ESE577 Final Project

Yijing Liao Ziyan Wang

November 30, 2022

1 Project Introduction

For the final project, we are required to implement the LSTM model to predict the word for a word sequence. Generally speaking, given several words input, one predicted word as output is expected. The whole project includes several processes: material selection, dataset preparation, word coding, training, and prediction.

2 RNN&LSTM Introduction

2.1 Recurrent Neural Networks (RNN)

Before introducing the LSTM, the more generated model is the recurrent neural network designed for a memory network. In a traditional CNN network, the input is considered identical, and each node in a layer will not affect the other node. Each data slice should affect the next data slice, like reading a sentence or a stock sequence. A identical RNN network is shown in figure1

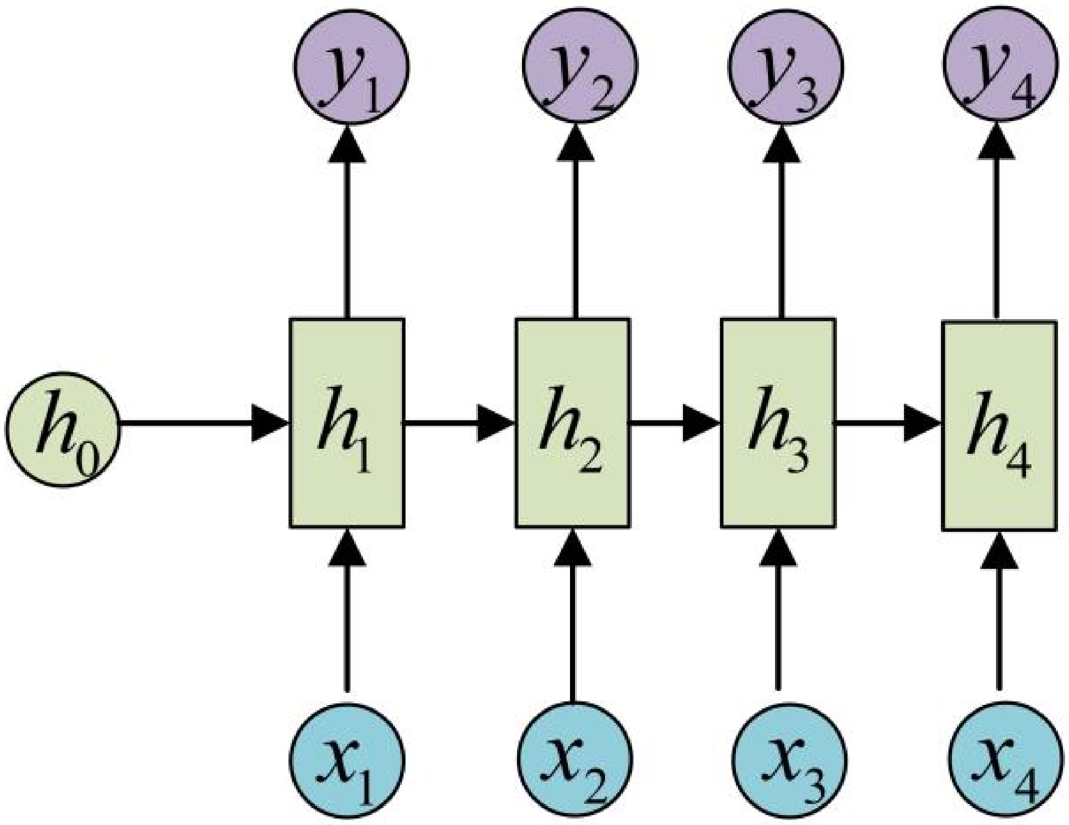


Figure 1: Identical RNN Structure

Where x_t mean inputs, h_t mean each node, and y_t mean each of the outputs from each node. For this layer, there is an initial state call h_0, which serves as the initial state value entering the h_1 node. After the h_1, the state value enters the h_2, representing if the information is remembered. One of the features of the RNN is that: for each node, the weight and the bias remain the same, meaning that each of the x_t is remembered collectively. However, to imitate the logit of the human brain, when we read a sentence, some of the words are remembered clearer than the other words, meaning that the input should be considered identically. The LSTM model is brought out to fix that question, and it is modified from RNN.

2.2 Long Short Term Memory(LSTM) Model

Considering the LSTM, LSTM means Long Short-Term Memory. Compared to RNN, the main difference is that the LSTM node not just passes one value but passes two values to the next node, representing the neuron state and level of remembering. Given a situation like this, when a person is reading a sentence or paragraph, he may not remember all the words and detail about it. He will only remember those “keywords”, which LSTM does toward a series stream of data.

The structure of the RNN node and the LSTM node is shown in figure 2.

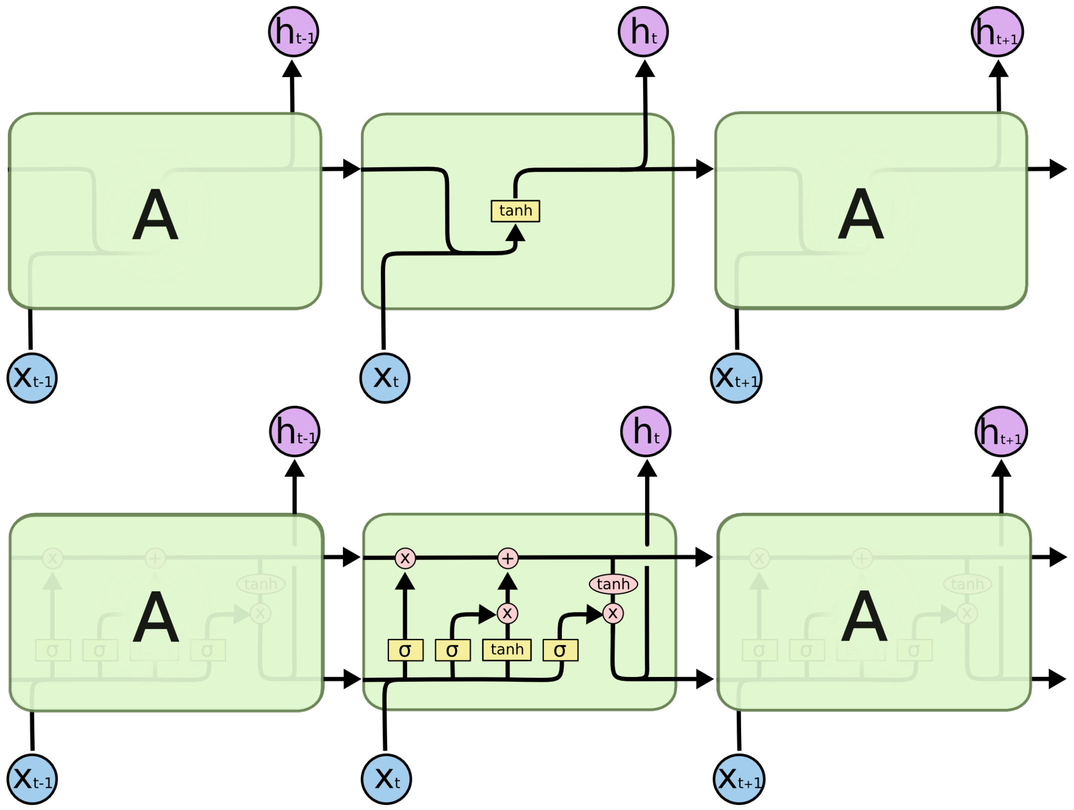
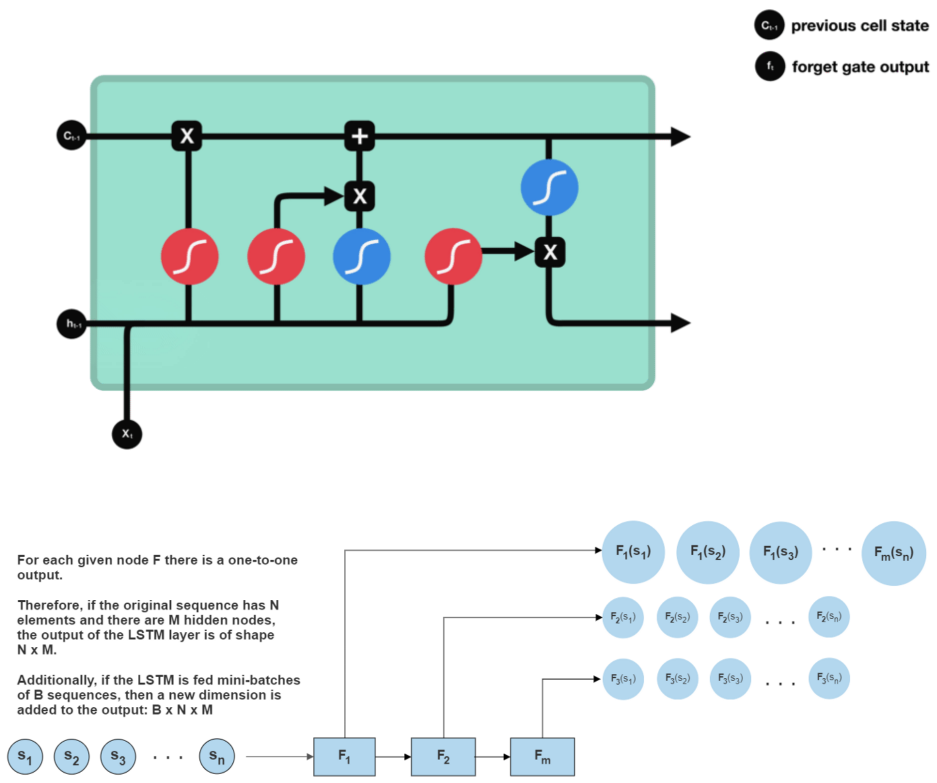


