Evaluating Convolutional and Recurrent Neural Network Types for EEG 4-class Motor Classification

Nick Brandis
UCLA ECE C147
nicbrandis@g.ucla.edu

Spencer Stice
UCLA ECE C147
smstice@ucla.edu

Nick Turner
UCLA ECE C247
nturner1@g.ucla.edu

Abstract

The goal of this project is to create a machine learning model that accurately classifies electroencephalograph (EEG) data into four different classes based on motor imaginary tasks: moving the left hand, right hand, both feet, or tongue. Our investigation analyzes and compares the performance of CNN, RNN, CNN+LSTM, and ResNet architectures as well as the pre-processing and hyperparameter tuning techniques we used. After tuning, we achieved our highest test accuracy of 69.3% on all subjects and 80.0% on subject #3 using our CNN+LSTM model.

1. Introduction

In this report we evaluate four types of networks: a basic Convolutional Neural Network (CNN), a basic Recurrent Neural Network (RNN), ResNet, and a CNN with Long Short-Term Memory (LSTM)..

1.1. CNN

We attempted to experiment with a model which we had previously had success with: a convolutional neural network. For this task, we implemented a network with 3 convolutional layers (see model architectures for details). For this model, we viewed the input as essentially an array of time sequence values where each time step contains 22 data points. We then performed our convolutions with appropriate padding to maintain the 2D shape of the input while progressively increasing the number of channels. We believed this approach may be effective considering previous successful work using convolutional networks with this data and considering the apparent spatial dependencies of the data when we visualized it. Additionally, we incorporated max pooling in our convolutional layers, dropout, and batch normalization. See the model architecture page for more details.

1.2. RNN

We decided to attempt to achieve desirable performance using a post-CNN model: the residual neural network. Given that our data is a time series from the electrodes, we determined that using a residual network may be effective at modeling the time dependencies. For this effort, we implemented various architectures and found most success with fewer layers rather than more. Most notably, we attempted to train a residual network with only one recurrent layer and a hidden size of 2 (see model architectures for details). We also incorporated dropout into the recurrent layer to prevent overfitting.

1.3. ResNet

We then thought it would be interesting to evaluate a more specific type of model we learned in class, ResNet. ResNet incorporates ResBlocks, which consist of a convolutional layer and activation function followed by another convolutional layer. Then, the input to the feed-forward ResBlock is added to the output of the second convolutional layer. This introduces "skip connections" that help address vanishing gradients [2]. While we weren't experiencing noticeably slow training in our other models, we thought it would allow us to test more complicated and deeper model designs than with a traditional CNN, while being less susceptible to vanishing gradients. We tested two main architectures: ResNet with 15 total ResBlocks; and ResNet with 6 total ResBlocks, dropout within each ResBlock, and weight decay.

1.4. CNN + LSTM

After considering the individual performances of the CNN and RNN architectures as previously described, we considered the possibility that the data may be most effectively viewed as having both spatial dependencies and a time-series structure. To test such a theory, we implemented a hybrid CNN and LSTM combination network with help from the ECE C147 TAs [1]. Most notably, we used a three layer convolutional architecture followed by a recurrent layer featuring LSTM blocks. As we describe later, we discovered that fewer convolutional layers (2-3) worked better than more (4).

1.5. Pre-processing Techniques

In addition to considering multiple architectures, we noticed some interesting characteristics of the dataset and devised some pre-training methods to lead to better training and testing. Specifically, we noticed that the bulk of the data for each example was in the first 500 time stamps. In visualizing the data, after around 500 time points, the spikes cease. The remainder appears to be mostly noise. To investigate whether the low amplitude noisy region impacts performance, we tested varying amounts of data trimming. Then, to extract the most dominant data, we added a max pooling operation on the training and validation data (separately). We then concatenated an array of averages plus some Gaussian noise, and subsampled every other index to add more noise, hoping these efforts would add some regularization to our training data [1].

2. Results

For most architectures, we trained with the Adam optimizer with batch size 64 and learning rate 1e-3. However, for our best architectures, CNN and CNN+LSTM, we experimented with different hyperparameters.

