```
predict_sentiment(model, tokenizer, "This film is great")
```

Out[18]: 0.7384241223335266

Positive Review

In [19]:
```
predict_sentiment(model,tokenizer,
"This film was actually good. I loved the acting and I really enjoyed the music. Definitely a must watch for all the fans.")
```

Out[19]: 0.9798356294631958

Negative Review

In [5]:
```
predict_sentiment(model,tokenizer,
"This film was horrible. I hated the acting and I despised the music. Avoid this movie it's not worth it.")
```

0.0231458347894251

# Conceptual Questions

1. Why is the residual connection is crucial in the Transformer architecture? [5 points]

We need residual connections in transformers to mitigate the vanishing gradient problem. They allow the gradients to flow through the network directly and help facilitate back propagation.

2. Why is Layer Normalization important in the Transformer architecture? [5 points]

Layer norm calculated the normalisation over each feature, instead of each batch. This stabilises the network and has an effect similar to batchnorm in CNNs. This results in substantially reduced training times.

3. Why do we use the scaling factor of $1/\sqrt{d_k}$ in Scaled Dot Product Attention? If we remove it, what is going to happen? [5 points]

The variables Q and K are independent and are of size dk*dk. Their dot product has a variance of dk, and a mean of 0. The dot products quickly become larger values and hence, the softmax of the same will be very high, and therfore, very small gradients will flow back. So, we scale it down by a factor of root(dk) to bring down the variance to 1. If we dont have the scaling factor, training will be very slow, since very little gradient can flow backward.

# Problem 2 - Transformers for Language Modeling and Sentiment Analysis

Open in Colab

In this problem we will learn how to implement the building blocks for "Transformer" models, and then implement a pre-training procedure for such models via BERT-style language modeling and then fine-tune a pre-trained model on sentiment analysis tasks on the IMDB movie review dataset. Typically, transformer models are very large and are pre-trained on language modeling tasks with massive datasets with huge computational resources. As such, we will only implement the pre-training *procedure*, without expecting you to pre-train a model to completion. We will then load in a pre-trained model for you to perform fine-tuning on a sentiment analysis task.

We will complete the following steps in this problem:

1. Implement a multi-head-attention (MHA) layer.
2. Implement "Transformer block" layers which use MHA layers, linear layers, and residual connections.
3. Implement a full Transformer model comprised of Transformer blocks.
4. Implement BERT-style language model pre-training for the Transformer model.
5. Fine-tune our trained language model on a sentiment analysis task.

In order to run on GPU in Colab go to `Runtime -> Change runtime type` and select `GPU` under the `Hardware accelerator` drop-down box.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

import math
import random
import numpy as np

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

## 1 - Scaled Dot Product Attention [8 points]

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. Here we use the following definition: *the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys.* In other words, we want to dynamically decide on which inputs we want to "attend" more than others based on their values. In particular, an attention mechanism has usually **4 parts** we need to specify:

- **Query**: The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys**: For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is "offering", or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- **Values**: For each input element, we also have a value vector. This feature vector is the one we want to average over.
- **Score function**: To rate which elements we want to pay attention to, we need to specify a score function $f_{attn}$. The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

The weights of the average are calculated by a softmax over all score function outputs. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query.

The attention applied inside the Transformer architecture is called self-attention. In self-attention, each sequence element provides a key, value, and query. For each element, we perform an attention layer where based on its query, we check the similarity of the all sequence elements' keys, and returned a different, averaged value vector for each element.

The core concept behind self-attention is the scaled dot product attention. The dot product attention takes as input a set of queries $Q \in R^{T \times d_k}$, keys $K \in R^{T \times d_k}$ and values $V \in R^{T \times d_v}$ where $T$ is the sequence length, and $d_k$ and $d_v$ are the hidden dimensionality for queries/keys and values respectively. The attention value from element $i$ to $j$ is based on its similarity of the query $Q_i$ and key $K_j$, using the dot product as the similarity metric. Mathmatically:

$$Attention(Q,K,V)=\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The matrix multiplication $QK^T$ performs the dot product for every possible pair of queries and keys, resulting in a matrix of the shape $T \times T$. Each row represents the attention logits for a specific element $i$ to all other elements in the sequence. We apply a softmax and multiply with the value vector to obtain a weighted mean (the weights being determined by the attention). The computation graph is visualized below.

```
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
```

```
#################################### TODO
####################################
# Implement the scaled dot product attention described above.
########################################################################
######
def scaled_dot_product(q, k, v, attn_drop_rate=0.1, mask=None):
    """
    Parameters:
      q: query, shape: (batch, # heads, seq len, head dimension)
      k: keys, shape: (batch, # heads, seq len, head dimension)
      v: value, shape: (batch, # heads, seq len, head dimension)
      attn_drop_rate: probability of an element to be zeroed,
      mask: the optional masking of specific entries in the attention
matrix.
              shape: (batch, seq len)
    """
    # TODO: get hidden dimensionality d_k for query/keys.
    d_k = q.size()[-1]
    # TODO: compute (QK^T)/d_k, use
https://pytorch.org/docs/stable/generated/torch.matmul.html.
    attn_logits = torch.matmul(q, k.transpose(-2, -1)) /
math.sqrt(d_k)

    # TODO: if mask is not None, apply mask. use
https://pytorch.org/docs/stable/generated/torch.Tensor.masked_fill_.ht
ml#torch.Tensor.masked_fill_.
    # Make sure that padding tokens cannot be attended to by
subtracting a
    # large negative value from the columns of attention weights
    # corresponding to the tokens that have mask = 1. These will
become 0
    # after the softmax.
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 1, -1 *
float('inf'))

    # TODO: compute softmax((QK^T)/d_k). Normalize attention weights
to sum to 1 with a softmax over the key dimension.
    attention = F.softmax(attn_logits, dim=-1)

    # TODO: Add dropout to attention weights w/ attn_drop_rate.
    attention = F.dropout(attention, attn_drop_rate)

    # TODO: compute softmax((QK^T)/d_k)V.
    values = torch.matmul(attention, v)

    return values, attention
```