Figure 2: RCNN & LSTM Comparison

For a stream data as a vector, each rnn node is connected in series. Each of the data sequences will be the input of each rnn node and output a remembered state to the next node, and each of the node output will be collected and processed afterward. In these nodes, each node will decide if this “word” will be “remembered” or not and use the tanh activation to squish the value.[Neia].

Then, for LSTM mode, the difference is the node structure. Instead, output one remember state,shown in figure 3. LSTM node will output a two-state value to show that a word can not be completely remembered or completely forgotten. So each lstm node will have three inputs: two-state values from the forward node and one input segment for this node, and output two-state values.[Neib]

Consider an LSTM layer that includes m nodes in this layer; after every segment(total s) enters each of the LSTM nodes, every output state is collected and enters the fully connected layer. Then by adjusting the structure of the layer, it can output the prediction result or the classification result.


previous cell state
forget gate output
C_{t-1}
h_{t-1}
X_t
$F_1(s_1)$ $F_1(s_2)$ $F_1(s_3)$ $F_m(s_n)$
$F_2(s_1)$ $F_2(s_2)$ $F_2(s_3)$ $F_2(s_n)$
$F_3(s_1)$ $F_3(s_2)$ $F_3(s_3)$ $F_3(s_n)$
s_1 s_2 s_3 s_n
F_1 F_2 F_m

For each given node F there is a one-to-one output.

Therefore, if the original sequence has N elements and there are M hidden nodes, the output of the LSTM layer is of shape N x M.

Additionally, if the LSTM is fed mini-batches of B sequences, then a new dimension is added to the output: B x N x M

Figure 3: LSTM Structure

We can implement backpropagation like the CNN network by defining the loss function and train the network.

2.3 Word Embedding

While the LSTM model will only receive the float value as input, these string words must first convert into some form of value. Word embedding can help transfer each word into a vector by considering the relationship with the word adjacent, which is a mapping. The mapping condition is shown in figure4.

The whole mapping is just a simple CNN network where the network gets the sequence as input and outputs the vector-matrix in the shape of `[vector,word]`. After training the network using CBOW, which considers the middle word with the surrounding word beside. The word to vector mapping is fixed and returns the map from string word to vector.[Kar]

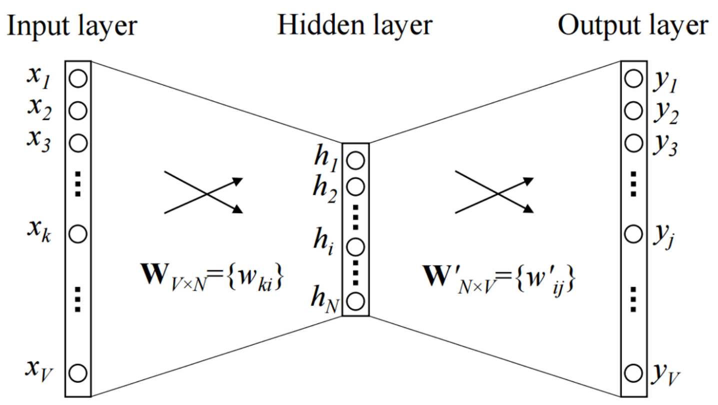
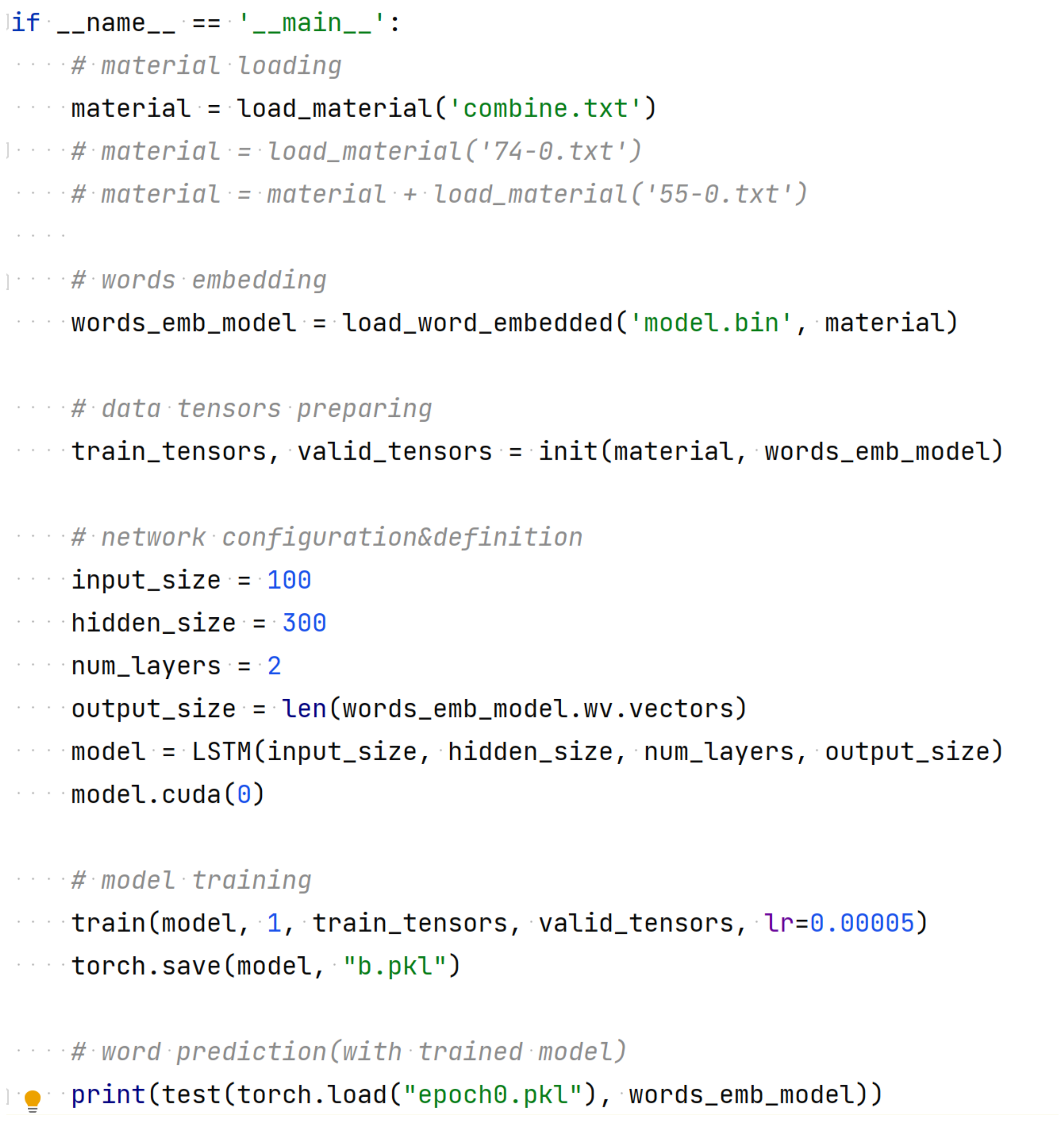


Figure 4: CBOW Mapping

3 LSTM Implementation

The implementation includes several processes, including material cleaning and preprocessing, LSTM model defining, network training, and prediction. The main function is shown in figure 5.

Function `load_material()` read the data in txt form and output the material as the word list. Then function `load_word_embedded()` will transfer these words into several 100 dimensions vectors using CBOW from the word2vec library. The output is represented as a high dimension matrix. Then, function `init()` transfers the material into the tensor prepared for the model, including input and the expected output labels. By using function `LSTM()`, the LSTM model is defined. Finally, use function `train()` to train the model and print the prediction result using function `test()`.

