Knowledge Distillation

————

New Jersey Institute of Technology

[CS677] Deep Learning

Professor Ioannis Koutis

Created by: Aponte, David; Duggan, William; Suthar, Pooja

————

Video source: Watch us!

Notebook source: Colab Notebook

Table of Contents

Abstract

The central aspect of intent classification in artificial intelligence and machine learning is to identify and classify user intent. This is done automatically when software receives a query to access some message from keywords or phrases, more commonly known are chatbots.

The goal of Natural Language Understanding (NLU), which is a branch of Natural Language Processing (NLP), is to improve machine reading comprehension by examining the grammar and context of words. To assist computers to identify exactly how to perform the appropriate response message back to humans for thousands of different languages all over the world seems tedious; even more so when calculating computational costs.

Our group experiments to perform model compression using DistilBERT as the smaller student model, and the large teacher model BERT. We aim to explore models encapsulating more out of scope training that can lead to improvements on out-of-scope performance, without compromising significant computational cost. In conjunction, we are interested in recent studies

[1]Back to the top

that consider the problem of multi-intent classification to be future work, and why smaller, less costly, datasets are preferred.

BERT

Bidirectional Encoder Representations from Transformers, BERT for short, is designed to pre-train representations from unlabeled text by jointly conditioning on both left and right context.

★ **Encoder representations** take text data in a way that captures the underlying language structure. This can be useful for a variety of natural language processing tasks, such as sentiment analysis, named entity recognition, and language translation.

★ **Transformer** architecture that BERT is based on uses self-attention mechanisms to learn the relationships between words in a sentence, allowing it to capture long-range dependencies in the text data. This is in contrast to traditional language models, which use recurrent neural networks and are limited to capturing short-range dependencies. It is a powerful NLP algorithm which was introduced in the paper "Attention is All You Need" published in 2017.

★ **Bidirectional** defines the training process by using left and right context when dealing with a word

The NLP vectorization method allows users to shape sentences, tokenizing and embedding strings to vectors, so that standard machines can better translate text or intent from binary. The basic requirements would be a vocabulary of known words and, then a measure of the presence of known words.

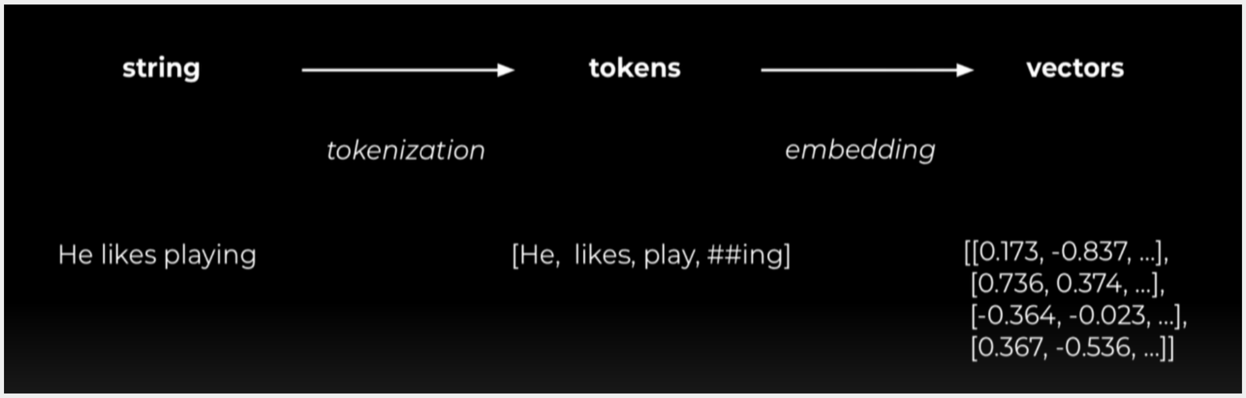


Figure 1- Basic representation of vectorization.

BERT can be fine tuned downstream with just one additional output layer to create state of the art models for a wide range of NLP tasks, for example: Relation Extraction, Question Answering, Chatbot Dialogue, Semantic Search Indexing, Knowledge Base Population, E-Discovery and Media Monitoring, and much more.

BERT is pre-trained in an unsupervised manner, on a large corpus of unlabelled text including the entire Wikipedia (2.5 billion words) and BookCorpus (800 million words), to create a general language representation model. This massive dataset equates to `2 quintillion (2.0 x 10`^{1}`)` possabilities. Modern ML/DL practitioners optimized BERT in a variety of ways, noting a few below:

★ **ALBERT** [2] by Google and more — This paper describes parameter reduction techniques to lower memory reduction and increase the training speed of BERT models.

★ **RoBERTa** [3] by Facebook — This paper for FAIR believes the original BERT models were under-trained and shows with more training/tuning it can outperform the initial results.

★ **ERNIE** [4] Enhanced Representation through Knowledge Integration by Baidu — It is inspired by the masking strategy of BERT and learns language representation enhanced by knowledge masking strategies, which includes entity-level masking and phrase-level masking.

★ **DistilBERT** [5] Smaller BERT using model distillation from Hugging Face that we will be using in our experiment.

BERT's architecture builds on top of a Transformer. Simply put, it is a stack of Transformer Encoders. It only uses the encoder part of the Transformer, which currently has two classes:

BERT Base	**BERT Large**

12 layers (encoder blocks)	24 layers (encoder blocks)
12 attention heads	16 attention heads
110 million parameters	340 million parameters

Figure 2 - Current BERT versions available

Moreover, BERT uses two unsupervised tasks for pre-training: masked language modeling and next sentence prediction. To be fully bidirectional, instead of learning to predict the next word of a sentence, we learn to predict a missing word within the sequence. In addition to learning the relationship between words, BERT is also trained to learn the relationships between sentences. For example, providing two sentences to ask if they logically come one after the other in a corpus or if they are randomly picked.

Training on a dataset this large takes a long time. BERT's training was made possible by the novel Transformer architecture and enhanced by using Tensor Processing Units (TPUs) which is Google's custom circuit built specifically for large Machine Learning models. 64 TPUs trained BERT over the course of 4 days, which is out of scope for the ordinary person to practice within a reasonable amount of time. To better appreciate this, tinkering with Google's Tensorflow [6] playground, encapsulates many computational avenues for neural networks to explore, which can exponentially demand higher computational power.

DistilBERT

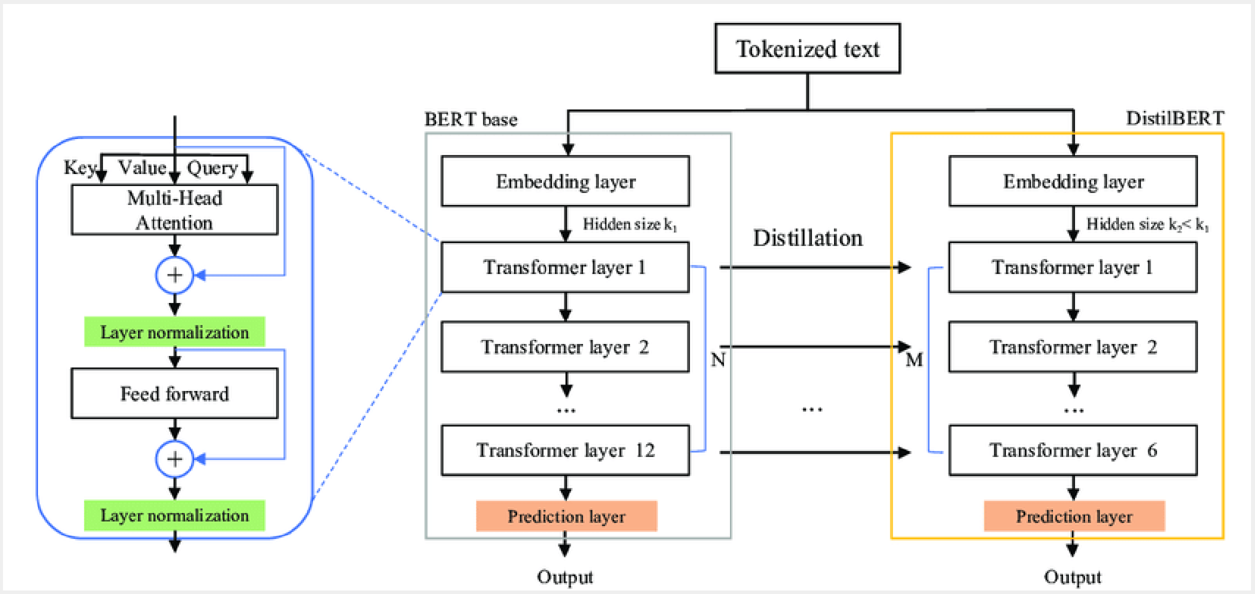


Figure 3 - DistilBERT model architecture and components (Image source)

Neural networks typically produce class probabilities by using a softmax output layer converting the logit, computed for each class into a probability, by comparing with the other logits. In its simplest form of distillation, knowledge is transferred to the distilled model by training it on a transfer set and using a soft target distribution for each case in the transfer set that is produced by using the cumbersome model with a high target value in its softmax. Using the logits of probability and combining it into the loss function to compress knowledge in an ensemble of models into a single model, this creates a more friendly environment for deployment, and fine tuning. There are generally two types of knowledge distillation:

- ★ Task-specific
- ★ Task-agnostic

Task-specific knowledge distillation is used to fine-tune a model on a given dataset. The idea came from the DistilBERT [7] paper, as well as the FastFormers [8] paper. DistilBERT was pretrained on the same data as BERT. We use the transformers Pipeline API which downloads and cahes this pretrained model.

