Deep Learning Project

1. **[Bonus question]** Demonstrate a good understanding of the data that involves visualizing data and show how this understanding is used in model design. You are not required to answer this question, but extra points will be offered if you answer this. **[5 points]**

**(Note: Full code provided in separate file)**

First, we load in "Set_1.npz". To start off, let's look at the shape of the data:

Examining data shape

```
# Train loader
for setID in train_set_idx:
  train_set = MyDataset(setID)
  train_loader = torch.utils.data.DataLoader(train_set,
                                             batch_size=128,
                                             shuffle=True)

  print(setID)
  for X_train, y_train in train_loader:
    print(f"X Shape: {X_train.shape}")
    print(f"y Shape: {y_train.shape}\n")
    break
```

```
1
X Shape: torch.Size([128, 4, 4000])
y Shape: torch.Size([128, 4000])
```

We have a batch size of 128. Our X is a Tensor of size 4*4000, and our y is a tensor of size 4000.

From the provided project description, we know that the X first dimension of X represents the 4 features, and the second dimension represents 4000 points in space along straight line.

Now, let's examine if the dataset has any nans.

```
# Train loader
for setID in train_set_idx:
  train_set = MyDataset(setID)
  train_loader = torch.utils.data.DataLoader(train_set,
                                             batch_size=128,
                                             shuffle=True)

  print(setID)
  for X_train, y_train in train_loader:
    print(f"Num y NaNs: {torch.sum(y_train.isnan())}")
    print(f"Num X NaNs: {torch.sum(X_train.isnan())}")
    break
```

```
1
Num y NaNs: 3998
Num X NaNs: 0
```

It appears that X doesn't have any nans, but y has 3998 nans about of 128*4000 total values. That is ~1% of the data in y. We will need to handle this somehow (maybe by setting the values to 0).

Now, let's look at the head of the data.

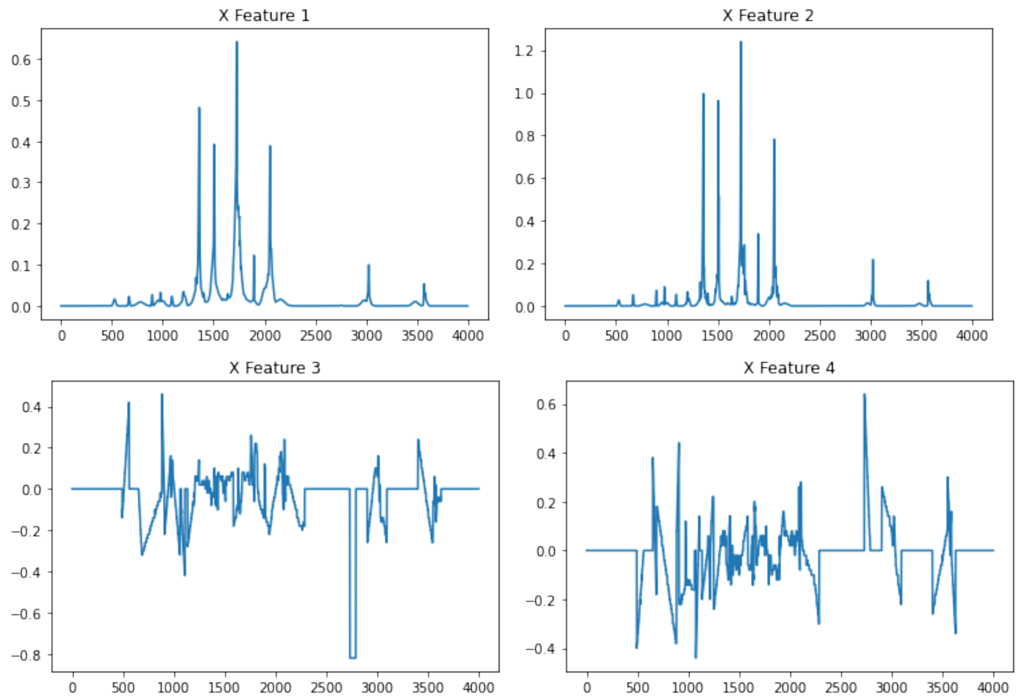
```
# Train loader
for setID in train_set_idx:
  train_set = MyDataset(setID)
  train_loader = torch.utils.data.DataLoader(train_set,
                                             batch_size=128,
                                             shuffle=True)

  print(setID)
  for X_train, y_train in train_loader:
    print(f"X: {X_train[0, :, 0:10]}")
    print(f"y: {y_train[0, 0:10]}\n")
    break
```

