The Prediction for Terrain Identification

Colin Moore
camoore7@ncsu.edu

Yaqin Si
ysi4@ncsu.edu

Ruohan Liao
rliao6@ncsu.edu

I. Methodology

In the prediction for terrains, we composed four different neural network architectures to select the best model based on accuracy and averaged F1-score in the validation set. The models we used were: a Fully-connected Neural Network(FNN), a Convolutional Neural Network (CNN), a Long Short-Term Memory (LSTM) Neural Network, and a Transformer Neural Network(TransNN), to predict a categorical distribution over the target classes.

A FNN has an input layer, an output layer and multiple hidden layers. As indicated in the name, all neurons in each layer is fully connected to the previous layer and the next layer, leading to a large number of parameters as well as an unclear structure. A FNN can be applied to solve classification problems. [3]

A CNN adds convolutional layers in a network to extract features by applying a summary statistic to the nearby inputs. In practice, it is usually accompanied with a max pooling layer to reduce dimensions. [3] Since we have 1-d structure for terrain recognition signals as input, 1-d convolutional layers are applied in this project to do feature summary.

One shortcoming of FNN and CNN is that they don't exploit temporal information. A LSTM layer is designed to deal with sequential data like the motion signals we have in the terrain classification task. The LSTM is a type of the Recurrent Neural Networks (RNN), which utilize history information in model training. The LSTM has longer memory than the usual RNN by controlling information fade overtime. [2]

- The Fully-connected neural network
  The FNN in this project had 1 input layer (192 nodes), 2 hidden layers (100 nodes and 50 nodes), and 1 output layer(4 nodes). The activation function applied between the input layer and hidden layers was an Relu function.
- The Convolutional Neural Network The CNN model is a natural extension of the FNN, by adding 2 layers of 1-d convolutional layer as well as a corresponding max pooling layer (size 4) at the very beginning (Cov1d(16), Cov1d(64)). To control the complexity of network, a drop out rate of (0.5) was added to each convolutional layer and each dense layer.
- The Long Short-Term memory (LSTM) Neural Network
  The LSTM is also an extension of the FNN, by adding a layers of bidirectional LSTM layer at the very beginning (LSTM(100, reverse), LSTM(100)). Drop out rate (0.5) was added to each LSTM layer and each dense layer.
  Among these three architectures, only the LSTM is shown in Figure (fig. 1)
- The Transformer Neural Network
  The transformer model, as described in the article [5], uses attention mechanisms to build an encoder-decoder model which transforms an input sequence to an output sequence. We apply a dropout of 0.2 to each token of the input before each transformer encoder layer. A more detailed summary of our model's architecture can be seen in Figure 2.

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 768, 100)	40800
dropout_6 (Dropout)	(None, 768, 100)	0
lstm_3 (LSTM)	(None, 100)	80400
dense_7 (Dense)	(None, 100)	10100
activation_6 (Activation)	(None, 100)	0
dropout_7 (Dropout)	(None, 100)	0
dense_8 (Dense)	(None, 50)	5050
activation_7 (Activation)	(None, 50)	0
dropout_8 (Dropout)	(None, 50)	0
dense_9 (Dense)	(None, 4)	204

Total params: 136,554

Fig. 1. The architecture of LSTM

II. Model training and selection

A. Data Preparation

The training process was based on labeled dataset. Considering the time-series data [4] specifically, we extracted features based on information from the current moments as well as the "borrowed" information from the previous time periods by grouping some number of signals (X) in the time window immediately before each y label. Considering that the changing of terrains would usually happen in seconds, and the frequency of signals (X) was 40Hz, we arbitrarily chose a window size of 128, which was the number of the time steps of X data grouped for each label y.

```
TransformerModel(
  (pos_encoder): PositionalEncoding(
    (dropout): Dropout(p=0.2, inplace=False)
  )
  (transformer_encoder): TransformerEncoder(
    (layers): ModuleList(
      (0): TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=200, out_features=200, bias=True)
        )
        (linear1): Linear(in_features=200, out_features=200, bias=True)
        (dropout): Dropout(p=0.2, inplace=False)
        (linear2): Linear(in_features=200, out_features=200, bias=True)
        (norm1): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.2, inplace=False)
        (dropout2): Dropout(p=0.2, inplace=False)
      )
      (1): TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=200, out_features=200, bias=True)
        )
        (linear1): Linear(in_features=200, out_features=200, bias=True)
        (dropout): Dropout(p=0.2, inplace=False)
        (linear2): Linear(in_features=200, out_features=200, bias=True)
        (norm1): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((200,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.2, inplace=False)
        (dropout2): Dropout(p=0.2, inplace=False)
      )
    )
  )
  (encoder): Linear(in_features=6, out_features=200, bias=True)
  (decoder): Linear(in_features=200, out_features=4, bias=True)
)
```