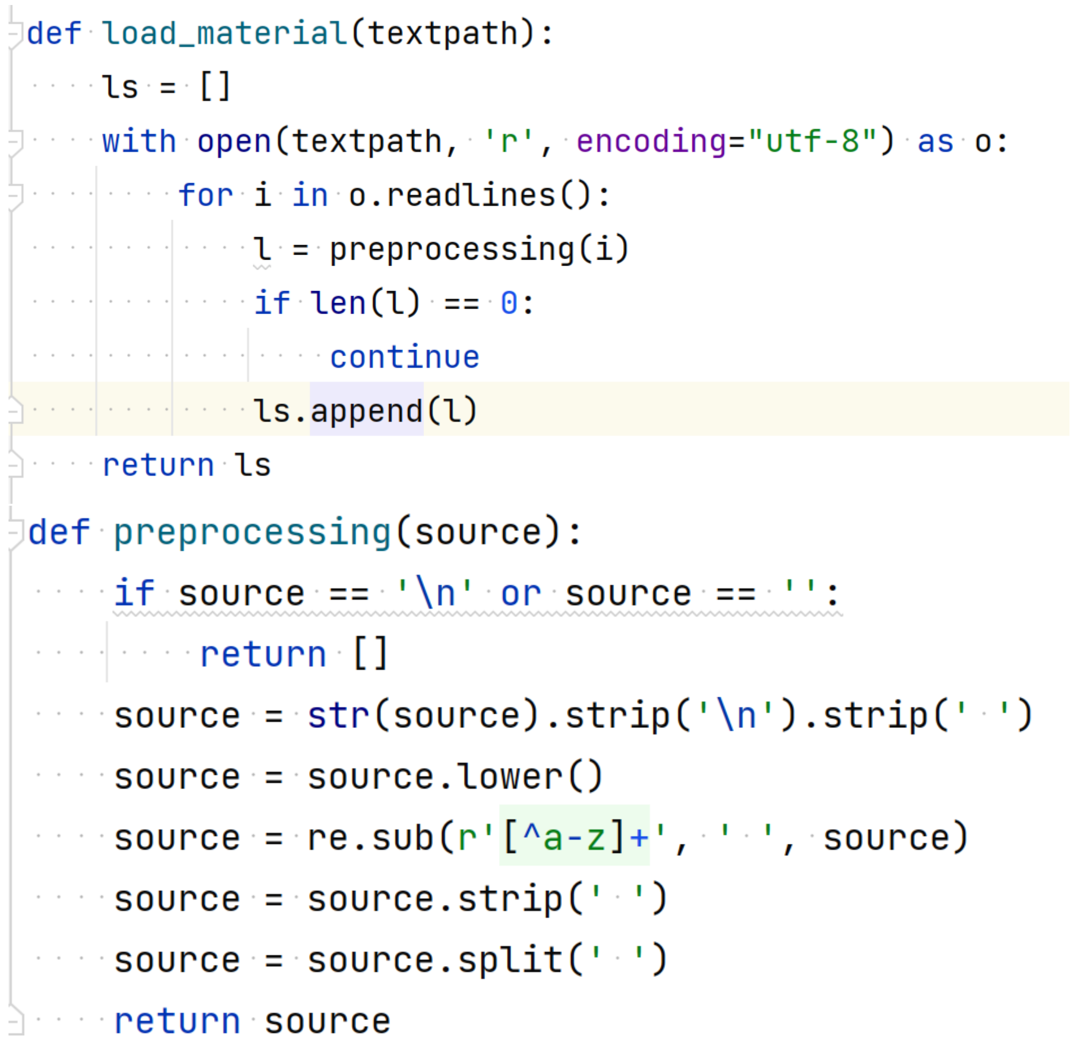


```python
if __name__ == '__main__':
    # material loading
    material = load_material('combine.txt')
    # material = load_material('74-0.txt')
    # material = material + load_material('55-0.txt')

    # words embedding
    words_emb_model = load_word_embedded('model.bin', material)

    # data tensors preparing
    train_tensors, valid_tensors = init(material, words_emb_model)

    # network configuration&definition
    input_size = 100
    hidden_size = 300
    num_layers = 2
    output_size = len(words_emb_model.wv.vectors)
    model = LSTM(input_size, hidden_size, num_layers, output_size)
    model.cuda(0)

    # model training
    train(model, 1, train_tensors, valid_tensors, lr=0.00005)
    torch.save(model, "b.pkl")

    # word prediction(with trained model)
    print(test(torch.load("epoch0.pkl"), words_emb_model))
```

Figure 5: Main Function

3.1 Material Loading&Prepossessing

Function `load_material()` is shown in figure 6. The first step is to open the `.txt` data and read the data line by line. Then use function `preprocessing()` to separate each of the words and append them to the word list.

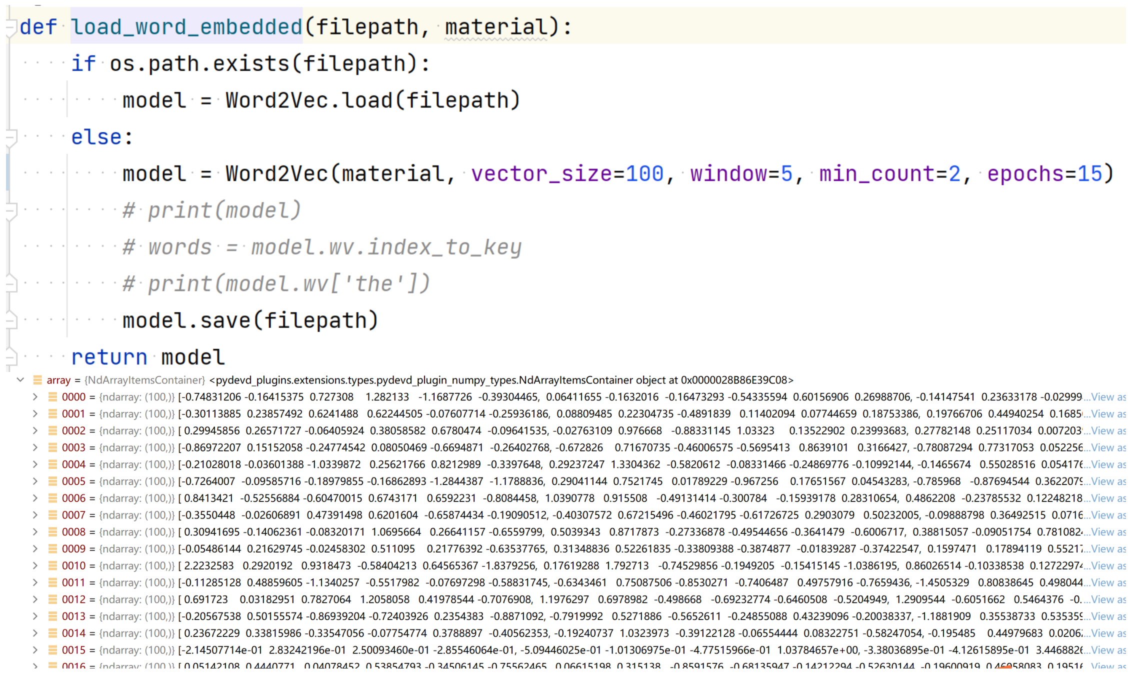


```
def load_material(textpath):
    ls = []
    with open(textpath, 'r', encoding="utf-8") as o:
        for i in o.readlines():
            l = preprocessing(i)
            if len(l) == 0:
                continue
            ls.append(l)
    return ls
def preprocessing(source):
    if source == '\n' or source == '':
        return []
    source = str(source).strip('\n').strip(' ')
    source = source.lower()
    source = re.sub(r'[^a-z]+', ' ', source)
    source = source.strip(' ')
    source = source.split(' ')
    return source
```

Figure 6: Material Loading&Prepossessing

3.2 Word Embedding

Function `load_word_embedded` is shown in figure 4. As in the traditional CNN network, the LSTM model cannot receive the string as input, so we first need to transform each word string into a vector. Here we can directly use the `Word2Verb()` library to train the mapping and convert the string into the 100-dimensions vector. Some of the word vectors are shown in figure 7.



