50.039 Theory and Practice of Deep Learning HW9
Loh De Rong (1003557)

The complete code and printouts can be found in the hw9.ipynb.

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.context_size = context_size
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view(1, -1)
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        out = F.log_softmax(out, dim=1)
        log_probs = torch.cat([out]*self.context_size*2) # weights are being shared
        log_probs = log_probs.view(2*self.context_size, -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(context, model, word2index, index2word):
    """
    This is a helper function to get prediction from our model.
    """


    # Get into eval() mode
    model.eval()
    context_id = words_to_tensor([context], word2index, dtype = torch.LongTensor)

    # Forward pass
    prediction = model(context_id)

    # Return top 4 predicted words
    predicted_words = []
    ids = torch.topk(prediction, 4).indices[0]
    for index in ids:
        predicted_words.append(index2word[index.item()])

    # print("Context word: {}. Target words: {}".format(context, predicted_words))

    return predicted_words


def check_accuracy(model, data, word2index, index2word):

    # Compute accuracy
    correct = 0
    for context, target in data:
        prediction = get_prediction(context, model, word2index, index2word)

        for word in target:
            if word in prediction:
                correct += 1

    return correct/(4 * len(data))
```

50.039 Theory and Practice of Deep Learning HW9
Loh De Rong (1003557)

```
def train(data, word2index, model, epochs, loss_func, optimizer):
    losses = []
    accuracies = []

    for epoch in range(epochs):
        total_loss = 0

        for context, target in data:

            # Prepare data
            ids = words_to_tensor([context], word2index, dtype = torch.LongTensor)
            target = words_to_tensor(target, word2index, dtype = torch.LongTensor)

            # Forward pass
            model.zero_grad()
            output = model(ids)
            loss = loss_func(output, target)

            # Backward pass and optim
            loss.backward()
            optimizer.step()

            # Loss update
            total_loss += loss.data.item()

        # Display
        if epoch % 10 == 0:
            accuracy = check_accuracy(model, data, word2index, index2word)
            print("Accuracy after epoch {} is {}".format(epoch, accuracy))
            accuracies.append(accuracy)
            losses.append(total_loss)

    return losses, accuracies, model


# Define training parameters
learning_rate = 0.001
epochs = 1000
torch.manual_seed(28)
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr = learning_rate)


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

50.039 Theory and Practice of Deep Learning HW9
Loh De Rong (1003557)

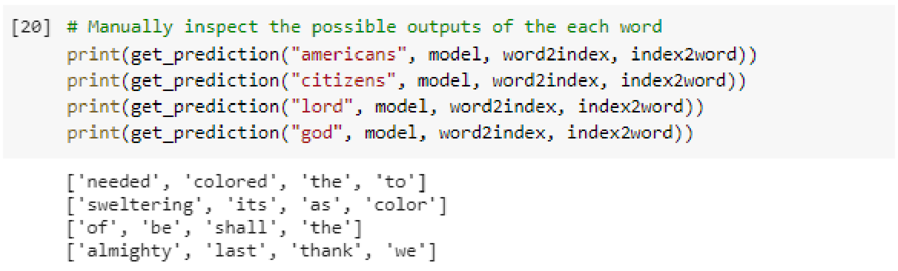
C. Why is the SkipGram model much more difficult to train than the CBoW. Is it problematic if it does not reach a 100% accuracy on the task it is being trained on?

CBoW model is trained to predict a single word from a fixed window size of context words, whereas the SkipGram model does the opposite, and tries to predict several context words from a single input word. Therefore, the SkipGram model is harder to train as it needs to get high probabilities for not just 1 right output word, but the right 2k output words surrounding the input word. In our case k=2.

It is not problematic if the SkipGram model does not reach a 100% accuracy on the task it is being trained on. This is because all we want is to train a good embedding W, which can be achieved as long as the “right” surrounding context words have high probabilities --- they can appear in any order and do not necessarily need to be in the top 2k shortlisted candidates.

