NLP Mid-term Project: Language Model

18340146 Computer Science and Technology 宋渝杰

Contents of research:

Use the word segmentation model trained in the previous experiment to re-segment the text in the training set and test set, train a simple LSTM language model, and use the trained model to generate text.

Plans of research/Design of experimental procedures:

I personally think that "Plans of research" and "Design of experimental procedures" are equivalent to **"Experimental process"**.

Since I learned neural networks in the "Artificial Intelligence" course, the main process of my research in this course project is relatively clear:

1. Re-segmentation: Use the BiLSTM+CRF model of the previous experiment to re-segment the training set `msr_training.utf8` (generate the training set `trainset.txt` of this experiment)
2. Generate tags: Generate a corresponding tag sequence for each word sequence in the training set. The specific method is: for each word in this word sequence, the label is the next word (the label of the last word in this word sequence is the end label: `<stop>`)

For example: the first line is the word sequence, and the second line is the corresponding label sequence:

这	次	作业	好	难
次	作业	好	难	<stop>

3. Establish an LSTM model (recurrent neural network), and use the word sequence and label sequence of the training set to train the network
4. After training for a certain number of times, verify the model (input a word as the beginning and observe what sentence it generates)

Statement of Principle:

The following will mainly explain the construction and operation principle of the LSTM language model, and then the model training and testing process

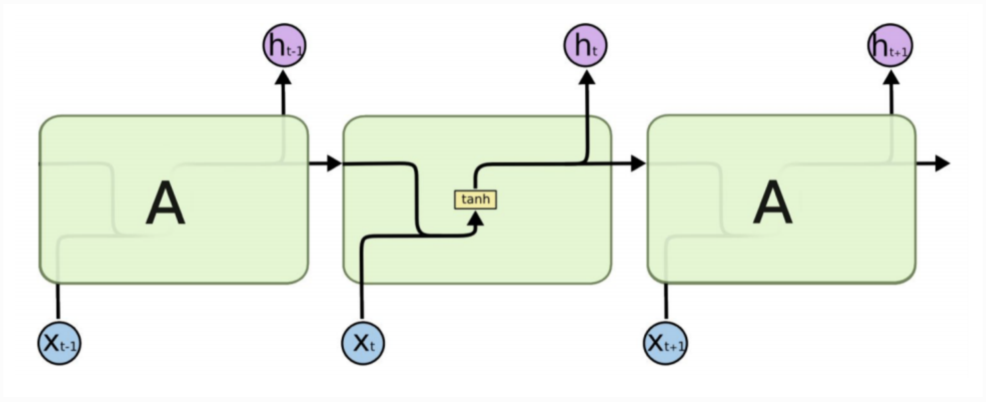
Recurrent Neural Network (RNN) :

RNN is a special kind of neural network (essentially a kind of mathematical model). Like ordinary neural networks, it is divided into input layer, hidden layer, and output layer. But the hidden layer is special, it will accept the last output of the neuron h_{t-1} as the second input and the original input x_t, which together affect the current output(h_t) of the neuron.

Standard RNN:

A standard RNN model consists of an input neuron, a hidden neuron, and an output neuron. However, due to its cyclic characteristics, it can be expanded into multiple input, hidden, and output neurons, and the number of expansions is the same as the dimension of the input data (in this experiment, it can be understood as the length of a sentence/the number of words).

The processing of hidden neurons is similar to the fully connected layer: the input x and the input h_{t-1} are used to form a fully connected layer $\mathrm{W}_{xh}\mathrm{x}_t$ and $\mathrm{W}_{hh}\mathrm{h}_{t-1}$, and then calculate the sum of the above two results with the two offsets b_x and b_h respectively, finally the output h_t is obtained through the activation function (usually tanh).



$$\mathrm{h}_t = \tanh(\mathrm{W}_{xh}\mathrm{x}_t + \mathrm{b}_x + \mathrm{W}_{hh}\mathrm{h}_{t-1} + \mathrm{b}_h)$$