2.1. Accuracy for Single Subject

We experimented with training on an individual subject, using the CNN+LSTM architecture. We compared this individually trained model with another model trained on the entire dataset. The test accuracy using the generally trained model ranged from 54.0% to 80.0%. The 80.0% test accuracy on subject #3 is surprisingly high, and it appears that the model used the other subjects as noise to boost its performance on an individual subject. The individually trained model was trained on subject 0, and achieved 68.0% accuracy. This is similar to the overall test accuracy of the general model, which means individual training likely does not influence the accuracy of the model. The individually trained model achieved test accuracies from 23.4% to 50.0% on the other, unseen subjects. The average accuracy on other subjects was 35.0%. We conclude that the CNN+LSTM architecture works well after training on a specific set of subjects, but it may not generalize well to unseen subjects.

2.2. Accuracy for All Subjects

After experimenting with the various architectures described above, we were able to achieve 69.3% test accuracy on all subjects with our best model, the standard convolutional architecture. Additionally, when combined with LSTM layers, we were able to achieve 68.3% test accuracy. Given more time for tuning and a higher number of epochs, we believe that incorporating the LSTM layers would push the performance beyond these figures. However, with our limited compute capabilities, we are satisfied with these results.

2.3. Accuracy over Different Time Periods

We experimented with a variable amount of data points to discard from the end of the samples. We trained several models, varying the amount of data points we dropped from the end. We found that training using only the first 500 time points actually increases performance, and drastically decreases the training time. Using the CNN+LSTM model, we found that decreasing from the full set to only the first half (500) points results in an increase in test accuracy of 7.9% after 300 epochs.

3. Discussion

In this section we discuss our results for the different models we created, as well as reasons for the choices we made on hyperparameters and architectures.

3.1. CNN

As noted earlier, we found that viewing the input as a 1D array with 22 input channels worked effectively. When designing this architecture, we determined that incorporating data from all channels during the convolution by viewing the channels as a depth, rather than viewing the channels as the height, may be more effective. We also performed hyperparameter tuning on this architecture given its early success. Specifically, we decreased the batch size from our standard 64 to 32 in an attempt to increase the stochastic properties of the optimization. Further, we increased the learning rate by a factor of 2 up to 2e-3. We realized that both of these changes will likely result in a more noisy, “jumping around” optimization procedure, but we noticed that our original attempts resulted in fairly smooth progress (see algorithm performances), so we determined that increasing the noise may be beneficial to escape local optima.

3.2. RNN

Given that we are dealing with time series data, a clear top choice is the recurrent network architecture. Even with just 2 recurrent layers, this model easily overfit the data, suggesting that the expressive capabilities of this model are indeed more than sufficient to understand the data. We then performed hyperparameter tuning in an attempt to improve pure RNN performance. We reduced the number of layers in an attempt to decrease the expressive power of the model and thus force it to not overfit, which we had success with. We also increased the hidden state size in an attempt to continue increasing the model complexity while not overfitting. However, the model still did not achieve desirable performance, so we believe that additional tuning and training time may help. However, given the immediate success of the convolutional models, we decided to halt efforts on pure RNNs and proceed using a combination of their strengths.

3.3. ResNet

The first ResNet architecture with 15 total ResBlocks was created to see how complicated we could make a model. With three, two, four, and six ResBlocks for the layers plus a convolutional layer at the beginning, this represented over 30 convolutional layers in total. This model was able to get near 100% training accuracy, but the validation accuracy plateaued around 31% (see results page).

To address overfitting, we reduced the ResBlock layers to each be two and removed one layer (yielding 6 total ResBlocks), implemented dropout within each ResBlock, and added L2 regularization to the Adam optimizer. We also decreased the learning rate to 1e-4 since the training accuracy shot up quickly in the previous model. After training for 150 epochs, we achieved a much better test accuracy of 52.6% on all participants.

3.4. CNN+LSTM

After much success with the pure CNN model, we implemented a CNN-LSTM combination model. This model consists of 3 convolutional layers, followed by a single LSTM layer. The motivation for this architecture is to enhance the strong performance of CNNs by exploiting the temporal nature of the problem at hand. LSTMs are useful for input sequences along an axis of time.