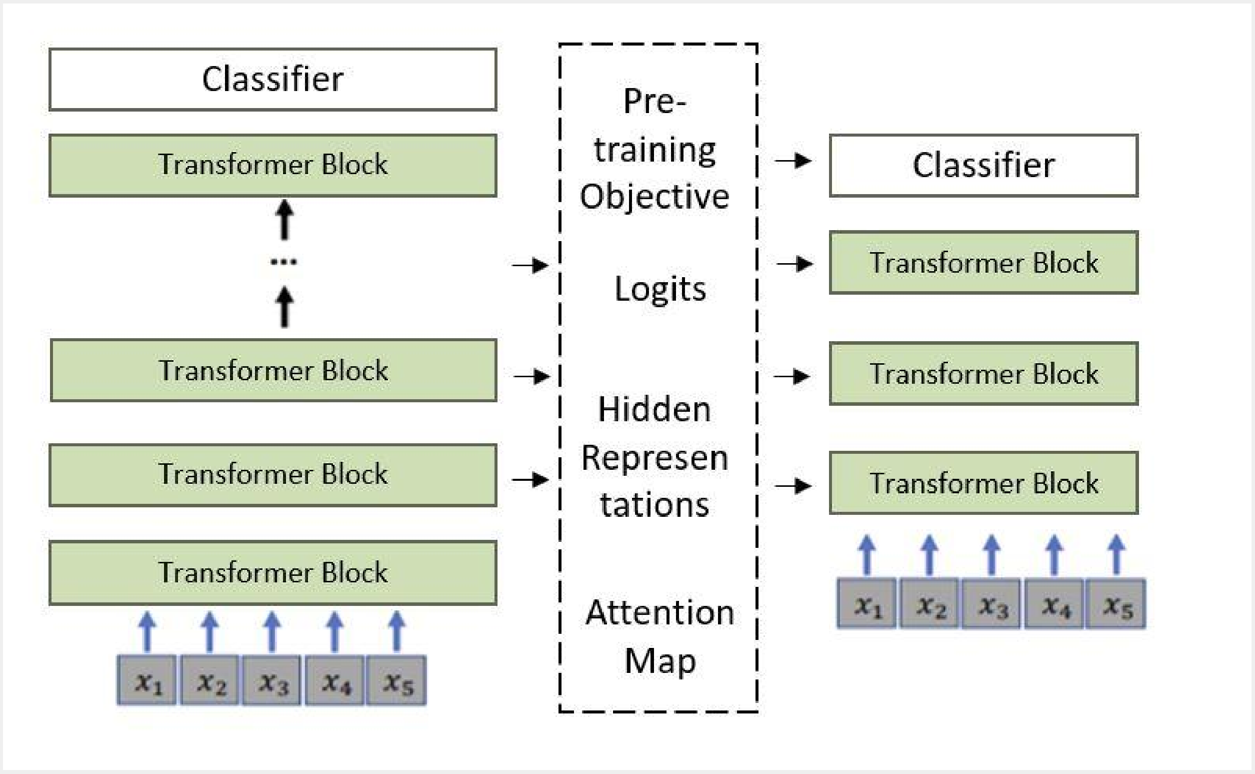


Figure 4 - Knowledge Distillation from Transformer

Knowledge distillation is performed during the pre-training phase to reduce the size of a BERT model by 40% (or 1.32 billion words), retaining a majority of its language understanding capabilities which performs much faster. This is because transfer learning does not require a lot of data as earlier features were already trained. Thus, adding a smaller dataset with a lot less weights, allows for more significant training performance. Intuitively, DistilBERT balances memory against storage in order to fit in RAM on standard machines. As disks are generally slower and memory is generally faster, the model reads data from memory while larger data files, such as images, can live on the disk.

Back to the top

Model Building

To use a pretrained BERT-based model, you will need to install the transformers library and PyTorch. Ideally, using a virtual environment such as Google Collab, Jupyter or equivalent.

We will be using the Google Collab environment. Here are some basic steps to use the pretrained BERT-based model for classification:

1. Install the transformers library and dependencies.
2. Import the necessary modules from the transformers library, as well as PyTorch.
3. Load the pretrained BERT-based model and tokenizer.
4. Define the input text and tokenize it using the BERT tokenizer.
5. Pass the input tensor through the BERT model to obtain the logits (predicted probabilities for each class).
6. Use the logits to predict the class labels for the input text.
7. If you want to fine-tune the BERT model on your own dataset, you can do so by running the following steps:

    ★ Define your dataset using the `TensorDataset` class from PyTorch, which takes as input the input tensors and the corresponding target labels.

    ★ Define a DataLoader for the dataset, which will be used to iterate through the dataset in mini-batches during training.

    ★ Define an optimizer and a loss function for training. For example, the `Adam` optimizer and the `cross-entropy` loss function.

    ★ Train the model by iterating through the dataset using the DataLoader and updating the model parameters using the optimizer and the loss function. You must first use the `model.train()` method to switch the model to training mode. Then you use the `model.eval()` method to switch onto the evaluation mode.

Implementing our model

Data:

- ★ We are using the clinc-oos dataset available at HuggingFace available here.
- ★ DistilBERT for smaller transformer pipeline API which downloads and caches the pretrained model

Hardware:

- ★ HuggingFace account is needed
- ★ Python 3.6 and above
- ★ PyTorch, Transformers and python stock libraries
- ★ CPU but allows for GPU enabled setup
- ★ Jupyter, GoogleColab or equivalent environment

Objective:

- ★ Fine-tune DistilBERT to classify text

Requirements

We will be fine tuning a transformer model for the multi-text classification problem. This is one of the most common business problems where a given piece of text, sentence, document needs to be classified into one of the categories out of the given list.

The following scripts leverage multiple tools designed by other teams to cooperate with implementation. Details of the tools used below must be present in the setup to successfully run our model.

Begin by installing basic dependencies libraries

```
!pip install torch
!pip install transformers datasets
!pip install matplotlib pandas
!pip install torchinfo
!pip install latexify-py==0.2.0
!pip install huggingface_hub
```

Try adding a summary from torchinfo as it provides unique output when calling it onto the model for understanding architectures, etc.

```
try:
    from torchinfo import summary
except:
    print("[INFO] Couldn't find torchinfo... installing it.")
    !pip install -q torchinfo
    from torchinfo import summary
```

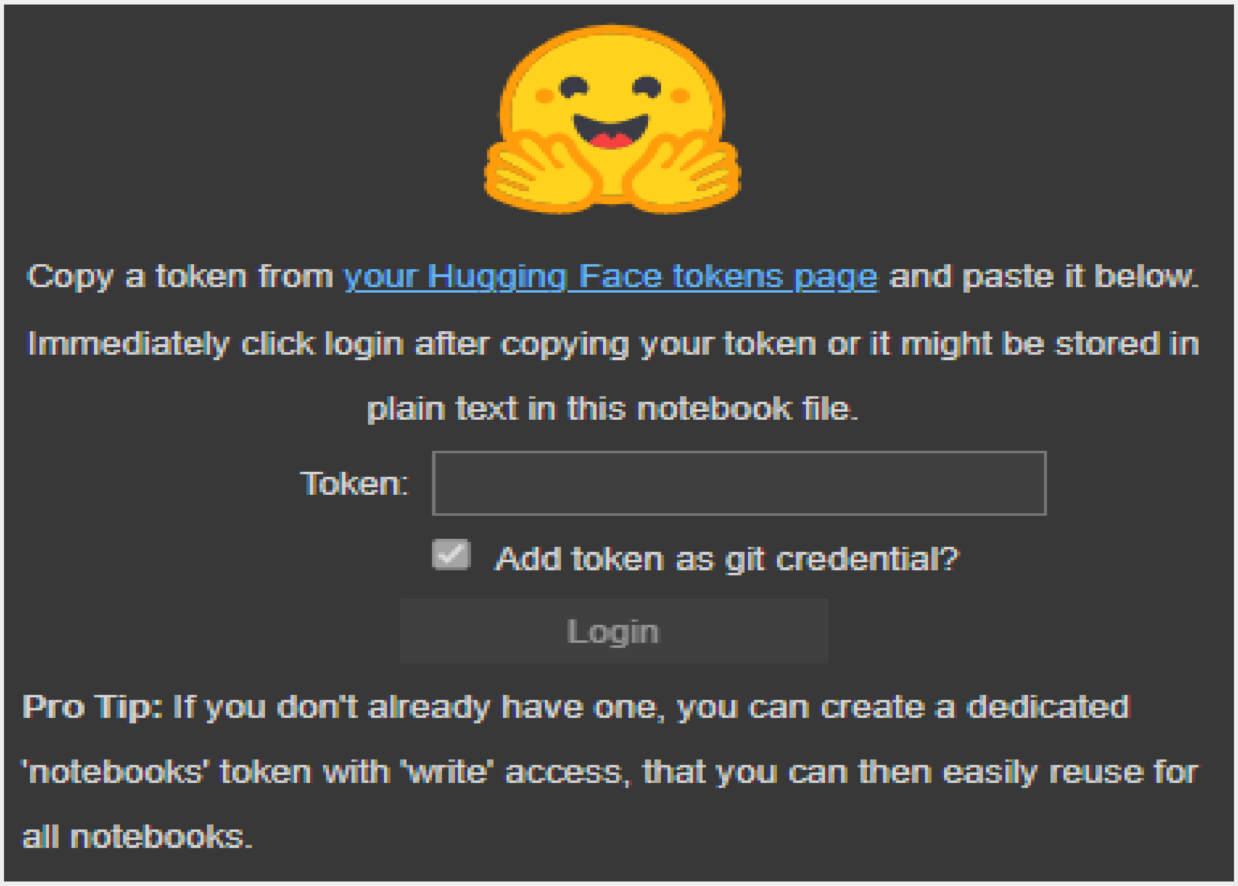
Login to HuggingFace

Login to huggingface from your local notebook. This enables access so that local notebook and huggingface notebook can communicate checkpoints.

```
     # This code will be hidden when the notebook is loaded.
     from huggingface_hub import notebook_login
```

```
    notebook_login()
```

This will prompt you by asking for your access Token provided automatically by Hugging Face. You can access your token in the settings section of your profile documented here [10]. Copy and paste your token into the local notebook (NOTE: you must be logged in to HuggFace to work).