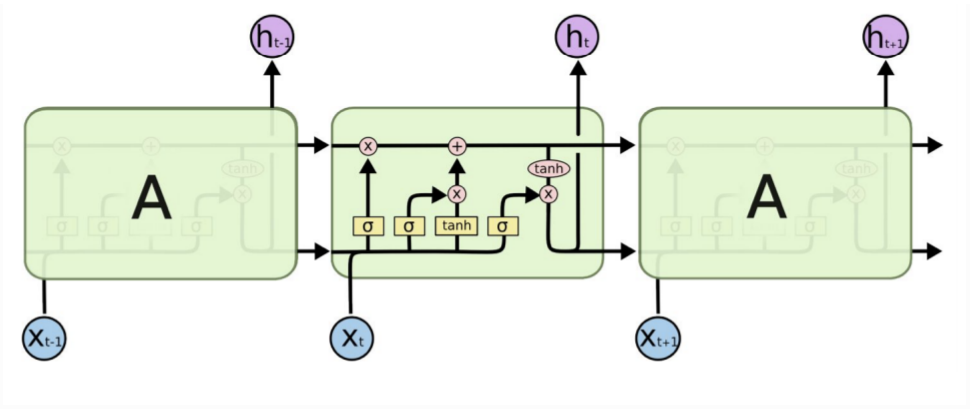
The processing of the output neuron is also similar to the fully connected layer: the input h_t is made into a fully connected layer to form $\mathrm{W}_{hz}\mathrm{h}_t$, and then sum it with the offset b_z, finally use the activation function (generally not to be used) to get the output z_t, which is the non-normalized probability of each label of the word.

$$\mathrm{z}_t = \mathrm{W}_{hz}\mathrm{h}_t + \mathrm{b}_z$$

Long Short-Term Memory Network (LSTM):

In the standard RNN, this repeated structural module has only a very simple tanh layer, and this simple level of RNN can only be inferred based on the previous relatively recent information, but if you need to use relatively distant information, the standard RNN is very Difficult to achieve results.

Long Short-Term Memory Network (LSTM) is a special RNN network to solve the long-term dependency problem. Compared with standard RNN, its hidden layer has four network layers:



Cell state: The LSTM not only transmits h_{t-1} cyclically, but also transmits a "cell state" C_{t-1} to save the historical information of the sequence. Cell state usually changes very slowly, so it can also save long-distance information.

Forgotten Gate: The first layer of LSTM decides to discard certain information about the cell state. It outputs a vector f_t between 0~1 by looking at the information of h_{t-1} and x_{t-1}. The value of 0~1 in this vector indicates which information in the cell state C_{t-1} needs to be retained or discarded. 0 means not to retain, and 1 means to retain all.

After the vector is obtained, it is **multiplied** by the cell state, which is to realize the forgetting operation of information.

$$f_t = \mathrm{sigmoid}(\mathrm{W}_f[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_f)$$
$$C'_{t-1} = C_{t-1} \cdot f_t$$

Input gate: The second and third layers of LSTM decide to add some new information to the cell state. It outputs a vector i_t between 0 and 1 and new candidate cell information $\tilde{C}_t$ by looking at the information of h_{t-1} and x_{t-1} , and then the value of 0~1 in the vector indicates which information in the new candidate cell information $\tilde{C}_t$ needs to be added and how much it needs to be added.

After getting the amount of added information, it is **summed** with the cell state, which is to realize the input operation of information:

$$i_t = \mathrm{sigmoid}(\mathrm{W}_i[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_i)$$
$$\tilde{C}_t = \tanh(\mathrm{W}_c[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_c)$$
$$C_t = C'_{t-1} + i_t * \tilde{C}_t$$

After the forget gate and input gate work, the cell state has also been **updated**.

