ML 2023 Final Report

Dohyun Lee
2018-12663

Abstract

In this ML final project, I construct a deep learning model for non-seen hand-written letter image sorting. I use convolutional neural networks (CNNs) as an encoder, recurrent neural networks (RNNs, especially LSTM) and Transformer as a decoder to enable this task. I use ***ResNet-like CNN structure and bidirectional LSTMs*** *and find optimal hyperparameters (e.g., RNN dropout ratio, and learning rate) To explain, model, ConvLSTM processes the input images through the CNN, followed by the LSTM encoder, generating an encoded hidden state. This hidden state is passed through the LSTM decoder, which generates the final output sequence. In experiment, the best validation accuracy was* ***83.31%*** *without data augmentation and* ***83.38%*** *with data augmentation. Best training accuracy was 94.52%. Thus, it shows that model generalizes well with augmented data. In addition, I gained useful information from Chat-GPT 3.5 while planning LSTM architecture in order to successfully implement bidirectional gradient flow. ResNet-like structure is revised by Chat-GPT, but overall structure is from [1], and revised by myself. Plus, all the figures are drawn by me. Let me first explain how I made this model in details.*

1. Introduction

Non-seen hand-written letter image sorting is a task of predicting non-seen letters in alphabetical order from given 10 alphabetic handwriting images. It is important to note that output sequence can be various (e.g., 16 ~ 25). First, I combine convolutional neural networks and recurrent neural networks to enable this task. By combining the strengths of CNNs and RNNs, it is possible to leverage both local feature extraction and sequence modeling capabilities. At first, I practiced various types of convolutional neural networks as follows:

1) Very deep conv layers (e.g. *VGG*)

2) Simple conv layers (e.g. *LeNet*)

3) Efficient models (e.g. *GoogLeNet*)

4) *ResNet*-like model

It turns out to be other models 1) to 3) has low validation accuracy. In case of *LeNet*, it's a relatively small architecture, making it fast to train. Training was fast, but validation accuracy was poor. VGG has simple-to-implement structure but shallower than *ResNet*. Lastly, "inception module" of *GoogLeNet* is somewhat attractive, but training was so slow. Therefore, I use *ResNet*-like CNN model. ('-like' is added because common task for *ResNet* is very complicated where as our ML final task is simple.)

For RNN structure, I first use Simple Vanilla structure which has a simple structure, the hidden state is passed from one time step to the next. They are computationally efficient. However, they have difficulties in capturing long-range dependencies due to the vanishing gradient problem. This means that they can't effectively learn from information that is many time steps in the past. Meanwhile, LSTM usually preferred for most practical applications due to their ability to capture long-range dependencies. It has a series of gates (input gate, forget gate, and output gate) and a cell state in addition to the hidden state. So, I implement **bidirectional LSTM** in order to learn different features. It should be noted that 'time' serves almost no meaning in our task, bidirectional LSTM is usually used for tasks of text translation or sentiment analysis. But they essentially double the number of hidden units compared to a unidirectional LSTM which means that it can generalizes well. Now, I'll explain the model architecture in detail.

2. Model Architecture

2.1. CNN Architecture

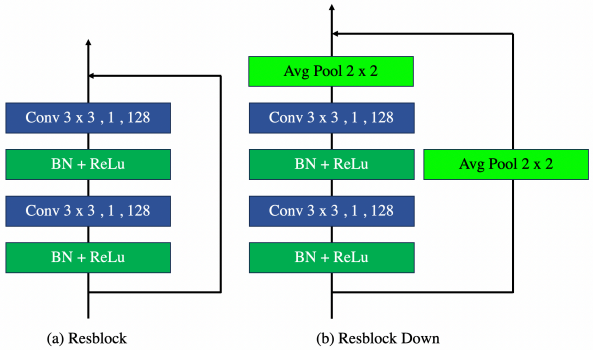

Figure. 1 Resblock and Resblock Down architectures for *ResNet*-like structure. Conv k × k, s, n denotes a convolutional network with size k × k, stride s, and n feature maps, BN is the batch normalization, and AvgPool denotes the average pooling layer with stride 2.

