50.039 Theory and Practice of Deep Learning - Homework 5

tags: `SUTD` `Theory and Practice of Deep Learning` `Homework`

Done by: Lin Huiqing (1003810)

Web view: https://hackmd.io/@ephemeral-instance/ryDB4DHHu

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.emb = nn.Embedding(vocab_size, embedding_dim)
        self.fc = nn.Linear(embedding_dim, 2 * context_size * vocab_size)

    def forward(self, inputs):
        x = self.emb(inputs).view((1, -1))
        x = self.fc(x)
        log_probs = F.log_softmax(x, dim=1).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_accuracy(model, data, word2index):
    model.eval()

    correct_count = 0
    for x, y in data:
        context_idxs = torch.tensor([word2index[x]], dtype=torch.long).cuda()
        log_probs = model(context_idxs)
        output_ids = torch.argmax(log_probs, dim=1)

        target_list = torch.tensor([word2index[w] for w in y], dtype=torch.long).cuda()

        for o_id, t_id in zip(output_ids, target_list):
            if o_id == t_id:
                correct_count += 1

    return correct_count / len(data)
```

```
def train(data, word2index, model, epochs, loss_func, optimizer):
    losses = []
    accuracies = []
    for epoch_no in range(epochs):
        model.train()

        e_loss = 0
        for x, y in data:
            # Step 1. Prepare the inputs to be passed to the model (i.e, turn
            # the words into integer indices and wrap them in tensors)

            context_idxs = torch.tensor([word2index[x]], dtype=torch.long).cuda()

            # Step 2. Recall that torch *accumulates* gradients. Before passing
            # in a new instance, you need to zero out the gradients from the old
            # instance
            optimizer.zero_grad()

            # Step 3. Run the forward pass, getting log probabilities over next
            # words
            log_probs = model(context_idxs)
            # print(log_probs.shape)

            # Step 4. Compute your loss function. (Again, Torch wants the target
            # word wrapped in a tensor)
            target_list = torch.tensor([word2index[w] for w in y], dtype=torch.long).cuda()
            # print(target_list.shape)
            loss = loss_function(log_probs, target_list)

            # Step 5. Do the backward pass and update the gradient
            loss.backward()
            optimizer.step()

            # Get the Python number from a 1-element Tensor by calling tensor.item()
            e_loss += loss.item()

        model.eval()

        with torch.no_grad():
            e_loss /= len(data)
            accuracy = get_accuracy(model, data, word2index)

            print(f"Epoch {epoch_no+1}: loss - {e_loss}; accuracy - {accuracy}")

            losses.append(e_loss)
            accuracies.append(accuracy)

    return losses, accuracies, model

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

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?

> Why is the SkipGram model much more difficult to train than the CBoW?

Because the SkipGram model tries to predict a word's context from a single word, as compared to the CBoW which tries to predict a single word from the word's context. As the SkipGram has to perform more elaborate predictions based on less data, this makes SkipGram a much more difficult model to train than CBoW.

> Is it problematic if it does not reach a 100% accuracy on the task it is being trained on?

No, because the motive of training these tasks are to obtain the embeddings learnt, rather than use the whole model. Even when the task being trained on does not reach 100% accuracy, the embedding layer might have already been sufficiently trained.

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!

> If we were to evaluate this model by using intrinsic methods, what could be a possible approach to do so?

We could check the relationship between the resulting embeddings from the trained embedding layer. The vectors should behave the following ways:

1. Cosine similarity between embeddings of words with similar meanings should be close to 1.
2. Embeddings should be able to recognise word analogies (e.g. 'man' is to 'woman' = 'king' is to 'queen')
3. Variabionts of words should have cosine similarity of close to 1.

> Please submit some code that will demonstrate the performance/problems of the word embedding you have trained!

The embedding layer and its weights are first loaded as follows:

```
embedding_layer = nn.Embedding(len(vocab), 20)
embedding_layer.weight = torch.nn.Parameter(model.state_dict()['emb.weight'])
embedding_layer.eval()
```

After which, 3 vectors are obtained from the embedding. These embeddings are of the following words: 'his', 'her', and 'boys'.

```
his_context_id = torch.zeros(len(vocab)).cuda().long()
his_context_id[word2index['his']] = 1.
his_vec = embedding_layer(his_context_id)

her_context_id = torch.zeros(len(vocab)).cuda().long()
her_context_id[word2index['her']] = 1.
her_vec = embedding_layer(her_context_id)

boys_context_id = torch.zeros(len(vocab)).cuda().long()
boys_context_id[word2index['boys']] = 1.
boys_vec = embedding_layer(boys_context_id)
```

After which, we can check the different cosine similarities between the different vectors.

```
his_her_sim = F.cosine_similarity(his_vec, her_vec, dim=0)
print(f"cosine similarity of 'his' and 'her': {his_her_sim}")
# cosine similarity of 'his' and 'her': tensor([1.0000, 0.9955, 0.9812,
# 0.9851, 1.0000, 0.9824, 0.9078, 0.9952, 0.9980, 0.9947, 0.9966, 0.9866,
# 0.9976, 0.9927, 0.9989, 0.8462, 1.0000, 0.9983, 0.9996, 0.9949],
# device='cuda:0', grad_fn=<DivBackward0>)

his_boys_sim = F.cosine_similarity(his_vec, boys_vec, dim=0)
print(f"cosine similarity of 'his' and 'boys': {his_boys_sim}")
# cosine similarity of 'his' and 'boys': tensor([1.0000, 0.9955, 0.9812,
# 0.9851, 1.0000, 0.9824, 0.9078, 0.9952, 0.9980, 0.9947, 0.9966, 0.9866,
# 0.9976, 0.9927, 0.9989, 0.8462, 1.0000, 0.9983, 0.9996, 0.9949],
# device='cuda:0', grad_fn=<DivBackward0>)

her_boys_sim = F.cosine_similarity(her_vec, boys_vec, dim=0)
print(f"cosine similarity of 'her' and 'boys': {her_boys_sim}")

# cosine similarity of 'her' and 'boys': tensor([1.0000, 0.9955, 0.9812,
# 0.9851, 1.0000, 0.9824, 0.9078, 0.9952, 0.9980, 0.9947, 0.9966, 0.9866,
# 0.9976, 0.9927, 0.9989, 0.8462, 1.0000, 0.9983, 0.9996, 0.9949],
# device='cuda:0', grad_fn=<DivBackward0>)
```

As seen, 'his' and 'her' is just as similar to 'his' and 'boys', and 'her' and 'boys'. This shows that the word embedding is limited as it is not able to differentiate gender, and is not able to detect the age component of the meaning when 'boys' is considered as well.