Fig. 2. The architecture of the transformer model

B. Training and Validation set

Once the data was prepared as described above, we used a simple random selection to segment the data into a training (80%) set and a validation(20%) set.

C. Dealing with the imbalanced labels in the training set and the validation set

The dataset has imbalanced number of labels between classes. Two different methods were considered to deal with this problem. (1) One method was to use an effective number-based weighted cross-entropy in measuring the loss [1]. The power term (P) allows for adjustment of relative weights between classes. P¿1 would increase the difference between the common classes and the rare classes, comparing to P¡1.

$$\text{effective number of class i} = [\frac{1}{\text{Number of samples of class i}}]^P$$

$$\text{weight for sample} = 4 \times \frac{\text{effective number of class i}}{\sum_{i=1}^{4} \text{effective number of class i}}$$

(2) Another method that we considered was a probability based re-sampling method. To be specific, we can calculate a re-sampling probability for each of the class in this way:

$$\text{resample\_prob\_class\_i} = \frac{\text{min\_class\_counts}}{\text{class\_x\_counts}} * a$$

For the re-sampling process, we generated a random number between 0 and 1, and once the re-sampling probability was greater than this random number, we dropped this data point. Otherwise, we added this record to the final input of our model. In this way, the class with large number of instances would be effectively decreased and then our input data would have rather balanced proportion for each of the class. Specifically, if we strictly limit our multiplier to be 1, then each terrain label after re-sampling would consist of around 1/4 of the input. However, this method would drop huge number of instances for the class with the largest proportion, when the original data was quite imbalanced. Thus, to avoid serious dropping issue which might cause over-waste of our data points, here we proposed to add an adjusting multiplier $\boldsymbol{a}$ in the above equation, which could be used to control the maximum number of the non-minority labels we would like to save in our final input. In the training process, the multiplayer is set equal to 1.5 to allow us to have a slightly skewed dataset.

D. Model training

1) FNN, CNN and LSTM Model Training: In training FNN, CNN and LSTM, the batch size was set as 512; the total number of epochs as 20; the optimizer as Adam with learning rate=0.0001. To deal with imbalanced data, probability based re-sampling strategy with multiplier=1.5 was applied.

The training loss is shown below in Figure 3.

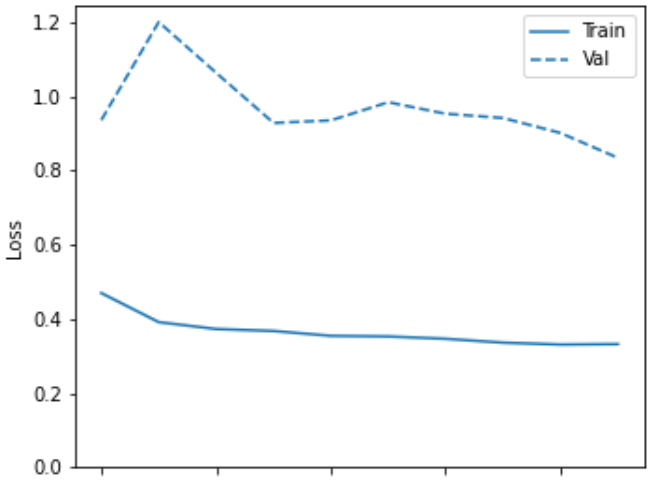

Fig. 3. Training loss for the LSTM model

2) Transformer Model Training: Because of the somewhat peculiar architecture of the transformer model, it required a slightly different approach to training. Additionally, we used an initial self-supervised pre-training step to initially train the transformer.

For all training steps, the batch size was set to 100, with a total of 200 epochs. Additionally, this model used an SGD optimizer with an initial learning rate of 0.5, and a learning rate decay of 0.95 each epoch.