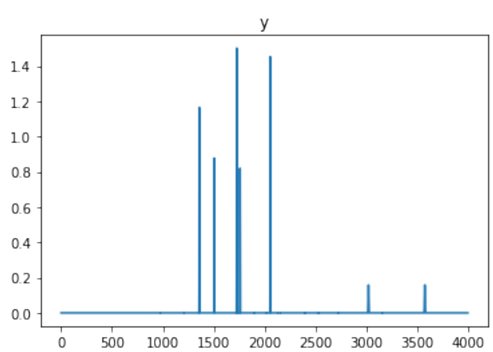
```
1
X: tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
y: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

That is a lot of zeroes.

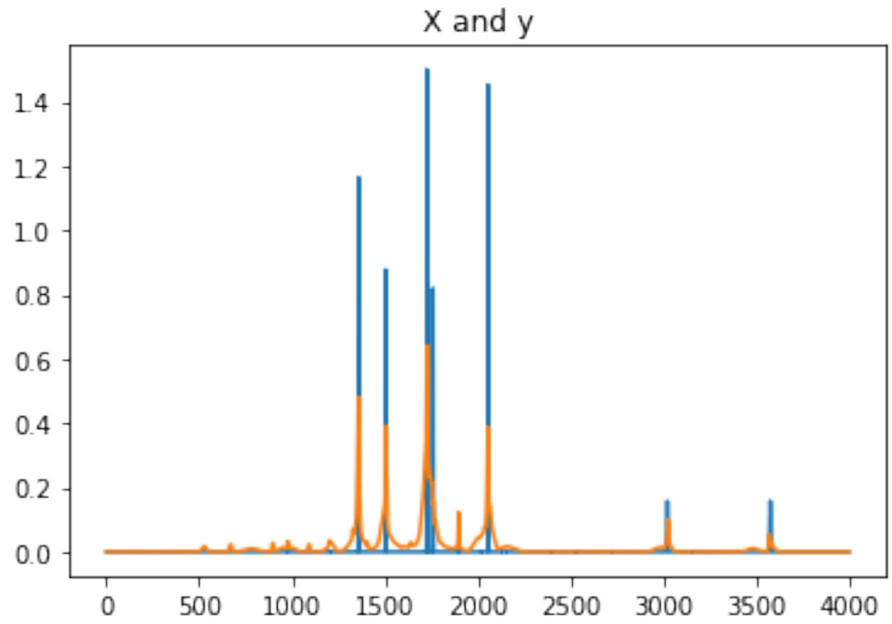
Let’s try graphing X:




And now y:




It looks almost like y is trying to predict location of the peaks in the first two features of X. Let’s try superimposing the graphs.




Yup. It seems like we are trying to find the peaks. I'm not exactly sure what the 3rd and 4th feature of X are, but it seems like we are trying to find the peaks in the first two features of X. I'm somewhat familiar with task similar to this. For example, I have some knowledge for processing audio to try and identify the time that certain events happen in the track. (This problem seems somewhat similar). For these problems, it is typical to use RNNs or 1 dimensional CNNs. So, I think that I will try both a bi-directional RNN and a CNN architecture. I might also train a fully connected network as a baseline.

2. Clearly show at least three different model designs and your rationale for choosing them, and the three performance metrics (loss, efficiency, fp_rate) on the validation set. You want to have these three models vastly different from one another. For example, choosing three MLP with different #hidden units would be a bad choice. You are welcome to explore more than three models, as this will increase your chance of arriving at a better model for optimum performance on test data. Novelty in model design is considered in grading this part. Highlight in your report what you consider as novelty in your design. The most infrequently reported model designs will get higher score compared to those frequently examined among other students. **[10 points]**

**Model 1 (Bi-directional RNN):**

The first model I tried is a bi-directional LSTM. I picked an RNN style model because of its ability to exploit the sequential nature of the data, i.e., the data points are contiguous in space. Additionally, I wanted the RNN to be bi-directional because, from the earlier data analysis, I determined that we are trying to predict the peaks in the training data. Conceptually, it should be much easier to find the peaks if we have data from both sides of the peak.

Novelty:

Concerning the novelty of this model, I doubt that very many people in this class tried a bi-directional LSTM simply because bi-directional RNNs weren't covered in much detail in this course. Additionally, I have what is essentially an average pooling layer at the end the network which empirically I found works better than the fully connected layer, but I believe that this is a somewhat unusual choice.

Performance:

