Exploration of CNN and RNN Model Performances on EEG Data Sets
CS239AS: UCLA Winter 2018

Debleena Sengupta (004945991)
Fan Hung (804319873)
Jayanth (405026111)

Abstract

Recent advancements in computer hardware and processor technology have enabled researchers all around the globe to vastly expand the existing knowledge about the complexity of the human brain and gain deeper insights into brain processes and structures. There are a variety of applications for EEG technology in today's world, such as social interactions, psychology and neuroscience, and brain computer interfaces. For this study, used the EEG data of 9 patients and we explored various deep learning algorithms to predict 1 of 4 possible activities a patient may be doing: movement of the left arm, right arm, foot or resting.

1. Introduction

Current advances in EEG technology has enables the collection of a plethora of data. This data can be analyzed to gain insight into brain activity and how it relates to different actions. For this study, we were given the EEG data of 9 patients performing 1of 4 different activities and the goal was to build a model that can predict the patient's activity based on the EEG. We decided to explore two main categories of models for this task: CNNs and RNNs. We built a custom CNN model (produced best results), a Deep CNN model [1], Vanilla RNN model, and an LSTM model.

1.1. Motivation for CNN Models

Convolutional neural nets have an interesting ability to learn spatial relations from raw data without prior feature selection or feature engineering. This end to end learning can be leveraged for classification tasks where the data looks like EEG signals because it is difficult to make prior assumptions about features for this sort of data set. However, EEG signals are relatively noisy and such signal properties make learning features more difficult for EEG data, as opposed to image data, for which CNNs work really well. We hypothesized that a CNN architecture could produce reasonably good results in this case as well. We designed a custom 3 layer convolutional neural net for the classification task and this model performed the best out of all the models. Please see Table 1 for performance comparisons and Table 3 for architecture details.

1.2. Motivation for RNN Models

Recurrent Neural Networks exploit short term and long term sequential dependencies in data. This essentially allows a model to have a sense of state. Since our EEG data set is a collection of time series, the initial hypothesis was that we could leverage the RNN's attribute of learning time dependencies and modelling evolving processes to help in the prediction process. We explored a few Vanilla RNN and LSTM architectures. Please see Table 5 for architecture details.

1.3. Motivation for RNN atop CNN Models

We tested models which used both RNN and CNN layers. These primarily added recurrent layers between the output the convolutional layers and the final classification layers of our some of our CNN architectures. The last state of the recurrent layers for this setup was passed into the fully connected classification layers. The motivation for these tests was that the convolutional layers would serve to smooth out noise and extract a sequence of feature vectors, and the recurrent layers would almost model a state machine reading in the sequence and deciding the sequence class.
A few experiments were also run where LSTM layers were placed between some of our CNN layers. For these layers, all of the recurrent network states from the sequence (not just the final one) were fed to the next convolutional layer in the model. The thought was that the recurrent layers used might work well if their entire output sequence was considered.
All of these recurrent models, however, were not very effective.

2. Creating the Train, Validation and Test Sets

The dataset was read from separate matlab files containing patient data and concatenated together. The size for the input data per patient was (288x25x1000), where 288 is the number of trials, 25 is the electrode channels and 1000 is the total timesteps for which the dataset is collected (250 readings per second collected for 4 seconds). We only utilized 22 EEG channels and none of the 3 EOG channels. The total dataset size was ((9x288)x22x1000) = (2592x22x1000). We removed cases with NaNs from the dataset and the final clean dataset size was (2558x22x1000). For initial experiments on our different models, this was then split as (1774x22x1000) as training set, (339x22x1000) as validation set and (445x22x1000) as testset i.e. 20% split as testing dataset. For all the models, the dataset was fed as it is to the model in format of (Batch size x Input_Channels x Timesteps) and was transposed, where the model required input channels as the last axis. The recurrent models themselves swap some of the data axes, due to the libraries that we use, so the (Batch size x Input_Channels x Timesteps) was converted right before the RNN/LSTM layers into (Batch size x RNN/LSTM_Dimentions) (sequentially trained over the timesteps).

3. Results

We explored CNN and RNN models in the study and we were able to obtain best results using a custom CNN model with 3 convolutional layers, including Batchnorm, ELU activation, Maxpooling, and Dropout after each layer. More details of this model's hyperparameters and architecture can be found in Table 1 and Table 3.