We implemented 3 convolutional layers followed by a single LSTM layer. We experimented with multiple LSTM layers, but they were found to be extremely prone to overfitting. We tried tuning the hyperparameters for the 3 CNN - 1 LSTM model. Namely, we changed the batch size, training rate, and increased dropout. The best values were simply our initial choices of 1e-3 and 64 for training rate and batch size, respectively. The model performed very well, achieving 68.62% test accuracy. This is slightly worse than the pure CNN network, which we attribute to lack of training data. Since the multiple LSTM layers were prone to overfitting, we believe additional training data, additional preprocessing, and/or data augmentation would bolster this model's performance to surpass the pure CNN model.

4. Conclusion

Of the CNN, RNN, ResNet, and CNN+LSTM architectures we trained and evaluated, we achieved the highest test accuracy of 69.3% on all subjects with our basic CNN, and 80.0% on one subject with our CNN+LSTM model.

References

[1] UCLA ECE C147 Teaching Team. Discussion 7 Jupyter Notebook.

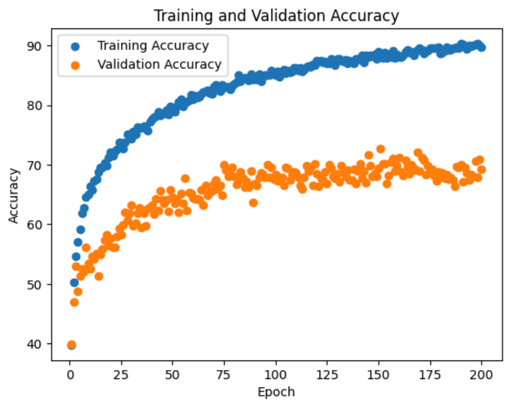
[2] Nouman. “Writing ResNet from Scratch in PyTorch”, *Paperspace*. 2022.

[3] Brunner, C. et al. BCI Competition 2008 Graz Data Set A.

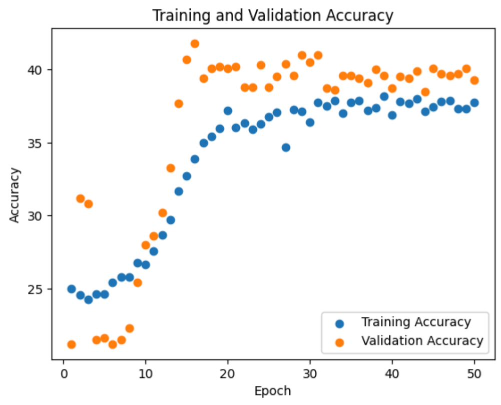
Summary of Algorithm Performances

Architecture Description	# Epochs	Batch Size	Learning Rate	Test Accuracy	Notes
RNN with 2 separate recurrent layers	50	64	1e-3	36.57%	Overfit
RNN with 1 recurrent layer	50	64	1e-3	36.117%	Underfit
RNN with 1 recurrent layer, larger hidden size	50	64	1e-3	29.57%	Overfit
RNN with 1 recurrent layer, larger hidden size, L2 regularization (2)	50	64	1e-3	38.6%	Performance ceiling
CNN with 3 layers, more channels in deeper layers	150	64	1e-3	66.13%	May do better with more training
CNN + LSTM with 3 conv layers and an LSTM layer	500	64	1e-3	68.62%	Test accuracy improved by 4.7% after 3 separate runs
ResNet with 15 total ResBlocks	50	64	1e-3	31%	Overfit
ResNet with 6 total ResBlocks, dropout within each ResBlock, and weight decay	150	64	1e-4	52.6%	May do better with more training
CNN with 3 layers, addition of L2 regularization (1)	**200**	**64**	**1e-3**	**69.3%**	**Solid performance**

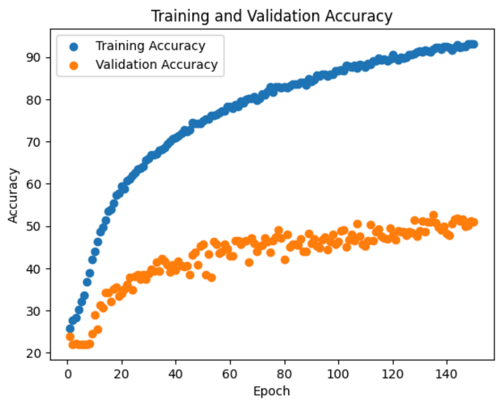
Note: We experimented with different hyperparameter configurations as well, which we describe in the report. This table summarizes our general findings and is not indicative of every experiment we performed.