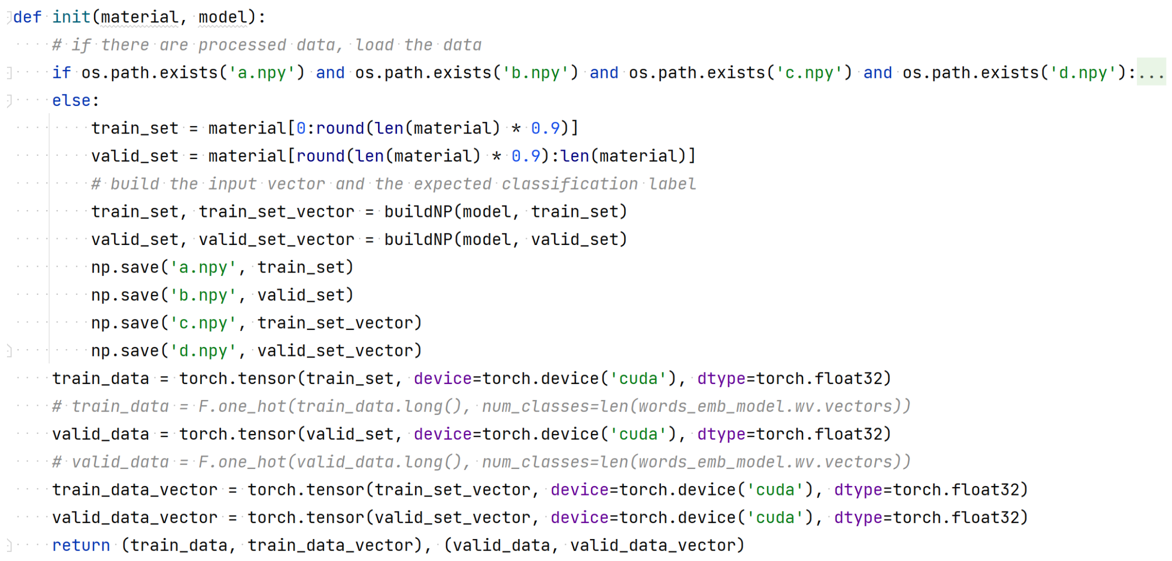
```
def load_word_embedded(filepath, material):
    if os.path.exists(filepath):
        model = Word2Vec.load(filepath)
    else:
        model = Word2Vec(material, vector_size=100, window=5, min_count=2, epochs=15)
        # print(model)
        # words = model.wv.index_to_key
        # print(model.wv['the'])
        model.save(filepath)
    return model
```

```
array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContainer object at 0x0000028B86E39C08>
  0000 = {ndarray: (100,)} [-0.74831206 -0.16415375  0.727308    1.282133   -1.1687726  -0.39304465,  0.06411655 -0.1632016  -0.16473293 -0.54335594  0.60156906  0.26988706, -0.14147541  0.23633178 -0.02999...View as
  0001 = {ndarray: (100,)} [-0.30113885  0.23857492  0.6241488   0.62244505 -0.07607714 -0.25936186,  0.08809485  0.22304735 -0.4891839   0.11402094  0.07744659  0.18753386,  0.19766706  0.44940254  0.1685...View as
  0002 = {ndarray: (100,)} [ 0.29945856  0.26571727 -0.06405924  0.38058582  0.6780474  -0.09641535, -0.02763109  0.976668   -0.88331145  1.03323     0.13522902  0.23993683,  0.27782148  0.25117034  0.007203...View as
  0003 = {ndarray: (100,)} [-0.86972207  0.15152058 -0.24774542  0.08050469 -0.6694871  -0.26402768, -0.672826    0.71670735 -0.46006575 -0.5695413   0.8639101   0.3166427, -0.78087294  0.77317053  0.052256...View as
  0004 = {ndarray: (100,)} [-0.21028018 -0.03601388 -1.0339872   0.25621766  0.8212989  -0.3397648,  0.29237247  1.3304362  -0.5820612  -0.08331466 -0.24869776 -0.10992144, -0.1465674   0.55028516  0.05417...View as
  0005 = {ndarray: (100,)} [-0.7264007  -0.09585716 -0.18979855 -0.16862893 -1.2844387  -1.1788836,  0.29041144  0.7521745   0.01789229 -0.967256    0.17651567  0.04543283, -0.785968   -0.87694544  0.362207...View as
  0006 = {ndarray: (100,)} [ 0.8413421  -0.52556884 -0.60470015  0.6743171   0.6592231  -0.8084458,  1.0390778   0.915508   -0.49131414 -0.300784   -0.15939178  0.28310654,  0.4862208  -0.23785532  0.12248218...View as
  0007 = {ndarray: (100,)} [-0.3550448  -0.02606891  0.47391498  0.6201604  -0.65874434 -0.19090512, -0.40307572  0.67215496 -0.46021795 -0.61726725  0.2903079   0.50232005, -0.09888798  0.36492515  0.0716...View as
  0008 = {ndarray: (100,)} [ 0.30941695 -0.14062361 -0.08320171  1.0695664   0.26641157 -0.6559799,  0.5039343   0.8717873  -0.27336878 -0.49544656 -0.3641479  -0.6006717,  0.38815057 -0.09051754  0.78108...View as
  0009 = {ndarray: (100,)} [-0.05486144  0.21629745 -0.02458302  0.511095    0.21776392 -0.63537765,  0.31348836  0.52261835 -0.33809388 -0.3874877  -0.01839287 -0.37422547,  0.1597471   0.17894119  0.5521...View as
  0010 = {ndarray: (100,)} [ 2.2232583   0.2920192   0.9318473  -0.58404213  0.64565367 -1.8379256,  0.17619288  1.792713   -0.74529856 -0.1949205  -0.15415145 -1.0386195,  0.86026514 -0.10338538  0.127229...View as
  0011 = {ndarray: (100,)} [-0.11285128  0.48859605 -1.1340257  -0.5517982  -0.07697298 -0.58831745, -0.6343461   0.75087506 -0.8530271  -0.7406487   0.49757916 -0.7659436, -1.4505329   0.80838645  0.498044...View as
  0012 = {ndarray: (100,)} [ 0.691723    0.03182951  0.7827064   1.2058058   0.41978544 -0.7076908,  1.1976297   0.6978982  -0.498668   -0.69232774 -0.6460508  -0.5204949,  1.2909544  -0.6051662   0.5464376  -0....View as
  0013 = {ndarray: (100,)} [-0.20567538  0.50155574 -0.86939204 -0.72403926  0.2354383  -0.8871092, -0.7919992   0.5271886  -0.5652611  -0.24855088  0.43239096 -0.20038337, -1.1881909   0.35538733  0.53535...View as
  0014 = {ndarray: (100,)} [ 0.23672229  0.33815986 -0.33547056 -0.07754774  0.3788897  -0.40562353, -0.19240737  1.0323973  -0.39122128 -0.06554444  0.08322751 -0.58247054, -0.195485    0.44979683  0.0206...View as
  0015 = {ndarray: (100,)} [-2.14507714e-01  2.83242196e-01  2.50093460e-01 -2.85546064e-01, -5.09446025e-01 -1.01306975e-01 -4.77515966e-01  1.03784657e+00, -3.38036895e-01 -4.12615895e-01  3.446882...View as
  0016 = {ndarray: (100,)} [ 0.05142108  0.4440771   0.04078452  0.53854793 -0.34506145 -0.75562465  0.06615198  0.315138   -0.8591576  -0.68135947 -0.14212294 -0.52630144 -0.19600919  0.46958083  0.1951...View as
```

Figure 7: Word Embedding

3.3 Data Initialization & Label Building

Then, considering these word vectors, we need to separate them into the training and validation data and build the expected label in vector form. Functions `init()` and `buildNP()` are shown in figure 8 and 9.