All of the the following experiments in section 3.1, 3.2, and 3.3 are based on the custom CNN model.

3.1 Optimization over Patient 1

In this section, we explored the effects of training the model on Patient 1's data and then testing across all patients. We hypothesized that in this case, optimizing over Patient 1 and then picking a test set across all samples would not perform well because the model would learn specificities in the activity of this one patient and thus would not generalize well. We created a new train data set and validation set from Patient 1's data and created a test set over all the modes. We noted the optimized model parameters in Table 2. The validation accuracy was around 52%. When training the model on a specific patient and testing across a sample from all patients, we get a test accuracy around 36%. This is much lower than expected. However, this result is understandable because, as we stated in our hypothesis, optimizing over one patient will overfit to that patient since human activity is very unique to an individual and testing across all patients will not generalize as well as if we trained across all patients.

3.2. Optimization Over All Patient Data

In this section, we explored the effects of training the model across all patient data and then testing across all patients. For this section, we hypothesized that training across all patients will perform better on the test set because the model will learn generalized features to distinguish movements, as opposed to overfitting on one patient. We created a train set, validation set and test set over all the patient data. We noted the optimized model parameters in Table 1. We were able to achieve close to 65% validation accuracy and 65-69% test accuracy. Comparing the results of this section to the results from section 3.1, we observe that our hypothesis was correct and indeed training the model across all patients produced better results. The model learns more generalized features related to each of the 4 activities patients perform.

3.3. Optimization Over Time Over all Patient Data

In this section, we trained the model across all patient data and then tested across all patients. We explored the effect of the classification accuracy as a function of sensor recording time.
For this section, we hypothesized that with more time steps, the accuracy will increase significantly because this gives the model larger features to train on an generalize on as well as more information to process. In Figure 1, we plotted a graph of number of timesteps the model was trained on vs the test accuracy of the model.

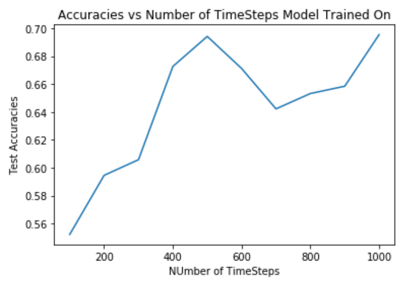

Figure 1: Test Accuracies vs Timesteps

As we can see, after around 500 timesteps the test accuracy fluctuates between 65-69%. More time steps gives the model more data to work with and allows for more feature extraction. Hence, we

Debleena Sengupta (004945991) Fan Hung (804319873) Jayanth (405026111)

can conclude that we need at least 500 time steps to get reasonably performing models. This matches our hypothesis that more time steps can help the model learn well.

4. Discussion (of different models)

4.1 Custom CNN Model

We decided to begin our search for the best model to classify the EEG data by implementing a custom CNN. This model was build in Keras with a tensorflow backend. We have included a description of the the architecture in Table 3. Since our data set is so small, we observed that adding fully connected layers after the convolutional layers was causing too much overfitting. Adding fully connected layers after convolutional layers would be more beneficial in models that have larger data sets to train on since they have larger parameter spaces, such as the ImageNet dataset. As a result, we decided to try building a model that consists of only convolutional layers and one linear layer. With this architecture, we were able to optimize the model to perform on average between 65-69% on test data across all patients.

4.2 Deep CNN Model

Seeing that the model proposed by [1] performed well on a similar problem, we decided to try out this architecture. We have included description of the the architecture in Table 4. We also used this to sanity check our training hyper parameters. With this model, we achieved around 50% training training validation, and testing accuracies.

4.3 RNN Models

We tried to train an architecture where a set of recurrent layers (both Vanilla and LSTM) took the input data directly. We had difficulty achieving any meaningful accuracies. Even when we added fully connected layers after the output of the recurrent layers, we did not see much success. We hypothesized that, without any smoothing, the temporal sequence of the sensors was too noisy and created an unnecessary amount of complexity for the state machine represented by the RNN to capture.