Output gate: The fourth layer of LSTM, output h_t. It outputs a vector O_t between 0 and 1 by looking at the information of h_{t-1} and x_{t-1}, then the 0~1 value in the vector indicates which information in the new cell state C_t needs to be output and how much needs to be output.

$$O_t = \mathrm{sigmoid}(\mathrm{W}_o[\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_o)$$
$$\mathrm{h}_t = O_t * \tanh(C_t)$$

So in general, the steps of a neuron in LSTM can be summarized into four steps: **forget, input, update, and output.**

LSTM language model:

After introducing the principle of LSTM, my LSTM language model is established as follows:

- LSTM layer: input x_t (the word sequence of the training set), output h_t (the captured feature information)
- Fully connected layer: input h_t (feature information captured by the LSTM layer), output z_t (the non-normalized probability of the next word for each word in the word sequence)

Therefore, the LSTM language model outputs the non-normalized probability of the next word of each word in the word sequence, and then the non-normalized probability can be taken max to obtain the word sequence with the highest probability, which is spliced to form a sentence.

Training:

After establishing the LSTM language model, the training process is as follows:

- Input a line of sentence (word sequence), the language model returns the non-normalized probability of the next word of each word in the word sequence
- Calculate cross entropy: calculate the cross entropy loss function according to the actual label sequence:
    - First calculate LogSoftmax, that is, use Softmax for normalization and then take the logarithmic form
    - Then calculate NLLLoss, that is, take the log-normalized probability of its actual label, and take the negative value to become the loss function value
    - After calculating the loss function value of each word, take the average value to become the final loss function value of the sentence (word sequence)

    $$\mathrm{loss} = -\sum_i \log\left(\frac{\exp(x_i[class_i])}{\sum_j \exp(x_i[j])}\right)/len(x)$$

    Where x is the input sentence (word sequence), x_i is the ith word, $class_i$ is the true label of the ith word, and $x_i[class_i]$ is the LSTM language model prediction The unnormalized probability of i words with label $class_i$

- Gradient descent reverse update: use stochastic gradient descent method to update the network parameters

Verification:

There are some differences between verification and training. The specific steps are as follows:

- Enter a starting word (a sequence of words with only one word), and the LSTM language model returns the non-normalized probability of the next word of this word
- Take max for the non-normalized probability to get the word with the highest probability, that is, the next word of the starting word
- Add this word to the word sequence, and then use the word sequence as input. The LSTM language model returns the non-normalized probability of the next word of the two words (the non-normalized probability of the next word output by the starting word will be the same The output of the first step is exactly the same)

- Take the non-normalized probability of the next word of the second word output by the LSTM language model, and take max to get the word with the highest probability, that is, the third word
- Add the third word to the word sequence, and then use the word sequence as input...
- Repeat the above process until the LSTM language model returns the non-normalized probability of the next word. Take max and get the "end" tag <stop>, that is, end the verification process, output the final word sequence, that is, enter the starting word and return sentence

Explanation of core code:

The explanation of the code in this article is mainly in the form of **comment**, and basically all key codes will be commented.

Re-segmentation of training set: The model is the model of the previous experiment

```
# Only extract the key code for display, please move to fenci.py for the complete code

def test_data(root): # data without spaces
    f = codecs.open(root, 'r', encoding = 'utf-8')
    data = []
    for s in f.readlines():
        data.append(list(''.join(s.strip('\n').split())))
    return data

test = test_data('msr_training.utf8') # data without spaces

model = torch.load('model.pkl').to(device) # GPU

data = []
for i in range(len(test)):
    if (len(test[i]) == 0): continue # there are empty lines in the training set
    sentence = pre(test[i], word_to_ix).to(device) # sentence->numerical sequence
    l, s = model(sentence), '' # the model that passed the previous experiment
    for j in range(len(l)): # reparticiple
        if l[j] == 0 or l[j] == 3: s = s + test[i][j] + ' '
        else: s = s + test[i][j]
    data.append(s)

def text(filename, data): # new data set write file
    file = open(filename,'w')
    for i in data: file.write(i+'\n')
    file.close()

text('trainset.txt', data)
```

Read in the data set:

```
def read(root): # training set data + label
    f = codecs.open(root, 'r', encoding = 'utf-8')
    data = []
    for s in f.readlines():
        data.append(s.strip('\n').split()+["<STOP>"]) # add a closing tag at the end
    return data

train = read('trainset.txt')[1000:3560] # optimization: take only part of the data
```