Before you continue, run the test code listed below. It will generate random queries, keys, and value vectors, and calculate the attention outputs. Make sure you can follow the calculation of the specific values here, and also check it by hand.

```
bs = 1
num_heads = 1
seq_len, d_k = 3, 2
q = torch.randn(bs, num_heads, seq_len, d_k)
k = torch.randn(bs, num_heads, seq_len, d_k)
v = torch.randn(bs, num_heads, seq_len, d_k)
mask = torch.bernoulli(0.5 * torch.ones(bs, seq_len))
values, attention = scaled_dot_product(q, k, v, 0.0, mask)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Mask\n", mask)
print("Values\n", values)
print("Attention\n", attention)
```

```
Q
 tensor([[[[ 0.0461,  0.4024],
          [-1.0115,  0.2167],
          [-0.6123,  0.5036]]]])
K
 tensor([[[[ 0.2310,  0.6931],
          [-0.2669,  2.1785],
          [ 0.1021, -0.2590]]]])
V
 tensor([[[[-0.1549, -1.3706],
          [-0.1319,  0.8848],
          [-0.2611,  0.6104]]]])
Mask
 tensor([[0., 1., 0.]])
Values
 tensor([[[[-0.2007, -0.5155],
          [-0.2066, -0.4067],
          [-0.2005, -0.5194]]]])
Attention
 tensor([[[[0.5683, 0.0000, 0.4317],
          [0.5134, 0.0000, 0.4866],
          [0.5703, 0.0000, 0.4297]]]])
```

## 2 - Build Multi-Head-Attention Layer [8 points]

Now we will implement multi-head-attention, first introduced by Attention is All you Need (Vaswani et al. 2017). The scaled dot product attention allows a network to attend over a sequence. However, often there are multiple different aspects a sequence element wants to attend to, and a single weighted average is not a good option for it. This is why we extend the attention mechanisms to multiple heads, i.e. multiple different query-key-value triplets on the same features.

A multi-head-attention layer works by employing several self-attention layers in parallel. Given a query, key, and value matrix, we transform those into $h$ sub-queries, sub-keys, and sub-values, which we pass through the scaled dot product attention independently where $h$ is the number of heads. Afterward, we concatenate the heads and combine them with a final weight matrix. Mathmatically,

$$Multihead(Q,K,V)=Concat(head_1,\ldots,head_h)W^O,$$

where

$$head_i=Attention(QW_i^Q,KW_i^K,VW_i^V).$$

We refer to this as Multi-Head Attention layer with the learnable parameters $W_{1\ldots h}^Q \in R^{d_{in}\times d_k}$, $W_{1\ldots h}^K \in R^{d_{in}\times d_k}$, $W_{1\ldots h}^V \in R^{d_{in}\times d_v}$, and $W^O \in R^{h\cdot d_k \times d_{out}}$ where $d_{in}$ is the input dimensionality, and $d_{out}$ is the output dimensionality. The visualized computational graph is shown below.

Looking at the computation graph above, a simple but effective implementation is to set the current feature map $X$ in a NN, $X \in R^{B\times T\times d_{model}}$, where $B$ is the batch size, $T$ is the sequence length, and $d_{model}$ is the hidden dimentionality of $X$.

```
#################################### TODO
####################################
# Implement Multi-head attention described above.
########################################################################
######
class MultiHeadAttention(nn.Module):
  def __init__(self, embed_dim, n_heads, attn_drop_rate):
    """
    Parameters:
      input_dim: The input dimension.
      embed_dim: The embedding dimension of the model
      n_heads: Number of attention heads
      attn_drop_rate: Dropout rate for attention weights (Q K^T)
    """
    super().__init__()
    self.embed_dim = embed_dim
    self.n_heads = n_heads
    self.head_dim = embed_dim // n_heads
    self.attn_drop_rate = attn_drop_rate

    # TODO: Add learnable parameters for computing query, key, and
value using nn.Linear.
    # Store all weight matrices W^Q, W^K, W^V 1...h together for
efficiency.
    self.qkv_proj = nn.Linear(embed_dim, 3*embed_dim)
```

```
    # TODO: Add learnable parameters W^O using nn.Linear.
    self.o_proj = nn.Linear(self.embed_dim, self.embed_dim)

    self._reset_parameters()

  def _reset_parameters(self):
      # Original Transformer initialization, see PyTorch documentation
      nn.init.xavier_uniform_(self.qkv_proj.weight)
      self.qkv_proj.bias.data.fill_(0)
      nn.init.xavier_uniform_(self.o_proj.weight)
      self.o_proj.bias.data.fill_(0)

  def forward(self, embedding, mask):
    """
    Inputs:
      embedding: Input embedding with shape (batch size, sequence
length, embedding dimension)
      mask: Mask specifying padding tokens with shape (batch_size,
sequence length)
        Value for tokens that should be masked out is 1 and 0
otherwise.
    Outputs:
      Attended values
    """



    # TODO: get batch_size, seq_length, embed_dim.
    batch_size, seq_length, embed_dim = embedding.size()

    # TODO: Compute queries, keys, and values (keep continguous for
now).
    qkv = self.qkv_proj(embedding)

    # TODO: Separate Q, K, V from linear output, give each shape
[batch, num_head, seq_len, head_dim] (may require
transposing/permuting dimensions)
    q, k, v = qkv.reshape(batch_size, seq_length, self.n_heads, 3 *
self.head_dim).permute(0, 2, 1, 3).chunk(3, dim=-1)

    # TODO: Determine value outputs, with shape [batch, seq_len,
num_head, head_dim]. (hint: use scaled_dot_product())
    if mask is not None:
      values, attention = scaled_dot_product(q, k, v,
mask=mask.reshape((mask.shape[0], 1, mask.shape[1], 1)))
    else:
      values, attention = scaled_dot_product(q, k, v, mask=mask)

    # TODO: Linearly project attention outputs w/ W^O.
    # The final dimensionality should match that of the inputs.
```

```
    attended_embeds = values.permute(0, 2, 1, 3).reshape(batch_size, 
seq_length, embed_dim)

    return attended_embeds
```