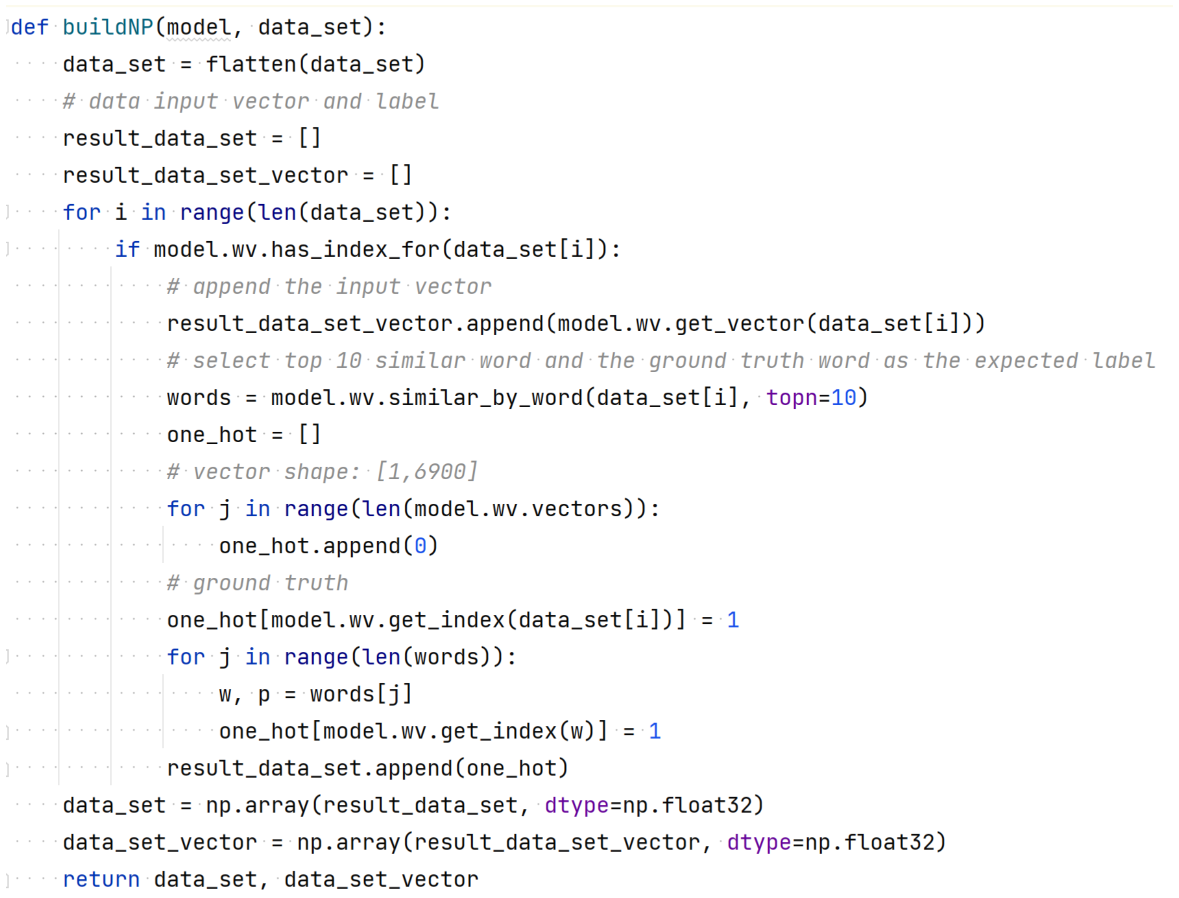


```
def init(material, model):
    # if there are processed data, load the data
    if os.path.exists('a.npy') and os.path.exists('b.npy') and os.path.exists('c.npy') and os.path.exists('d.npy'):...
    else:
        train_set = material[0:round(len(material) * 0.9)]
        valid_set = material[round(len(material) * 0.9):len(material)]
        # build the input vector and the expected classification label
        train_set, train_set_vector = buildNP(model, train_set)
        valid_set, valid_set_vector = buildNP(model, valid_set)
        np.save('a.npy', train_set)
        np.save('b.npy', valid_set)
        np.save('c.npy', train_set_vector)
        np.save('d.npy', valid_set_vector)
    train_data = torch.tensor(train_set, device=torch.device('cuda'), dtype=torch.float32)
    # train_data = F.one_hot(train_data.long(), num_classes=len(words_emb_model.wv.vectors))
    valid_data = torch.tensor(valid_set, device=torch.device('cuda'), dtype=torch.float32)
    # valid_data = F.one_hot(valid_data.long(), num_classes=len(words_emb_model.wv.vectors))
    train_data_vector = torch.tensor(train_set_vector, device=torch.device('cuda'), dtype=torch.float32)
    valid_data_vector = torch.tensor(valid_set_vector, device=torch.device('cuda'), dtype=torch.float32)
    return (train_data, train_data_vector), (valid_data, valid_data_vector)
```

Figure 8: Data Initialization

For function `init()`, the first step is to separate the material into the train set and the validation set. Then use function `buildNP` to build the input vector and corresponding expected label vector as `train_set_vector` and `train_set`. Then convert them into the tensor prepared for the torch model.

For function `buildNP`, to build the input vector and corresponding label, the first step is to flatten the data and prepare two lists for saving the data. Then for each index of the word, first get the corresponding input vector into the input vector list, then use this word to find the top 10 similar



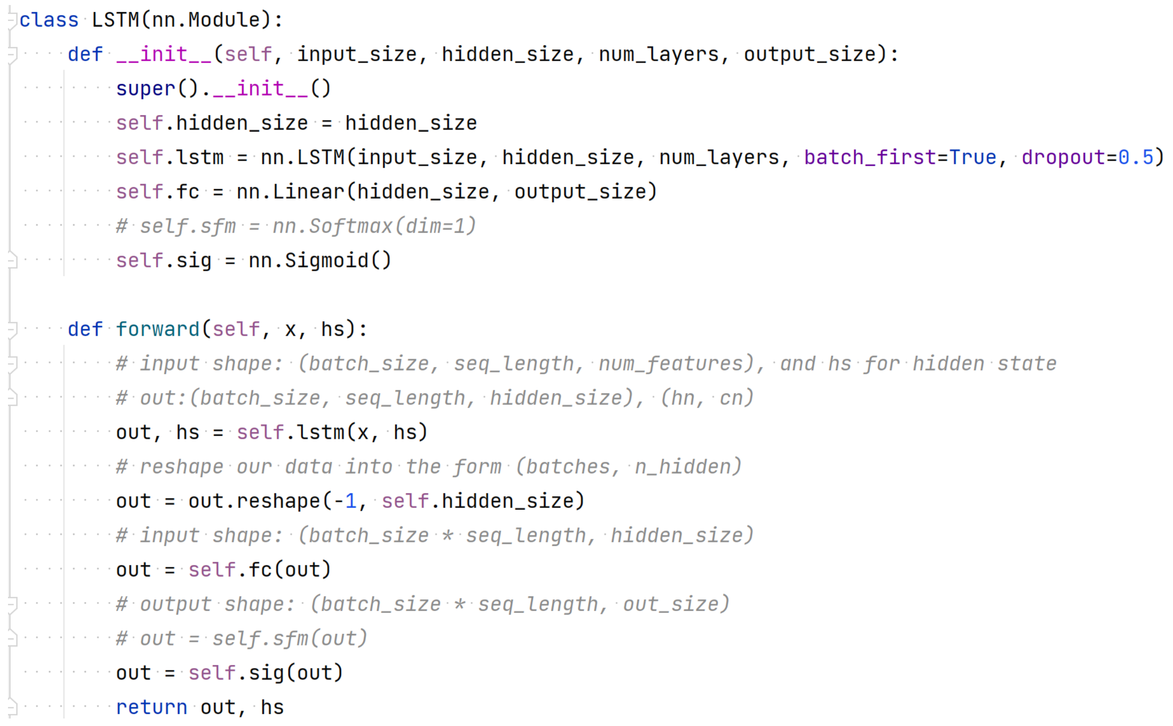
```python
def buildNP(model, data_set):
    data_set = flatten(data_set)
    # data input vector and label
    result_data_set = []
    result_data_set_vector = []
    for i in range(len(data_set)):
        if model.wv.has_index_for(data_set[i]):
            # append the input vector
            result_data_set_vector.append(model.wv.get_vector(data_set[i]))
            # select top 10 similar word and the ground truth word as the expected label
            words = model.wv.similar_by_word(data_set[i], topn=10)
            one_hot = []
            # vector shape: [1,6900]
            for j in range(len(model.wv.vectors)):
                one_hot.append(0)
            # ground truth
            one_hot[model.wv.get_index(data_set[i])] = 1
            for j in range(len(words)):
                w, p = words[j]
                one_hot[model.wv.get_index(w)] = 1
            result_data_set.append(one_hot)
    data_set = np.array(result_data_set, dtype=np.float32)
    data_set_vector = np.array(result_data_set_vector, dtype=np.float32)
    return data_set, data_set_vector
```

Figure 9: Label Building

words using function `model.wv.similar_by_word()` as the expected output. That is because we find out that it is hard to let the network find the one expected word among nearly 7,000-word classifications. Then we use one-hot coding the create the label and transfer all these data into the `np.array` prepared for the model.

3.4 Model Definition & Building

The definition of the LSTM model is shown in figure 10. The model includes one input layer, two hidden layers, and one fully connected layer, the output layer. The hidden layer, which is the LSTM layer, includes 300 LSTM nodes. `num_layers` is 2. `output_size` is the quantity of the class of all the words. The input vector first enters the LSTM model and gets some of the hidden state values and the corresponding output. Then the output vector enters the fully connected layer to implement classification in the form of vectors. Then, these vectors go through the sigmoid layer to get each of the word prediction probability as output. Function `forward()` defines the forward propagation process for one LSTM layer, where x is the input for each node and hs is the state value from the forward LSTM node.

