DistilBERT and ALBERT

Distilbert:

Distilbert aims to optimize the training by reducing the size of BERT and increase the speed of BERT – all while trying to retain as much performance as possible. Specifically, Distilbert is 40% smaller than the original BERT-base model, is 60% faster than it, and retains 97% of its functionality.

How does Distilbert do this? It uses roughly the same general architecture as BERT, but with only 6 encoder blocks (recall that BERT base has 12). These encoder blocks are also initialized by merely taking 1 out of every 2 pretrained encoder blocks of BERT. Furthermore, the token-type embeddings and the pooling functionalities of BERT are also removed from Distilbert.

Unlike BERT, Distilbert is only pretrained using masked language modeling (recall that BERT was trained using MLM and Next Sentence Prediction). Distilbert is trained using triple loss/three loss functions:

- The same language model loss that is used by BERT
- The distillation loss measures the similarity of the outputs between Distilbert and BERT.
- The cosine-distance loss measures how similar the hidden states of Distilbert and BERT are.

The combination of these loss functions mimics a student-teacher learning relationship between Distilbert and BERT. Distilbert also uses several of the same hyperparameters that Roberta uses, like larger batch size, dynamic masking, and as I mentioned before, no pretraining on next sentence prediction. Looking at the code for Roberta (and BERT) from above, using Distilbert from hugging face is very easy to do.

```
from transformers import DistilBertModel
import torch
import torch.nn as nn
class DistilBERT_Model(nn.Module):
 def __init__(self, classes):
  super(DistilBERT_Model, self).__init__()
  self.distilbert = DistilBertModel.from_pretrained('distilbert
                                    base-uncased')
  self.out = nn.Linear(self.distilbert.config.hidden_size, classes)
  self.sigmoid = nn.Sigmoid()
 def forward(self, input, attention_mask):
  _, output = self.distilbert(input, attention_mask
                          = attention_mask)
  out = self.sigmoid(self.out(output))
  return out
```

ALBERT:

Albert was published/introduced at around the same time as Distilbert, and also has some of the same motivations presented in the paper. Just like Distilbert, Albert reduces the model size of BERT (18x fewer parameters) and also can be trained 1.7x faster. Unlike Distilbert, however, Albert does not have a tradeoff in performance (Distilbert does have a slight tradeoff in performance). This comes from just the core difference in the way the Distilbert and Albert experiments are structured. Distilbert is trained in such a way to use BERT as the teacher for its training/distillation process. Albert, on the other hand, is trained from scratch like BERT. Better yet, Albert outperforms all previous models including BERT, Roberta, Distilbert, and XLNet.

Albert is able to attain the results it does with the smaller model architecture with these parameter-reduction techniques:

- Factorized Embedding Parameterization: To ensure the size of the hidden layers and the embedding dimensions are different, Alberta deconstructs the embedding matrix into 2 pieces. This allows it to essentially increase the size of the hidden layer without truly modifying the actual embedding dimension. After decomposing the embedding matrix, Alberta adds a linear layer/fully connected layer onto the embedding matrices after the embedding phase is done, and this maps/ensures the dimension of the embedding dimension is the same correct. You can read more about this here.
- Cross-layer parameter sharing: recall that BERT and Alberta have 12 encoder blocks each. In Alberta, these encoder blocks share all parameters. This reduces the parameter size 12-fold and also increases the regularization of the model (regularization is a technique when calibrating ML models that are used to prevent overfitting/underfitting)
- Alberta removes dropout layers: a dropout layer is a technique where neurons that are randomly selected are ignored during training. This means that they are no longer being trained and are essentially useless temporarily.

```
from transformers import AlbertModel
import torch
import torch.nn as nn
class ALBERT_Model(nn.Module):
 def __init__(self, classes):
  super(ALBERT_Model, self).__init__()
  self.albert = AlbertModel.from_pretrained('albert-base-v2')
  self.out = nn.Linear(self.albert.config.hidden_size, classes)
  self.sigmoid = nn.Sigmoid()
 def forward(self, input, attention_mask):
  _, output = self.albert(input, attention_mask = attention_mask)
  out = self.sigmoid(self.out(output))
  return out
```