Let's check that your MHA layer works and returns a tensor of the correct shape

```
embed_dim = 16
n_heads = 4
attn_drop_rate = 0.1
layer = MultiHeadAttention(embed_dim, n_heads, attn_drop_rate)

bs = 3
seq_len = 2
inputs = torch.randn(bs, seq_len, embed_dim)
mask = torch.zeros(bs, seq_len)
outputs = layer(inputs, mask)
out_bs, out_seq_len, out_hidden = outputs.shape
print("Output shape: ", (out_bs, out_seq_len, out_hidden))
assert out_bs == bs and out_seq_len == seq_len and out_hidden == 
embed_dim, "Unexpected output shape"
```

```
Output shape:  (3, 2, 16)
```

# 3 - Build Transformer Blocks [8 points]

Now we construct the blocks from which transformer models are comprised of.

Originally, the Transformer model was designed for machine translation. Hence, it got an encoder-decoder structure where the encoder takes as input the sentence in the original language and generates an attention-based representation, the decoder attends over the encoded information and generates the translated sentence in an autoregressive manner, as in a standard RNN. The visualized computational graph is shown below. Here we will mainly focus on the encoder part and implement the encoder block.

A Transformer encoder block consists of the following modules in this order:

1. Multi-Head Attention (we implemented above)
2. Dropout
3. Residual connection to the input (simply add the input of the block to the output of the previous dropout layer).
4. Layer Norm - https://arxiv.org/abs/1607.06450
5. Linear layer
6. Activation function (typically gelu - https://arxiv.org/abs/1606.08415)
7. Linear layer
8. Dropout
9. Residual connection to 4 (add the output of 4 to 8)

10. Layer Norm

According to the listed modules, please implement:

```
class TransformerBlock(nn.Module)
```

```
################################### TODO
###################################
# Implement transformer encoder block
######################################################################
######
class TransformerBlock(nn.Module):
  def __init__(self, embed_dim, n_heads, attn_drop_rate,
layer_drop_rate):
    """
    Parameters:
      input_dim: Dimensionality of the input
      embed_dim: The embedding dimension of the model
      n_heads: Number of attention heads
      attn_drop_rate: Dropout rate for attention weights (Q K^T)
      layer_drop_rate: Dropout rate for activations
    """
    super().__init__()
    self.embed_dim = embed_dim
    self.n_heads = n_heads
    self.layer_dropout = nn.Dropout(layer_drop_rate)

    # TODO: define attention layer
    self.self_attn = MultiHeadAttention(embed_dim, n_heads,
attn_drop_rate)

    # TODO: define a network (using nn.Sequential) with:
    # 1) a linear layer, 2) an activation layer, 3) another linear
layer, 4) a dropout layer.
    self.linear_net = nn.Sequential(nn.Linear(embed_dim,
embed_dim),nn.GELU(),nn.Linear(embed_dim, embed_dim),
    nn.Dropout(layer_drop_rate))

    # TODO: define 2 norm layers, 1 dropout layer.
    self.norm1 = nn.LayerNorm(embed_dim)
    self.norm2 = nn.LayerNorm(embed_dim)
    self.dropout = nn.Dropout(layer_drop_rate)

  def forward(self, inputs):
    embedding, mask = inputs

    # TODO: 1. compute multi-head attention
    attn_out = self.self_attn(embedding, mask=mask)

    # TODO: 2. add dropout
```

```python
    dropout_out = self.dropout(attn_out)

    # TODO: 3. add residual connection to the input
    embedding = embedding + dropout_out

    # TODO: 4. apply layernorm
    embedding = self.norm1(embedding)

    # TODO: 5-8. compute 1) a linear layer, 2) an activation layer, 3) another linear layer, 4) a dropout layer.
    linear_out = self.linear_net(embedding)

    # TODO: 9. add residual connection
    embedding = embedding + linear_out

    # TODO: 10. apply layer norm
    embedding = self.norm2(embedding)

    return embedding, mask
```

Let's once again check that the code runs without error and outputs the correct shape (note, this is not a guarantee that you have implemented it correctly).

```python
embed_dim = 16
n_heads = 4
attn_drop_rate = 0.1
layer_drop_rate = 0.1
block = TransformerBlock(embed_dim, n_heads, attn_drop_rate,
layer_drop_rate)

bs = 3
seq_len = 2
embeds = torch.randn(bs, seq_len, embed_dim)
mask = torch.zeros(bs, seq_len)
outputs, _ = block((embeds, mask))
out_bs, out_seq_len, out_hidden = outputs.shape
print("Output shape: ", (out_bs, out_seq_len, out_hidden))
assert out_bs == bs and out_seq_len == seq_len and out_hidden ==
embed_dim, "Unexpected output shape"
```

```
Output shape:  (3, 2, 16)
```