LSTM language model:

```
class myLSTM(nn.Module):

    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(myLSTM, self).__init__()
        self.word_embeds = nn.Embedding(vocab_size, embedding_dim) # word embedding
        # self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers = 1, batch_first = True)
        self.lstm = nn.GRU(embedding_dim, hidden_dim, num_layers = 1, batch_first = True) #
optimization: use GRU model
        self.hidden2tag = nn.Linear(hidden_dim, vocab_size) # the fully connected layer
outputs the predicted score of each label

    def forward(self, sentence):
        embeds = self.word_embeds(sentence).unsqueeze(dim=0)
        lstm_out, _ = self.lstm(embeds)
        lstm_feats = self.hidden2tag(lstm_out.squeeze())
        if len(sentence) == 1: lstm_feats = lstm_feats.unsqueeze(0) # when the sentence
length is 1, add one dimension
        return lstm_feats

model = myLSTM(len(word_to_ix), EMBEDDING_DIM, HIDDEN_DIM).to(device)
```

Training:

```
def pre(seq, to_ix): # sentence->numerical sequence
    seqs = torch.tensor([to_ix[w] if w in to_ix else 0 for w in seq[0:len(seq)-1]], dtype =
torch.long)
    tags = torch.tensor([to_ix[w] if w in to_ix else 0 for w in seq[1:]], dtype = torch.long)
# the tag sequence is the sentence left by one position
    return seqs.to(device), tags.to(device)

loss_func = nn.CrossEntropyLoss() # loss function: cross entropy
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9, weight_decay = 5e-4) #
optimization method: SGDM
progressive = tqdm(range(100), total = 100, ncols = 50, leave = False, unit = "b") # visual
progress bar

bestAcc, Acc = 1e9, []
for epoch in progressive:
    if epoch == 40: optimizer = optim.SGD(model.parameters(), lr = 0.001, momentum = 0.9,
weight_decay = 5e-4) # optimization: the learning rate drops in stages
    if epoch == 80: optimizer = optim.SGD(model.parameters(), lr = 0.0001, momentum = 0.9,
weight_decay = 5e-4)
    ans = 0.0
    for i in range(len(train)):
        sentence_in_pad, targets_pad = pre(train[i], word_to_ix) # get training data and
labels
        out = model(sentence_in_pad) # forward spread
        loss = loss_func(out, targets_pad)
        optimizer.zero_grad() # BPNN normal steps
        loss.backward()
        optimizer.step()
        ans += loss.item()
    print('\n', ans/len(train))
```

Testing:

```
def test(l): # enter the list, such as:['我们']
    count = 30;
    while count:
        count -= 1 # the generated sentences are up to 30 words (anti-loop)
        sentence = torch.tensor([word_to_ix[w] if w in word_to_ix else 0 for w in l], dtype =
torch.long).to(device)
        out = model(sentence) # forward spread
        _, predicted = torch.max(out.data, 1) # return the maximum value and its index in
each row
        l.append(ix_to_word[predicted.cpu().numpy()[-1]]) # add the next word generated to
the input list
        if l[-1] == "<STOP>": break
    return l

print(''.join(test(["我们"])))
print(''.join(test(["一些"])))
print(''.join(test(["发展"])))
```

Optimizations:

After implementing the basic part, I adopted the following optimization plan:

Innovation in model structure and method:

Processing of data set

This time the data set is large (86000+ sentences, 90000+ words), the output neurons of the final fully connected layer of the neural network model are also tens of thousands of times larger than the previous experiment (4 -> 90000+). Therefore, using this training set for neural network training will be very time-consuming (it takes 90 hours for 80 iterations of full-data training for the pro-test, about 4 days). If you add a series of tests such as parameter tuning, the time is definitely not enough.