4.4 RNN on CNN Models

We have included description of this architecture in Table 5. The training accuracies for these models were able to reach greater than 50%, but did not exceed 65% . The per-epoch validation scores did not in general exceed 40%, so this setup was able to slowly overfit, but not really perform well on the data. Additionally, post training test set accuracy for these models stayed at or below 30%. In general, it seemed that the recurrent layers did not really help.

To test our hypothesis regarding data smoothing, an additional experiment was run where the recurrent layers were removed and the final fully connected layer's input size was increased so that each neural network node had an input attached to each of the (number of channels X number of timesteps = 8*36) outputs of the CNN layers. This model achieved training accuracy above 90%, per epoch validation accuracies in the mod 50%'s and post training test accuracy of 50%. It appears then, that adding the recurrent layers in this setup overcomplicated the model.

After we implemented the models from Schirrmeister et al [1], we also tried adding an LSTM at the network's outputs (after Block 4 in Table 4) and also in between blocks. The time sequence length for the LSTM was short, because it was only to account for the remaining reduction from our longer signal. These also achieve accuracies that are worse than our original implementations of the CNNs with larger kernel and strides in their pooling layers.

We hypothesize that our recurrent layer sizes were possibly too small to model a state machine capable of making a single scanning pass over the network and deciding the class of our input signals. During training, we ran into long runtimes with the layer sizes that we had, so we did not explore beyond 200 hidden state nodes per layer or beyond 3 layers. Also, it may be that a sequential processing of our input data in general is much less efficient than a combinational logic type pass represented by our normal CNN to final fully connected networks.

Debleena Sengupta (004945991) Fan Hung (804319873) Jayanth (405026111)

References

[1] Tibor Schirrmeister et al. Deep learning with convolutional neural networks for brain mapping and decoding of movement-related information from the human EEG. 2017. https://arxiv.org/abs/1703.05051

[2] Clevert et al. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). 2015. https://arxiv.org/abs/1511.07289

Performance Summary:

Model	Batch size	Epochs	Lr	Opt.	Act.	Dropout	Val Acc	Test Acc
*Custom CNN	250	100	3e-3	Adam	ELU	0.4	0.6549	**0.6943**
Deep CNN	10	20	3e-5	Adam	ELU	0.5	0.71	0.50
Vanilla RNN on CNN	100	40	1e-4	Adam	ELU	None	0.35	0.30
3 Stacked LSTM	100	50	0.0025	Adam	Relu	None	0.24	0.25
LSTM on CNN	100	30	1e-3	Adam	Relu	None	0.64	0.30

Table 1: Overall performance on all models trained and tested across all 9 patients
*Performance varies between 65-69%: Best performance reported here

Patient #	Batch size	Epochs	Lr	Opt.	Act.	Dropout	Val Acc	Test Acc
1	200	100	3e-3	Adam	ELU	0.5	0.5263	0.3595

Table 2 : Overall performance of custom CNN model trained and optimized on Patient 1 and tested across all patients

Architecture and Training Details (All Models Used Adam Opt.):

Custom CNN (Best Performance Model):

Layer Type	Layer Details	Output Shape
1-D Convolution	22 Channels to 128 Channels 1-D Kernel Size 7, padding 1, stride 1	Nx128x1000
Batch Normalization	128 Means and Variations for the 128 Channels	Nx128x1000
Activation	ELU	Nx128x1000
Max Pooling	Kernel size 2	Nx128x500
Dropout	Fraction of the input units to drop: 0.4	Nx128x500
1-D Convolution	128 Channels to 32 Channels 1-D Kernel Size 5, padding 1, stride 1	Nx32x500
Batch Normalization	32 Means and Variations for the 32 Channels	Nx32x500
Activation	ELU	Nx32x500
Max Pooling	Kernel size 2	Nx32x250
Dropout	Fraction of the input units to drop: 0.4	Nx32x250
1-D Convolution	32 Channels to 8 Channels 1-D Kernel Size 3, padding 1, stride 1	Nx8x250
Batch Normalization	8 Means and Variations for the 8 Channels	Nx8x250
Activation	ELU	Nx8x250
Max Pooling	Kernel Size 2	Nx8x125
Dropout	Fraction of the input units to drop: 0.4	Nx8x125
Linear	Input: 1000 Output: 4	Nx4
Softmax		Nx4

Table 3: Custom CNN Architecture

Deep CNN:

In our implementation, we used 2-D convolutions for convenience, and for convenience in referencing the paper by Schirrmeister et al. We treated the temporal, new channel, and sensor channel axes as width, height, and channel width respectively as in the image in the original paper.