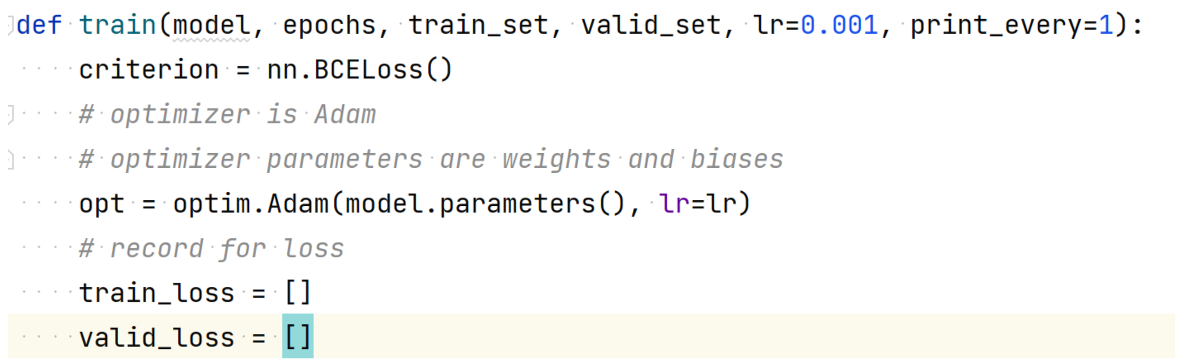


```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, dropout=0.5)
        self.fc = nn.Linear(hidden_size, output_size)
        # self.sfm = nn.Softmax(dim=1)
        self.sig = nn.Sigmoid()

    def forward(self, x, hs):
        # input shape: (batch_size, seq_length, num_features), and hs for hidden state
        # out:(batch_size, seq_length, hidden_size), (hn, cn)
        out, hs = self.lstm(x, hs)
        # reshape our data into the form (batches, n_hidden)
        out = out.reshape(-1, self.hidden_size)
        # input shape: (batch_size * seq_length, hidden_size)
        out = self.fc(out)
        # output shape: (batch_size * seq_length, out_size)
        # out = self.sfm(out)
        out = self.sig(out)
        return out, hs
```

Figure 10: Model Definition & Building

3.5 Model Training

The training configuration is shown in figure 11. We choose `binary_cross_entropy()` as the loss function and `Adam()` as the optimizer. Then prepare two lists to fill the training loss and valid loss.

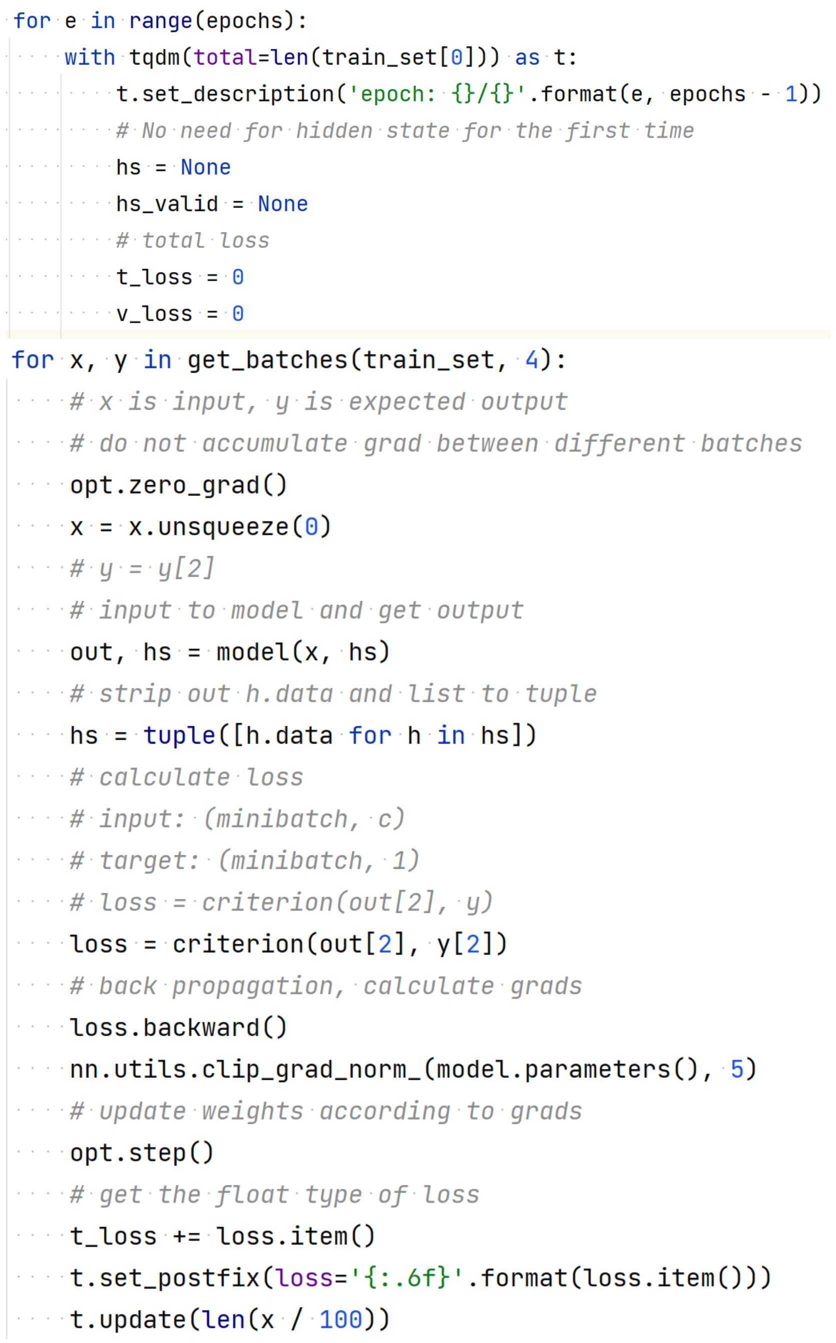


```python
def train(model, epochs, train_set, valid_set, lr=0.001, print_every=1):
    criterion = nn.BCELoss()
    # optimizer is Adam
    # optimizer parameters are weights and biases
    opt = optim.Adam(model.parameters(), lr=lr)
    # record for loss
    train_loss = []
    valid_loss = []
```

Figure 11: Model Training Configuration

In figure 12, for each epoch, prepare a `tqdm` counter to visualize the process. Then initial the hidden state value and loss value for training and validation.

Then, the first step for each batch is to get a batch using function `get_batches()`. X is the input, `y` is the expected output. Our batch size is 4. The `x` enters the model and gets the corresponding output:`out` and hidden state:`hs`. Then use the third output as the prediction result:`out[2]` to compare with the third word:`y[2]` in the label list, which is the ground truth of the word behind the third word, and compute the loss. Finally, use the loss to backward propagation and train the network.

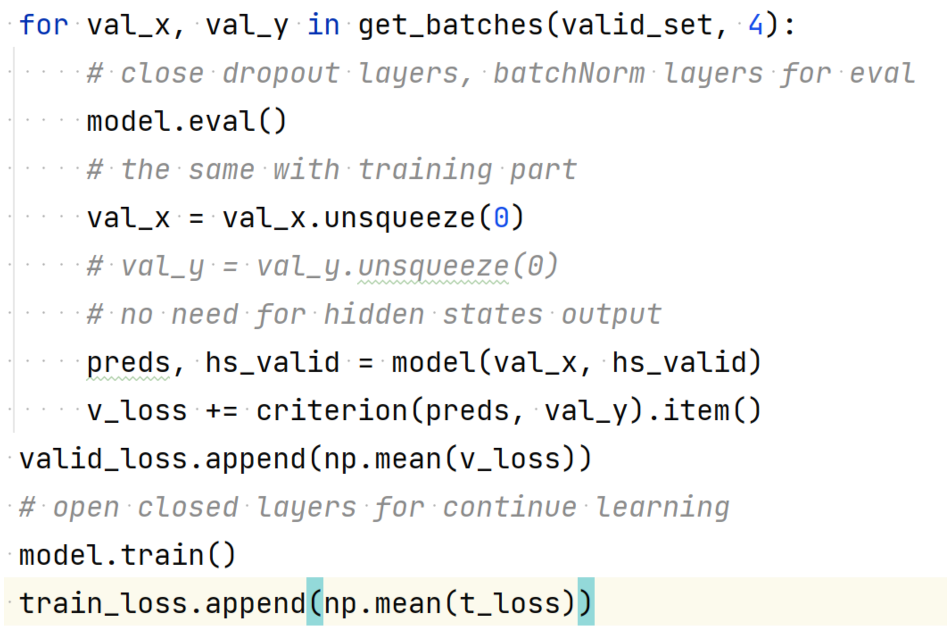


```python
for e in range(epochs):
    with tqdm(total=len(train_set[0])) as t:
        t.set_description('epoch: {}/{}'.format(e, epochs - 1))
        # No need for hidden state for the first time
        hs = None
        hs_valid = None
        # total loss
        t_loss = 0
        v_loss = 0
```

```python
for x, y in get_batches(train_set, 4):
    # x is input, y is expected output
    # do not accumulate grad between different batches
    opt.zero_grad()
    x = x.unsqueeze(0)
    # y = y[2]
    # input to model and get output
    out, hs = model(x, hs)
    # strip out h.data and list to tuple
    hs = tuple([h.data for h in hs])
    # calculate loss
    # input: (minibatch, c)
    # target: (minibatch, 1)
    # loss = criterion(out[2], y)
    loss = criterion(out[2], y[2])
    # back propagation, calculate grads
    loss.backward()
    nn.utils.clip_grad_norm_(model.parameters(), 5)
    # update weights according to grads
    opt.step()
    # get the float type of loss
    t_loss += loss.item()
    t.set_postfix(loss='{:.6f}'.format(loss.item()))
    t.update(len(x / 100))
```

