Cheat Sheet: AI Models and Language Understanding

<table>
<tr><th>Package/
Method</th><th>Description</th><th>Code example</th></tr>
<tr>
<td>PyTorch/Embedding and Embedding Bag</td>
<td>Embedding is a class that represents an embedding layer. It accepts token indices and produces embedding vectors.

EmbeddingBag is a class that aggregates embeddings using mean or sum operations.

Embedding and EmbeddingBag are part of the torch.nn module.

The code example shows how you can use Embedding and EmbeddingBag in PyTorch.</td>
<td>

```
# Defining a data set

dataset = [
"I like cats",
"I hate dogs",
"I'm impartial to hippos"
]

# Initializing the tokenizer, iterator from the data set, and vocabulary

tokenizer = get_tokenizer('spacy', language='en_core_web_sm’)
def yield_tokens(data_iter):
      for data_sample in data_iter:
            yield tokenizer(data_sample)
data_iter = iter(dataset)
vocab = build_vocab_from_iterator(yield_tokens(data_iter))


# Tokenizing and generating indices

input_ids=lambda x:[torch.tensor(vocab(tokenizer(data_sample))) for
data_sample in dataset]

index=input_ids(dataset)
print(index)

# Initiating the embedding layer, specifying the dimension size for the
embeddings,
# determining the count of unique tokens present in the vocabulary, and
creating the embedding layer

embedding_dim = 3
```

</td>
</tr>
</table>

		`n_embedding = len(vocab)` `n_embedding:9` `embeds = nn.Embedding(n_embedding, embedding_dim)` `# Applying the embedding object` `i_like_cats=embeds(index[0])` `i_like_cats` `impartial_to_hippos=embeds(index[-1])` `impartial_to_hippos` `# Initializing the embedding bag layer` `embedding_dim = 3` `n_embedding = len(vocab)` `n_embedding:9` `embedding_bag = nn.EmbeddingBag(n_embedding, embedding_dim)` `# Output the embedding bag` `dataset = ["I like cats","I hate dogs","I'm impartial to hippos"]` `index:[tensor([0, 7, 2]), tensor([0, 4, 3]), tensor([0, 1, 6, 8, 5])]` `i_like_cats=embedding_bag(index[0],offsets=torch.tensor([0]))` `i_like_cats`
Batch function	The batch size defines the number of samples that will be propagated through the network.	`def collate_batch(batch):` `    target_list, context_list, offsets = [], [], [0]` `    for _context, _target in batch:` `        target_list.append(vocab[_target])` `        processed_context = torch.tensor(text_pipeline(_context),` `        dtype=torch.int64)` `        context_list.append(processed_context)` `        offsets.append(processed_context.size(0))` `        target_list = torch.tensor(target_list, dtype=torch.int64)`

		`offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)` `context_list = torch.cat(context_list)` `return target_list.to(device), context_list.to(device), offsets.to(device)` `BATCH_SIZE = 64 # batch size for training` `dataloader_cbow = DataLoader(` `cobw_data, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_batch)`
Forward pass	Refers to the computation and storage of intermediate variables (including outputs) for a neural network in order from the input to the output layer.	`def forward(self, text):`
Stanford's pre-trained GloVe	Leverage large-scale data for word embeddings. It can be integrated into PyTorch for improved NLP tasks such as classification.	`from torchtext.vocab import GloVe,vocab` `# creating an instance of the 6B version of Glove() model` `glove_vectors_6B = GloVe(name ='6B') # you can specify the model with the following format: GloVe(name='840B', dim=300)` `# Build vocab from glove_vectors` `vocab = vocab(glove_vectors_6B.stoi, 0,specials=('<unk>', '<pad>'))` `vocab.set_default_index(vocab["<unk>"])`
vocab	The vocab object is part of the PyTorch torchtext library. It maps tokens to indices. The code example shows how you can apply the vocab object to tokens directly.	`# Takes an iterator as input and extracts the next tokenized sentence. Creates a list of token indices using the vocab dictionary for each token.` `def get_tokenized_sentence_and_indices(iterator):` `    tokenized_sentence = next(iterator)` `    token_indices = [vocab[token] for token in tokenized_sentence]` `    return tokenized_sentence, token_indices`

```
# Returns the tokenized sentences and the corresponding token indices. Repeats the process.

tokenized_sentence, token_indices = get_tokenized_sentence_and_indices(my_iterator)
next(my_iterator)

# Prints the tokenized sentence and its corresponding token indices.

print("Tokenized Sentence:", tokenized_sentence)
print("Token Indices:", token_indices)
```

Special tokens in PyTorch: <eos> and <bos>

Special tokens are tokens introduced to input sequences to convey specific information or serve a particular purpose during training.