Therefore, this experiment only uses part of the data set for experimentation, specifically 1000-3560 rows of the data set (the first few hundred rows of data are of poor quality and are all meaningless quotation marks at the beginning)

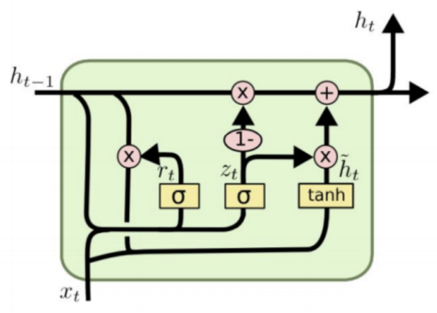
Let the learning rate drop in stages

When a certain learning rate becomes stable in the verification index score, the learning rate is divided by 10 to continue learning.During the test, it was found that a certain index score was improved, so this test was optimized by this scheme, and simplified to: the learning rate of the first 40 times is 0.01, the learning rate of 40-80 times is 0.001, and the learning rate of 80-100 times is 0.0001, a total of 100 iterations

```
    optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9, weight_decay = 5e-4)
# optimization method: SGDM
    if epoch == 40: optimizer = optim.SGD(model.parameters(), lr = 0.001, momentum = 0.9,
weight_decay = 5e-4) # optimization: the learning rate drops in stages
    if epoch == 80: optimizer = optim.SGD(model.parameters(), lr = 0.0001, momentum = 0.9,
weight_decay = 5e-4)
```

Gated Recurrent Unit (GRU):

GRU is a variant of LSTM. It combines the forget gate and input gate of LSTM into a single **update gate**, combining the cell states C_{t-1} and h_{t-1}, although the principles and results are not much different from LSTM, the parameters of the **model are reduced by about a quarter, and the training is faster**, so it has gradually begun to be an optimized substitute for LSTM.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Among them, σ is the activation function $\mathrm{sigmoid}$, formulas 1～3 omit the expression of offset

Since the principle of GRU and LSTM is similar (but the gate operation is different), this article will not repeat its principle part

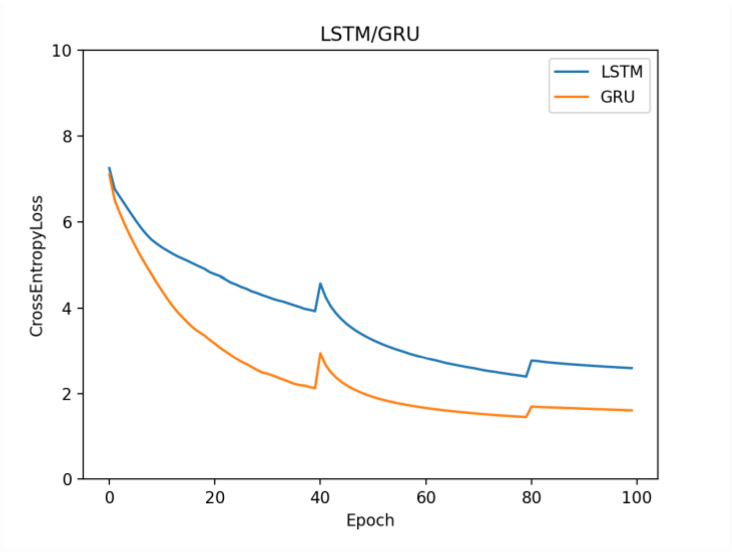
The optimized results are described below

Results of experimental:

Experimental results of LSTM and GRU:

Run the experimental source code (the code for training the model is train.py, the code for reading the model and outputting the test results is test.py), train iteratively 100 times (about 1 hour) under GPU (Testa T4), configure the hyperparameter combination And the cross entropy results during training are as follows:

Hyperparameter	Value
Word vector dimension	256
Hidden layer dimensions	1024
Optimization method	SGDM
Learning rate	0.01～0.0001
Number of training iterations	50

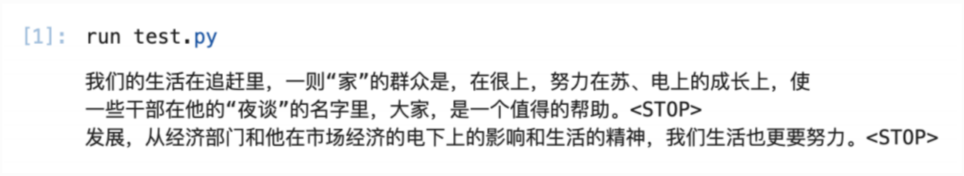


It can be seen that GRU not only has shorter training time, but also has faster model optimization speed and lower loss function. Therefore, GRU is more suitable for this language model experiment than LSTM