## 4 - Position Encoding [0 points]

In tasks like language understanding, the position is important for interpreting the input words. The position information can therefore be added via the input features. We could learn a embedding for every possible position, but this would not generalize to a dynamical input sequence length. Hence, the better option is to use feature patterns that the network can identify from the features and potentially generalize to larger sequences. Mathmatically:

$$PE(pos,i) = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{model}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{model}}}\right) & \text{otherwise} \end{cases}$$

$PE(pos,i)$ represents the position encoding at position $pos$ in the sequence, and hidden dimensionality $i$. These values, concatenated for all hidden dimensions, are added to the original input features, and constitute the position information. The intuition behind this encoding is that you can represent $PE(pos+k,:)$ as a linear function of $PE(pos,:)$, which might allow the model to easily attend to relative positions.

```
class PositionalEncoding(nn.Module):

    def __init__(self, embed_dim: int, drop_rate=0.1, max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(p=drop_rate)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, embed_dim, 2) * (-
math.log(10000.0) / embed_dim))
        pe = torch.zeros(1, max_len, embed_dim)
        pe[0, :, 0::2] = torch.sin(position * div_term)
        pe[0, :, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Args:
            x: Tensor, shape [batch_size, seq_len, embedding_dim]
        """
        x = x + self.pe[:, :x.size(1)]
        return self.dropout(x)
```

## 5 - Build a BERT model [8 points]

A BERT model consists of:

1. **An input embedding layer.** This converts a token index into a vector embedding. Make sure to include an extra embedding for the masked tokens! In other words, learn vocab_size + 1 embeddings.
2. **Positional encodings.** This layer (implemented for you already) encodes the position of each token since multi-head-attention layers have no notion of positional

locality or order. It takes as input the the token embeddings from (1) and returns them with positional embeddings added.

3. Several stacked **Transformer blocks** (the number specified by n_layers)
4. **Output linear layer** that predicts masked words for pre-training. Takes final embedding of last block and outputs probability distribution over the vocabulary.

```
################################### TODO
###################################
# Add the requisite modules for a BERT model
######################################################################
######
class BertModel(nn.Module):
  def __init__(self, n_layers, vocab_size, embed_dim, n_heads,
attn_drop_rate, layer_drop_rate):
    super().__init__()

    # TODO: 1. add input embedding layer (hint: use nn.Embedding) -
don't forget about the mask token
    self.embed = nn.Embedding(vocab_size+1, embed_dim)
    #vocabsize+1 for mask token

    # TODO: 2. add positional encoding
    self.pos_embed = PositionalEncoding(embed_dim, layer_drop_rate)

    # TODO: 3. add stacked transformer blocks (use nn.Sequential)
    ss = [TransformerBlock(embed_dim, n_heads, attn_drop_rate,
layer_drop_rate)]
    for i in range(n_layers-1):
      ss.append(TransformerBlock(embed_dim, n_heads, attn_drop_rate,
layer_drop_rate))
    #Refer Sequential document
    self.net = nn.Sequential(*ss)
    # TODO: 4. add output linear layer that predicts masked words for
pre-training
    self.mask_pred = nn.Linear(embed_dim, vocab_size)

  def forward(self, batch_text, mask=None):
    # TODO: implement forward pass (embedding -> stacked blocks ->
output masked word predictions)
    embed, m =
self.net.forward((self.pos_embed.forward(self.embed(batch_text)),
mask))
    mask_preds = self.mask_pred(embed)

    #mask_preds =
self.mask_pred(self.net(self.pos_embed(self.embed(batch_text, mask))))
    return mask_preds
```

Let's once again check that the code runs without error and outputs the correct shape (note, this is not a guarantee that you have implemented it correctly).

```
embed_dim = 16
n_heads = 4
n_layers = 2
vocab_size = 10
attn_drop_rate = 0.1
layer_drop_rate = 0.1
model = BertModel(n_layers, vocab_size, embed_dim, n_heads,
attn_drop_rate, layer_drop_rate)

bs = 3
seq_len = 2
inputs = torch.randint(0, vocab_size, (bs, seq_len))
mask_preds = model(inputs)
out_bs, out_seq_len, out_vocab = mask_preds.shape
print("Mask predictions shape: ", (out_bs, out_seq_len, out_vocab))
assert out_bs == bs and out_seq_len == seq_len and out_vocab ==
vocab_size, "Unexpected mask prediction output shape"
```

```
Mask predictions shape:  (3, 2, 10)
```

## 6 - Implement BERT Pre-Training [8 points]

In order to pre-train our language model, we randomly permute `mask_rate`% of the tokens and attempt to predict the original tokens. The permutation is as follows:

- In 80% of these cases we replace the token with a `<mask>` token. Use `MASK_TOKEN_IND` as the index of this token.
- In 10% of these cases we replace the token with a random token.
- In the final 10% we do not permute the token.

The prediction task is then to predict the original token for *only* the permuted tokens. You should use `nn.CrossEntropyLoss`. Note that this module has a keyword argument `ignore_index` which specifies a label index for which we do not compute the loss. It is `-100` by default. This can be used to **only** do prediction for the permuted tokens.

For more details, please look at Task 1 in Section 3.1 of the BERT paper. We do not consider the second pre-training task (Next Sentence Prediction) for this assignment.

