pre-processed the feature vectors using the StandardScaler and the labels with MinMaxScaler classes from the scikit-learn library. Supposing that the data is normally distributed within each feature, the StandardScaler scales it, making the distribution centered around zero, while having a standard deviation equal to 1.

For each feature and label, this method computes both the mean and the standard deviation. The scaling is then applied, defined as follows for each data point x:

$$\frac{x_i - mean(x)}{stddev(x)} \tag{1}$$

In order to obtain the original label range, we apply the inverse scaling transform on the predictions and labels before evaluating them.

IV. Model Architecture

Our model is composed of one Long Short-Term Memory (LSTM) layers, and one fully-connected Linear layer. We use Sigmoid as the activation function before feeding LSTM layer's output into the fully-connected layer. LSTM Networks are a specific type of Recurrent Neural Networks (RNN), which have the ability of learning long-term dependencies. They are defined by the following equations:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$C_t^{tilde} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t * C_{t-1} + i_t * C_t^{tilde}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

with C_t being the cell state, C_t^{tilde} the candidates values that could be added to the cell state, h_t the hidden layer, f_t the forgetting gate, i_t the input gate, and o_t the output gate all at time t.

A. Loss Function

As our goal is to evaluate the model based on the correlation coefficient, we chose to use a cost function based on this metric. Firstly, we defined the loss function as the inverse of the correlation coefficient. However, one difficulty we encountered was that this function was not numerically stable in the case where all predictions or all labels considered were equal to zero. In order to avoid this, we then defined the cost as 1 minus the correlation coefficient between the prediction and the labels. The minimization of this loss function ensured maximization of the correlation coefficient, while keeping it numerically stable in all cases. After visualizing the predictions however, we noticed a lot of noise being generated when using the correlation coefficient as part of the loss measure. This led us to use the MSE loss function where the generated noise in the predictions was far lower.

V. Weight Binarization and Fixed-Point Quantization

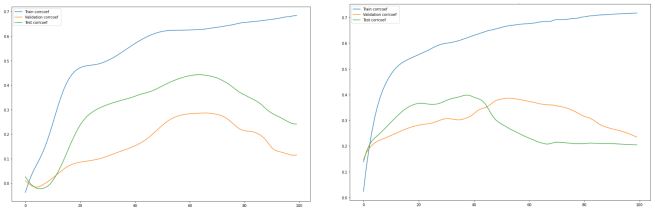
Weight binarization is a method that is expected to substantially improve power-efficiency, by reducing the memory size for storing the weights and replacing most arithmetic operations with bit-wise operations. These Binarized Neural Networks are created by restricting the weights and activations to hold only the value -1 or 1, as explained in [1]. We perform weight binarization during the forward pass, by setting the sign of the weight as its new binary value, with all negative weights set to -1 and the rest to 1.

However, this causes a significant performance drop when compared for 50 epochs of training, as shown in Table I.

To improve the performance, while keeping the power-efficiency, we expand the range for the weights with more integers. After inspecting the weights value, we can notice that they are always between -0.2 and 0.2. This leads us to use fixed point quantization, by rounding the weights to obtain one digit precision floating point numbers. After this transformation, we are left with a weight value set of cardinality 5. As only 3 bits are needed to represent any weight in this range, this solution is very memory efficient. These fixed point weight values are fed as inputs to both LSTM layers during the forward pass of the network. However, this approximate gives very low performance results, as seen in Table I, leading us to further expand the value space. We obtain better correlation results when using 3 decimal precision restriction on the weights as seen in Figure 2. This method would require 8 bits to represent each weight instead of the originally needed 32 bits. Nevertheless, this approach can be further improved by using weight clustering, as shown in Section VII.

Finger	**Ref.**	-1/1	1 dec. pt.	2 dec. pt.	3 dec. pt.
0	0.3913	-0.0009	-0.0769	0.0579	0.3987
1	0.2835	0.0069	0.0077	-0.0134	0.3080
2	0.1655	0.0348	0.0019	-0.0743	0.1534
3	0.2544	0.0079	-0.0365	-0.0291	0.2015
4	0.3228	0.0195	-0.0179	0.0572	0.2869

Table I: Comparing the max correlation coefficient after 50 epochs of training, optimizer learning rate 0.003, Subject 10. The reference is the model without binarization, the next column is with binarization and last 3 columns are using different fixed point quantization



(a) Baseline model

(b) With 3 decimals fixed point quantization

Figure 2: Finger 0, Subject 10 - Train (blue), validation (orange) and test (green) correlation coefficient measures after training for 100 epochs

VI. Pruning

We perform network pruning in order to further compress our model. Pruning has also been proved to reduce the possibility of overfitting. We use the pruning approach described in [2]. This method is composed of three distinct steps of iterative weight pruning. First, we train our model. Then, we prune the weights of the model using a ThresholdPruning class, which removes all weights which absolute value are below a threshold. Finally, in order to minimize the impact of pruning on the precision of our model, we retrain the pruned model. After pruning, the accuracy of the predictions decreases. The iterative pruning methods presents the advantage of having a third step of re-training the model, which is necessary to increase the accuracy of pruned model, thus allowing us to obtain a compressed model with an accuracy comparable to the one of the original model.

