Exercise 12: Solution

FeedForwardNeuralNetwork

```python
def __init__(self,
             d_model: int,
             d_ff: int,
             dropout: float = 0.0):
    """

    Args:
        d_model: Dimension of Embedding
        d_ff: Dimension of hidden layer
        dropout: Dropout probability
    """
    super().__init__()

    self.linear_1 = None
    self.relu = None
    self.linear_2 = None
    self.dropout = None

    ########################################################################
    # TODO:                                                                #
    #   Task 3: Initialize the feed forward network                        #
    #   Task 11: Initialize the dropout layer (torch.nn implementation)    #
    #                                                                      #
    ########################################################################

    self.linear_1 = nn.Linear(in_features=d_model, out_features=d_ff)
    self.relu = nn.ReLU()
    self.linear_2 = nn.Linear(in_features=d_ff, out_features=d_model)
    self.dropout = nn.Dropout(p=dropout)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

```python
def forward(self,
            inputs: torch.Tensor) -> torch.Tensor:
    """

    Args:
        inputs: Inputs to the Feed Forward Network

    Shape:
        - inputs: (batch_size, sequence_length_queries, d_model)
        - outputs: (batch_size, sequence_length_queries, d_model)
    """
    outputs = None

    ########################################################################
    # TODO:                                                                #
    #   Task 3: Implement forward pass of feed forward layer               #
    #   Task 11: Pass the output through a dropout layer as a final step   #
    #                                                                      #
    ########################################################################

    outputs = self.linear_1(inputs)
    outputs = self.relu(outputs)
    outputs = self.linear_2(outputs)
    outputs = self.dropout(outputs)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return outputs
```

EncoderBlock

```python
def __init__(self,
             d_model: int,
             d_k: int,
             d_v: int,
             n_heads: int,
             d_ff: int,
             dropout: float = 0.0):
    super().__init__()

    self.multi_head = None
    self.layer_norm1 = None
    self.ffn = None
    self.layer_norm2 = None

    ########################################################################
    # TODO:                                                                #
    #   Task 4: Initialize the Encoder Block                               #
    #           You will need:                                             #
    #                           - Multi-Head Self-Attention layer          #
    #                           - Layer Normalization                      #
    #                           - Feed forward neural network layer        #
    #                           - Layer Normalization                      #
    #                                                                      #
    # Hint 4: Check out the pytorch layer norm module                      #
    ########################################################################

    self.multi_head = MultiHeadAttention(d_model=d_model, d_k=d_k, d_v=d_v, n_heads=n_heads, dropout=dropout)
    self.layer_norm1 = nn.LayerNorm(normalized_shape=d_model)
    self.ffn = FeedForwardNeuralNetwork(d_model=d_model, d_ff=d_ff, dropout=dropout)
    self.layer_norm2 = nn.LayerNorm(normalized_shape=d_model)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

```python
def forward(self,
            inputs: torch.Tensor,
            pad_mask: torch.Tensor = None) -> torch.Tensor:
    """

    Args:
        inputs: Inputs to the Encoder Block
        pad_mask: Optional Padding Mask

    Shape:
        - inputs: (batch_size, sequence_length, d_model)
        - pad_mask: (batch_size, sequence_length, sequence_length)
        - outputs: (batch_size, sequence_length, d_model)
    """
    outputs = None

    ########################################################################
    # TODO:                                                                #
    #   Task 4: Implement the forward pass of the encoder block            #
    #   Task 10: Pass on the padding mask                                  #
    #                                                                      #
    # Hint 4: Don't forget the residual connection! You can forget about   #
    #         the pad_mask for now!                                        #
    ########################################################################

    outputs = self.multi_head(q=inputs, k=inputs, v=inputs, mask=pad_mask) + inputs
    outputs = self.layer_norm1(outputs)
    outputs = self.ffn(outputs) + outputs
    outputs = self.layer_norm2(outputs)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return outputs
```

ScaledDotAttention

```python
def __init__(self,
             d_k,
             dropout: float = 0.0):
    """

    Args:
        d_k: Dimension of Keys and Queries
        dropout: Dropout probability
    """
    super().__init__()
    self.d_k = d_k

    self.softmax = nn.Softmax(dim=-1)
    self.dropout = None

    ########################################################################
    # TODO:                                                                #
    #   Task 11: Initialize the dropout layer (torch.nn implementation)    #
    #                                                                      #
    ########################################################################

    self.dropout = nn.Dropout(p=dropout)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

```python
def forward(self,
            q: torch.Tensor,
            k: torch.Tensor,
            v: torch.Tensor,
            mask: torch.Tensor = None) -> torch.Tensor:
    scores = torch.matmul(q, k.transpose(-2, -1)) / (self.d_k ** 0.5)

