Optimizing performance of decoding EEG dataset using CNN and RNN architecture

Keyu Ji
University of California, Los Angeles
Los Angeles, CA 90095
g.jikeyu@gmail.com

Abstract

I built a CNN, an RNN, and a CRNN using PyTorch for the EEG dataset, aiming to maximize the test accuracy on classification. The CNN is able to achieve 69% accuracy. The CRNN, with LSTM layers inserted in the modified CNN, is able to achieve 61% accuracy. The RNN, with convolution layers removed from the CRNN, is able to achieve 59% accuracy. I ensembled multiple CNN's and it is able to achieve 70% test accuracy. The downside of my algorithm is that it requires thousands of epochs for training. Some potential improvements include ensembling more uncorrelated trained models, data augmentation, mixed precision training, and more investigation on RNN.

1. Introduction

I built a CNN, a CRNN, and an RNN using PyTorch for the EEG dataset.

For CNN, I first used the PyTorch implementation of AlexNet as the starting point. Compared with CIFAR-10 dataset whose image is each 32*32*3, each EEG data is 22*1000. Though each EEG data has more entries, it is one dimensional. As a result, EEG should not require a much more complex model to describe it. The empirical results is consistent, as the network easily overfits the training data.

The changes I made include the following. All the layers are changed from 2d to 1d. I added BatchNorm after Conv1d and Dropout after MaxPool. The stride distances are kept smaller than 2 to capture more correlation between consecutive data points. Kernel sizes are chosen in the way that fc layers will not have too little information to process. Dropout layers after Conv1d have dropout probability of 0.8, because worse validation accuracy is observed with smaller probability. Dropout layers among fc layers have dropout probability of 0.5 because the model struggles to learn when probability is as high as 0.8. In deciding how many units I should have in the fc layers, I once again referred to AlexNet, which has 4096 units, and chose 5376 for my own. The network predicted only one class at the beginning, which motivated me to change ReLU to LeakyReLU, since ReLU has no learning for $x <= 0$. In attempting to achieve better performance, I changed the structures of layers to figure from class notes and added one more Conv1d. I also removed the avgpool in the PyTorch implementation of AlexNet, as I found it does not improve the performance.

The CNN was trained with learning rate = 0.001 and L2 regularization = 0.0005, which come from the optimal model I have from HW5.

Because initially I was not able to get RNN learn the data, I turned to deploy a CRNN with reduced convolution layers and LSTM layers inserted between convolution layers and fc layers. I used three stacked layers to make RNN layers more dominant in this model. As RNN layers seem to overfit more easily, I increased L2 regularization to 0.01.

After the CRNN started to learn, I also tried to remove the convolution layers and observe the performance. The resultant RNN only has three stacked LSTM followed by three affine layers with BatchNorm layers, and two LeakyReLU layers. I chose hidden size to be 15 and dropout to be 0.8 to avoid overfitting. I also transposed the data so that the time dimension comes before the dimension of electrodes (i.e., it is now 2115,1000,22 instead of original 2115,22,1000).

The loss function I used is cross entropy loss, as this is multi-class classification. The optimizer I used is Adam, which is a personal preference. The architecture summaries are in table 6, 7, and 8.

2. Results

I split train_valid into 80% training and 20% validation. The batch size is 128, same as AlexNet. I trained the models with an RTX 2060. I recorded the model with the best validation error from each training.

I trained my CNN with 2000 epochs and the best model achieves 65% test accuracy. I further trained my CNN

with 5000 epochs and the best model (CNN #1 in table 1) achieves 69% test accuracy. The second best model (CNN #2 in table 1) achieves 68% test accuracy.

I also tried ensembling method by combining the outputs from 5 models, each independently trained with 5000 epochs, and is able to achieve 70% test accuracy. This is because, as explained in the lecture, that independently trained models have small chance of making the same mistakes. The five ensembled models (Ensembled CNN's in table 1), besides CNN #1, can each achieve 67% or 66% test accuracy.

In general, my CNN model is better at learning behavior 769, 770, test accuracy typically exceeds 80%, whereas for 771 and 772, the model struggles to achieve accuracy above 70%. It may be because that behavior 769 and 770 have more examples than 771 and 772, thus when trying to minimize the training loss, the algorithm tends towards getting 769 and 770 correct and sacrifices the correctness on 771 and 772.