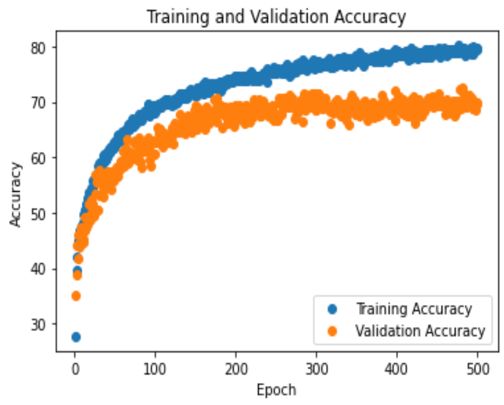
(1) CNN with 3 layers



(2) RNN with 1 recurrent layer



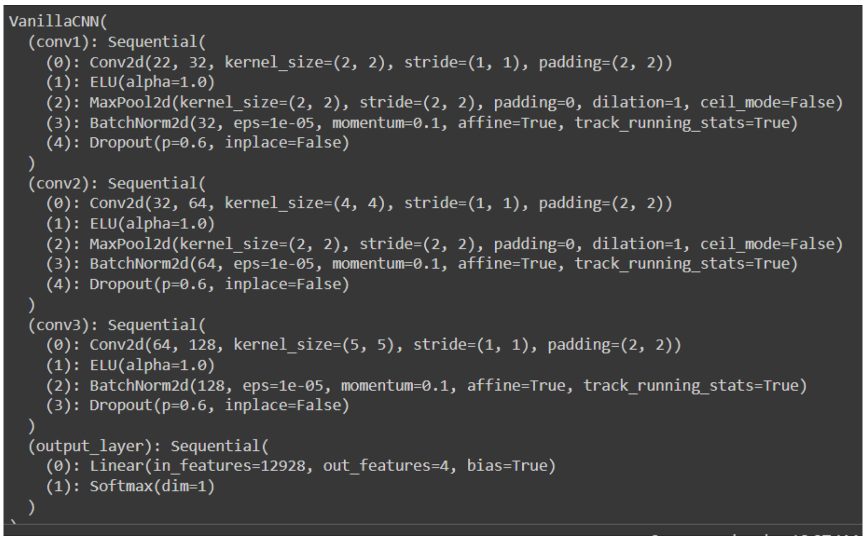
(3) ResNet with 6 ResBlocks



(4) CNN + LSTM with 3 conv layers

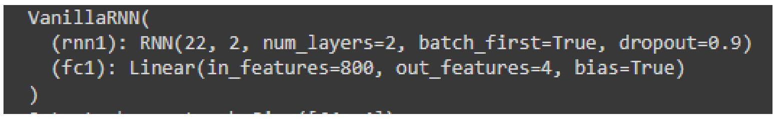
Architecture Summaries

Naive CNN:



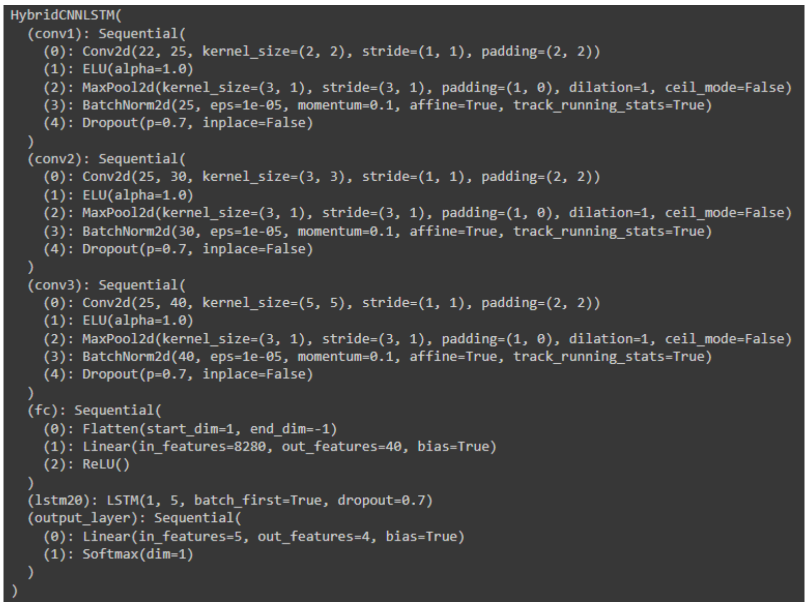
```
VanillaCNN(
  (conv1): Sequential(
    (0): Conv2d(22, 32, kernel_size=(2, 2), stride=(1, 1), padding=(2, 2))
    (1): ELU(alpha=1.0)
    (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Dropout(p=0.6, inplace=False)
  )
  (conv2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(4, 4), stride=(1, 1), padding=(2, 2))
    (1): ELU(alpha=1.0)
    (2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Dropout(p=0.6, inplace=False)
  )
  (conv3): Sequential(
    (0): Conv2d(64, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ELU(alpha=1.0)
    (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Dropout(p=0.6, inplace=False)
  )
  (output_layer): Sequential(
    (0): Linear(in_features=12928, out_features=4, bias=True)
    (1): Softmax(dim=1)
  )
```