    ########################################################################
    # TODO:                                                                #
    #   Task 6:                                                            #
    #       - Add a negative infinity mask if a mask is given              #
    #                                                                      #
    # Hint 6:                                                              #
    #       - Have a look at Tensor.masked_fill_() or use torch.where()    #
    ########################################################################

    if mask is not None:
        scores.masked_fill_(~mask, -torch.inf)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    scores = self.softmax(scores)

    ########################################################################
    # TODO:                                                                #
    #   Task 11:                                                           #
    #       - Add dropout to the scores                                    #
    ########################################################################

    scores = self.dropout(scores)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    outputs = torch.matmul(scores, v)

    SCORE_SAVER.save(scores)

    return outputs
```

Remark:
Fill with –Inf to make the scores after the softmax 0.

MultiHeadAttention - __init__()

```python
def __init__(self,
             d_model: int,
             d_k: int,
             d_v: int,
             n_heads: int,
             dropout: float = 0.0):
    super().__init__()

    self.n_heads = n_heads
    self.d_k = d_k
    self.d_v = d_v

    self.weights_q = nn.Linear(in_features=d_model, out_features=n_heads * d_k, bias=False)
    self.weights_k = nn.Linear(in_features=d_model, out_features=n_heads * d_k, bias=False)
    self.weights_v = nn.Linear(in_features=d_model, out_features=n_heads * d_v, bias=False)

    self.attention = ScaledDotAttention(d_k=d_k, dropout=dropout)

    self.project = nn.Linear(in_features=n_heads * d_v, out_features=d_model, bias=False)

    self.dropout = None

    ########################################################################
    # TODO:                                                                #
    #  Task 11:                                                            #
    #       -Initialize the dropout layer (torch.nn implementation)       #
    #                                                                      #
    ########################################################################

    self.dropout = nn.Dropout(p=dropout)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

MultiHeadAttention – forward()

```python
def forward(self,
            q: torch.Tensor,
            k: torch.Tensor,
            v: torch.Tensor,
            mask: torch.Tensor = None) -> torch.Tensor:
    batch_size, sequence_length_queries, _ = q.size()
    _, sequence_length_keys, _ = k.size()

    q = self.weights_q(q)
    k = self.weights_k(k)
    v = self.weights_v(v)

    q = q.reshape(batch_size, sequence_length_queries, self.n_heads, self.d_k)
    q = q.transpose(-3, -2)

    k = k.reshape(batch_size, sequence_length_keys, self.n_heads, self.d_k)
    k = k.transpose(-3, -2)

    v = v.reshape(batch_size, sequence_length_keys, self.n_heads, self.d_v)
    v = v.transpose(-3, -2)

    ########################################################################
    # TODO:                                                                #
    #   Task 6:                                                            #
    #       - If a mask is given, add an empty dimension at dim=1          #
    #       - Pass the mask to the ScaledDotAttention layer                #
    #                                                                      #
    # Hints 6:                                                             #
    #       - Use unsqueeze() to add dimensions at the correct location    #
    ########################################################################

    if mask is not None:
        mask = mask.unsqueeze(1)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

```python
outputs = self.attention(q, k, v, mask)

outputs = outputs.transpose(-3, -2)
outputs = outputs.reshape(batch_size, sequence_length_queries, self.n_heads * self.d_v)

outputs = self.project(outputs)

########################################################################
# TODO:                                                                #
#  Task 11:                                                            #
#       - Add dropout as a final step after the projection layer       #
#                                                                      #
########################################################################

outputs = self.dropout(outputs)

########################################################################
#                           END OF YOUR CODE                           #
########################################################################

return outputs
```

DecoderBlock – __init__()

```python
def __init__(self,
             d_model: int,
             d_k: int,
             d_v: int,
             n_heads: int,
             d_ff: int,
             dropout: float = 0.0):
    super().__init__()

    self.causal_multi_head = None
    self.layer_norm1 = None
    self.cross_multi_head = None
    self.layer_norm2 = None
    self.ffn = None
    self.layer_norm3 = None