**We do not expect you to complete the pre-training procedure, which is not feasible given your computational resources. We are simply asking you to implement one step of training with synthetic data.**

```
batch_size = 128
learning_rate = 1e-4
n_layers = 2  # number of transformer blocks in model
embed_dim = 64
n_heads = 4
attn_drop_rate = 0.1  # dropout rate on attention weights
layer_drop_rate = 0.1  # dropout rate on activations
```

```
mask_rate = 0.15  # rate at which we permute words in order to predict
them
vocab_size = 100
MASK_TOKEN_IND = vocab_size
PAD_IND = 0
model = BertModel(n_layers, vocab_size, embed_dim, n_heads,
attn_drop_rate, layer_drop_rate)
opt = torch.optim.AdamW(model.parameters(), lr=learning_rate,
weight_decay=0.01)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model.to(device)
model.train()

def mask_inputs(text, only_mask=False):
  """
  Inputs:
    text: Batch of sequences of shape (batch_size, seq_len) and type
torch.Long
          Each token is represented by its index in the vocabulary.
    only_mask: If this is true, only replace tokens with <mask>
tokens, no
               random tokens, or keeping tokens the same. This is used
for
               evaluation only.
  Outputs:
    masked_text: Permuted inputs based on rules defined in description
above.
    mask_labels: Labels for prediction. Use label -100 for tokens that
we do not
                want to predict. Should have the same shape as input
text.

  """
  masked_text = text.clone()
  mask_labels = text.clone()
  #################################### TODO
###################################
  # Implement random permutation of tokens based on mask_rate, store
the masked
  # sequences in masked_text. Note, you have access to mask_rate,
  # MASK_TOKEN_IND, etc. inside this function. Also store the
prediction labels
  # for the pre-training task in mask_labels. Make sure to set the
labels for
  # non-permuted tokens as well as padding tokens to -100

######################################################################
```

```
######
  # print(masked_text)
  shuffle_idxs = torch.randperm(torch.unique(masked_text).shape[0])
  print(shuffle_idxs)
  tokens = torch.unique(masked_text)[shuffle_idxs]#.to(device)
  mask = torch.rand_like(text.double())
  mask = mask < mask_rate
  mask_locs = (mask == True).nonzero()
  range_80s = ( int(mask_locs.size()[0] * 0.80))
  for i in range(range_80s):
    masked_text[mask_locs[i][0], mask_locs[i][1]] = MASK_TOKEN_IND
  for i, j in enumerate(range(range_80s, range_80s +
int(mask_locs.size()[0] * 0.10))):
    masked_text[mask_locs[j][0], mask_locs[j][1]] = tokens[i%
tokens.size()[0]]

  mask_labels = mask_labels.masked_fill(~mask, -100)
  ################################ END TODO
################################
  return masked_text, mask_labels

text = torch.randint(1, vocab_size, (batch_size, 128)).to(device)
pad_mask = (text == PAD_IND).to(torch.uint8).to(device)  # this is a
different type of mask (used to prevent attending to padding tokens)
masked_text, mask_labels = mask_inputs(text)

changed = (text != masked_text)
masked = (masked_text == MASK_TOKEN_IND)
print("Proportion of text changed (should be around 0.135): ",
changed.float().mean().cpu().item())
print("Proportion of text masked (should be around 0.12): ",
masked.float().mean().cpu().item())

labeled = (mask_labels != -100)
print("Proportion of data labeled for pre-training (should be around
0.15)", labeled.float().mean().cpu().item())

mask_preds = model(masked_text, pad_mask)
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(mask_preds.reshape((-1, vocab_size)),
mask_labels.flatten())
opt.zero_grad()
#loss.backward()
opt.step()
print("Training step successfully completed! Loss value (should be
around 4.6): ", loss.cpu().item())
```

```
tensor([40, 52, 86, 98, 54, 19, 16, 22, 78, 50, 95, 84,  7, 97, 53,
66, 26,  6,
        35, 58, 38, 31, 96, 11, 34, 49, 76, 43, 51, 55, 28, 27, 63,
```

```
88, 87, 65,
        25, 91, 15,  2,  9, 30, 45, 85, 13, 64, 29, 36, 69, 70, 24,
90, 92, 71,
        23,  0, 93, 14, 21, 12,  1, 60, 47, 44, 75, 10, 89, 42,  3,
61, 80, 72,
        39, 77, 18, 32, 73, 37, 62, 56, 48, 74,  4, 82, 17, 67,  8,
57, 20, 41,
        68, 81, 83, 79, 59, 94, 46, 33,  5])
Proportion of text changed (should be around 0.135):  0.134521484375
Proportion of text masked (should be around 0.12):  0.11968994140625
Proportion of data labeled for pre-training (should be around 0.15)
0.149658203125
Training step successfully completed! Loss value (should be around
4.6):  4.799966812133789
```

## 7 - Fine-Tune Pre-Trained Model on Sentiment Analysis [8 points]

In the previous section we implemented the pre-training procedure specified in the BERT paper. Now, we will take a fully-trained BERT model and use its learned representations for performing a sentiment analysis task.

We will use the transformers library to get pre-trained transformers and use them as our embedding layers. We will freeze (not train) the transformer and only train the remainder of the model which learns from the representations produced by the transformer. In this case we will be using a multi-layer bi-directional GRU, however any model can learn from these representations.

The goal of this sentiment analysis task is to predict the "sentiment" of a particular sequence. In this case the sequences are movie reviews are we're predicting whether they are positive or negative. Our model outputs a probability of positive sentiment for each input sequence. Use `nn.BCEWithLogitsLoss` to fine-tune the model on this task.