I also tried to apply the CNN algorithm on individual subject. It turns out that the CNN only predicts one single class on test data, which leads to 24% to 28% test accuracy. One example is shown in table 5

I trained my CRNN with 2000 epochs and the best model achieves 61% test accuracy.

I trained my RNN with 2000 epochs and the best model achieves 59% test accuracy.

More detailed statistics are shown in tables on page 5, with accuracies on respective classes.

3. Discussion

I think there are several advantages of my CNN that lead to its decent general performance. First, the model is neither too simple nor too complicated to describe the data. I did attempt to add more layers but data showed it did not improve performance. For the last three convolution layers, I also referred to the insights gained from VGGNet to have smaller but deeper filters. Second, high dropout probability and L2 regularization are used to prevent overfitting. I observed that EEG dataset is much easier to overfit, compared with CIFAR-10, so dropout and L2 regularization are especially important in generalization. Three, BatchNorm is used to help with the training. Four, I spent time attempting to optimize the architecture and hyperparameter. Five, I used a large number of epochs to train the models.

The observation from table 1 shows that ensembling is not combining strong classifiers to make stronger ones but about combining classifiers with different strengths, even if they are relatively weaker. CNN # 2 has higher test accuracy than most of other ensembled models individually. However, test accuracy drops to 64% when CNN #2 is added to the ensembling. From table 1, we can observe that CNN #2 does especially well on class 770, which does not help much because 770 is a class that my model has been doing well for the most time. For class 771 and 772, CNN #2 is worse than any other model; and since my models do often struggle with class 771 and 772, the addition of CNN #2 is destructive to the performance. For other CNNs in table 1, some CNNs do better in 772 and some do better in 771, so that when their output is combined, they make fewer mistakes.

Regarding individual subject training and the problem of only predicting one class, my hypothesis is that the architecture is too complex. I received suggestions to reduce the number of hidden layers and start from simple models. It is possible that each person has his own distinct EEG pattern, thus the data of each individual subject is much simpler, compared with all the data combined together. I had encountered this phenomenon when I first applied my prototype CNN on all the data. Since several changes had been made before the problem was resolved, I am now unable to identify what the problems were.

I further checked the performance of CNN #1 and #2 on individual subject, which is shown in table 3 and 4. Interestingly, these two CNNs both perform worst on subject 1 and subject 7. From per-class numbers, we can see that the CNNs perform especially poorly on classifying 771 and 772 for these two subjects, with accuracies around 20%-40%. However, the accuracies of 771 and 772 for other subjects can be as high as 70%-90%. It is possible that subject 1 and subject 7 are different from other subjects in some ways so that their data exhibit different patterns.

I also checked the models' performances when they are trained over 2000 epochs with data over different periods of time. The result is shown in figure 1. The smallest number of time points I have is 360, because that is effectively the smallest my model can take, which makes the shape of output of the last pooling layer (256*1). From the figure, we can observe a general trend of increasing test accuracy with more time points available. However, even if we only have 360 time points, test accuracy of 62% is still reasonable. A simpler network could be designed in the future to see the result of having even less time points.

On the other hand, the performances of CRNN and RNN are not so satisfying. I think this is partially because I spent much less time trying out different RNN architectures and training different RNN models. Since the performance of a neural network largely depends on initialization, and initialization is different each time, I expect a trained model with better performance from my CRNN and RNN. The architecture is also subject to improvement. Due to time restriction and individual working, I did not get to explore RNN more. Given the EEG dataset is a time-series data, I expect a decent RNN architecture can well surpass the performance of CNN as CNN does not capture the "early" information of the data. For now, the history of hidden state and cell state

in LSTM is directly detached after each iteration and backpropagation does not go through the history. If it does, it should help with learning of CRNN and RNN.

4. Improvements

There are several improvements could be made in the future.

I used to think that Adam is the best optimizer for neural network, which is apparently not always true according to the paper by Wilson and Wilson's colleagues[1]. The paper finds that in some cases adaptive methods generalize significantly worse than SGD and concludes that we "should reconsider the use of adaptive methods to train neural networks". As a result, I would like to try other optimizers and see how the output changes in the future.

Ensembling can be improved with different CNN architecture, as in this way trained models can capture different information from the data and are more independent.

Data augmentation is also one consideration, though I have not been able to think about one. This would require more insight in the data. For instance, if there exists any periodicity in the data, I can shuffle these cycles around and produce more data for training.