Copy a token from your Hugging Face tokens page and paste it below.

Immediately click login after copying your token or it might be stored in plain text in this notebook file.

Pro Tip: If you don't already have one, you can create a dedicated 'notebooks' token with 'write' access, that you can then easily reuse for all notebooks.

Mount to Google Drive

Optional for those looking to provide reproducibility and add style or context.

```
#@title Mount to Google Drive {display-mode: "form"}

# This code will be hidden when the notebook is loaded.
from google.colab import drive
```

```
drive.mount('/content/gdrive')
```

Back to the top

Teacher Model

Load the pretrained BERT-based model and tokenizer by running the following code.

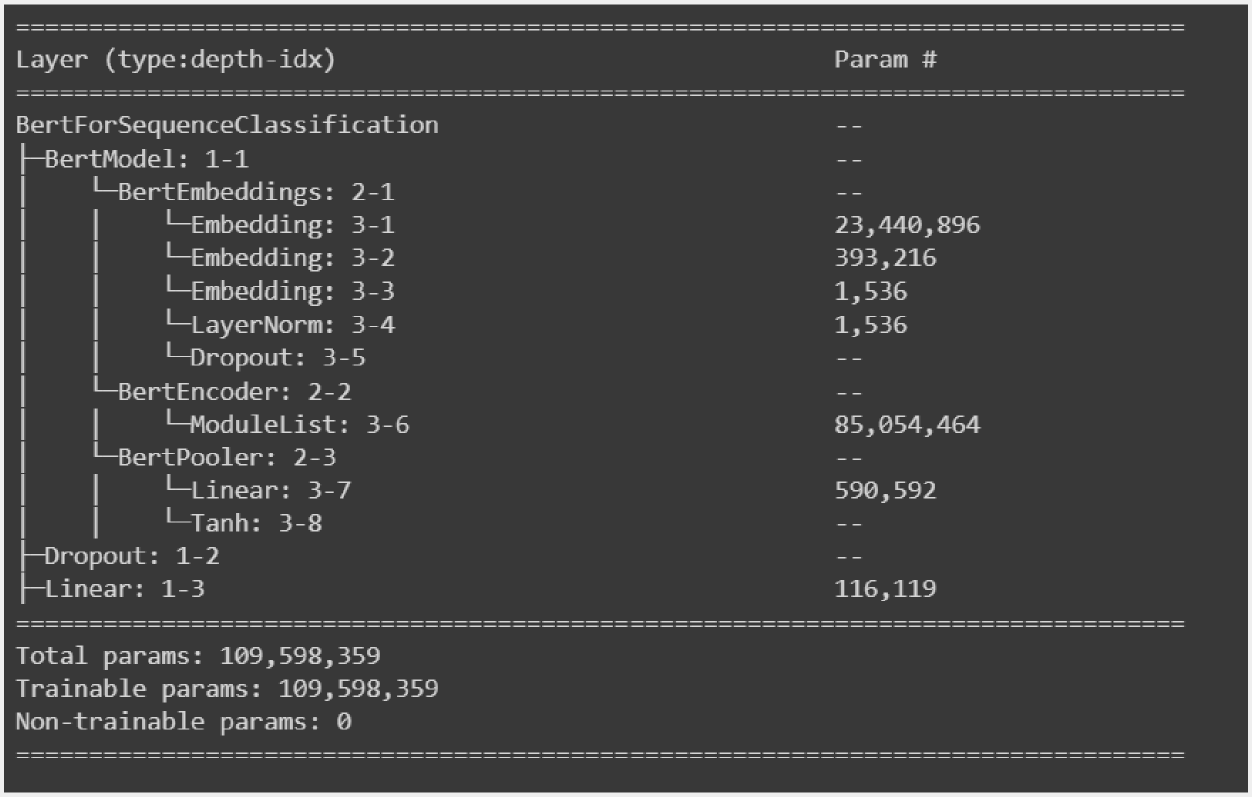
```
from transformers import pipeline

# Load pretrained teacher model from huggingface
teacher_id = "transformersbook/bert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=teacher_id)
```

By combining tokenization, forward pass and backward pass operations, transformer pipelines can be set up for our specific task of text classification.

```
teacher_module = pipe.model
teacher_module
```

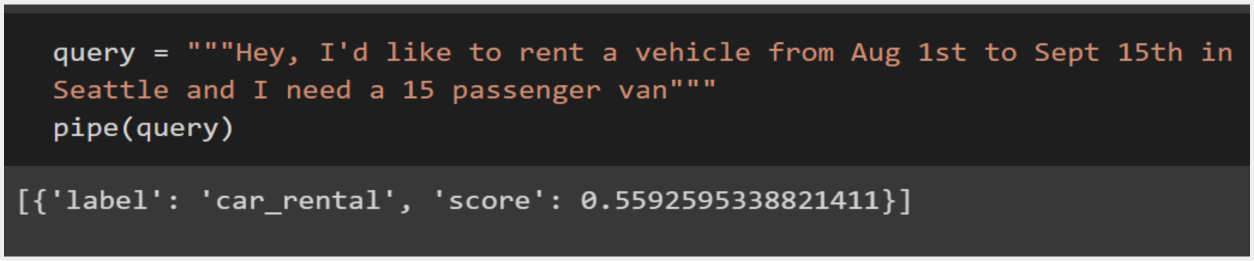
Inspect the teacher_model using torch summary to display layers and parameters.

```
import torch
from torchinfo import summary

teacher_module.to(torch.device("cpu"))
summary(teacher_module,                         dtypes=['torch.IntTensor'],
device=torch.device("cpu"))
```



```
================================================================================
Layer (type:depth-idx)                                  Param #
================================================================================
BertForSequenceClassification                           --
├─BertModel: 1-1                                        --
│    └─BertEmbeddings: 2-1                              --
│    │    └─Embedding: 3-1                              23,440,896
│    │    └─Embedding: 3-2                              393,216
│    │    └─Embedding: 3-3                              1,536
│    │    └─LayerNorm: 3-4                              1,536
│    │    └─Dropout: 3-5                                --
│    └─BertEncoder: 2-2                                 --
│    │    └─ModuleList: 3-6                             85,054,464
│    └─BertPooler: 2-3                                  --
│    │    └─Linear: 3-7                                 590,592
│    │    └─Tanh: 3-8                                   --
├─Dropout: 1-2                                          --
├─Linear: 1-3                                           116,119
================================================================================
Total params: 109,598,359
Trainable params: 109,598,359
Non-trainable params: 0
================================================================================
```

Define a variable string; our sample from the teacher_model, piped query scores 55% accuracy targeting the label `car_rental` below.



```
query = """Hey, I'd like to rent a vehicle from Aug 1st to Sept 15th in
Seattle and I need a 15 passenger van"""
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.5592595338821411}]
```

Preprocessing

Load in the dataset from datasets utility. The clinic-oos dataset contains:

★ 23,700 queries (22,500 in-scope queries covering 150 intents), which will be grouped into 10 general, categorical domains.
★ 1,200 out-of-scope queries as well, and can be found on the hugging face website.

Load the dataset from the PyTorch utility and sample a test index label to check that it is loaded properly. Before training, define the intents to transfer dataset objects.

```
from datasets import load_dataset

clinc = load_dataset("clinc_oos", "plus")
```
===
```
sample = clinc["test"][33]
print(f"Text | Intent = {sample}")
```
===
```
intents = clinc["test"].features["intent"]
intents.int2str(sample["intent"])
```
===

Importing load_metric from PyTorch to measure accuracy score for the model.

```
from datasets import load_metric

accuracy_score = load_metric("accuracy")
```
===

Benchmarking Performance

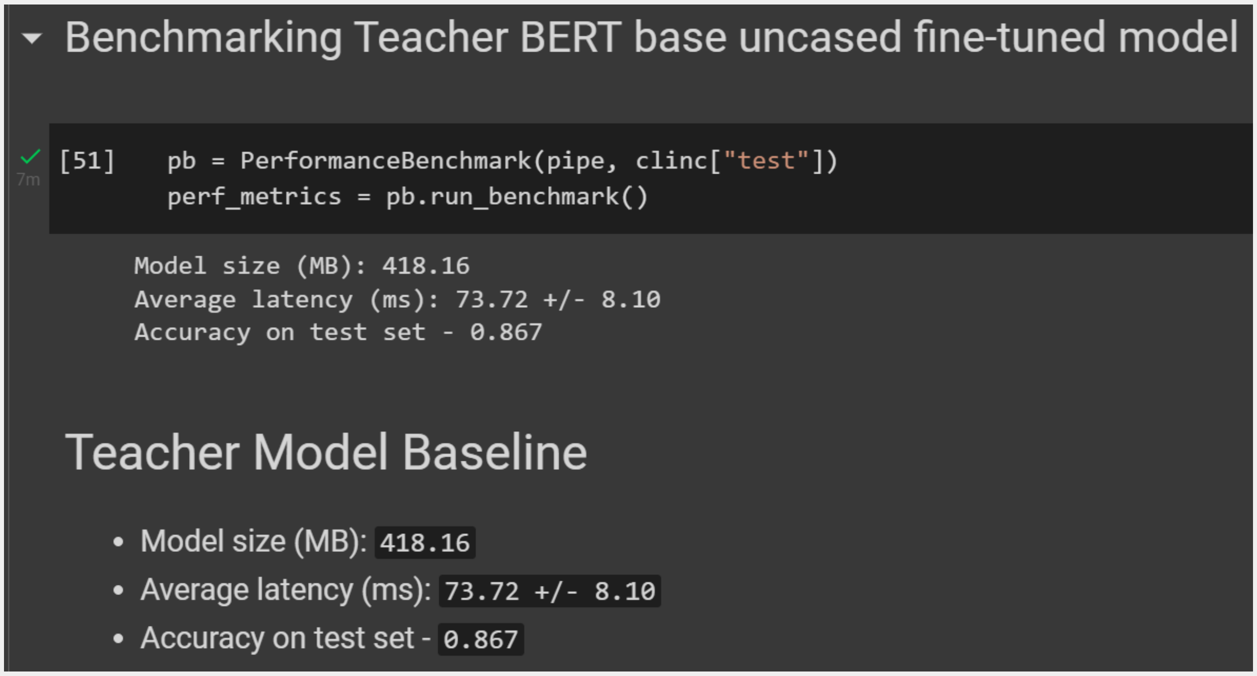
To measure the performance improvements from knowledge distillation, we will benchmark our results to measure the following:

- ★ Size of the model in megabytes (MB).
- ★ Accuracy of the model.
- ★ Average latency in milliseconds (ms). To report the average and standard deviation, we will conduct multiple trials to measure execution time of the forward pass (including tokenization since we are using a pipeline).

We will record performance metrics in a `perf_metrics` dictionary that we will update for each benchmarking run.

```
from pathlib import Path
import numpy as np
import torch
from time import perf_counter

class PerformanceBenchmark:
    def __init__(self, pipeline, dataset, optim_type="teacher-bert"):
        self.pipeline = pipeline
        self.dataset = dataset
        self.optim_type = optim_type

    def compute_accuracy(self):
      preds, labels = [], []
      for example in self.dataset:
          pred = self.pipeline(example["text"])[0]["label"]
          label = example["intent"]
          preds.append(intents.str2int(pred))
          labels.append(label)
      accuracy = accuracy_score.compute(predictions=preds, references=labels)
      print(f"Accuracy on test set - {accuracy['accuracy']:.3f}")
      return accuracy

    def compute_size(self):
        state_dict = self.pipeline.model.state_dict()
        tmp_path = Path(f"{self.optim_type}-model.pt")
        torch.save(state_dict, tmp_path)
        # Measure in megabytes
        size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
        tmp_path.unlink()
        print(f"Model size (MB): {size_mb:.2f}")
        return {"size_mb" : size_mb}

    def time_pipeline(self, trials, query="What is the pin number for my
account?"):
        latencies = []
        # warm up CPU/GPU
        for _ in range(10):
          _ = self.pipeline(query)
        # Run trials and record latency
        for _ in range(trials):
          start = perf_counter()
          _ = self.pipeline(query)
          latency = perf_counter() - start
          latencies.append(latency)
        # Get mean and std
        time_avg_ms = np.mean(latencies)*1000
        time_std_ms = np.std(latencies)*1000
        print(f"Average latency (ms): {time_avg_ms:.2f} +/- {time_std_ms:.2f}")
        return {"time_avg_ms": time_avg_ms, "time_std_ms": time_std_ms}

    def run_benchmark(self, trials=100):
        metrics = {}
        metrics[self.optim_type] = self.compute_size()
        metrics[self.optim_type].update(self.time_pipeline(trials=trials))
        metrics[self.optim_type].update(self.compute_accuracy())
        return metrics
```

Benchmarking Teacher BERT base uncased fine-tuned



Benchmarking Teacher BERT base uncased fine-tuned model

```
[51] pb = PerformanceBenchmark(pipe, clinc["test"])
     perf_metrics = pb.run_benchmark()

     Model size (MB): 418.16
     Average latency (ms): 73.72 +/- 8.10
     Accuracy on test set - 0.867
```

Teacher Model Baseline

- Model size (MB): `418.16`
- Average latency (ms): `73.72 +/- 8.10`
- Accuracy on test set - `0.867`

Figure 5 - Teacher Model Baseline results

Back to the top

Knowledge Distillation

Arguments

Knowledge distillation arguments included:

★ Alpha = 0.5

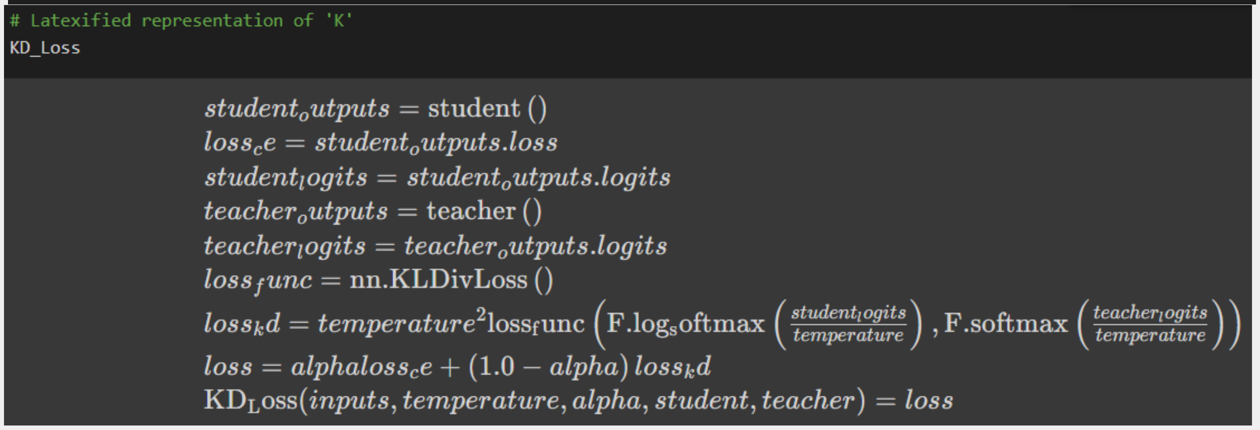
- ★ Temperature = 2.0
- ★ Note: these control the relative weight of the distillation loss and how much the probability distribution of the labels should be rendered by.

```
from transformers import TrainingArguments

class KDArguments(TrainingArguments):
    def __init__(self, *args, alpha=0.5, temperature=2.0, **kwargs):
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.temperature = temperature
```

Loss Function

We define KD_Loss to calculate the difference in the output text created by the model, and the actual output text. We also attempt to use the python package latexify function to pretty print our compiled functions. NOTE: This package is a cosmetic option and not necessary

```
#@title Knowledge Distillation Loss Function {display-mode: "form"}

# This code will be hidden when the notebook is loaded.
import latexify

@latexify.with_latex
def KD_Loss(inputs, temperature, alpha,  student, teacher):
    # Extract cross-entropy loss and logits from student
    student_outputs = student(**inputs)
    loss_ce = student_outputs.loss
    student_logits = student_outputs.logits

    # Extract logits from teacher
    with torch.no_grad():
        teacher_outputs = teacher(**inputs)
        teacher_logits = teacher_outputs.logits

    # Soften probabilities and compute distillation loss
    loss_func = nn.KLDivLoss(reduction="batchmean")
    loss_kd = temperature ** 2 * loss_func(
        F.log_softmax(student_logits / temperature, dim=-1),
        F.softmax(teacher_logits / temperature, dim=-1))
    # Return weighted student loss
    loss = alpha * loss_ce + (1. - alpha) * loss_kd
    return loss
KD_Loss
```



```
# Latexified representation of 'K'
KD_Loss
```

$$\begin{aligned}
&student_outputs = \mathrm{student}\,()\\
&loss_ce = student_outputs.loss\\
&student_logits = student_outputs.logits\\
&teacher_outputs = \mathrm{teacher}\,()\\
&teacher_logits = teacher_outputs.logits\\
&loss_func = \mathrm{nn.KLDivLoss}\,()\\
&loss_kd = temperature^2 \mathrm{loss_func}\left(\mathrm{F.log_softmax}\left(\frac{student_logits}{temperature}\right), \mathrm{F.softmax}\left(\frac{teacher_logits}{temperature}\right)\right)\\
&loss = alphaloss_ce + (1.0 - alpha)\,loss_kd\\
&\mathrm{KD_Loss}(inputs, temperature, alpha, student, teacher) = loss
\end{aligned}$$

Kullback-Leibler loss `nn.KLDivLoss` computes the gradients with respect to q (student distribution) to obtain the same gradients. `reduction="batchmean"` takes the batch size dimension average to measure similarity of two distributions. It also allows leveraging PyTorch implementation for faster computation:

$$KL(p||q) \;=\; E(log(p/q)) \;=\; \sum_i pi * log(pi) \;-\; \sum_i pi * log(qi)$$

Student Model

We choose a student model that is similar to the teacher model but smaller (e.g, We are using BERT as the parent model, so we can use DistilBERT as the student). As discussed earlier, there are others to choose from that may better fit your needs.

```
import torch.nn as nn
import torch.nn.functional as F
from transformers import Trainer


class KDTrainer(Trainer):
  def __init__(self, *args, teacher_model=None, **kwargs):
    super().__init__(*args, **kwargs)
    self.teacher_model = teacher_model
    # place teacher on same device as student
    self._move_model_to_device(self.teacher_model, self.model.device)
    # Keep teacher in eval mode
    self.teacher_model.eval()

  # Overwrite compute_loss function of trainer
  def compute_loss(self, model, inputs, return_outputs=False):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # device found globally
    inputs = inputs.to(device)

    student_outputs = model(**inputs)
    # Extract cross-entropy loss and logits from student
    loss_ce = student_outputs.loss
    student_logits = student_outputs.logits

    # Extract logits from teacher
    with torch.no_grad():
        teacher_outputs = self.teacher_model(**inputs)
        teacher_logits = teacher_outputs.logits

    # Soften probabilities and compute distillation loss
    loss_func = nn.KLDivLoss(reduction="batchmean")
    loss_kd = self.args.temperature ** 2 * loss_func(
        F.log_softmax(student_logits / self.args.temperature, dim=-1),
        F.softmax(teacher_logits / self.args.temperature, dim=-1))

    # Return weighted student loss
    loss = self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
    return (loss, student_outputs) if return_outputs else loss
```