The sparsity of our pruned model was quantified by computing the percentage of weights that were pruned over the original number of weights of the model. The key step is finding a threshold allowing a satisfying trade-off between the sparsity of the weight matrix and the prediction accuracy.

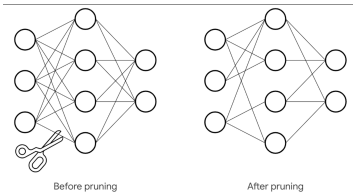


Figure 3: Schematic description of weight pruning taken from [3]

A. *Threshold Tuning*

In order to fine-tune the value of the Threshold used for pruning, we evaluate both compression percentage and correlation coefficient for distinct thresholds, on the data of each finger from Subject 10. Since our weights are consistently in the interval [-0.2, 0.2], we choose the threshold values accordingly, ranging for 0.05 to 0.15. Results on test set are provided in Table II.

	Ref.	Thrs=0.05	Thrs=0.1	Thrs=0.125	Thrs=0.15
k%	0	22.39	45.11	56.22	67.18
0	0.3709	0.3788	0.3111	0.2384	0.2771
1	0.2631	0.2696	0.2015	0.1976	0.1775
2	0.1663	0.2146	0.1303	0.1226	0.1248
3	0.3212	0.1911	0.0913	0.2774	0.2638
4	0.3153	0.3067	0.0486	0.2474	0.1872

Table II: Comparing the correlation coefficient on test set after training for 40 epochs, threshold pruning and retraining for 30 epochs on Subject 10. Ref. is the model without pruning, trained on 70 epochs. k% is the percentage of weight pruned

We observe that pruning decreased the correlation coefficient and thus the performance of our model. However, for a small percentage of pruned weights (22%) the performance actually increased compared to the model on Finger 0, 1, and 2. We explain this by the hypothesis that pruning help reduce any overfitting happening in the reference model, allowing a better performance on the test set. Further we will investigate whether changing the number of epochs happening before or/and after pruning can minimize the loss in performance.

B. *Evaluating with a threshold of 0.125*

A threshold of 0.125 offers an opportunity of removing more than 56% of the original weights. However, we saw that performance is significantly affected by such pruning. We are interested in observing whether or not adapting the number of epochs can have an positive impact on the performance when using this threshold. We choose to try increase epochs respectively before and after performing pruning. Results are provided in Table III. In both cases, the correlation coefficient tends to increase when increasing the number of epochs. The model trained in 40 epochs and then 50 more has a performance approaching the one of the unpruned model. We witness a risk of overfitting both when increasing the number of epochs before pruning (visible in Finger 2, 3) and after (Finger 3).

	Ref.	40/30	60/30	40/50
k%	0	56.22	56.29	56.24
0	0.3709	0.2384	0.3309	0.3519
1	0.2631	0.1976	0.3726	0.2576
2	0.1663	0.1226	0.1151	0.1958
3	0.3212	0.2774	0.2606	0.2323
4	0.3153	0.2474	0.3345	0.3426

Table III: Comparing the correlation coefficient on test set after training the pruned model (Thrs = 0.125) for different numbers of epochs, on Subject 10. Ref. is the model without pruning, trained on 70 epochs. k% is the percentage of weight pruned. The first row indicates the number of epochs "before/after" pruning

VII. Trained Quantization and Weight Sharing

Trained Quantization and Weight Sharing introduced in [2] is another compression method we try to make our model more space-efficient. This method reduces the number of bits required to represent each weight by clustering the weights of the connections in each layer using k-means clustering, and storing the cluster indices instead of the 32 bit weights. Trained Quantization and Weight Sharing consists of two main steps: weight clustering, and feed-forward and back-propagation.

A. *Weight Clustering*

We use k-means clustering to group the weights in k clusters. So the weights for each connection in a layer are replaced by a $log(k)$ bit number indicating the index of the cluster it corresponds to. As a result instead if a 32 bit float number for each connection, we only store a low bit width number. Figure 4 shows the probability distribution of LSTM's last layer's quantized input-hidden weights for each finger of subject 10.

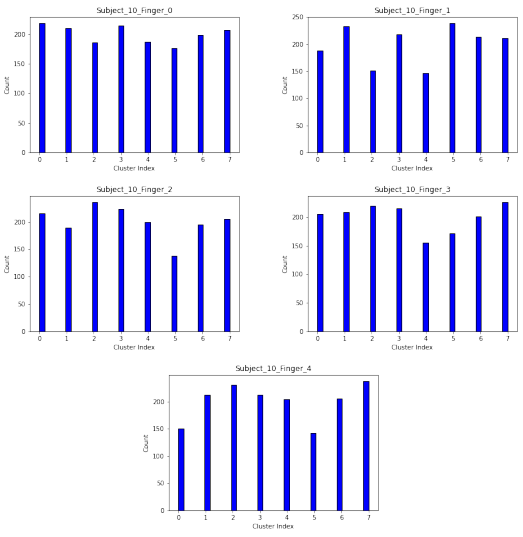


Figure 4: **Cluster Distribution**. Connection weights are clustered into 8 bins using k-means clustering.