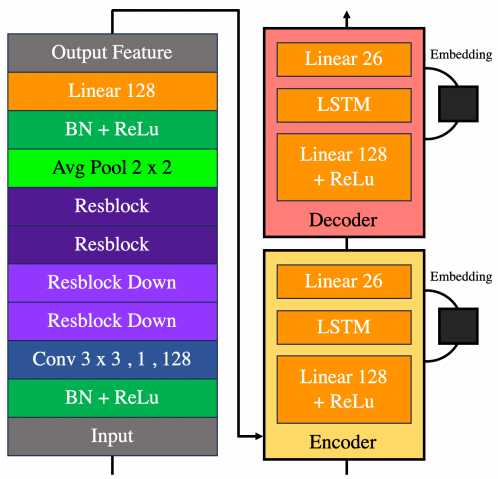

Figure. 2 Overview of 'ConvLSTM' model. *ResNet*-like Custom CNN is used for feature extraction. 2 LSTMs are used as encoder and decoder. Note that LSTM is bidirectional, so hidden dimension is 256 in total.

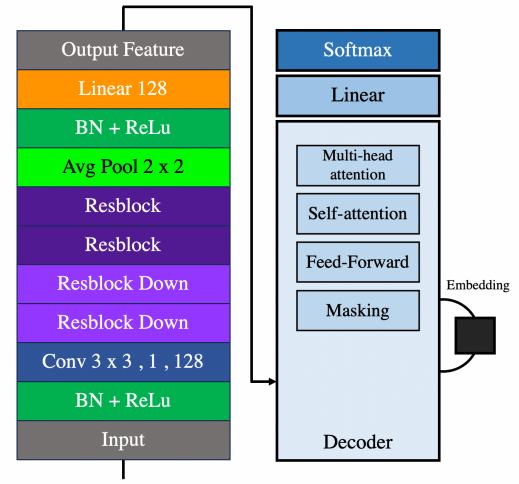

Figure. 3 Overview of 'ConvTransformerDecoder' model. Again, *ResNet*-like Custom CNN is used for feature extraction. Transformer is used as decoder; note that algorithms are fully depends on publicly released Github repositories.

As I pointed earlier, various types of CNN architectures have been used to improve train accuracy, and it turns out to be ***ResNet*-like structure** has a best performance. This is because *ResNet* (Residual Network) address the problem of vanishing gradients in deep networks. It was proposed by Kaiming He et al. in the paper "Deep Residual Learning for Image Recognition" [1]. The key idea behind *ResNet* is the use of residual connections or skip connections, which enable the network to learn residual mappings. These connections allow the network to bypass certain layers and propagate the gradient more effectively during training. (In our final term exam, residual connection is compared with '*A Highway*'.) **Figure. 1** illustrates the Resblock and Resblock Down architectures for *ResNet*-like structure.

2.2. RNN Architecture

Here, 2 LSTM layers used as an **encoder-decoder** architecture which allows the model to learn and capture both spatial and temporal dependencies in the input sequence, making it suitable for tasks such as sequence prediction or sequence generation. Encoder is an LSTM layer responsible for processing the CNN output and capturing the temporal dependencies in the input sequence. During the forward pass, the encoder takes the CNN output, hidden state, and cell state, and updates its hidden and cell states using the forward method of the encoder LSTM layer:

```
_, hidden_state, cell_state =
  self.encoder(out, hidden_state, cell_state)
```

The decoder is also an LSTM layer responsible for generating the predictions for the output sequence based on the hidden states from the encoder. During the forward pass, the decoder takes the inputs (initialized as a tensor of zeros) and the hidden and cell states from the encoder. The forward method of the decoder is called as follows:

```
out, _, _ = self.decoder(inputs, hidden_state,
cell_state)
```

2.3. Transformer Architecture

As illustrated in **Figure. 3**, Instead of usage LSTM, I define a '**ConvTransformer Decoder'** which is used in sequence2sequence problems. The CNN architecture first extracts features and then feeds those features into a transformer decoder for further processing. Here, substructures of **ConvTransformer Decoder** are summarized:

1. CNN Layer: Extracts important features from the input data.
2. Transformer Decoder: Processes the extracted features further.
3. Subsequent Masking: Ensures that in the decoder the predicted output can't peek into future tokens.
4. Forward Method: Defines the steps performed in the forward pass of the model.
5. Embedding: Converts categorical data into dense vectors.
6. Position-wise Feed-Forward Networks: Applies the same MLP to each position separately and identically.
7. **Attention Mechanism**: A mechanism in the Transformer model to weigh the input features.
8. Multi-Headed Attention: The mechanism to allow the model to focus on different positions.