Preprocess Data

```
from transformers import AutoTokenizer

student_id = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_id)

def tokenize_batch(batch):
  return student_tokenizer(batch["text"], truncation=True)

clinc_encoded = clinc.map(tokenize_batch, batched=True,
remove_columns=['text'])
clinc_encoded = clinc_encoded.rename_column("intent", "labels")
```

```
def compute_metrics(pred):
  predictions, labels = pred
  predictions = np.argmax(predictions, axis=1)
  return accuracy_score.compute(predictions=predictions,
references=labels)
```

```
batch_size = 48

finetuned_id = "kd-distilBERT-clinc"
student_training_args = KDArguments(
    output_dir=finetuned_id,
    evaluation_strategy="epoch",
    num_train_epochs=5,
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    alpha=1,
    weight_decay=0.01,
    push_to_hub=True,
)
student_training_args.logging_steps = len(clinc_encoded['train']) //
batch_size
student_training_args.disable_tqdm = False
student_training_args.save_steps = 1e9
```

Student model needs to be mapped from intents.

```
id2label = pipe.model.config.id2label
label2id = pipe.model.config.label2id
num_labels = intents.num_classes
```

Load Teacher checkpoint without tokenizer

```
from transformers import AutoModelForSequenceClassification,
DataCollatorWithPadding

# define data_collator
data_collator = DataCollatorWithPadding(tokenizer=student_tokenizer)

# Define teacher model
teacher_model = AutoModelForSequenceClassification.from_pretrained(
    teacher_id,
    num_labels=num_labels,
    id2label=id2label,
    label2id=label2id,
)

# Define student model
student_model = AutoModelForSequenceClassification.from_pretrained(
    student_id,
    num_labels=num_labels,
    id2label=id2label,
    label2id=label2id,
)
```

WARNING: Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing DistilBertForSequenceClassification:

```
['vocab_transform.bias', 'vocab_projector.weight', 'vocab_layer_norm.bias',
'vocab_projector.bias', 'vocab_transform.weight',
'vocab_layer_norm.weight']
['classifier.bias', 'classifier.weight', 'pre_classifier.weight',
'pre_classifier.bias']
```

★ This **IS** expected if you are initializing DistilBertForSequenceClassification 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 DistilBertForSequenceClassification from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).

Back to the top

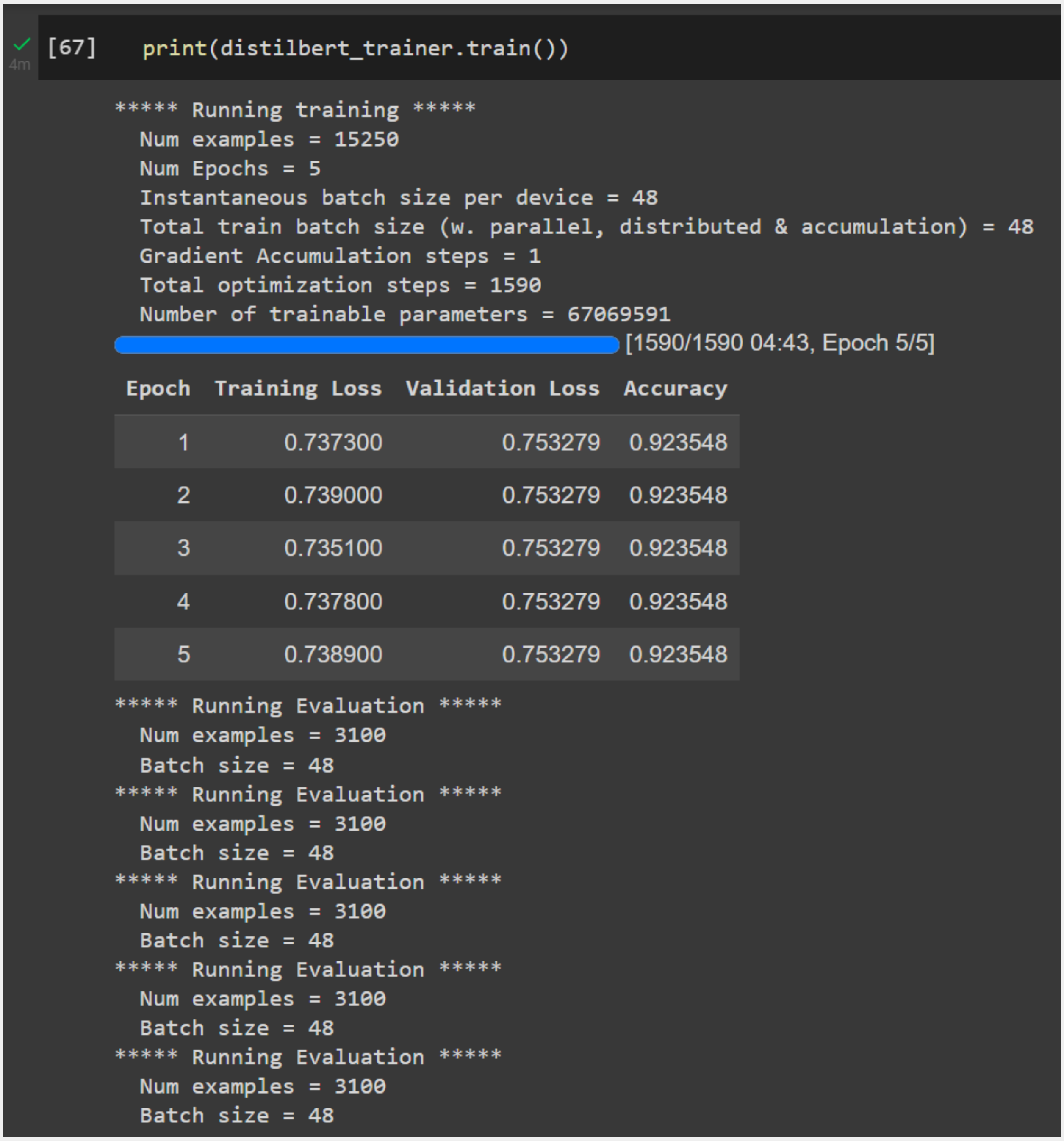
Training Procedure

DistilBERT task-specific student model

We further study the use of DistilBERT on downstream tasks under efficient inference constraints. We use our compact pre-trained language model by fine-tuning it with a classification task, which mixes distillation pre-training and transfer learning.

```
distilbert_trainer = KDTrainer(
    student_model,
    student_training_args,ñññ
    teacher_model=teacher_model,
    train_dataset=clinc_encoded['train'],
    eval_dataset=clinc_encoded['validation'],
    data_collator=data_collator,
    tokenizer=student_tokenizer,
    compute_metrics=compute_metrics,
  )
```

Just over 67 million trainable parameters, we take 15,250 examples in 48 batches of 3,100 each over 5 epochs to evaluate.



```
[67]   print(distilbert_trainer.train())

***** Running training *****
  Num examples = 15250
  Num Epochs = 5
  Instantaneous batch size per device = 48
  Total train batch size (w. parallel, distributed & accumulation) = 48
  Gradient Accumulation steps = 1
  Total optimization steps = 1590
  Number of trainable parameters = 67069591
```

[1590/1590 04:43, Epoch 5/5]

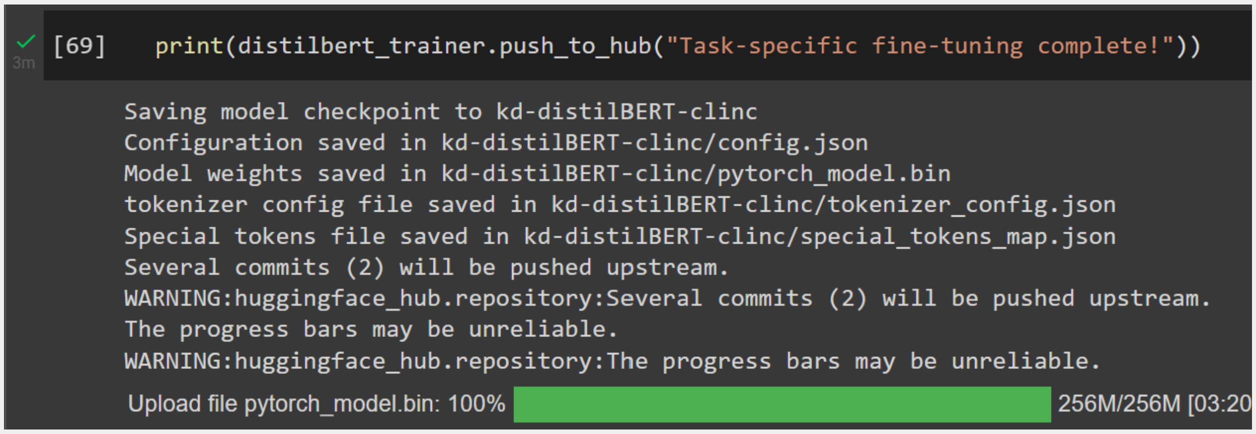
Epoch	Training Loss	Validation Loss	Accuracy
1	0.737300	0.753279	0.923548
2	0.739000	0.753279	0.923548
3	0.735100	0.753279	0.923548
4	0.737800	0.753279	0.923548
5	0.738900	0.753279	0.923548

```
***** Running Evaluation *****
  Num examples = 3100
  Batch size = 48
***** Running Evaluation *****
  Num examples = 3100
  Batch size = 48
***** Running Evaluation *****
  Num examples = 3100
  Batch size = 48
***** Running Evaluation *****
  Num examples = 3100
  Batch size = 48
***** Running Evaluation *****
  Num examples = 3100
  Batch size = 48
```