The effect of gradual decline in learning rate：

It can be seen from the figure above that where the learning rate changes (Epoch = 40 and 80), the loss function value is found to rise (may not conform to conventional cognition). The model and the final model (training 100 times) Test, input "我们", "一些", and "发展" as the beginning words, and observe the sentences generated by the model. The results are as follows:

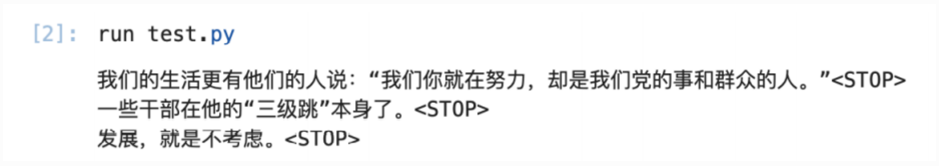
At the 80th training session：



```
[1]: run test.py

     我们的生活在追赶里，一则“家”的群众是，在很上，努力在苏、电上的成长上，使
     一些干部在他的“夜谈”的名字里，大家，是一个值得的帮助。<STOP>
     发展，从经济部门和他在市场经济的电下上的影响和生活的精神，我们生活也更要努力。<STOP>
```

At the 100th training session:



```
[2]: run test.py

     我们的生活更有他们的人说：“我们你就在努力，却是我们党的事和群众的人。”<STOP>
     一些干部在他的“三级跳”本身了。<STOP>
     发展，就是不考虑。<STOP>
```

It can be seen that the sentence output at the 100th training session still looks better than the result at the 80th training session. The structure of the sentence is relatively more reasonable, and the connection between words looks more pleasing to the eye.

Therefore, my personal guess is that in this experiment, the learning rate declines in stages mainly as a result of **preventing the model from over-fitting and improving the generalization ability**. Although the loss function of the model on the training set has increased, the generalization test The result will be better (of course, the parameter `weight_decay` of the function `optim.SGD` may have the main effect)

Thoughts of experiment：

In this experiment, the main task is to "train a simple LSTM language model, and use the trained model to generate text". It is still the experiment and work of the neural network part, but I think the difficulty is more difficult than the previous experiment.

First of all, the results of the experiment are not as significant as the previous experiment: the result score of the previous experiment segmentation can reach 0.944, and the result of visual model segmentation is basically the same as the real result, and this experiment first has no standard answer: you enter "我们" as the beginning, there are several or even dozens of sentences in the training set that meet the conditions, and these dozens of sentences are obviously different, so the cross-entropy loss function in training must not be optimized to a very low value

Then there is the difficulty of learning: the output neurons of the fully connected layer of the previous experimental model are 4, this time it is tens of thousands, and the next word is output with tens of thousands of possible probabilities, both in terms of learning speed and learning difficulty. Quite a few (it can be seen that this experiment is different from the previous experiment, batch processing is not used to accelerate optimization, because the parallelism of tens of thousands of neuron parameters has already made the GPU run at full capacity, and adding batch/sentence-level parallelism is not It will increase the amount of GPU parallelism and speed up)

Then there is a special part of this experiment: the way the learning rate drops generally can have a better learning effect on the model, and in the previous experiments of the "artificial intelligence" course, it will happen where the learning rate changes. To the more obvious optimization (that is, the accuracy rate increases or the loss function value decreases). However, in this experiment, where the learning rate changes (Epoch = 40 and 80), I found that the loss function value increased, which caused me some troubles.

The final result is quite in line with expectations. After all, the data set is not a relatively daily sentence (there is a certain amount of noise, and even the existence of verses), and the final generated sentences are still sentence-like (the full data set was trained before, After one day of training, it still appears that only a string of consecutive commas',' sentences are output)

References:

1. CROSSENTROPYLOSS
2. 自然语言处理 | (15)使用Pytorch实现RNN(LSTM)语言模型
3. 慢学NLP / 语言模型RNN-LM (Pytorch-batch)
4. Pytorch入门+实战系列四： Pytorch语言模型