9. Linear Transformation and Softmax: The final layers in the Transformer model.

As discussed in the lecture, the use of the transformer's self-attention mechanism allows the model to consider the entire input sequence simultaneously, which can potentially improve performance on tasks where the relationship between elements is important. Self-attention can be calculated using equations as below:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathbf{T}}}{\sqrt{\mathbf{d_k}}}\right)\mathbf{V}$$

```
   def attention(query, key, value, mask=None,
dropout=None):
   d_k = query.size(-1)
   scores = torch.matmul
   (query, key.transpose(-2, -1)) / math.sqrt(d_k)
   if mask is not None:
   scores = scores.masked_fill(mask == 0, -1e9)
   p_attn = F.softmax(scores, dim=-1)
   if dropout is not None:
   p_attn = dropout(p_attn)
return torch.matmul(p_attn, value)
```

Where Q is query, K is key, and V is Value. The softmax function is applied along the last dimension of the dot product result to obtain attention weights. These weights are then used to weight the values V, resulting in the final attended output.

2.4. Data augmentation

Data augmentation is an important technique for improving generalization in machine learning models. It involves applying various transformations or modifications to the training data, creating additional augmented samples that are similar to the original data but exhibit variations in different aspects. As illustrated in **Figure. 4**, I applied *RandomAffine* (Applies random affine transformations to the image.) and *RandomInvert* (Randomly inverts the colors of the image, but note that gray scale image is used). **The validation accuracy increased by 0.07%**, which is insignificant, but it does shows it improves generalization.

2.5. Optimization

I used ***Adam* optimizer and its adaptive learning rates** for each parameter. Since Adam keeps track of an exponentially decaying average of past gradients and the square of these gradients, it uses to adapt the learning rate for each weight in the model. **Learning rate schedulers** can be used because they adjust the learning rate during training to balance the speed of convergence and the risk of overshooting the global minimum. I tried *CosineAnnealing WarmRestarts, ReduceLROnPlateau, ExponentialLR*. Two of them except *ReduceLROnPlateau* are based on mathematics equation. I founded out that two schedulers are somewhat inflexible; reducing the validation accuracy. Besides, *ReduceLROnPlateau* reduces the learning rate when a metric has stopped improving, so it's dynamic. I wanted to use it but many errors had occurred while implementing it. So, **learning rate schedulers are NOT used.**

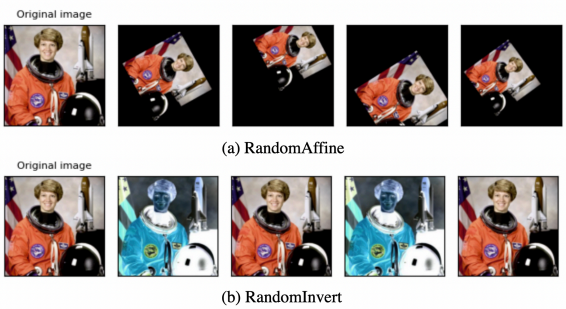

Figure. 4 Data augmentation techniques used in this paper (Captured from [2])

2.6. Regularization

I used **Dropout and L2 regularization** for regul -arization techniques. I set the value of **drop out ratio = 0.1** (both for RNN and CNN). In the lecture, we have taught that drop out ratio is typically 0.5, but small ratio was enough. It prevents overfitting by randomly dropping out neurons during training. By introducing noise into the training process, model's reliance on any single neuron and encourages the network to learn more robust and generalizable features. I set **l2 regularization coefficient = 0.00005.** It adds a penalty term to the loss function that is proportional to the sum of the squares of the model parameters. This reduces the model's complexity and make it less likely to overfit.