    ########################################################################
    # TODO:                                                                #
    #   Task 7: Initialize the Decoder Block                               #
    #           You will need:                                             #
    #                           - Causal Multi-Head Self-Attention layer   #
    #                           - Layer Normalization                      #
    #                           - Multi-Head Cross-Attention layer         #
    #                           - Layer Normalization                      #
    #                           - Feed forward neural network layer        #
    #                           - Layer Normalization                      #
    #                                                                      #
    # Hint 7: Check out the pytorch layer norm module                      #
    ########################################################################

    self.causal_multi_head = MultiHeadAttention(d_model=d_model, d_k=d_k, d_v=d_v, n_heads=n_heads, dropout=dropout)
    self.layer_norm1 = nn.LayerNorm(normalized_shape=d_model)
    self.cross_multi_head = MultiHeadAttention(d_model=d_model, d_k=d_k, d_v=d_v, n_heads=n_heads, dropout=dropout)
    self.layer_norm2 = nn.LayerNorm(normalized_shape=d_model)
    self.ffn = FeedForwardNeuralNetwork(d_model=d_model, d_ff=d_ff, dropout=dropout)
    self.layer_norm3 = nn.LayerNorm(normalized_shape=d_model)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

DecoderBlock – forward()

```python
def forward(self,
            inputs: torch.Tensor,
            context: torch.Tensor,
            causal_mask: torch.Tensor,
            pad_mask: torch.Tensor = None) -> torch.Tensor:
    outputs = None

    ########################################################################
    # TODO:                                                                #
    #   Task 7: Implement the forward pass of the decoder block            #
    #   Task 10: Pass on the padding mask                                  #
    #                                                                      #
    # Hint 7:                                                              #
    #       - Don't forget the residual connections!                       #
    #       - Remember where we need the causal mask, forget about the     #
    #         other mask for now!                                          #
    # Hints 10:                                                            #
    #       - We have already combined the causal_mask with the pad_mask   #
    #         for you, all you have to do is pass it on to the "other"     #
    #         module                                                       #
    ########################################################################

    outputs = self.causal_multi_head(q=inputs, k=inputs, v=inputs, mask=causal_mask) + inputs
    outputs = self.layer_norm1(outputs)
    outputs = self.cross_multi_head(q=outputs, k=context, v=context, mask=pad_mask) + outputs
    outputs = self.layer_norm2(outputs)
    outputs = self.ffn(outputs) + outputs
    outputs = self.layer_norm3(outputs)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return outputs
```

Transformer

```python
################################################################################
# TODO:                                                                        #
#   Task 9: Initialize the transformer!                                        #
#            You will need:                                                    #
#               - An embedding layer                                           #
#               - An encoder                                                   #
#               - A decoder                                                    #
#               - An output layer                                              #
#                                                                              #
# Hint 9: Have a look at the output shape of the decoder and the               #
#         output shape of the transformer model to figure out the              #
#         dimensions of the output layer! We will not need a bias!             #
################################################################################

self.embedding = Embedding(vocab_size=self.vocab_size,
                           d_model=self.d_model,
                           max_length=self.max_length,
                           dropout=self.dropout)

self.encoder = Encoder(d_model=self.d_model,
                       d_k=self.d_k,
                       d_v=self.d_v,
                       n_heads=self.n_heads,
                       d_ff=self.d_ff,
                       n=self.n,
                       dropout=self.dropout)

self.decoder = Decoder(d_model=self.d_model,
                       d_k=self.d_k,
                       d_v=self.d_v,
                       n_heads=self.n_heads,
                       d_ff=self.d_ff,
                       n=self.n,
                       dropout=self.dropout)

self.output_layer = nn.Linear(in_features=self.d_model,
                              out_features=self.vocab_size,
                              bias=False)

################################################################################
#                           END OF YOUR CODE                                   #
################################################################################
```

```python
def forward(self,
            encoder_inputs: torch.Tensor,
            decoder_inputs: torch.Tensor,
            encoder_mask: torch.Tensor = None,
            decoder_mask: torch.Tensor = None) -> torch.Tensor:
    outputs = None

    ############################################################################
    # TODO:                                                                    #
    #   Task 9: Implement the forward pass of the transformer!                 #
    #            You will need to:                                             #
    #               - Compute the encoder embeddings                           #
    #               - Compute the forward pass through the encoder             #
    #               - Compute the decoder embeddings                           #
    #               - Compute the forward pass through the decoder             #
    #               - Compute the output logits                                #
    #   Task 10: Pass on the encoder and decoder padding masks!                #
    #                                                                          #
    # Hints 10: Have a look at the forward pass of the encoder and decoder     #
    #           to figure out which masks to pass on!                          #
    ############################################################################

    encoder_inputs = self.embedding(encoder_inputs)
    encoder_outputs = self.encoder(encoder_inputs,
                                   encoder_mask=encoder_mask)

    decoder_inputs = self.embedding(decoder_inputs)
    decoder_outputs = self.decoder(decoder_inputs,
                                   encoder_outputs,
                                   decoder_mask=decoder_mask,
                                   encoder_mask=encoder_mask)

    outputs = self.output_layer(decoder_outputs)

    ############################################################################
    #                           END OF YOUR CODE                               #
    ############################################################################

    return outputs
```