Figure 12: Model Training

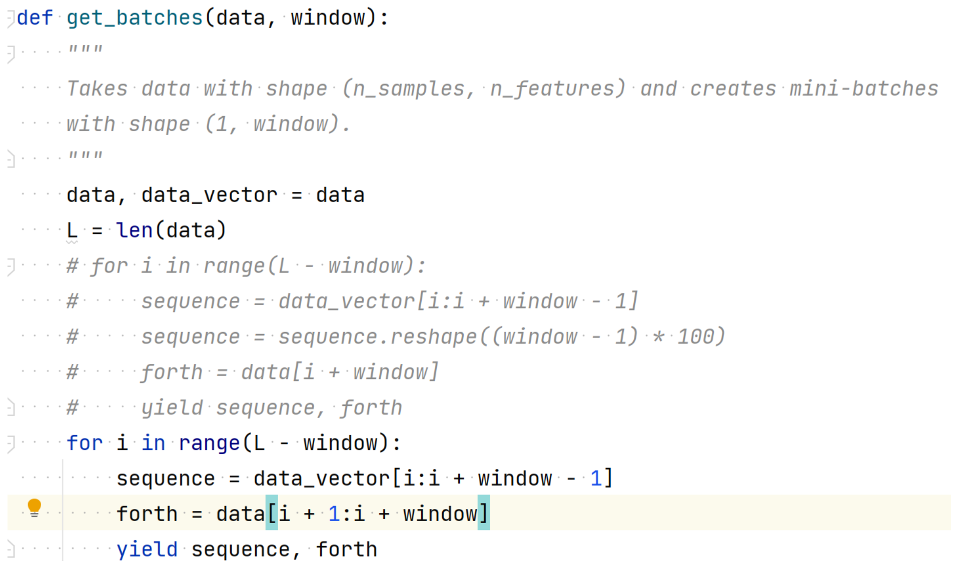
While for validation in figure 13, similar to training, after getting batch from function `get_batched()` the validation set enters the model and gets a corresponding prediction result. Here the point is to remember not to drop the hidden state:`hs_valid` because it is supposed to remain for the next LSTM node prediction.



```python
for val_x, val_y in get_batches(valid_set, 4):
    # close dropout layers, batchNorm layers for eval
    model.eval()
    # the same with training part
    val_x = val_x.unsqueeze(0)
    # val_y = val_y.unsqueeze(0)
    # no need for hidden states output
    preds, hs_valid = model(val_x, hs_valid)
    v_loss += criterion(preds, val_y).item()
valid_loss.append(np.mean(v_loss))
# open closed layers for continue learning
model.train()
train_loss.append(np.mean(t_loss))
```

Figure 13: Model Training

Function `get_batches()` is shown in figure 14.



```python
def get_batches(data, window):
    """
    Takes data with shape (n_samples, n_features) and creates mini-batches
    with shape (1, window).
    """
    data, data_vector = data
    L = len(data)
    # for i in range(L - window):
    #     sequence = data_vector[i:i + window - 1]
    #     sequence = sequence.reshape((window - 1) * 100)
    #     forth = data[i + window]
    #     yield sequence, forth
    for i in range(L - window):
        sequence = data_vector[i:i + window - 1]
        forth = data[i + 1:i + window]
        yield sequence, forth
```

Figure 14: Model Training

For a data stream, a sequence is defined as the first three words of the sequence, and the fourth is selected from the first one behind the third word so that the first three words can enter the sequence as input and the fourth vector as the expected label.

Overall speaking, the workflow of the model is shown in figure 15.

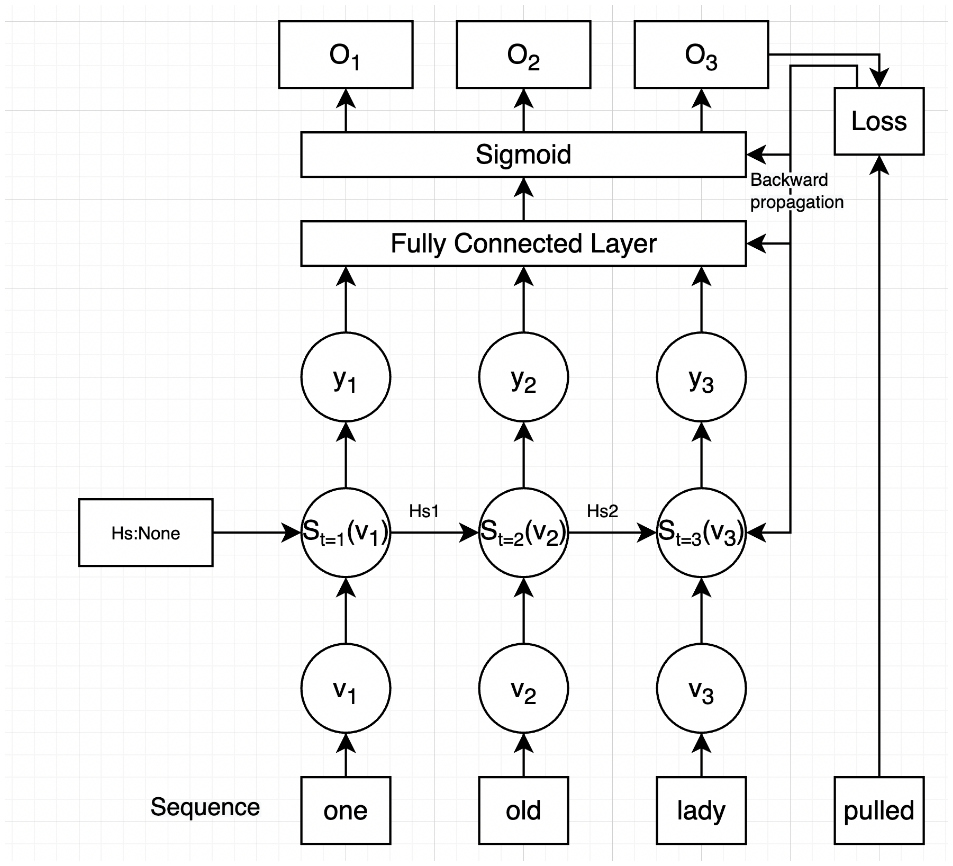


Figure 15: Model Workflow

4 Result & Conclusion

After the training in 50 epochs, the `Loss vs epoch` is shown in figure 16. We can see that training loss decreases fast compared to the validation loss. That is because the quantity of the Loss is not the mean but the total sum, while the training g set is much bigger than the validation set. Both of the losses decrease, meaning the model is training properly.

We select one of the predicting results from the referring list for the prediction. When we choose 'one,' 'old,' and 'lady' as three inputs, one of the sequences we generate after them is:

`['way','its','an','made','this','made','an','whole','an','whole','behind']`

which is shown in figure 17.