2.7. Learning rate and hidden dimension

The choice of learning rate is a crucial aspect of training a machine learning model, especially for gradient-based optimization algorithms. A good learning rate is often essential for convergence and to ensure that the training process is not too slow or unstable. In the lecture, we have taught that Learning rate is the most significant factors in training period. If time allows, I would have tried hundreds of various learning rates; I used a typical value of **learning rate = 1e-3.** I tried high learning rate (e.g., 1 or 0.1), the learning algorithm overshoot the optimal point in the parameter space. It results in divergence, since I don't used learning rate scheduler. Also, when learning rate is too low (e.g., 1e-6 or 1e-5), the learning process becomes very slow (almost 6 hours per 1 training).

Also, hidden dimension of LSTM (i.e. hidden state dimension) is considered to be another crucial factors. For **hidden dimension of Bidirectional LSTM is chosen to be 128** (256 in total). If the hidden dimension is too large, model may overfit. If dimension is too small, RNN will suffer from an information bottleneck which causes high bias.

3. LSTM vs Transformer

I have attached Screenshots below this part. At 15th epoch, 'ConvLSTM' model shows **82.21%** of valid accuracy, while 'ConvTransformerDecoder' shows **82.99 %** of valid accuracy. In fact, two models are similar. Because in 'ConvLSTM' model, the CNN-LSTM is used as an encoder-decoder setup, making the LSTM encoder-decoder responsible for predicting the next step based on the learned information. Only difference lies in the application of the Transformer Decoder instead of an LSTM for processing the sequential data. 'ConvTransformerDecoder' shows better accuracy because of its **self-attention mechanism.** It captures more complex dependencies in the input data than the LSTM in 'ConvLSTM'. Despite this, 'ConvLSTM' is also good and efficient because bidirectional LSTM cells have their strengths, such as being able to maintain and manipulate a "cell state" to retain long-term information. Also, 'ConvLSTM' was **easy to train** (Specifically, it takes 1 hours lesser than Transformer) and less computationally expensive than Transformer models.

4. Screenshots of Validation accuracy.

Figure. 5 ConvLSTM w/o data augmentation.

Figure. 6 ConvLSTM with data augmentation.

```
Valid epoch: 15, Valid accuracy: 80.44, Best valid accuracy: 82.21
epoch: 16 step: 15 loss: 0.2335292398929596 accuracy: 90.96808605209353
epoch: 16 step: 30 loss: 0.2691846787929535 accuracy: 90.74237360976296
epoch: 16 step: 45 loss: 0.2822820246219635 accuracy: 90.93388405252912
epoch: 16 step: 60 loss: 0.27779725193977356 accuracy: 90.72118325208977
epoch: 16 step: 75 loss: 0.33089587092399597 accuracy: 90.33202169394224
epoch: 16 step: 90 loss: 0.29598692059516907 accuracy: 90.13556766643185
epoch: 16 step: 105 loss: 0.24717019498348236 accuracy: 90.1448784207405
```

Figure. 7 ConvLSTM at epoch 15.

```
Valid epoch: 15, Valid accuracy: 79.07, Best valid accuracy: 82.99
epoch: 16 step: 15 loss: 0.39314424991607666 accuracy: 87.83721722641145
epoch: 16 step: 30 loss: 0.3471207022666931 accuracy: 88.41199173290467
epoch: 16 step: 45 loss: 0.40269342064857483 accuracy: 88.46510892966481
epoch: 16 step: 60 loss: 0.45323801040649414 accuracy: 88.12891946052666
epoch: 16 step: 75 loss: 0.4174362123012543 accuracy: 87.97197283831328
epoch: 16 step: 90 loss: 0.31702807545661926 accuracy: 88.07051251972608
epoch: 16 step: 105 loss: 0.38269898295402527 accuracy: 88.07450575141033
epoch: 16 step: 120 loss: 0.35226812958717346 accuracy: 88.16645354978596
epoch: 16 step: 135 loss: 0.32585930824279785 accuracy: 88.25846046402842
```

```
# load and evaluate ConvTransformerDecoder model
load_path = './model_ConvTransformerDecoder.pt'
eval(valid_dl, load_path)
```

Figure. 8. ConvTransformerDecoder at epoch 15.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py, Retrieved June 21, 2023.