Embedding

```python
def __init__(self,
             vocab_size: int,
             d_model: int,
             max_length: int,
             dropout: float = 0.0):
    """

    Args:
        vocab_size: Number of elements in the vocabulary
        d_model: Dimension of Embedding
        max_length: Maximum sequence length
    """
    super().__init__()

    self.embedding = nn.Embedding(num_embeddings=vocab_size,
                                  embedding_dim=d_model)

    self.pos_encoding = nn.Parameter(data=positional_encoding(d_model=d_model, max_length=max_length),
                                     requires_grad=False)

    self.dropout = None

    ########################################################################
    # TODO:                                                                #
    #   Task 11: Initialize the dropout layer (torch.nn implementation)    #
    #                                                                      #
    ########################################################################

    self.dropout = nn.Dropout(p=dropout)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

```python
def forward(self,
            inputs: torch.Tensor) -> torch.Tensor:
    """
    The forward function takes in tensors of token ids and transforms them into vector embeddings.
    It then adds the positional encoding to the embeddings, and if configured, performs dropout on the layer!

    Args:
        inputs: Batched Sequence of Token Ids

    Shape:
        - inputs: (batch_size, sequence_length)
        - outputs: (batch_size, sequence_length, d_model)
    """

    sequence_length = inputs.shape[-1]
    outputs = self.embedding(inputs) + self.pos_encoding[:sequence_length]

    ########################################################################
    # TODO:                                                                #
    #   Task 11:                                                           #
    #       - Add dropout to the outputs                                   #
    ########################################################################

    outputs = self.dropout(outputs)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################

    return outputs
```

Trainer

```python
def _forward(self, batch: dict, metrics):
    """
    Forward pass through the model. Updates metric object with current stats.

    Args:
        batch (dict): Input data batch.
        metrics: Metrics object for tracking.
    """
    loss = None
    outputs = None
    labels = None
    label_mask = None

    ########################################################################
    # TODO:                                                                #
    #   Task 13:                                                           #
    #       - Unpack the batch                                             #
    #       - Move all tensors to self.device                              #
    #       - Compute the outputs of self.model                            #
    #       - Compute the loss using self.loss_func                        #
    #                                                                      #
    # Hints: Inspect the outputs of collate method  - __call__()  - in     #
    #        data/collator.py                                              #
    #        Inspect the inputs of the SmoothCrossEntropy                  #
    #        Make sure to pass all masks to the model!                     #
    ########################################################################

    encoder_inputs = batch['encoder_inputs'].to(self.device)
    decoder_inputs = batch['decoder_inputs'].to(self.device)
    encoder_mask = batch['encoder_mask'].to(self.device)
    decoder_mask = batch['decoder_mask'].to(self.device)

    outputs = self.model(encoder_inputs, decoder_inputs, encoder_mask, decoder_mask)

    labels = batch['labels'].to(self.device)
    label_mask = batch['label_mask'].to(self.device)
    lengths = batch['label_length'].to(self.device)

    loss = self.loss_func(outputs, labels, label_mask, lengths)

    ########################################################################
    #                            END OF YOUR CODE                          #
    ########################################################################

    metrics.update_loss(loss.item())
    metrics.update_words(batch['label_length'].sum().item())
    metrics.update_correct_words(torch.sum((torch.argmax(outputs, -1) == labels) * label_mask).item())

    return loss
```

Hyperparameters

```
############################################################################
# TODO:                                                                    #
#   Initialize your tokenizer. You train your own tokenizer and load      #
#   load it like we did in the first notebook, or load the pretrained     #
#   version!                                                               #
#                                                                          #
# Hint: Scroll up a couple of cells for the default tokenizer             #
############################################################################

tokenizer = load_pretrained_fast()

############################################################################
#                             END OF YOUR CODE                             #
############################################################################

hparams = None

############################################################################
# TODO:                                                                    #
#   Implement you model here                                               #
#                                                                          #
############################################################################

hparams = {
    'd_model': 256,
    'd_k': 32,
    'd_v': 32,
    'd_ff': 512,
    'n_heads': 3,
    'n': 2,
    'dropout': 0
}

############################################################################
#                             END OF YOUR CODE                             #
############################################################################
model = Transformer(vocab_size=len(tokenizer),
                    eos_token_id=tokenizer.eos_token_id,
                    hparams=hparams)
```

```
############################################################################
# TODO:                                                                    #
#   Define the optimizer and optionally scheduler - not really needed     #
#                                                                          #
############################################################################

optimizer = Adam(model.parameters(), lr=1e-3)

epochs = 100
batch_size = 20

############################################################################
#                             END OF YOUR CODE                             #
############################################################################
```

Questions? Piazza ☺