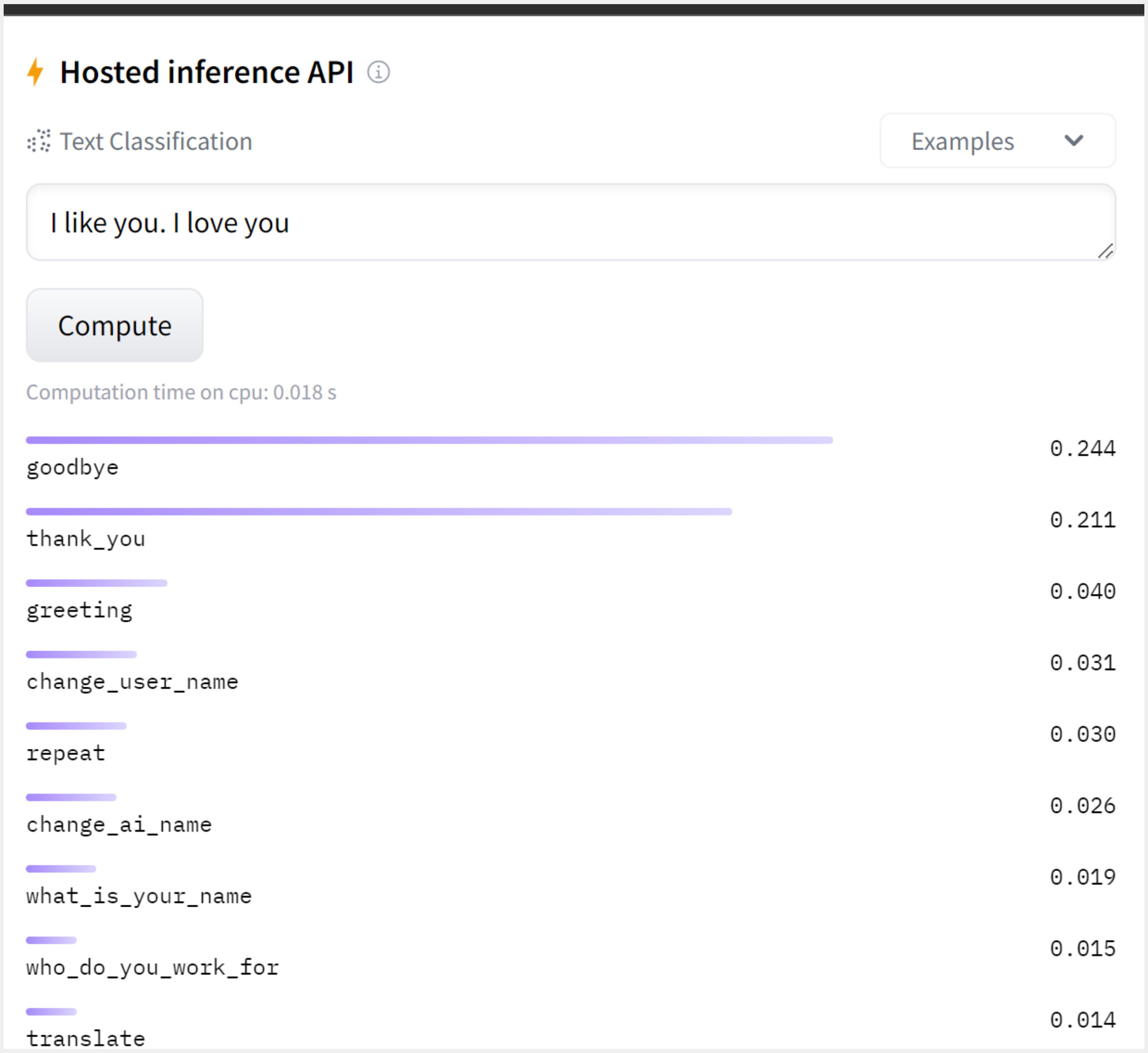
Figure 6 - Evaluation of task-specific trainer

Saving to Hugging Face 😊

As you can see below results classifying intent “i like you. i love you”, some interesting labels emerge. Using `"your_trainer".push_to_hub()` commits and pushes to your profile per account model card inference each time a training run is completed on your specific fine tuned model.



```
[69]   print(distilbert_trainer.push_to_hub("Task-specific fine-tuning complete!"))

       Saving model checkpoint to kd-distilBERT-clinc
       Configuration saved in kd-distilBERT-clinc/config.json
       Model weights saved in kd-distilBERT-clinc/pytorch_model.bin
       tokenizer config file saved in kd-distilBERT-clinc/tokenizer_config.json
       Special tokens file saved in kd-distilBERT-clinc/special_tokens_map.json
       Several commits (2) will be pushed upstream.
       WARNING:huggingface_hub.repository:Several commits (2) will be pushed upstream.
       The progress bars may be unreliable.
       WARNING:huggingface_hub.repository:The progress bars may be unreliable.
        Upload file pytorch_model.bin: 100%                                256M/256M [03:20
```