For the self-supervised pre-training step, the transformer was trained to predict the data values both two and four time steps in the future. This approach was designed to be somewhat similar to autoencoder pre-training approaches which have previously been shown to have good results for generalization. A cursory evaluation of the model's results was not highly promising: the model's error would be very low at the beginning and end of each training sample, when the subject was not moving much, but then would have a very high error through the bulk of the training sample. Regardless of this, we used this model as a starting point for future models,

as later models trained in this manner converged significantly faster than models without any pre-training.

For training the classification models, the decoder portion of the self-supervised transformer model was replaced with a classification head, and trained to predict the terrain classification at each timestep. Although the transformer model was trained to output a prediction for each of the 128 input tokens, we only used the prediction from the last token in the sequence when performing predictions on the test set.

III. Evaluation

Table I shows the overall classification accuracy of the best performing models of several different varieties. Surprisingly, the Transformer model trained without weighted cross-entropy very slightly outperformed the Transformer with identical architecture on the validation set. This held true for both classification accuracy and F1 scores.

TABLE I
VALIDATION SET ACCURACY OF BEST PERFORMING MODELS

Model	Accuracy
FNN	0.73
CNN	0.90
LSTM	0.92
Transformer-Unweighted	**0.8493**
Transformer-Weighted	0.8486

A. LSTM Model

Among the FNN, CNN and LSTM model, LSTM model with re-sampled data achieved the best accuracy and F1-score in the validation set.

TABLE II
PERFORMANCE OF THE LSTM MODEL

Class	Precision	Recall	F1-Score
0	0.99	0.90	0.94
1	0.80	1.00	0.88
2	0.79	1.00	0.88
3	0.76	0.93	0.84
macro avg	0.83	0.96	0.89

Tabel II reports the performance of the LSTM model trained. Although the LSTM model achieved the best precision and best F1-score in the validation set, the performance on the final test set is just as good as random guess.

B. Transformer Model

Table III reports the performance of the Transformer model trained with unweighted categorical cross-entropy. This was our best performing model on the final test set.

Although we had some models which attempted to handle the unbalanced classes, this model made no attempt to do so. This indicates that a better performing model may be possible to produce using the other class re-sampling methodology, however, we were unable to perform this analysis due to time constraints. In Figure 4, we can clearly see that the model still has a tendency to over predict class 0, giving strong evidence that more working in dealing with unbalanced classes would likely improve performance.

TABLE III
PERFORMANCE OF THE TRANSFORMER MODEL

Class	Precision	Recall	F1-Score
0	0.95	0.85	0.90
1	0.76	0.92	0.83
2	0.78	0.94	0.85
3	0.53	0.78	0.63
macro avg	0.76	0.87	0.80

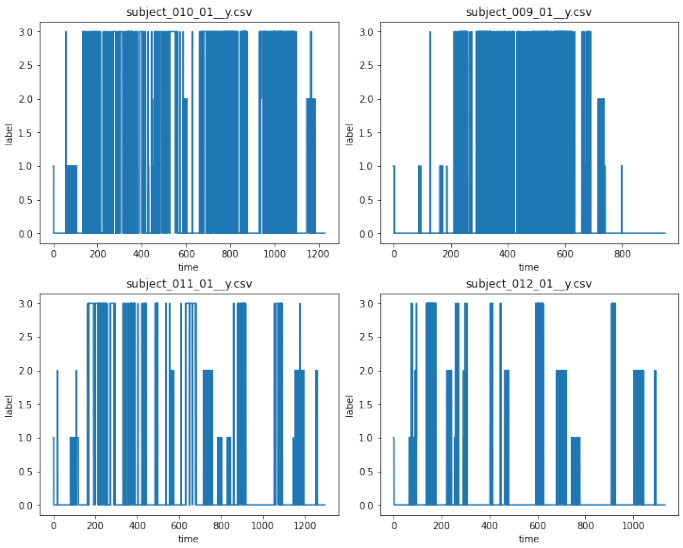

Fig. 4. Predictions on the test set for the final transformer model

References

[1] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9268–9277, 2019.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[3] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[4] Avishek Pal and PKS Prakash. *Practical time series analysis: master time series data processing, visualization, and modeling using python*. Packt Publishing Ltd, 2017.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.