```
Loss: 0.08754191547632217
Eff: 0.7617046376170463
FP: 0.16056788642271547
```

Model Code:

```
# Bidirectional recurrent neural network (many-to-many)
class BiRNN_2(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(BiRNN_2, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm_1 = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, bidirectional=True)
        self.lstm_2 = nn.LSTM(2*hidden_size, 1, 1, batch_first=True, bidirectional=True)

    def forward(self, x):
        x = torch.transpose(x, 2, 1)
        # x dim: [batch size, sentence length, feature dim]

        out, _ = self.lstm_1(x)  # out: tensor of shape (batch_size, seq_length, hidden_size*2)
        # output dim: [batch size, sentence length, hidden dim*2]
        # hidden dim: [2, batch size, hidden dim]

        out, _ = self.lstm_2(out)

        # average pooling
        out = (out[:, :, 0] + out[:, :, 1])/2

        return out.squeeze()
```

## Model 2 (CNN):

The second model that I tried was a 1d CNN. I got the idea for this model through some experience I have with using neural networks to identify a particular sound (e.g., a snare drum being struck) in audio tracks. For these types of problems, it is common to use 1d CNN architectures. As this data for this project seems similar to the data used for identifying sounds in an audio track, I posited that I similar model might work.

Novelty:

I don't think that 1d convolutions or 1d transpose convolutions were mentioned in class, so I doubt many other people in the class tried this approach. That said, I do believe that this makes sense, and I have seen similar approaches used on similar data. Additionally, I decided to reduce the size of the length of the data similar to an encoder and then increase the image width similar to a decoder. I did this because it would both allow me to cover a larger portion of the data with a smaller kernel and speed up computation. This is likely somewhat unique for this problem, although I am aware of similar CNN designs.

Performance:

```
Loss: 2.2011327743530273
Eff: 0.8917643589176436
FP: 1.624875024995001
```

Model:

```
[ ] class CNN(torch.nn.Module):
      def __init__(self):
        super(CNN, self).__init__()

        # Encoder

        # 4000
        self.conv1 = torch.nn.Conv1d(in_channels=4, out_channels=64, kernel_size=5, stride=4, padding=2)
        self.bn1 = torch.nn.BatchNorm1d(num_features=64)

        # 1000
        self.conv2 = torch.nn.Conv1d(in_channels=64, out_channels=128, kernel_size=5, stride=2, padding=2)
        self.bn2 = torch.nn.BatchNorm1d(num_features=128)

        # 500
        self.conv3 = torch.nn.Conv1d(in_channels=128, out_channels=256, kernel_size=5, stride=2, padding=2)
        self.bn3 = torch.nn.BatchNorm1d(num_features=256)

        # 250
        self.conv4 = torch.nn.Conv1d(in_channels=256, out_channels=256, kernel_size=5, stride=2, padding=2)
        self.bn4 = torch.nn.BatchNorm1d(num_features=256)

        # Decoder

        # 125
        self.tconv1 = torch.nn.ConvTranspose1d(in_channels=256, out_channels=256, kernel_size=5, stride=2, padding=2, output_padding=1)
        self.bn5 = torch.nn.BatchNorm1d(num_features=256)

        # 250
        self.tconv2 = torch.nn.ConvTranspose1d(in_channels=256, out_channels=128, kernel_size=5, stride=2, padding=2, output_padding=1)
        self.bn6 = torch.nn.BatchNorm1d(num_features=128)

        # 500
        self.tconv3 = torch.nn.ConvTranspose1d(in_channels=128, out_channels=64, kernel_size=5, stride=2, padding=2, output_padding=1)
        self.bn7 = torch.nn.BatchNorm1d(num_features=64)

        # 1000
        self.tconv4 = torch.nn.ConvTranspose1d(in_channels=64, out_channels=32, kernel_size=5, stride=4, padding=2, output_padding=3)
        self.bn8 = torch.nn.BatchNorm1d(num_features=(32))