Layer Type	Layer Details	Output Shape
	Block 1	
Batch Normalization	22 Means and Variations for the 22 Channels	Nx22x1000
	Inserted a dimension for 2-D Convolution	Nx1x22x1000
2-D Convolution (In effect, only 1-D) (temporal convolution)	1 Channel to 25 Channels 2-D Kernel Size: (Height: 1 for the number of sensors, Width: 10 for the time sequence), Default Padding(0), Stride(1)	Nx25x22x991
2-D Convolution (no kernel sliding combine and reduce 22 sensors)	25 Channels to 25 Channels 2-D Kernel Size: (Height: 22 for the number of sensors, Width: 1 for the time sequence), Default Padding(0), Stride(1)	Nx25x1x991

	Removing a dimension from 2-D convolution	Nx25x991
Batch Normalization	25 Means and Variations for the 25 Channels	Nx25x991
ELU		Nx25x991
Max Pool	Along right-most (time) dimension Kernel Size 3, Stride 3	Nx25x330
	Block 2	
1-D Convolution	25 Channels to 50 Channels 1-D Kernel Size: 10, Default Padding(0), Stride(1)	Nx50x321
Batch Normalization	50 Means and Variations for the 50 Channels	Nx50x321
ELU		Nx50x321
Max Pool	Along right-most (time) dimension Kernel Size 5, Stride 5	Nx50x64
	Block 3	
1-D Convolution	50 Channels to 100 Channels 1-D Kernel Size: 10, Default Padding(0), Stride(1)	Nx100x55
Batch Normalization	100 Means and Variations for the 100 Channels	Nx100x55
ELU		Nx100x55
Max Pool	Along right-most (time) dimension Kernel Size 4, Stride 4	Nx100x13
	Block 4	
1-D Convolution	100 Channels to 200 Channels 1-D Kernel Size: 10, Default Padding(0), Stride(1)	Nx200x4
Batch Normalization	200 Means and Variations for the 200 Channels	Nx200x4
ELU		Nx200x4
Max Pool	Along right-most (time) dimension Kernel Size 3, Stride 3	Nx200x1
	Classification	
Fully Connected	Input: 200, Output: 4	Nx4
Softmax		Nx4

Table 4: Deep CNN Architecture

Vanilla RNN on CNN:

Layer Type	Layer Details	Output Shape
Batch Normalization	22 Means and Variations for the 22 Channels	Nx22x1000
1-D Convolution	22 Channels to 16 Channels 1-D Kernel Size 25, padding 5, stride 5	Nx16x198
1-D Convolution	16 Channels to 8 Channels 1-D Kernel Size 25, padding 1, stride 5	Nx16x36
Batch Normalization	8 Means and Variations for the 16 Channels	Nx8x36
Vanilla RNN	Hidden State Width 32 Layers 2 Output Last Hidden State	Nx(32*2)
Batch Normalization	32*2=64 Means and Variations	Nx(32*2)
Fully Connected	Input: 64, Output: 4	Nx4
Softmax		Nx4

Table 5: Vanilla RNN on CNN Architecture

Modifications for RNNs atop CNNs:

When we started training the Vanilla RNN networks, little training occurred. Checking the means of the gradients of weights and finding that they were low, we determined that our gradients were vanishing. To accommodate this, a couple changes were made. First, all the hidden state to hidden state weights of the Vanilla RNN layers were initialized to identity matrices. Second, the regularizer mentioned class, (from Pascanu and colleagues 2012) was also added to the loss. After adding these, the network began meaningfully training.

For testing LSTMs atop CNNs, a nonlinearity layer (ELU) was added after the CNN layers and the Vanilla RNN layers themselves were replaced LSTM layers. The identity matrix state weight initialization and the regularization were removed.

Also, initially, to implement the Deep CNN, we increased the size and stride of the pooling layers, when we added the LSTM to the outputs, we reverted the pooling layer sizes and strides back to their original values (of 3) from the paper.