The code example shows the use of <bos> and <eos> during tokenization. The <bos> token denotes the beginning of the input sequence, and the <eos> token denotes the end.

```
# Appends <bos> at the beginning and <eos> at the end of the tokenized sentences using a loop that iterates over the sentences in the input data

tokenizer_en = get_tokenizer('spacy', language='en_core_web_sm')
tokens = []
max_length = 0

for line in lines:
    tokenized_line = tokenizer_en(line)
    tokenized_line = ['<bos>'] + tokenized_line + ['<eos>']
    tokens.append(tokenized_line)
    max_length = max(max_length, len(tokenized_line))
```

Special tokens in PyTorch: <pad>

The code example shows the use of <pad> token to ensure all sentences have the same length.

```
# Pads the tokenized lines
for i in range(len(tokens)):
     tokens[i] = tokens[i] + ['<pad>'] * (max_length - len(tokens[i]))
```

Cross entropy loss

Metric used in machine learning (ML) to evaluate the performance of a classification model. The loss is measured as the probability value between 0 (perfect model) and 1. Typically,

```
from torch.nn import CrossEntropyLoss
model = TextClassificationModel(vocab_size,emsize,num_class)

loss_fn = CrossEntropyLoss()
predicted_label = model(text, offsets)
loss = criterion(predicted_label, label)
```

	the aim is to bring the model as close to 0 as possible.	
Optimization	Method to reduce losses in a model.	`optimizer = torch.optim.SGD(model.parameters(), lr=0.1)` `scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)` `optimizer.zero_grad()` `predicted_label = model(text, offsets)` `loss = criterion(predicted_label, label)` `loss.backward()` `torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)` `optimizer.step()`
sentence_bleu()	NLTK (or Natural Language Toolkit) provides this function to evaluate a hypothesis sentence against one or more reference sentences. The reference sentences must be presented as a list of sentences where each reference is a list of tokens.	`from nltk.translate.bleu_score import sentence_bleu` `def calculate_bleu_score(generated_translation, reference_translations):` `    # Convert the generated translations and reference translations into the expected format for sentence_bleu` `    references = [reference.split() for reference in reference_translations]` `    hypothesis = generated_translation.split()` `    # Calculate the BLEU score` `    bleu_score = sentence_bleu(references, hypothesis)` `    return bleu_score` `reference_translations = ["Asian man sweeping the walkway .","An asian man sweeping the walkway .","An Asian man sweeps the sidewalk .","An Asian man is sweeping the sidewalk .","An asian man is sweeping the walkway .","Asian man sweeping the sidewalk ."]` `bleu_score = calculate_bleu_score(generated_translation, reference_translations)`

<table>
<tr>
<td>Encoder RNN model</td>
<td>The encoder-decoder seq2seq model works together to transform an input sequence into an output sequence. Encoder is a series of RNNs that process the input sequence individually, passing their hidden states to their next RNN.</td>
<td><pre>
class Encoder(nn.Module):
    def __init__(self, vocab_len, emb_dim, hid_dim, n_layers, dropout_prob):
        super().__init__()

        self.hid_dim = hid_dim
        self.n_layers = n_layers

        self.embedding = nn.Embedding(vocab_len, emb_dim)
        self.lstm = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout_prob)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, input_batch):
        embed = self.dropout(self.embedding(input_batch))
        embed = embed.to(device)
        outputs, (hidden, cell) = self.lstm(embed)

        return hidden, cell
</pre></td>
</tr>
<tr>
<td>Decoder RNN model</td>
<td>The encoder-decoder seq2seq model works together to transform an input sequence into an output sequence.

The decoder module is a series of RNNs that autoregressively generates the translation as one token at a time. Each generated token goes back into the next RNN along with the hidden state to generate the next token of the output sequence until the end token is generated.</td>
<td><pre>
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.output_dim = output_dim
        self.hid_dim = hid_dim
        self.n_layers = n_layers

        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.lstm = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)
        self.fc_out = nn.Linear(hid_dim, output_dim)
        self.softmax = nn.LogSoftmax(dim=1)
        self.dropout = nn.Dropout(dropout)
</pre></td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>

```
    def forward(self, input, hidden, cell):

        input = input.unsqueeze(0)
        embedded = self.dropout(self.embedding(input))

        output, (hidden, cell) = self.lstm(embedded, (hidden,
        cell))
        prediction_logit = self.fc_out(output.squeeze(0))
        prediction = self.softmax(prediction_logit)

        return prediction, hidden, cell
```