The number of clusters plays a key role in the accuracy of the quantized model. We expected to have better correlations with higher number of clusters. However, our experiments showed that having more clusters does not always lead to higher accuracy. As an example, for finger 1 the accuracy of the quantized network deceases as we increase the number of clusters. We believe this is caused by the fact that using less clusters reduces the variance of the model leading to higher test accuracy. Moreover, other factors including centroid initialization in k-means clustering impact the quality of clustering and thus the quantized model's performance. Table IV summarizes the results of our experiments using $4, 8$, and 16 clusters for quantizing the model for each finger of subject 10.

	Ref.	k=4	k=8	k=16
0	0.3483	0.3124	0.3380	0.3403
1	0.2631	0.2781	0.2643	0.2619
2	0.1607	0.1953	0.1659	0.1605
3	0.2147	0.2200	0.2214	0.2057
4	0.3449	0.3574	0.3620	0.3522

Table IV: Comparing the performance (correlation coefficient) of the quantized model before centroid-tuning for different number of clusters. The reference is the model before quantization.

B. Feed-Forward and Back-Propagation

Replacing the connection weights by the centroid of their corresponding cluster decreased the prediction accuracy in some cases, like finger 0. To prevent this, we re-trained the model and updated the centroids using the following formula:

$$\frac{\partial \mathcal{L}}{\partial C_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial W_{ij}} \frac{\partial W_{ij}}{\partial C_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial W_{ij}} \mathbb{1}(I_{ij} = k) \qquad (2)$$

Figure 5 illustrates weight sharing using k-means clustering and centroid fine-tuning with 4 clusters for a layer with 4 input neurons and 4 output neurons.

After re-training we obtain the quantized model by replacing the weights of each layer by the index of their corresponding cluster. Once this is done, the final weights are computed from the indices and the centroids for each layer in each forward pass, which adds an extra step to the prediction pipeline.

Table V shows the correlation coefficient between the labels of the test data and the predictions of our model after re-training the quantized model for different number of epochs. Our experiments show that re-training the model after quantization does not contribute to model's performance considerably.

As a result, using Trained Quantization and Weight Sharing, we can compress the model by a rate of $32/k$, k begin the number of bins used to cluster the weights, while maintaining the model's performance. Figure 6 shows the

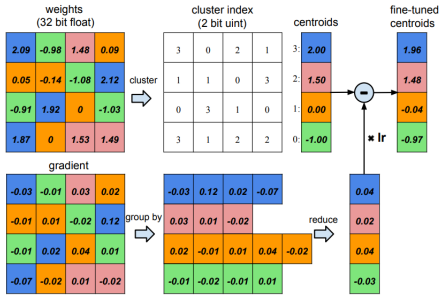
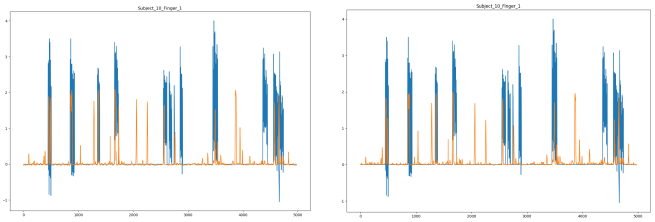


Figure 5: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom). (Taken from [2])

	Ref.	epochs=30	k=epochs=60	epochs=90
0	0.3380	0.3379	0.3381	0.3380
1	0.2643	0.2643	0.2643	0.2643
2	0.1659	0.1660	0.1660	0.1659
3	0.2214	0.2212	0.2212	0.2213
4	0.3620	0.3621	0.3621	0.3621

Table V: Comparing the performance (correlation coefficient) of the quantized model after centroid-tuning for different number of epochs. The reference is the quantized model before centroid tuning. The qunatiztion is performed using 8 clusters.



(a) before trained quantization (b) after trained quantization

Figure 6: **Predictions**. Predictions (orange) and ground truth (blue) before trained quantization (left) and afterward (right).

model's predictions and ground truth for finger 1 subject 10. As indicated, there is no considerable change it model's prediction after quantization.

VIII. Conclusion

We implemented an artificial recurrent neural network (RNN) model architecture, using Long Short Term Memory networks and trained it on EoG signals to predict the movement patterns of fingers. Moreover, we deployed Pruning, and Trained Quantization and Weight Sharing, two state-of-the-art deep compression methods. We showed that using any of these three techniques results in an efficient model with considerably reduced size without a substantial loss of accuracy, with Trained Quantization giving the best trade-off. The compression rate can be further increased by quantizing the pruned model and building separate model architecture for each finger, which is left as a future work.

REFERENCES

[1] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio1, "Binarized neural networks: Training neural networks with weights and activations constrained to +1 or 1," 2016.

[2] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization, and huffman coding," *ICLR*, 2016.

[3] [Online]. Available: https://medium.com/tensorflow/tensorflow-model-optimization-toolkit-pruning-api-42cac9157a6a