### Preparing Data

The transformer has already been trained with a specific vocabulary, which means we need to train with the exact same vocabulary and also tokenize our data in the same way that the transformer did when it was initially trained.

Luckily, the transformers library has tokenizers for each of the transformer models provided. In this case we are using the BERT model which ignores casing (i.e. will lower case every word). We get this by loading the pre-trained `bert-base-uncased` tokenizer.

```
!pip install transformers

from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
Requirement already satisfied: transformers in
/opt/conda/lib/python3.7/site-packages (4.16.2)
Requirement already satisfied: importlib-metadata in
/opt/conda/lib/python3.7/site-packages (from transformers) (4.11.3)
Requirement already satisfied: pyyaml>=5.1 in
/opt/conda/lib/python3.7/site-packages (from transformers) (6.0)
Requirement already satisfied: packaging>=20.0 in
/opt/conda/lib/python3.7/site-packages (from transformers) (21.3)
Requirement already satisfied: huggingface-hub<1.0,>=0.1.0 in
/opt/conda/lib/python3.7/site-packages (from transformers) (0.4.0)
Requirement already satisfied: filelock in
/opt/conda/lib/python3.7/site-packages (from transformers) (3.6.0)
Requirement already satisfied: requests in
/opt/conda/lib/python3.7/site-packages (from transformers) (2.26.0)
Requirement already satisfied: tokenizers!=0.11.3,>=0.10.1 in
/opt/conda/lib/python3.7/site-packages (from transformers) (0.11.6)
Requirement already satisfied: tqdm>=4.27 in
/opt/conda/lib/python3.7/site-packages (from transformers) (4.62.3)
Requirement already satisfied: numpy>=1.17 in
/opt/conda/lib/python3.7/site-packages (from transformers) (1.20.3)
Requirement already satisfied: regex!=2019.12.17 in
/opt/conda/lib/python3.7/site-packages (from transformers)
(2021.11.10)
Requirement already satisfied: sacremoses in
/opt/conda/lib/python3.7/site-packages (from transformers) (0.0.49)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/opt/conda/lib/python3.7/site-packages (from huggingface-
hub<1.0,>=0.1.0->transformers) (4.1.1)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/opt/conda/lib/python3.7/site-packages (from packaging>=20.0-
>transformers) (3.0.6)
Requirement already satisfied: zipp>=0.5 in
/opt/conda/lib/python3.7/site-packages (from importlib-metadata-
>transformers) (3.6.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers)
(1.26.7)
Requirement already satisfied: idna<4,>=2.5 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers)
(3.1)
Requirement already satisfied: charset-normalizer~=2.0.0 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers)
(2.0.9)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers)
(2021.10.8)
Requirement already satisfied: click in /opt/conda/lib/python3.7/site-
packages (from sacremoses->transformers) (8.0.3)
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-
packages (from sacremoses->transformers) (1.16.0)
```

```
Requirement already satisfied: joblib in
/opt/conda/lib/python3.7/site-packages (from sacremoses->transformers)
(1.1.0)
WARNING: Running pip as the 'root' user can result in broken
permissions and conflicting behaviour with the system package manager.
It is recommended to use a virtual environment instead:
https://pip.pypa.io/warnings/venv
```

{"version_major":2,"version_minor":0,"model_id":"82752992def944a1b391b98ece909d1a"}

{"version_major":2,"version_minor":0,"model_id":"efc1bed383ff4287930d7afbc6b19319"}

{"version_major":2,"version_minor":0,"model_id":"f7f47df87c0f40c0b44e325c652bf1fd"}

{"version_major":2,"version_minor":0,"model_id":"c057c1c341e3409ba9e7ca7f9144e9e8"}

Set constants regarding text tokenization and processing such that we are consistent with how the model was trained.

```
init_token_idx = tokenizer.cls_token_id
eos_token_idx = tokenizer.sep_token_id
pad_token_idx = tokenizer.pad_token_id
unk_token_idx = tokenizer.unk_token_id
max_input_length = tokenizer.max_model_input_sizes['bert-base-
uncased']
```

Define tokenization functions and set up IMDB dataset

```
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    return tokens

from torchtext.legacy import data

TEXT = data.Field(batch_first = True,
                  use_vocab = False,
                  tokenize = tokenize_and_cut,
                  preprocessing = tokenizer.convert_tokens_to_ids,
                  init_token = init_token_idx,
                  eos_token = eos_token_idx,
                  pad_token = pad_token_idx,
                  unk_token = unk_token_idx)

LABEL = data.LabelField(dtype = torch.float)

from torchtext.legacy import datasets
```

```
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split(random_state = 
random.seed(SEED))

LABEL.build_vocab(train_data)

print(f"Number of training examples: {len(train_data)}")
print(f"Number of validation examples: {len(valid_data)}")
print(f"Number of testing examples: {len(test_data)}")
```

```
downloading aclImdb_v1.tar.gz
```

```
aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:08<00:00, 
10.4MB/s]
```

```
Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
```

Create iterator to sample batches from the dataset.

```
BATCH_SIZE = 32

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = 
data.BucketIterator.splits(
    (train_data, valid_data, test_data), 
    batch_size = BATCH_SIZE,
    device = device)
```

## Build the Model

Next, we'll load the pre-trained model, making sure to load the same model as we did for the tokenizer.

```
from transformers import BertTokenizer, BertModel

bert = BertModel.from_pretrained('bert-base-uncased')
```

{"version_major":2,"version_minor":0,"model_id":"d0d446a7a7674703ad066
6151128c9e9"}

```
Some weights of the model checkpoint at bert-base-uncased were not 
used when initializing BertModel: 
['cls.predictions.transform.dense.weight', 
'cls.seq_relationship.bias', 'cls.predictions.transform.dense.bias', 
'cls.predictions.bias', 'cls.seq_relationship.weight', 
'cls.predictions.transform.LayerNorm.weight',
```

```
'cls.predictions.decoder.weight',
'cls.predictions.transform.LayerNorm.bias']
- This IS expected if you are initializing BertModel from the
checkpoint of a model trained on another task or with another
architecture (e.g. initializing a BertForSequenceClassification model
from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the
checkpoint of a model that you expect to be exactly identical
(initializing a BertForSequenceClassification model from a
BertForSequenceClassification model).
```

Next, we'll define our actual model.