</td>
</tr>
<tr>
<td>Skip-gram model</td>
<td>Predicts surrounding context words from a specific target word. It predicts one context word at a time from a target word.</td>
<td>

```
class SkipGram_Model(nn.Module):

    def __init__(self, vocab_size, embed_dim):

        super(SkipGram_Model, self).__init__()


        # Define the embeddings layer

        self.embeddings = nn.Embedding(num_embeddings=vocab_size,
        embedding_dim=embed_dim)


        # Define the fully connected layer

        self.fc = nn.Linear(in_features=embed_dim,
        out_features=vocab_size)


    # Perform the forward pass

    def forward(self, text):


        # Pass the input text through the embeddings layer

        out = self.embeddings(text)
```

</td>
</tr>
</table>

		`# Pass the output of the embeddings layer through the fully connected layer` `# Apply the ReLU activation function` `out = torch.relu(out)` `out = self.fc(out)` `return out` `model_sg = SkipGram_Model(vocab_size, emsize).to(device)` `# Sequence generation function` `CONTEXT_SIZE = 2` `skip_data = []` `for i in range(CONTEXT_SIZE, len(tokenized_toy_data) - CONTEXT_SIZE):` `context = (` `[tokenized_toy_data[i - j - 1] for j in range(CONTEXT_SIZE)] # Preceding words` `+ [tokenized_toy_data[i + j + 1] for j in range(CONTEXT_SIZE)] # Succeeding words)` `target = tokenized_toy_data[i]` `skip_data.append((target, context))` `skip_data=[('i', ['wish', 'i', 'was', 'little']), ('was', ['i', 'wish', 'little', 'bit’])],..`
collate_fn	Processes the list of samples to form a batch. The `batch` argument is a list of all your samples.	`def collate_fn(batch):` `target_list, context_list = [], []`

		`` for _context, _target in batch: target_list.append(vocab[_target]) context_list.append(vocab[_context]) target_list = torch.tensor(target_list, dtype=torch.int64) context_list = torch.tensor(context_list, dtype=torch.int64) return target_list.to(device), context_list.to(device) ```
Training function	Trains the model for a specified number of epochs. It also includes a condition to check whether the input is for skip-gram or CBOW. The output of this function includes the trained model and a list of average losses for each epoch.	``` def train_model(model, dataloader, criterion, optimizer, num_epochs=1000): # List to store running loss for each epoch epoch_losses = [] for epoch in tqdm(range(num_epochs)): # Storing running loss values for the current epoch running_loss = 0.0 # Using tqdm for a progress bar for idx, samples in enumerate(dataloader): optimizer.zero_grad() # Check for EmbeddingBag layer in the model CBOW if any(isinstance(module, nn.EmbeddingBag) for _, module in model.named_modules()): target, context, offsets = samples predicted = model(context, offsets) ```

		`` # Check for Embedding layer in the model skip gram elif any(isinstance(module, nn.Embedding) for _, module in model.named_modules()): target, context = samples predicted = model(context) loss = criterion(predicted, target) loss.backward() torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1) optimizer.step() running_loss += loss.item() # Append average loss for the epoch epoch_losses.append(running_loss / len(dataloader)) return model, epoch_losses ```
CBOW model	Utilizes context words to predict a target word and generate its embedding.	``` class CBOW(nn.Module): # Initialize the CBOW model def __init__(self, vocab_size, embed_dim, num_class): super(CBOW, self).__init__() # Define the embedding layer using nn.EmbeddingBag self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, sparse=False) # Define the fully connected layer self.fc = nn.Linear(embed_dim, vocab_size) ```

		<pre> def forward(self, text, offsets): # Pass the input text and offsets through the embedding layer out = self.embedding(text, offsets) # Apply the ReLU activation function to the output of the first linear layer out = torch.relu(out) # Pass the output of the ReLU activation through the fully connected layer return self.fc(out) vocab_size = len(vocab) emsize = 24 model_cbow = CBOW(vocab_size, emsize, vocab_size).to(device)</pre>
Training loop	It enumerates data from the DataLoader and, on each pass of the loop, gets a batch of training data from the DataLoader, zeros the optimizer's gradients, and performs an inference (gets predictions from the model for an input batch).	<pre>for epoch in tqdm(range(1, EPOCHS + 1)): model.train() cum_loss=0 for idx, (label, text, offsets) in enumerate(train_dataloader): optimizer.zero_grad() predicted_label = model(text, offsets) loss = criterion(predicted_label, label) loss.backward() torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1) optimizer.step() cum_loss+=loss.item() cum_loss_list.append(cum_loss) accu_val = evaluate(valid_dataloader) acc_epoch.append(accu_val) if accu_val > acc_old: acc_old= accu_val torch.save(model.state_dict(), 'my_model.pth')</pre>