Wilbert Aristo Guntoro
ISTD - 1003742

Deep Learning Homework 9 Report

A. Copy and paste your SkipGram class code (Task #1 in the notebook)

```
class SkipGram(nn.Module):
  """
  Your SkipGram model here!
  """

  def __init__(self, context_size, embedding_dim, vocab_size):
      super(SkipGram, self).__init__()
      self.total_neighbors = 2 * context_size
      self.embeddings = nn.Embedding(vocab_size, embedding_dim)
      self.linear1 = nn.Linear(embedding_dim, 512)
      self.linear2 = nn.Linear(512, self.total_neighbors * vocab_size)

  def forward(self, inputs):
      embeds = self.embeddings(inputs).view((1, -1))
      output = F.relu(self.linear1(embeds))
      output = self.linear2(output)
      log_probs = F.log_softmax(output, dim=1).view(self.total_neighbors, -1)
      return log_probs
```

B. Copy and paste your train function (Task #2 in the notebook), along with any helper functions you might have used (e.g. a function to compute the accuracy of your model after each iteration). Please also copy and paste the function call with the parameters you used for the train() function.

```
def get_prediction(target, model, word2index):
  """
  This is a helper function to get logprobs and prediction indexes from model
  """
  tensor_target = torch.tensor(word2index[target], dtype=torch.long).cuda()
  log_probs = model(tensor_target)
  context_pred_idxs = log_probs.argmax(dim=1, keepdim=True)
  return log_probs, context_pred_idxs.squeeze().tolist()
```

```
# Define training parameters
learning_rate = 0.001
epochs = 300 # <- CUSTOM EPOCH
torch.manual_seed(28)
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr = learning_rate)
```

```python
def train(data, word2index, model, epochs, loss_func, optimizer):
  context_size = 2
  losses = []
  accuracies = []
  for epoch in range(epochs):
    # Setup / Reset correct counter and total loss
    correct_count = 0
    total_loss = 0

    for target, context in data:
      # Get context indexes
      context_idxs = [word2index[w] for w in context]
      tensor_context_idxs = torch.tensor(context_idxs,dtype=torch.long).cuda()

      # Zero out gradients from old instance
      model.zero_grad()

      # Get log probabilities and most probable context predictions
      log_probs, context_pred_idxs = get_prediction(target, model, word2index)

      # If any of the prediction is correct, add correct counter
      correct_count += len(set(context_idxs).intersection(context_pred_idxs))

      # Calculate Loss
      loss = loss_func(log_probs, tensor_context_idxs)
      total_loss += loss.item()

      # Backward pass and update gradient
      loss.backward()
      optimizer.step()

    # Calculate accuracy & loss for this epoch
    accuracy = round(correct_count / ( len(data) * 2 * context_size ), 3)
    loss = total_loss / len(data)
    print("Epoch {}: Accuracy = {} Loss = {}".format(epoch, accuracy, loss))

    # Append accuracy and loss for visualization
    accuracies.append(accuracy)
    losses.append(loss)

    return losses, accuracies, model

losses, accuracies, model = train(data, word2index, model, epochs,
                                  loss_function, optimizer)
```

Results Screenshot (Full Print Logs available in the jupyter notebook attached):

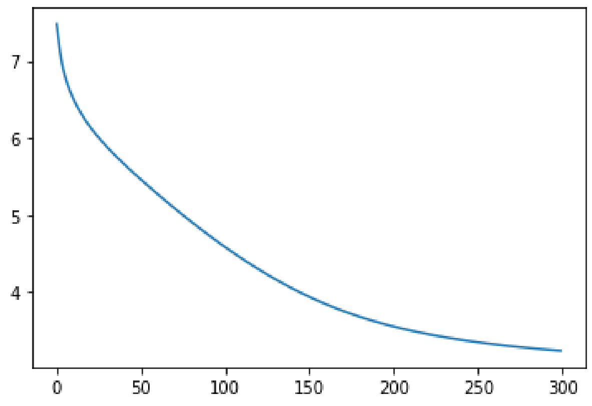
```
Epoch 0: Accuracy = 0.064 Loss = 7.477917604344595
Epoch 1: Accuracy = 0.116 Loss = 7.292670212945904
Epoch 2: Accuracy = 0.112 Loss = 7.127611689380903
Epoch 3: Accuracy = 0.111 Loss = 6.993345310849227
Epoch 4: Accuracy = 0.112 Loss = 6.887997459601677
Epoch 5: Accuracy = 0.115 Loss = 6.802183051126283
Epoch 6: Accuracy = 0.121 Loss = 6.728959598405505
Epoch 7: Accuracy = 0.131 Loss = 6.664303091327491
Epoch 8: Accuracy = 0.136 Loss = 6.606063589591573
Epoch 9: Accuracy = 0.142 Loss = 6.552941310448155
Epoch 10: Accuracy = 0.15 Loss = 6.503987601134276
```