Instead of using an embedding layer to get embeddings for our text, we'll be using the pre-trained transformer model. These embeddings will then be fed into a GRU to produce a prediction for the sentiment of the input sentence. We get the embedding dimension size (called the `hidden_size`) from the transformer via its config attribute. The rest of the initialization is standard.

Within the forward pass, we wrap the transformer in a `no_grad` to ensure no gradients are calculated over this part of the model. The transformer actually returns the embeddings for the whole sequence as well as a *pooled* output. The documentation states that the pooled output is "usually not a good summary of the semantic content of the input, you're often better with averaging or pooling the sequence of hidden-states for the whole input sequence", hence we will not be using it. The rest of the forward pass is the standard implementation of a recurrent model, where we take the hidden state over the final time-step, and pass it through a linear layer to get our predictions. **When using a bidirectional GRU, we concatenate the final step of the forward and backward direction**

```python
from inspect import Parameter
import torch.nn as nn

class BERTGRUSentiment(nn.Module):
    def __init__(self,
                 bert,
                 hidden_dim,
                 output_dim,
                 n_layers,
                 bidirectional,
                 dropout):
        """
        Parameters:
          bert: pre-trained BERT model
          hidden_dim: hidden dimensionality of GRU
          output_dim: output dimensionality of output linear layer
(when non-bidirectional)
          n_layers: number of GRU layers
          bidirectional: True if GRU is bi-directional, False if
otherwise.
          dropout: dropout rate for the dropout layer
```

```
        """
        super().__init__()

        self.bert = bert

        # TODO: get the embedding dimension size 'hidden_size' from
the transformer via its config attribute
        embedding_dim = bert.config.to_dict()['hidden_size']

        # TODO: add an n_layers GRU (you may use nn.GRU) - make sure
to include kwargs 'bidirectional', 'batch_first=True', and 'dropout'
        self.rnn = nn.GRU(embedding_dim,
                          hidden_dim,
                          num_layers = n_layers,
                          bidirectional = bidirectional,
                          batch_first = True,
                          dropout = 0 if n_layers < 2 else dropout)

        # TODO: add output linear layer (recall that we concatenate
two hidden vectors when using bidirectional GRU)
        self.out = nn.Linear(hidden_dim * 2 if bidirectional else
hidden_dim, output_dim)

        # TODO: add dropout layer
        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        # TODO: Compute the forward pass of the transformer inside a
`torch.no_grad()` context.
        with torch.no_grad():
          embedded = self.bert(text)[0]


        # TODO: pass embeddings through recurrent network
        _, hidden = self.rnn(embedded)

        # TODO: Select the hidden state to use - last step for
unidirectional -
        # last step of forward and backward iteration concatenated for
bidirectional
        # (hint: look at the docs for nn.GRU -
https://pytorch.org/docs/stable/generated/torch.nn.GRU.html)
        if self.rnn.bidirectional:
            hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-
1,:,:]), dim = 1))
        else:
            hidden = self.dropout(hidden[-1,:,:])
        # TODO: pass through dropout layer

        # TODO: pass through output linear layer
```

```
        output = self.out(hidden)
        return output
```

Next, we create an instance of our model. You need to select hyperparameters.

In order to freeze BERT paramers (not train them) we need to set their `requires_grad` attribute to `False`. To do this, we simply loop through all of the `named_parameters` in our model and if they're a part of the `bert` transformer model, we set `requires_grad = False`.

```
################################### TODO
###################################
# Adjust these hyperparameters as you see fit
######################################################################
######
HIDDEN_DIM = 256
N_LAYERS = 3
BIDIRECTIONAL = True
DROPOUT = 0.30
LEARNING_RATE = 3e-5
N_EPOCHS = 15
################################ END TODO
##################################

model = BERTGRUSentiment(bert,
                         HIDDEN_DIM,
                         1,
                         N_LAYERS,
                         BIDIRECTIONAL,
                         DROPOUT)

for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

## Train the Model

As is standard, we define our optimizer and criterion (loss function).

```
import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
criterion = nn.BCEWithLogitsLoss()

# Place the model and criterion onto the GPU (if available)
model = model.to(device)
criterion = criterion.to(device)
```

Next, we'll define functions for: calculating accuracy, performing a training epoch, performing an evaluation epoch and calculating how long a training/evaluation epoch takes.

```
def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this
returns 0.8, NOT 8
    """
    # TODO: compute the binary_accuracy
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for
division
    acc = correct.sum() / len(correct)
    return acc

def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():
```

```
        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Finally, we'll train our model.