        # 4000
        # reduces out_channels
        self.conv5 = torch.nn.Conv1d(in_channels=(32), out_channels=1, kernel_size=1, stride=1, padding=0)



      def forward(self, input_x):
        # x dim: [batch size, feature dim=4, sentence length=4000]

        conv1_x = self.conv1(input_x)
        x = self.bn1(conv1_x)

        conv2_x = self.conv2(x)
        x = self.bn2(conv2_x)

        conv3_x = self.conv3(x)
        x = self.bn3(conv3_x)

        conv4_x = self.conv4(x)
        x = self.bn4(conv4_x)

        x = self.tconv1(x)
        x = self.bn5(x)

        x = self.tconv2(x)
        x = self.bn6(x)

        x = self.tconv3(x)
        x = self.bn7(x)

        x = self.tconv4(x)
        x = self.bn8(x)

        x = self.conv5(x)

        return x.squeeze()
```

**Model 3 (RCNN):**

For my third model, I used an RCNN. The idea behind this model was that, when examining the data, it seemed to me that data from about 100 "distance" (where "distance" is whatever this distance between points in the data is) could be useful in identifying the peaks. A RNN isn't capable to remembering dependencies for this long, so I figured that I could reduce that length of the data from 4000 down to 500 and then run a bi-directional LSTM over the data. Transpose convolutions could then be used to return the image to its original shape. Additionally, I used skip connections in an attempt to allow gradients to backpropagate as far as possible.

Novelty:

I believe this model to be fairly unique. It combines the concepts of 1d convolutions, 1d transpose convolutions, bi-direction LSTMs, and skip connects. None of these concepts where talked about much, if at all, in lecture, so I find it unlikely that many other student have a similar architecture. That said, while RCNNs are somewhat uncommon, they are still used.

Performance:

```
Loss: 0.05866293981671333
Eff: 0.7837867728378677
FP: 0.17436512697460507
```

Model:

```
[ ] class RCNN(torch.nn.Module):
        def __init__(self, hidden_size, num_layers):
          super(RCNN, self).__init__()

          # Encoder

          # 4000
          self.conv1 = torch.nn.Conv1d(in_channels=4, out_channels=64, kernel_size=5, stride=2, padding=2)
          self.bn1 = torch.nn.BatchNorm1d(num_features=64)

          # 2000
          self.conv2 = torch.nn.Conv1d(in_channels=64, out_channels=128, kernel_size=5, stride=2, padding=2)
          self.bn2 = torch.nn.BatchNorm1d(num_features=128)

          # 1000
          self.conv3 = torch.nn.Conv1d(in_channels=128, out_channels=256, kernel_size=5, stride=2, padding=2)
          self.bn3 = torch.nn.BatchNorm1d(num_features=256)

          # 500
          self.lstm_1 = nn.LSTM(256, hidden_size, num_layers, batch_first=True, bidirectional=True)

          # Decoder

          # 500
          self.tconv1 = torch.nn.ConvTranspose1d(in_channels=2*hidden_size, out_channels=128, kernel_size=5, stride=2, padding=2, output_padding=1)
          self.bn5 = torch.nn.BatchNorm1d(num_features=2*128)

          # 1000
          self.tconv2 = torch.nn.ConvTranspose1d(in_channels=2*128, out_channels=64, kernel_size=5, stride=2, padding=2, output_padding=1)
          self.bn6 = torch.nn.BatchNorm1d(num_features=2*64)

          # 2000
          self.tconv3 = torch.nn.ConvTranspose1d(in_channels=2*64, out_channels=32, kernel_size=5, stride=2, padding=2, output_padding=1)
          self.bn7 = torch.nn.BatchNorm1d(num_features=(32+4))

          # 4000
          # reduces out_channels
          self.conv5 = torch.nn.Conv1d(in_channels=(32+4), out_channels=1, kernel_size=1, stride=1, padding=0)



        def forward(self, input_x):
          # x dim: [batch size, feature dim=4, sentence length=4000]

          # Encoder
          conv1_x = self.conv1(input_x)
          x = self.bn1(conv1_x)

          conv2_x = self.conv2(x)
          x = self.bn2(conv2_x)

          conv3_x = self.conv3(x)
          x = self.bn3(conv3_x)