Naive RNN:



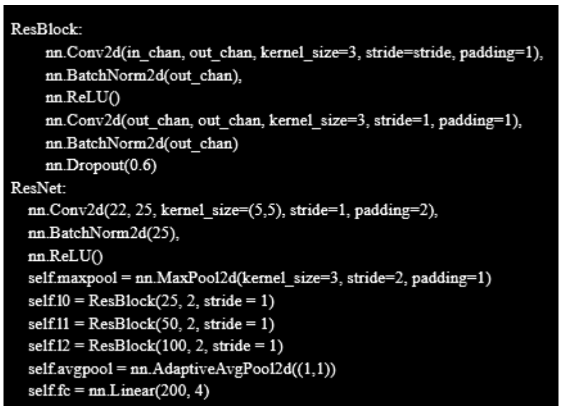
```
VanillaRNN(
  (rnn1): RNN(22, 2, num_layers=2, batch_first=True, dropout=0.9)
  (fc1): Linear(in_features=800, out_features=4, bias=True)
)
```

CNN + LSTM hybrid:



```
HybridCNNLSTM(
  (conv1): Sequential(
    (0): Conv2d(22, 25, kernel_size=(2, 2), stride=(1, 1), padding=(2, 2))
    (1): ELU(alpha=1.0)
    (2): MaxPool2d(kernel_size=(3, 1), stride=(3, 1), padding=(1, 0), dilation=1, ceil_mode=False)
    (3): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Dropout(p=0.7, inplace=False)
  )
  (conv2): Sequential(
    (0): Conv2d(25, 30, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2))
    (1): ELU(alpha=1.0)
    (2): MaxPool2d(kernel_size=(3, 1), stride=(3, 1), padding=(1, 0), dilation=1, ceil_mode=False)
    (3): BatchNorm2d(30, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Dropout(p=0.7, inplace=False)
  )
  (conv3): Sequential(
    (0): Conv2d(25, 40, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ELU(alpha=1.0)
    (2): MaxPool2d(kernel_size=(3, 1), stride=(3, 1), padding=(1, 0), dilation=1, ceil_mode=False)
    (3): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Dropout(p=0.7, inplace=False)
  )
  (fc): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=8280, out_features=40, bias=True)
    (2): ReLU()
  )
  (lstm20): LSTM(1, 5, batch_first=True, dropout=0.7)
  (output_layer): Sequential(
    (0): Linear(in_features=5, out_features=4, bias=True)
    (1): Softmax(dim=1)
  )
)
```

ResNet:



```
ResBlock:
      nn.Conv2d(in_chan, out_chan, kernel_size=3, stride=stride, padding=1),
      nn.BatchNorm2d(out_chan),
      nn.ReLU()
      nn.Conv2d(out_chan, out_chan, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(out_chan)
      nn.Dropout(0.6)
ResNet:
   nn.Conv2d(22, 25, kernel_size=(5,5), stride=1, padding=2),
   nn.BatchNorm2d(25),
   nn.ReLU()
   self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
   self.l0 = ResBlock(25, 2, stride = 1)
   self.l1 = ResBlock(50, 2, stride = 1)
   self.l2 = ResBlock(100, 2, stride = 1)
   self.avgpool = nn.AdaptiveAvgPool2d((1,1))
   self.fc = nn.Linear(200, 4)
```

Note: All architectures were trained using Adam optimizer and used data augmentation and pre-processing techniques detailed in the report