Hosted inference API

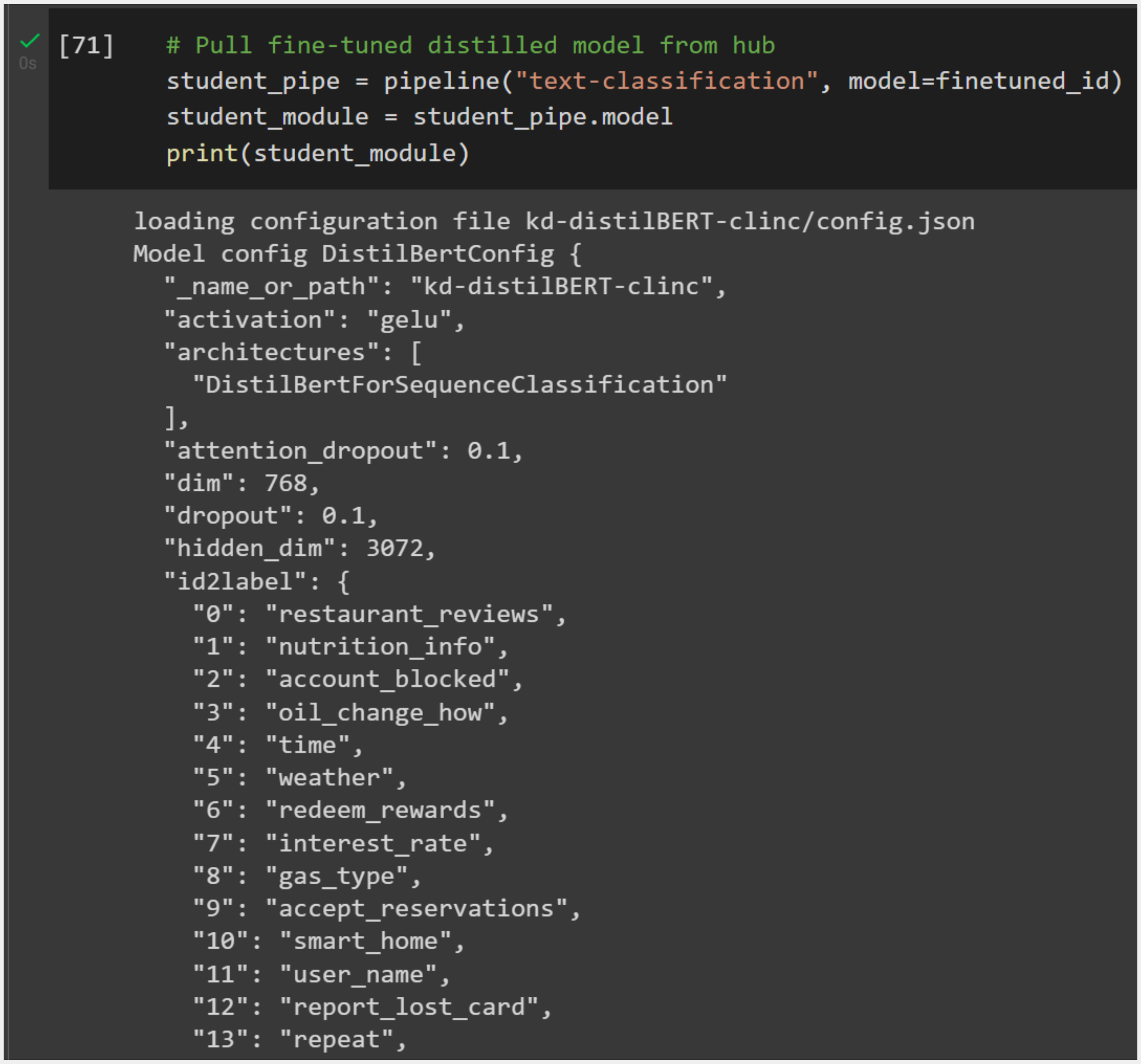
Text Classification   Examples

I like you. I love you

Compute

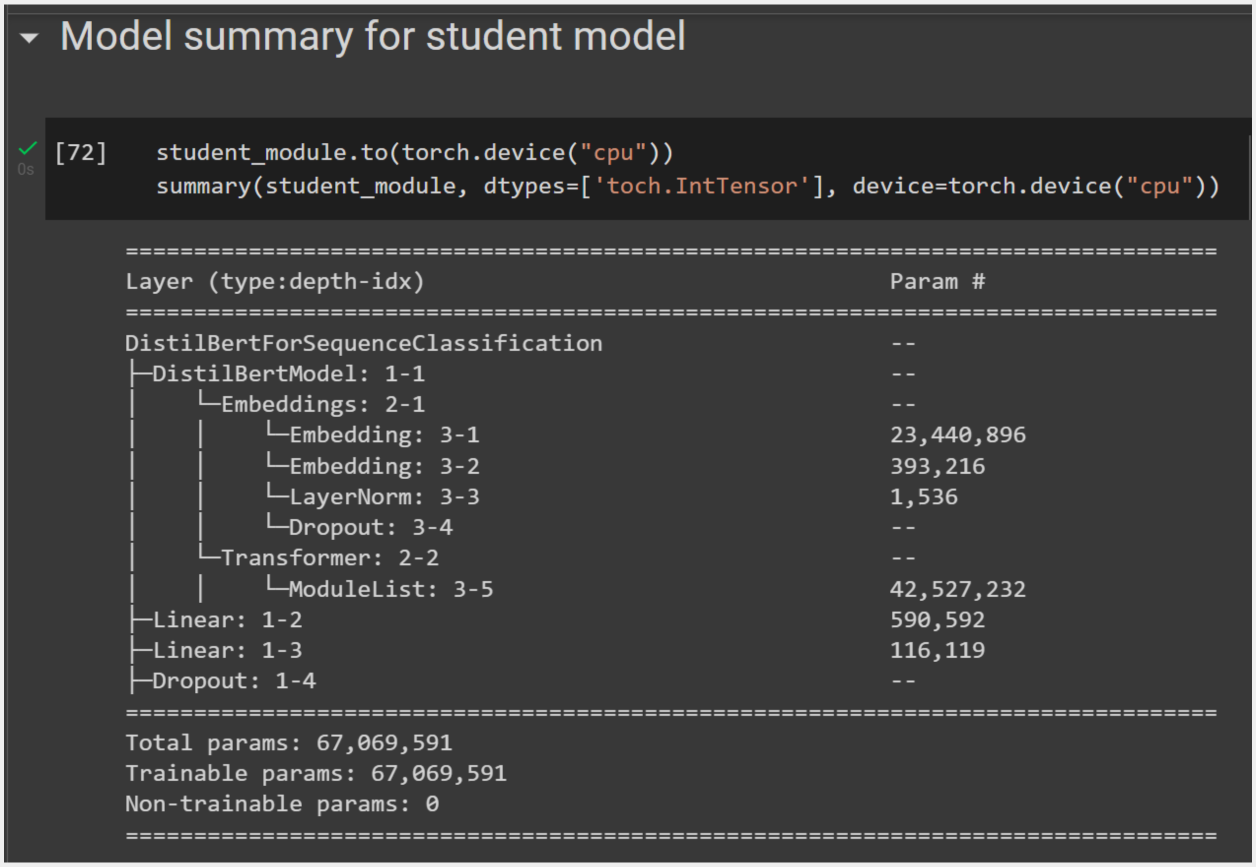
Computation time on cpu: 0.018 s

```
goodbye                0.244
thank_you              0.211
greeting               0.040
change_user_name       0.031
repeat                 0.030
change_ai_name         0.026
what_is_your_name      0.019
who_do_you_work_for    0.015
translate              0.014
```



```
[71]   # Pull fine-tuned distilled model from hub
       student_pipe = pipeline("text-classification", model=finetuned_id)
       student_module = student_pipe.model
       print(student_module)
```

```
loading configuration file kd-distilBERT-clinc/config.json
Model config DistilBertConfig {
  "_name_or_path": "kd-distilBERT-clinc",
  "activation": "gelu",
  "architectures": [
    "DistilBertForSequenceClassification"
  ],
  "attention_dropout": 0.1,
  "dim": 768,
  "dropout": 0.1,
  "hidden_dim": 3072,
  "id2label": {
    "0": "restaurant_reviews",
    "1": "nutrition_info",
    "2": "account_blocked",
    "3": "oil_change_how",
    "4": "time",
    "5": "weather",
    "6": "redeem_rewards",
    "7": "interest_rate",
    "8": "gas_type",
    "9": "accept_reservations",
    "10": "smart_home",
    "11": "user_name",
    "12": "report_lost_card",
    "13": "repeat",
```



Model summary for student model

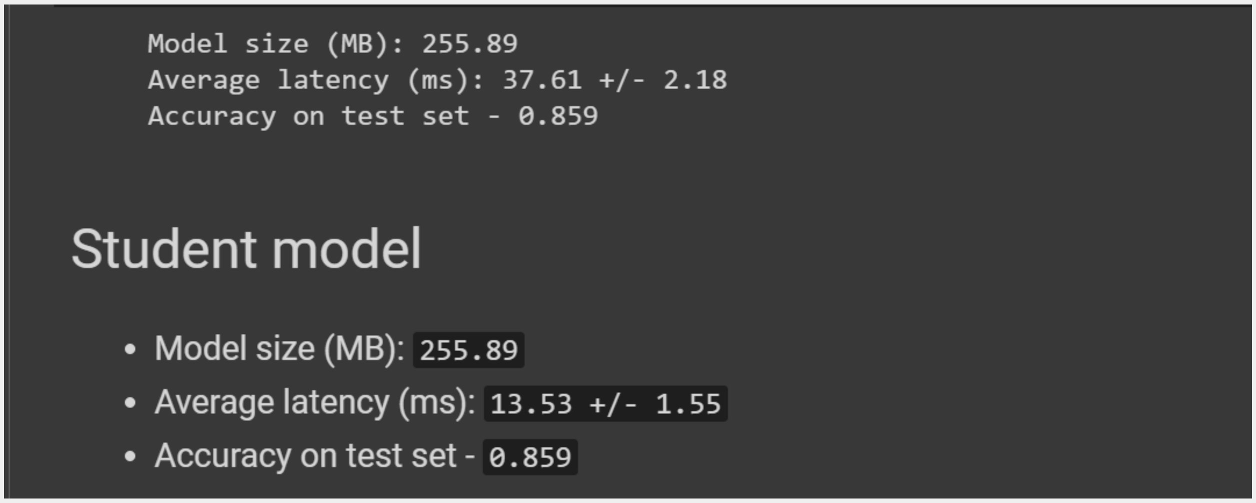
[72]
```
student_module.to(torch.device("cpu"))
summary(student_module, dtypes=['toch.IntTensor'], device=torch.device("cpu"))
```

```
================================================================================
Layer (type:depth-idx)                                        Param #
================================================================================
DistilBertForSequenceClassification                           --
├─DistilBertModel: 1-1                                        --
│    └─Embeddings: 2-1                                        --
│    │    └─Embedding: 3-1                                    23,440,896
│    │    └─Embedding: 3-2                                    393,216
│    │    └─LayerNorm: 3-3                                    1,536
│    │    └─Dropout: 3-4                                      --
│    └─Transformer: 2-2                                       --
│    │    └─ModuleList: 3-5                                   42,527,232
├─Linear: 1-2                                                 590,592
├─Linear: 1-3                                                 116,119
├─Dropout: 1-4                                                --
================================================================================
Total params: 67,069,591
Trainable params: 67,069,591
Non-trainable params: 0
================================================================================
```

Figure 7 - Student Model summary

Benchmarking distilled student fine-tuned model results

```
optim_type = "kd-DistilBERT"
pb = PerformanceBenchmark(
    student_pipe,
    clinc["test"],
    optim_type=optim_type,
)
perf_metrics.update(pb.run_benchmark())
```



```
Model size (MB): 255.89
Average latency (ms): 37.61 +/- 2.18
Accuracy on test set - 0.859
```

Student model

- Model size (MB): `255.89`
- Average latency (ms): `13.53 +/- 1.55`
- Accuracy on test set - `0.859`

Figure 8 - Student model performance

Analysis and evaluation

```python
import pandas as pd
import matplotlib.pyplot as plt

def plot_metrics(perf_metrics, current_optim_type):
    df = pd.DataFrame.from_dict(perf_metrics, orient='index')

    for idx in df.index:
        df_opt = df.loc[idx]
        # Add a dashed circle around the current optimization type
        if idx == current_optim_type:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        alpha=0.5, s=df_opt["size_mb"], label=idx,
                        marker='$\u25CC$')
        else:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        s=df_opt["size_mb"], label=idx, alpha=0.5)

    legend = plt.legend(bbox_to_anchor=(1,1))
    for handle in legend.legendHandles:
        handle.set_sizes([20])
```