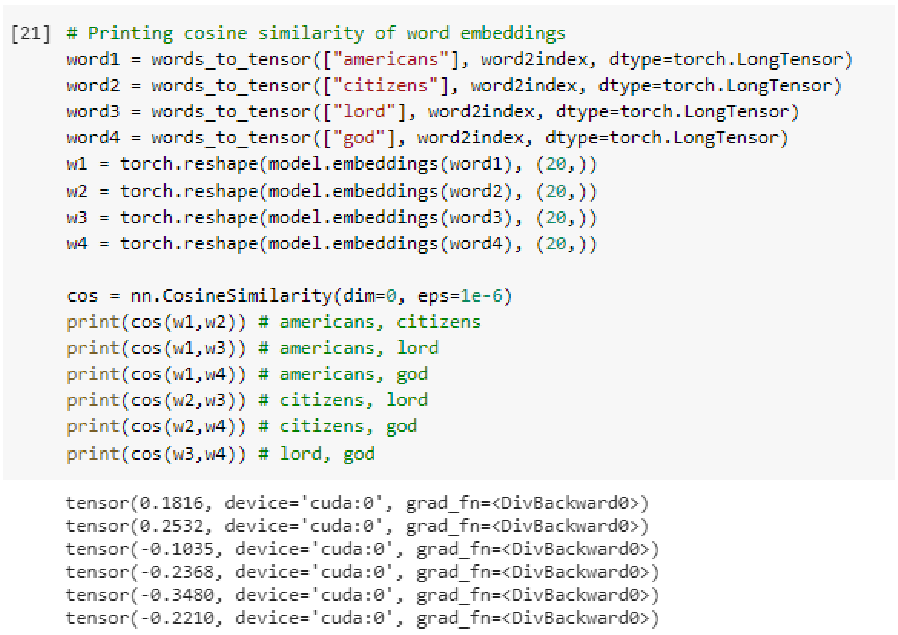
D. If we were to evaluate this model by using intrinsic methods, what could be a possible approach to do so. Please submit some code that will demonstrate the performance/problems of the word embedding you have trained!

The model does not appear to be performing very well. This is likely because our model is trained on a small corpus, so there is insufficient data to generate meaningful embeddings. For example, even though americans and citizens are used interchangeably in this racial text, and they both have syntactically similar outputs e.g. "colored" and "color" respectively, the model treats them as different words. This contributes to their low cosine similarity and poorly captures their semantic relationship. A larger text corpus is required.



```
[20] # Manually inspect the possible outputs of the each word
     print(get_prediction("americans", model, word2index, index2word))
     print(get_prediction("citizens", model, word2index, index2word))
     print(get_prediction("lord", model, word2index, index2word))
     print(get_prediction("god", model, word2index, index2word))
```

```
['needed', 'colored', 'the', 'to']
['sweltering', 'its', 'as', 'color']
['of', 'be', 'shall', 'the']
['almighty', 'last', 'thank', 'we']
```



```
[21] # Printing cosine similarity of word embeddings
     word1 = words_to_tensor(["americans"], word2index, dtype=torch.LongTensor)
     word2 = words_to_tensor(["citizens"], word2index, dtype=torch.LongTensor)
     word3 = words_to_tensor(["lord"], word2index, dtype=torch.LongTensor)
     word4 = words_to_tensor(["god"], word2index, dtype=torch.LongTensor)
     w1 = torch.reshape(model.embeddings(word1), (20,))
     w2 = torch.reshape(model.embeddings(word2), (20,))
     w3 = torch.reshape(model.embeddings(word3), (20,))
     w4 = torch.reshape(model.embeddings(word4), (20,))

     cos = nn.CosineSimilarity(dim=0, eps=1e-6)
     print(cos(w1,w2)) # americans, citizens
     print(cos(w1,w3)) # americans, lord
     print(cos(w1,w4)) # americans, god
     print(cos(w2,w3)) # citizens, lord
     print(cos(w2,w4)) # citizens, god
     print(cos(w3,w4)) # lord, god
```

```
tensor(0.1816, device='cuda:0', grad_fn=<DivBackward0>)
tensor(0.2532, device='cuda:0', grad_fn=<DivBackward0>)
tensor(-0.1035, device='cuda:0', grad_fn=<DivBackward0>)
tensor(-0.2368, device='cuda:0', grad_fn=<DivBackward0>)
tensor(-0.3480, device='cuda:0', grad_fn=<DivBackward0>)
tensor(-0.2210, device='cuda:0', grad_fn=<DivBackward0>)
```