**Please train your model such that it reaches 90% validation accuracy.** This is possible to accomplish within 15 minutes of training on GPU with the correct implementation and hyperparameters. Feel free to adjust the hyperparameters defined above in order to get the desired performance. Your points received will scale linearly from 0 for 50% accuracy to 8 for at least 90% accuracy.

```
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, 
criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'best-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m 
{epoch_secs}s')
```

```
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: 
{train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: 
{valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 10m 49s
	Train Loss: 0.677 | Train Acc: 59.12%
	 Val. Loss: 0.627 |  Val. Acc: 70.08%
Epoch: 02 | Epoch Time: 10m 47s
	Train Loss: 0.569 | Train Acc: 71.41%
	 Val. Loss: 0.502 |  Val. Acc: 75.84%
Epoch: 03 | Epoch Time: 10m 46s
	Train Loss: 0.494 | Train Acc: 76.41%
	 Val. Loss: 0.459 |  Val. Acc: 78.89%
Epoch: 04 | Epoch Time: 10m 47s
	Train Loss: 0.431 | Train Acc: 81.06%
	 Val. Loss: 0.408 |  Val. Acc: 83.51%
Epoch: 05 | Epoch Time: 10m 47s
	Train Loss: 0.382 | Train Acc: 84.31%
	 Val. Loss: 0.334 |  Val. Acc: 86.71%
Epoch: 06 | Epoch Time: 10m 47s
	Train Loss: 0.356 | Train Acc: 85.50%
	 Val. Loss: 0.321 |  Val. Acc: 87.28%
Epoch: 07 | Epoch Time: 10m 47s
	Train Loss: 0.343 | Train Acc: 86.01%
	 Val. Loss: 0.325 |  Val. Acc: 86.75%
Epoch: 08 | Epoch Time: 10m 47s
	Train Loss: 0.332 | Train Acc: 86.53%
	 Val. Loss: 0.298 |  Val. Acc: 88.32%
Epoch: 09 | Epoch Time: 10m 47s
	Train Loss: 0.318 | Train Acc: 87.21%
	 Val. Loss: 0.290 |  Val. Acc: 88.46%
Epoch: 10 | Epoch Time: 10m 47s
	Train Loss: 0.304 | Train Acc: 87.77%
	 Val. Loss: 0.282 |  Val. Acc: 88.98%
Epoch: 11 | Epoch Time: 10m 47s
	Train Loss: 0.294 | Train Acc: 88.16%
	 Val. Loss: 0.273 |  Val. Acc: 89.10%
Epoch: 12 | Epoch Time: 10m 46s
	Train Loss: 0.286 | Train Acc: 88.52%
	 Val. Loss: 0.270 |  Val. Acc: 89.22%
Epoch: 13 | Epoch Time: 10m 47s
	Train Loss: 0.278 | Train Acc: 88.83%
	 Val. Loss: 0.266 |  Val. Acc: 89.47%
Epoch: 14 | Epoch Time: 10m 47s
	Train Loss: 0.272 | Train Acc: 89.06%
	 Val. Loss: 0.269 |  Val. Acc: 89.31%
Epoch: 15 | Epoch Time: 10m 47s
	Train Loss: 0.268 | Train Acc: 89.12%
	 Val. Loss: 0.260 |  Val. Acc: 89.86%
```

Load up the parameters that gave us the best validation loss and try these on the test set

```
model.load_state_dict(torch.load("best-model.pt"))
```

```
test_loss, test_acc = evaluate(model, test_iterator, criterion)
```

```
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.240 | Test Acc: 90.63%
```

## Inference

We'll then use the model to test the sentiment of some sequences. We tokenize the input sequence, trim it down to the maximum length, add the special tokens to either side, convert it to a tensor, add a fake batch dimension and then pass it through our model. Feel free to add more test cases!

```
def predict_sentiment(model, tokenizer, sentence):
    model.eval()
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    indexed = [init_token_idx] + 
tokenizer.convert_tokens_to_ids(tokens) + [eos_token_idx]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(0)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()
```

```
predict_sentiment(model, tokenizer, "This film is terrible")
```

```
predict_sentiment(model, tokenizer, "This film is great")
```

```
0.7384241223335266
```

### Reviews for testing:

Positive Review

```
predict_sentiment(model,tokenizer,
"This film was actually good. I loved the acting and I really enjoyed
the music. Definitely a must watch for all the fans.")
```

```
0.9798356294631958
```

Negative Review

```
predict_sentiment(model,tokenizer,
"This film was horrible. I hated the acting and I despised the music.
Avoid this movie it's not worth it.")
```

```
0.0231458347894251
```

## Conceptual Questions

1. Why is the residual connection is crucial in the Transformer architecture? [5 points]

We need residual connections in transformers to mitigate the vanishing gradient problem. They allow the gradients to flow through the network directly and help facilitate back propagation.

1. Why is Layer Normalization important in the Transformer architecture? [5 points]

Layer norm calculated the normalisation over each feature, instead of each batch. This stabilises the network and has an effect similar to batchnorm in CNNs. This results in substantially reduced training times.

1. Why do we use the scaling factor of $1/\sqrt{d_k}$ in Scaled Dot Product Attention? If we remove it, what is going to happen? [5 points]

The variables Q and K are independent and are of size dk*dk. Their dot product has a variance of dk, and a mean of 0. The dot products quickly become larger values and hence, the softmax of the same will be very high, and therfore, very small gradients will flow back. So, we scale it down by a factor of root(dk) to bring down the variance to 1. If we dont have the scaling factor, training will be very slow, since very little gradient can flow backward.

## Submission PDF

As in assignment 1, please prepare a separate submission PDF for each problem. **Do not simply render your notebook as a pdf**. For this problem, please include the following in a PDF called `problem_2_solution.pdf`:

1. Some short, one-sentence movie reviews that you wrote yourself, with your model's predicted sentiment.
2. Answers to the conceptual questions above.

Note that you still need to submit the jupyter notebook with all generated solutions.