Because RTX 2060 has tensor cores to boost performance in 16-bit floating point matrix multiplication, mixed precision training could be used to speed up the training. This is important because for now training my model with 5000 epochs will take around 42 minutes.

5. Other Findings

There are some other interesting findings that I would like to share.

I found that CUDA support of PyTorch is (more) unstable in the Windows 10 environment. For RNN, `step()` consistently outputs "CUDA Error: unspecified launch failure" when I began the training. For CNN, I sometimes encounter CUDA error at around the 1800th epoch. For these reasons, I had to switch to Ubuntu, and I also found that CUDA has a much better performance in Ubuntu – around 0.5s for each epoch with batch size of 128 in CNN training, compared with more than 1s in Windows 10.

From figure 2 and 3 (screenshots taken from tensorboard), one can see that small changes in training loss can cause large fluctuation in validation loss, which shows the difficulty of generalizing one model.

PyTorch RNN is sensitive to the last unfull dataset from dataloader. Had to set `drop_last=True` to avoid the error. It does not happen to the CNN as data length does not matter in CNN but does in RNN.

Ensembling is a challenge to VRAM. I initially intended to ensemble 10 models on GPU, and CUDA reported "out of memory" when I was about to add the 7th model. Because of that, I had to switch back to CPU to see the output.

References

[1] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht. The marginal value of adaptive gradient methods in machine learning, 2017.

Performance Summary

Table 1. CNN Performance, trained with 5000 epochs

Accuracy/%	769	770	771	772	Overall
CNN #1	75	76	57	64	69
CNN #2	75	82	57	54	68
Ensembled CNN's	75	76	57	64	69
	64	74	64	62	67
	67	68	58	68	66
	69	70	58	67	67
	62	75	60	70	67
Ensembling	72	74	62	69	70

Table 2. CRNN and RNN Performance, trained with 5000 epochs

Accuracy/%	769	770	771	772	Overall
RNN	65	51	50	69	59
CRNN	75	63	57	46	61

Table 3. CNN #1 Performance on individual subject

Accuracy/%	769	770	771	772	Overall
Subject 0	75	75	50	58	64
Subject 1	83	60	20	30	50
Subject 2	87	100	66	71	82
Subject 3	50	64	57	92	68
Subject 4	82	90	75	45	74
Subject 5	66	80	60	66	69
Subject 6	69	77	75	63	72
Subject 7	57	75	45	68	64
Subject 8	84	75	75	80	78

Table 4. CNN #2 Performance on individual subject

Accuracy/%	769	770	771	772	Overall
Subject 0	75	83	42	75	68
Subject 1	75	66	20	38	52
Subject 2	87	90	55	50	72
Subject 3	50	82	71	76	74
Subject 4	76	90	62	27	65
Subject 5	86	80	70	55	75
Subject 6	61	83	75	63	72
Subject 7	42	81	45	37	54
Subject 8	92	91	75	70	82

Table 5. CNN predicting only one class, trained with data from subject 0 only

Accuracy/%	769	770	771	772	Overall
Subject 0	0	0	100	0	28

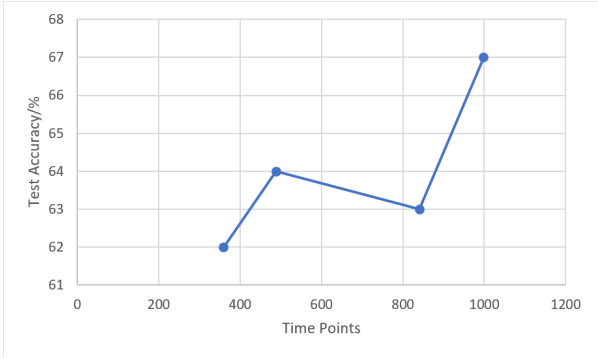

Figure 1. Test accuracies of trained models with different number of time points available, trained with 2000 epochs