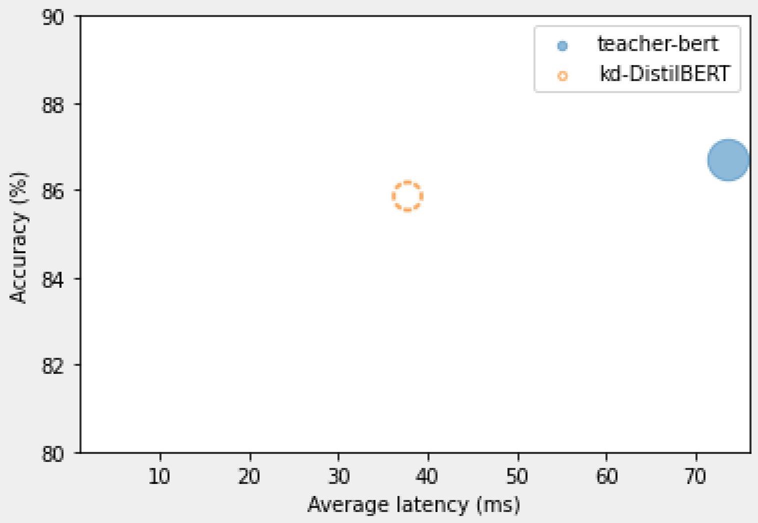
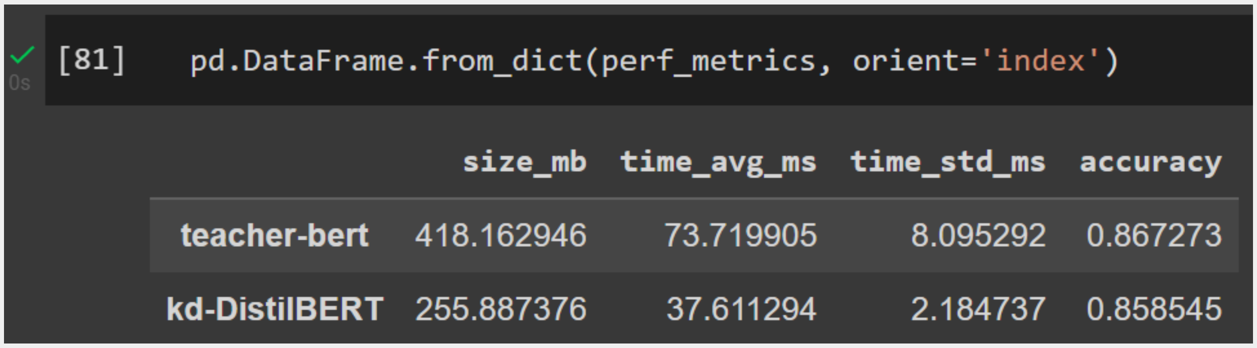
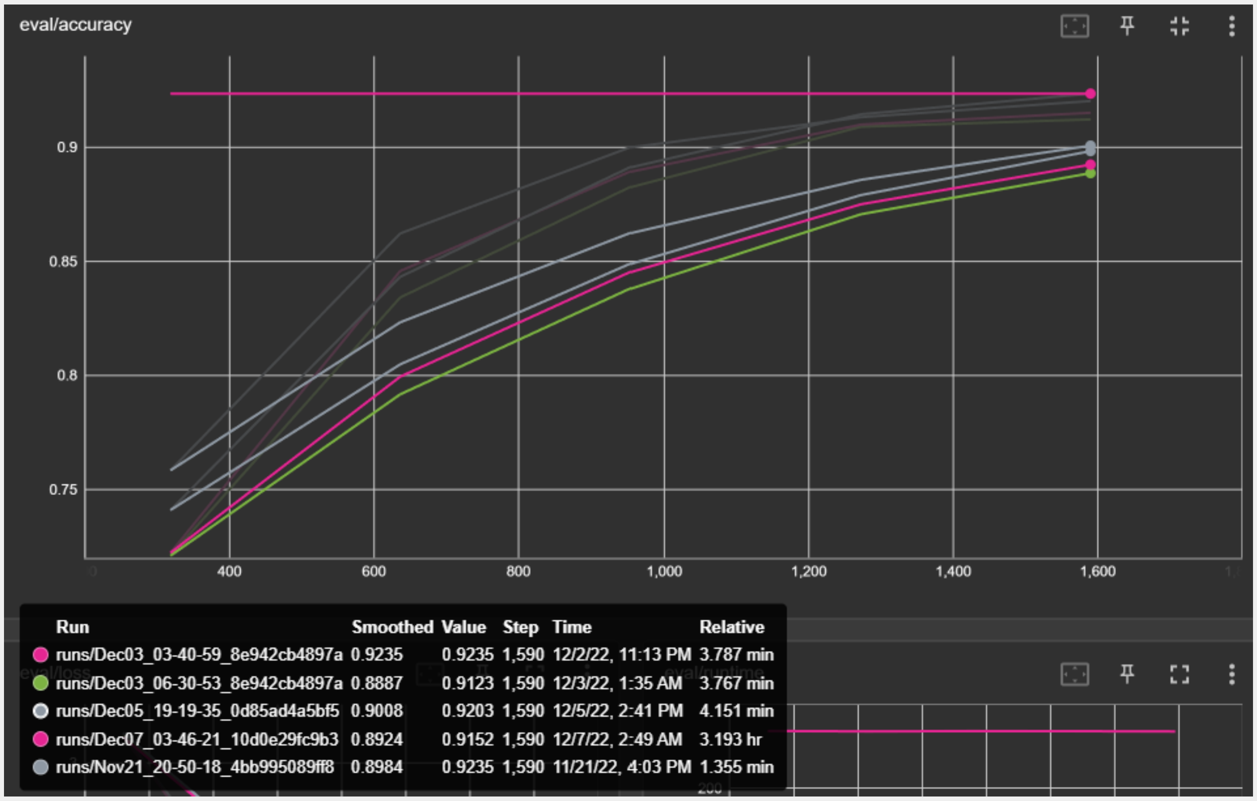


Figure 9 - teacher vs student comparison of accuracy (%) over latency (ms)



```
[81]  pd.DataFrame.from_dict(perf_metrics, orient='index')
```

	size_mb	time_avg_ms	time_std_ms	accuracy
teacher-bert	418.162946	73.719905	8.095292	0.867273
kd-DistilBERT	255.887376	37.611294	2.184737	0.858545



	Run	Smoothed	Value	Step	Time	Relative
●	runs/Dec03_03-40-59_8e942cb4897a	0.9235	0.9235	1,590	12/2/22, 11:13 PM	3.787 min
●	runs/Dec03_06-30-53_8e942cb4897a	0.8887	0.9123	1,590	12/3/22, 1:35 AM	3.767 min
●	runs/Dec05_19-19-35_0d85ad4a5bf5	0.9008	0.9203	1,590	12/5/22, 2:41 PM	4.151 min
●	runs/Dec07_03-46-21_10d0e29fc9b3	0.8924	0.9152	1,590	12/7/22, 2:49 AM	3.193 hr
●	runs/Nov21_20-50-18_4bb995089ff8	0.8984	0.9235	1,590	11/21/22, 4:03 PM	1.355 min

Back to the top

Conclusion

This model is a fine-tuned version of distilbert-base-uncased on the clinc_oos dataset. Using the teacher signal, we were able to train our student model, the smaller language model DistilBERT, from the supervision of BERT-base-uncased version of BERT.

Following Hinton et al., the training loss is a linear combination of the distillation loss and the masked language modeling loss. In doing so, we achieved our target objective by reducing the total number of parameters, retaining accuracy of BERT's performance on the understanding benchmark.

```
This model is a fine-tuned version of
[distilbert-base-uncased](https://huggingface.co/distilbert-base-uncased)
on the clinc_oos dataset.
It achieves the following results on the evaluation set:
- Loss: 0.7533
- Accuracy: 0.9235
```

Discussing novel future applications of the model

Some potential future applications of the model could include:

- Using the model as part of a larger system for automatic translation of languages, allowing for faster and more efficient translation of text from one language to another
- Developing new techniques for fine-tuning the model on specific datasets, to improve its performance on a wide range of natural language processing tasks
- Incorporating the model into intelligent assistants or chatbots, allowing for more natural and effective communication with users
- Using the model as part of a system for generating personalized content, such as articles or social media posts, based on a user's interests and preferences

These are just a few examples of the potential future applications of the Knowledge-Distillation-DistilBERT-Base-uncased-fine-tuned model. As the field of natural language processing continues to evolve, it is likely that new and novel applications of the model will be developed.

Recent works in knowledge distillation propose task-agnostic as well as task-specific methods to compress the models, with task-specific ones often yielding higher compression rate.

XtremeDistilTransformers is a technique for task-agnostic distillation, which is a method for training a smaller model (called the "student") to perform a specific task by using a larger, pre-trained model (called the "teacher") as a guide.

It is designed to improve the transfer of knowledge from the teacher model to the student model. It does this by using a combination of distillation and transfer learning methods, which allows the student model to learn more effectively from the teacher and adapt to the specific characteristics of the target task.

This can be especially useful when the student model is smaller and more efficient than the teacher, making it more practical for deployment in real-world applications

References

[1] https://huggingface.co/datasets/clinc_oos ("clinc_oos · Datasets at Hugging Face." *Hugging Face*, https://huggingface.co/datasets/clinc_oos.)

[2] https://arxiv.1909.11942 ("[1909.11942] ALBERT: A Lite BERT for Self-supervised Learning of Language Representations")

[3] https://arxiv.org/abs/1907.11692 ("[1907.11692] RoBERTa: A Robustly Optimized BERT Pretraining Approach." *arXiv*, 26 July 2019, https://arxiv.org/abs/1907.11692.)

[4] https://arxiv.org/abs/1904.09223 ("[1904.09223] ERNIE: Enhanced Representation through Knowledge Integration." *arXiv*, 19 April 2019, https://arxiv.org/abs/1904.09223. "[1907.11692])

[5] https://arxiv.org/abs/1910.01108 ("[1910.01108] DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter")

[7] https://arxiv.org/abs/2010.13382 (Kim, Young Jin, and Hany Hassan.) "[2010.13382] FastFormers: Highly Efficient Transformer Models for Natural Language Understanding." *arXiv*, 26 October 2020, https://arxiv.org/abs/2010.13382.

[8] https://arxiv.org/abs/1607.01759 ("[1607.01759] Bag of Tricks for Efficient Text Classification")

[10] https://arxiv.org/abs/2106.04563 ("[2106.04563] XtremeDistilTransformers: Task Transfer for Task-agnostic Distillation")

Quick Links

[6] Tensorflow Playground

[9] Accessing you HuggingFace Token

[11] Completed Colab Notebook