          # RNN
          x = x.permute(0, 2, 1)
          x, _ = self.lstm_1(x)
          x = x.permute(0, 2, 1)

          # Decoder
          x = torch.cat((self.tconv1(x), conv2_x), dim=1)
          x = self.bn5(x)

          x = torch.cat((self.tconv2(x), conv1_x), dim=1)
          x = self.bn6(x)

          x = torch.cat((self.tconv3(x), input_x), dim=1)
          x = self.bn7(x)

          x = self.conv5(x)

          return x.squeeze()
```

(Note: I also tried a fully connected network to try and get a baseline for performance. But the model failed to learn much of anything, so I decided to leave it out.)

3. Report how you performed appropriate hyperparameter tuning. List all the hyperparameters considered and the combination of hyperparameters explored. Clearly demonstrate how you selected the optimum choice of hyperparameters for your models. **[5 points]**

**(Note: code included in separate file. Also, I am counting the hidden dimension and number of layers a hyperparameters. I do this because Andrew Ng counts them as hyperparameters in the provided lecture video. Although, people could disagree about whether these should be counted as hyperparameters or whether they are more fundamental to the architecture.)**

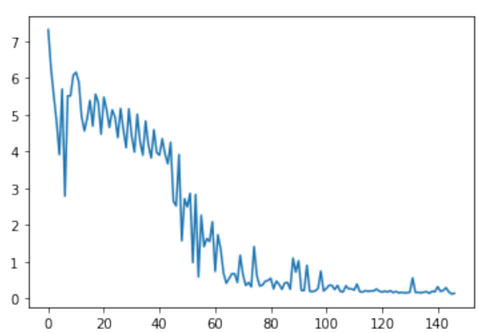
I used what is sometimes called “Grad Student Descent” or what Andrew Ng called “The Panda Approach” to select my hyperparameters. Basically, this means that I picked a set of hyperparameters, run the model, looked at the performance, and then tried to update the hyperparameters to perform better based on the results I saw from the old hyperparameters. I used this approach as opposed to grid search or random search because I didn’t feel that I had the compute available to run a large number of models. I determined which hyperparameters were best based on validation loss.

**<u>RNN:</u>**

Hyperparameters:

lr=0.001, hidden_size=64, num_layers=2

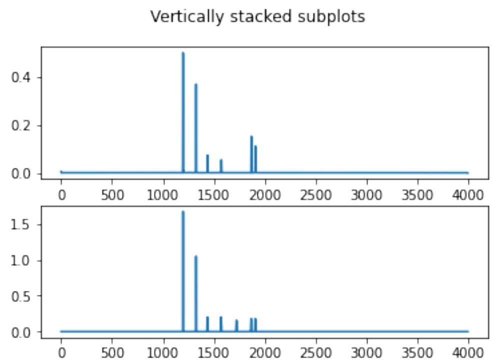
Loss:




Validation Performance:

```
Loss: 0.14711976051330566
Eff: 0.6983705669837057
FP: 0.10237952409518096
```

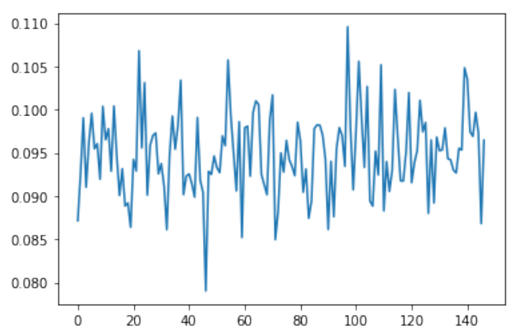
Sample Output (predicted on top):




Hyperparameters:

lr=0.005, hidden_size=64, num_layers=2

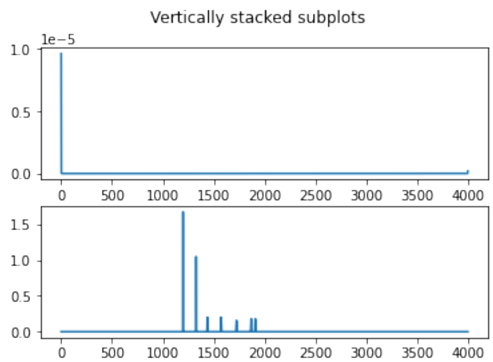
Loss:




Validation Performance:

```
Loss: 0.09495712071657181
Eff: 0.0
FP: 0.0
```

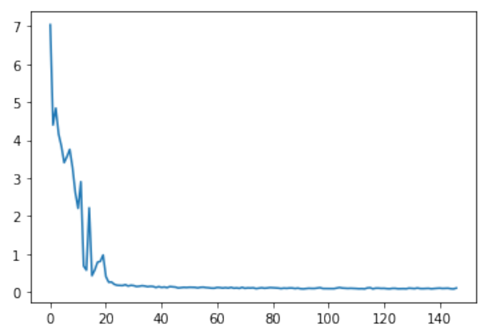
Sample Output (predicted on top):




Hyperparameters:

lr=0.001, hidden_size=96, num_layers=2

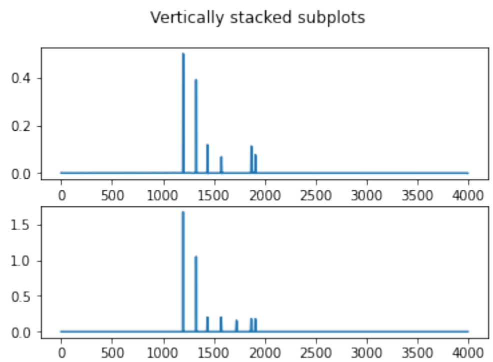
Loss:



Validation Performance:

```
Loss: 0.09109372645616531
Eff: 0.769114502691145
FP: 0.16176764647070585
```

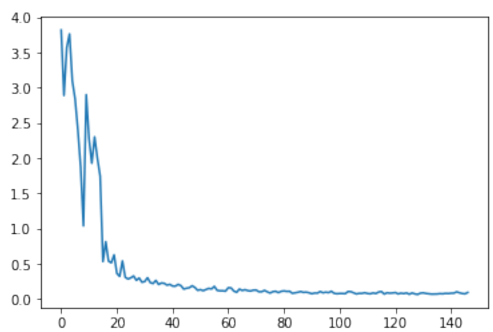
Sample Output (predicted on top):




Hyperparameters:

lr=0.0005, hidden_size=96, num_layers=2

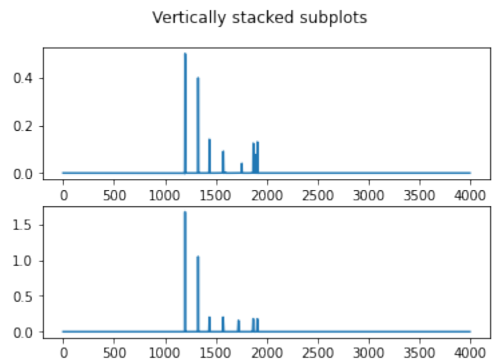
Loss:




Validation Performance:

```
Loss: 0.08754191547632217
Eff: 0.7617046376170463
FP: 0.16056788642271547
```

Sample Output (predicted on top):

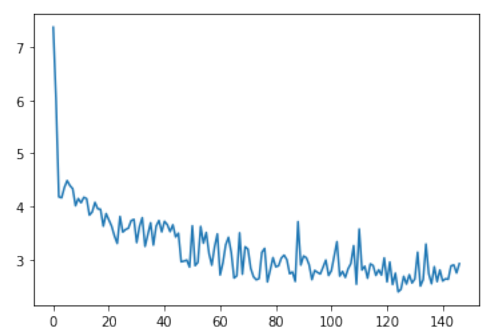



**CNN:**

Hyperparameters:

lr=0.001, beta1=0.9, beta2=0.999

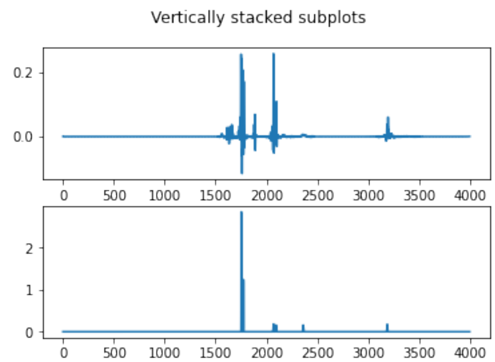
Loss:




Validation Performance:

```
Loss: 3.8393235206604004
Eff: 0.8776450637764507
FP: 1.3637272545490902
```

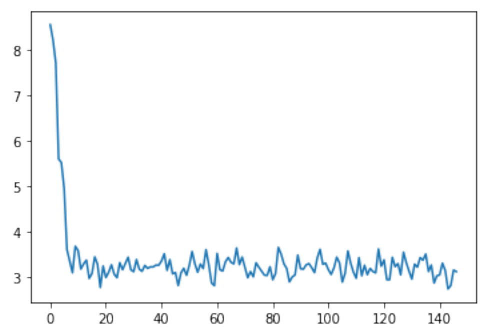
Sample Output (predicted on top):




Hyperparameters:

lr=0.0005, beta1=0.95, beta2=0.9995

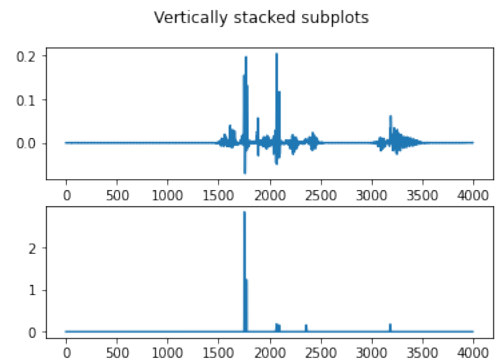
Loss:




Validation Performance:

```
Loss: 3.062436103820801
Eff: 0.8727420187274202
FP: 1.45750849830034
```

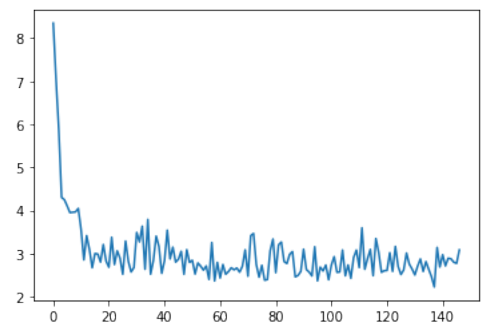
Sample Output (predicted on top):




Hyperparameters:

lr=0.001, beta1=0.9, beta2=0.9

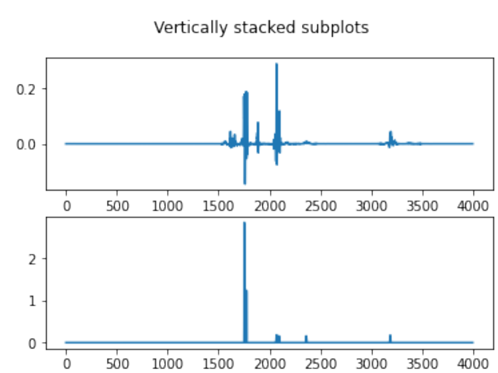
Loss:




Validation Performance:

```
Loss: 2.2011327743530273
Eff: 0.8917643589176436
FP: 1.624875024995001
```

Sample Output (predicted on top):

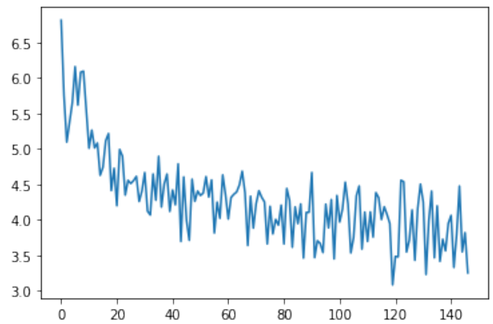


**RCNN:**

Hyperparameters:

lr=0.001, hidden_size=64, num_layers=2

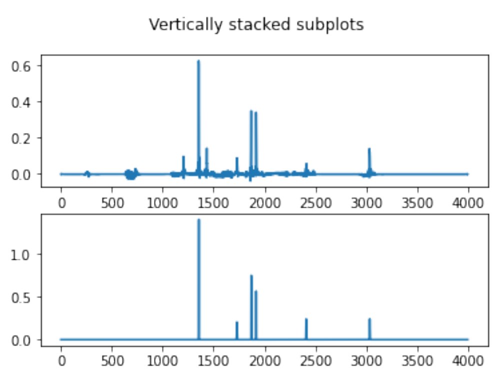
Loss:




Validation Performance:

```
Loss: 4.4561028480529785
Eff: 0.8607240286072403
FP: 1.3273345330933812
```

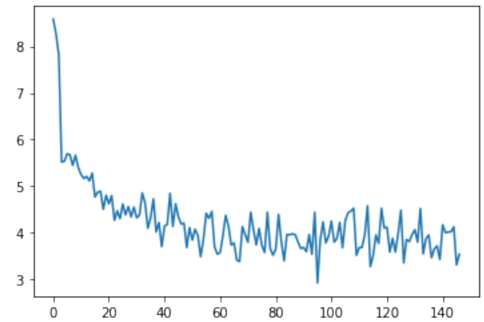
Sample Output (predicted on top):




Hyperparameters:

lr=0.0005, hidden_size=64, num_layers=2

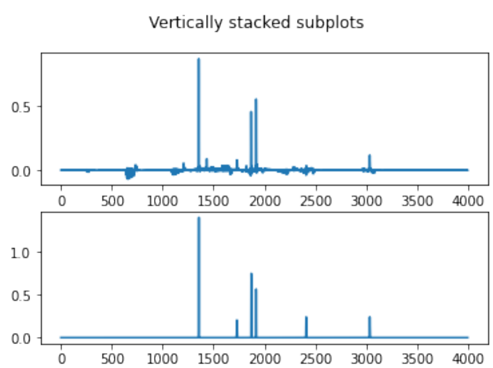
Loss:




Validation Performance:

```
Loss: 4.017274379730225
Eff: 0.8290569932905699
FP: 0.7096580683863227
```

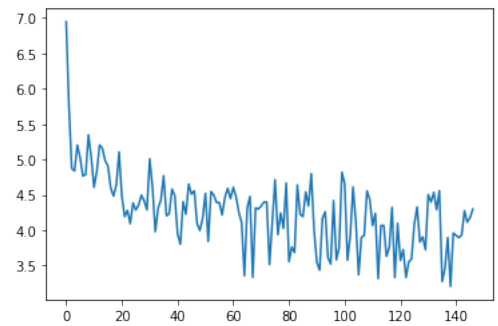
Sample output (predicted on top):




Hyperparameters:

lr=0.0005, hidden_size=96, num_layers=2

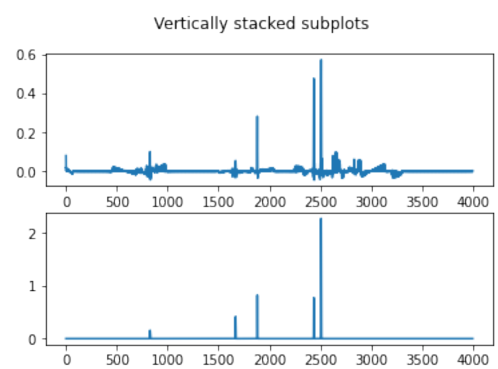
Loss:



Validation Performance:

```
Loss: 4.517522811889648
Eff: 0.8171864631718646
FP: 1.2723455308938212
```

Sample Output:




4. Show how you studied the impact of sample size on validation set efficiency for each of the models considered. **[5 points]**

To study the impact of sample size on validation set efficiency I did three runs for each model with an increasing sample size. Below are the results:

**<u>RNN (lr=0.0005, hidden_size=96, num_layers=2):</u>**

5,000 sample points:

```
Loss: 0.08754191547632217
Eff: 0.7617046376170463
FP: 0.16056788642271547
```

10,000 sample points:

```
Loss: 0.05557173117995262
Eff: 0.784597802845978
FP: 0.14037192561487702
```

15,000 sample points:

```
Loss: 0.05449014529585838
Eff: 0.7900169579001696
FP: 0.15476904619076184
```

**RCNN (lr=0.0005, hidden_size=64, num_layers=3):**

5,000 sample points:

```
Loss: 0.05866293981671333
Eff: 0.7837867728378677
FP: 0.17436512697460507
```

10,000 sample points:

```
Loss: 0.053513798862695694
Eff: 0.8207992332079923
FP: 0.21715656868626274
```

15,000 sample points:

```
Loss: 0.05029470846056938
Eff: 0.7874732728747327
FP: 0.14437112577484504
```

**CNN (lr=0.001, beta1=0.9, beta2=0.9):**

5,000 sample points:

```
Loss: 2.2011327743530273
Eff: 0.8917643589176436
FP: 1.624875024995001
```

10,000 sample points:

```
Loss: 2.7491302490234375
Eff: 0.8739954287399543
FP: 1.3117376524695061
```

15,000 sample points:

```
Loss: 2.052672863006592
Eff: 0.8903634889036349
FP: 1.5336932613477305
```

From these results, it appears that running the models on larger samples sizes tends to slightly increase their efficiency. But, my tests did not find a strong relationship between the two.