Training Process Illustration

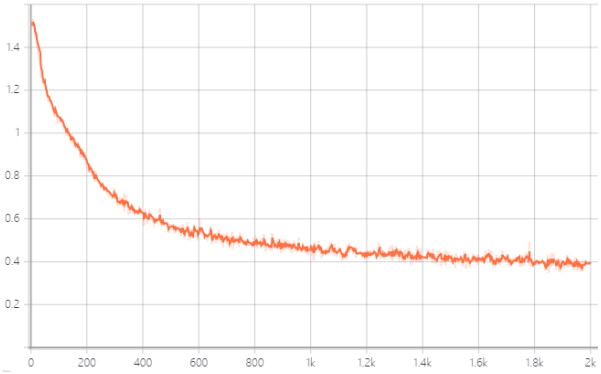

Figure 2. Training loss in 2000 epochs

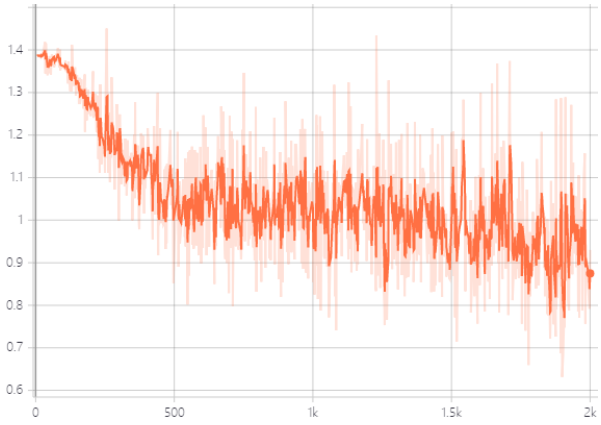

Figure 3. Validation loss in 2000 epochs

Architecture Summary

Table 6. CNN Architecture

Channel	Length	Layer	Parameters							
22	1000		#filters	kernel size	stride	padding	num_features	affine	p	out_features
64	490	Conv1d	64	22	2	0				
		BatchNorm1d					64	FALSE		
		LeakyReLU								
64	240	MaxPool1d	64	12	2	0				
		Dropout							0.8	
192	229	Conv1d	192	12	1	0				
		BatchNorm1d					192	FALSE		
		LeakyReLU								
192	114	MaxPool1d	192	3	2	0				
		Dropout							0.8	
384	56	Conv1d	384	4	2	0				
		BatchNorm1d					384	FALSE		
		LeakyReLU								
		Dropout							0.8	
256	27	Conv1d	256	4	2	0				
		BatchNorm1d					256	FALSE		
		LeakyReLU								
		Dropout							0.8	
256	24	Conv1d	256	4	1	0				
		BatchNorm1d					256	FALSE		
		LeakyReLU								
256	21	MaxPool1d	256	4	1	0				
		Flatten								
		Dropout							0.5	
		BatchNorm1d					5376	TRUE		
		LeakyReLU								
		BatchNorm1d					5376	TRUE		
		LeakyReLU								
		Linear					5376			4
Loss Function		Cross Entropy								
Optimizer		Adam, weight_decay=0.0005, lr=1e-3								

Table 7. CRNN Architecture

Channel	Length	Layer	Parameters							
22	1000		#filters	kernel size	stride	padding	#features	affine	p	out
64	490	Conv1d	64	22	2	0				
		BatchNorm1d					64	FALSE		
		LeakyReLU								
64	240	MaxPool1d	64	12	2	0				
		Dropout							0.8	
192	229	Conv1d	192	12	1	0				
		BatchNorm1d					192	FALSE		
		LeakyReLU								
192	114	MaxPool1d	192	3	2	0				
		Dropout							0.8	
384	56	Conv1d	384	4	2	0				
		BatchNorm1d					384	FALSE		
256	53	MaxPool1d	256	4	1	0				
			input_size	hid_size	#layers	batchfirst				
		LSTM	53	21	3	TRUE			0.8	
		Flatten								
		Dropout							0.5	
		BatchNorm1d					8064	TRUE		
		LeakyReLU								
		BatchNorm1d					8064	TRUE		
		LeakyReLU								
		Linear					8064			4
Loss Function		Cross Entropy								
Optimizer		Adam, weight_decay=0.01, lr=1e-3								

Table 8. RNN Architecture

Channel	Length	Layer	Parameters				
22	1000		input_size	hidden_size	num_layers	batch_first	dropout
		LSTM	22	15	3	TRUE	0.8
			in_features	affine	out_features		
		Flatten					
		Dropout					0.5
		BatchNorm1d	15000	TRUE			
		LeakyReLU					
		BatchNorm1d	15000	TRUE			
		LeakyReLU					
		Linear	15000		4		
Loss Function		Cross Entropy					
Optimizer		Adam, weight_decay=0.01, lr=1e-3					