.
.
.

```
Epoch 290: Accuracy = 0.462 Loss = 3.2528630339802374
Epoch 291: Accuracy = 0.462 Loss = 3.2509669606795937
Epoch 292: Accuracy = 0.462 Loss = 3.2490860760848292
Epoch 293: Accuracy = 0.462 Loss = 3.2472302166168374
Epoch 294: Accuracy = 0.462 Loss = 3.2453922210639057
Epoch 295: Accuracy = 0.463 Loss = 3.2435694059018987
Epoch 296: Accuracy = 0.463 Loss = 3.241768512267659
Epoch 297: Accuracy = 0.463 Loss = 3.23998290414912
Epoch 298: Accuracy = 0.463 Loss = 3.238220363918997
Epoch 299: Accuracy = 0.462 Loss = 3.236470586040266
```

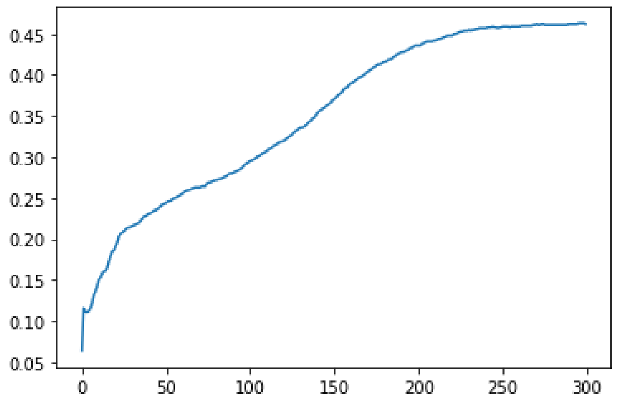
Loss vs. Epoch

```
# Display losses over time
plt.figure()
plt.plot(losses)
plt.show()
```



Accuracy vs. Epoch

```
# Display accuracies over time
plt.figure()
plt.plot(accuracies)
plt.show()
```



C. Why is the SkipGram model much more difficult to train than the CBoW? Is it problematic if it does not reach a 100% accuracy on the task it is being trained on?

The SkipGram model is much more difficult to be trained compared to CBoW because it requires the model to guess the surrounding context of a center word. This means the SkipGram model has to learn multiple contexts to which the current center word could pertain to, instead of just learning a possible center word given a fixed context (CBoW). SkipGram goes through more training instances compared to CBoW on the same training dataset.

It is not problematic for the SkipGram model to not reach 100% accuracy because all we want is a good embedding W (the weights of the hidden embedding layer). We are learning the relationship between words, and how a certain word can be surrounded by other words that will ultimately form a context that makes sense. The resulting sensible context from the model does not have to be the same as the original context provided in the training sample, thus accuracy is not a main concern.

D. If we were to evaluate this model by using intrinsic methods, what could be a possible approach to do so. Please submit some code that will demonstrate the performance/problems of the word embedding you have trained!

For a given corpus, we can choose one word randomly as a control sample. For example, look at the first sentence in the corpus given for this assignment: "*I am happy to join with you today in what will go down in history as **the greatest demonstration for freedom** in the history of our nation.*"

Let's say we choose the word "demonstration" as a control sample input.

First, we ask an untrained SkipGram model to predict the context words for "demonstration". The context words should be quite random and insensible, suggesting very bad word embeddings.

Next, we ask our previously trained model to predict the context words for "demonstration". Now, the context words should be sensible when paired with the word "demonstration", and even can be the same as the original context words in the corpus. This suggests good word embeddings.

I actually ran a test as described above and obtained very good results. The untrained model produced random & insensible context words ["if", "glory", "when", "millions"], whereas the trained model produced very sensible context words ["the", "greatest", "for", "freedom"]. In fact, the context words produced by the trained model are the original context words in the corpus! (100% accuracy for this particular word "demonstration").

```python
def predict_context_words(model, sample_input):
    _, context_pred_idxs = get_prediction(sample_input, model, word2index)
    context_preds = [index2word[idx] for idx in context_pred_idxs]
    return context_preds

# Initialize another SkipGram model (untrained)
untrained_model = SkipGram(context_size = context_size, embedding_dim = 20, vocab_size = len(vocab))
untrained_model.cuda()

# Get predictions
untrained_predictions = predict_context_words(untrained_model, "demonstration")
trained_predictions = predict_context_words(model, "demonstration")

print("Predicted Context Words from Untrained Model: {}".format(untrained_predictions))
print("Predicted Context Words from Trained Model: {}".format(trained_predictions))
```

```
Predicted Context Words from Untrained Model: ['if', 'glory', 'when', 'millions']
Predicted Context Words from Trained Model: ['the', 'greatest', 'for', 'freedom']
```