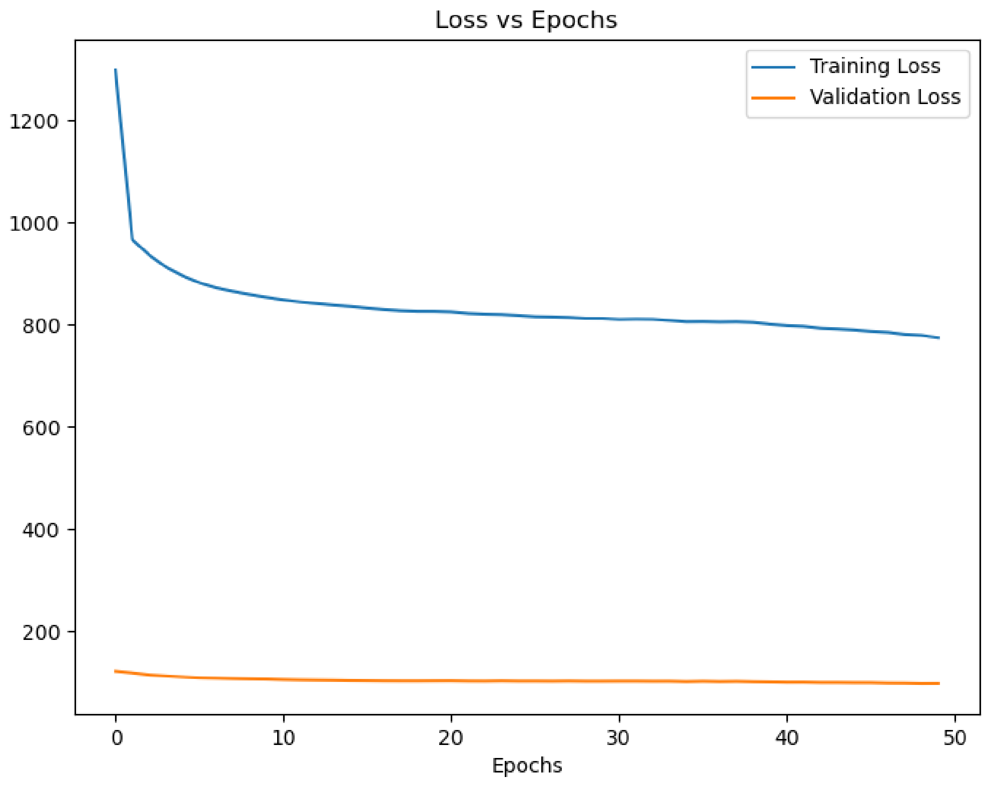
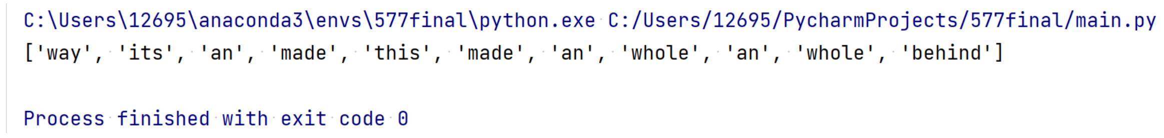


Figure 16: Loss Vs Epoch



```
C:\Users\12695\anaconda3\envs\577final\python.exe C:/Users/12695/PycharmProjects/577final/main.py
['way', 'its', 'an', 'made', 'this', 'made', 'an', 'whole', 'an', 'whole', 'behind']

Process finished with exit code 0
```

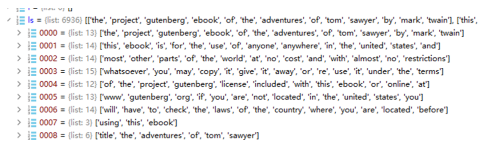
Figure 17: Prediction

5 Optimization

This part includes several optimizations we implement to help train the model better.

5.1 Sentences Readline Splitting

When we loaded the material, we split the sentences with the commas and then combined the word list. This action splite result is shown in figure 18.



```
ls = {list: 6936} [['the', 'project', 'gutenberg', 'ebook', 'of', 'the', 'adventures', 'of', 'tom', 'sawyer', 'by', 'mark', 'twain'], ['this',
  0000 = {list: 13} ['the', 'project', 'gutenberg', 'ebook', 'of', 'the', 'adventures', 'of', 'tom', 'sawyer', 'by', 'mark', 'twain']
  0001 = {list: 14} ['this', 'ebook', 'is', 'for', 'the', 'use', 'of', 'anyone', 'anywhere', 'in', 'the', 'united', 'states', 'and']
  0002 = {list: 14} ['most', 'other', 'parts', 'of', 'the', 'world', 'at', 'no', 'cost', 'and', 'with', 'almost', 'no', 'restrictions']
  0003 = {list: 15} ['whatsoever', 'you', 'may', 'copy', 'it', 'give', 'it', 'away', 'or', 're', 'use', 'it', 'under', 'the', 'terms']
  0004 = {list: 12} ['of', 'the', 'project', 'gutenberg', 'license', 'included', 'with', 'this', 'ebook', 'or', 'online', 'at']
  0005 = {list: 13} ['www', 'gutenberg', 'org', 'if', 'you', 'are', 'not', 'located', 'in', 'the', 'united', 'states', 'you']
  0006 = {list: 14} ['will', 'have', 'to', 'check', 'the', 'laws', 'of', 'the', 'country', 'where', 'you', 'are', 'located', 'before']
  0007 = {list: 3} ['using', 'this', 'ebook']
  0008 = {list: 6} ['title', 'the', 'adventures', 'of', 'tom', 'sawyer']
```

Figure 18: Sentences Splitting

5.2 Label Building

When we generate the label, we think it is hard to let the network predict one correct word among nearly 7,000 words. So we generate the top 10 similar words as the expected result using function `similar_by_word()`. The word similarity is shown in figure1.

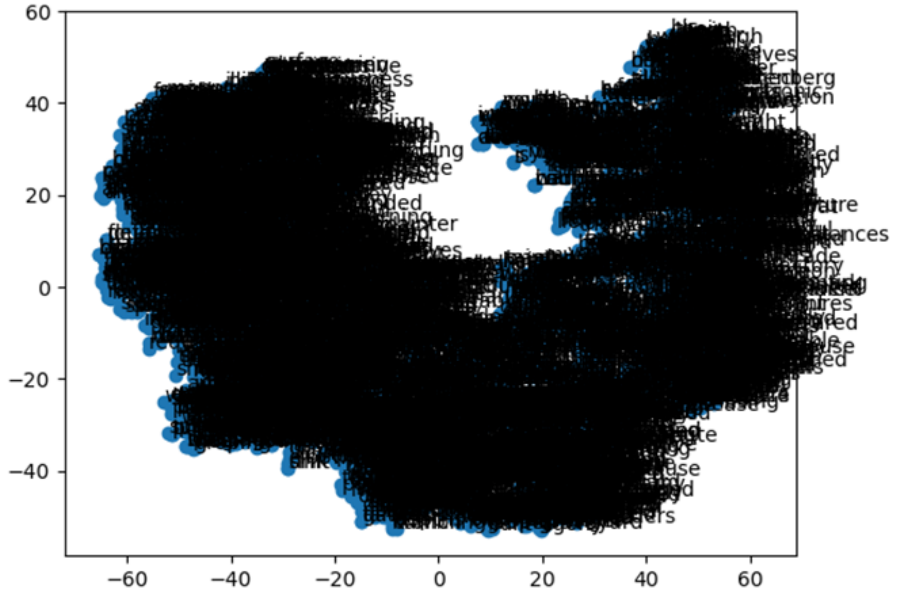
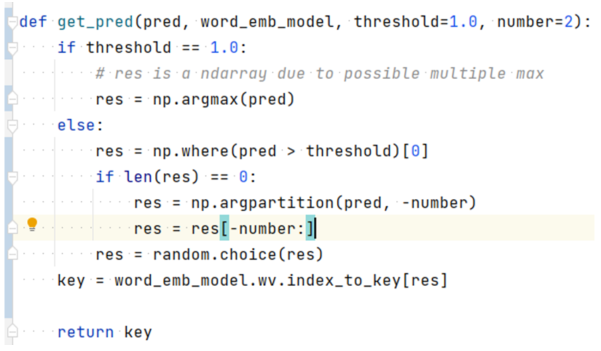


Figure 19: Label Building

5.3 Prediction Optimization

When we get the prediction result, if one of the resulting probabilities is precisely one, we select it as the one predicting word. Or we set a threshold, only those probability that meets the requirement could be the candidates, and then we choose the top of them as predicting word. If there is nobody who meets the threshold, we choose the best of them.



```python
def get_pred(pred, word_emb_model, threshold=1.0, number=2):
    if threshold == 1.0:
        # res is a ndarray due to possible multiple max
        res = np.argmax(pred)
    else:
        res = np.where(pred > threshold)[0]
        if len(res) == 0:
            res = np.argpartition(pred, -number)
            res = res[-number:]
        res = random.choice(res)
    key = word_emb_model.wv.index_to_key[res]

    return key
```

Figure 20: Prediction Optimization

References

[Kar] Dhruvil Karani. Introduction to word embedding and word2vec towards data science. https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa. (Accessed on 05/17/2022).

[Neia] Wesley Neill. Lstms in pytorch, towards data science.

[Neib] Wesley Neill. Lstms in pytorch understanding the lstm architecture and data flow. https://towardsdatascience.com/lstms-in-pytorch-528b0440244. (Accessed on 05/17/2022).