OPTIMIZING TRANSFORMER MODELS

Distillation, Quantization and ONNX Explained

AI Books

Optimizing Transformer Models: Distillation, Quantization and ONNX Explained

Transformer models are powerful but often too large and slow for real-time applications. To address this, we'll look at three key optimization techniques **knowledge distillation**, **quantization** and **ONNX graph optimization** . It can significantly reduce inference time and memory usage.

To illustrate the benefits and trade-offs associated with each technique, we'll use intent detection as a case study since it's an important component of text-based assistants, where low latencies are critical for maintaining a conversation in real-time.

Benchmarking Transformer Models for Production

Much like any machine learning model, deploying Transformers in production isn't just about accuracy — it's about making smart trade-offs between competing system-level demands. Three key constraints consistently rise to the surface:

- **Model Performance**
  How effectively does the model generalize to real-world data? In domains where mistakes are costly — whether due to regulatory risks, user impact, or scale — even slight improvements in precision or recall can have massive downstream effects. In high-risk scenarios, integrating a human-in-the-loop can help minimize critical errors.

- **Latency**
  How quickly can the model return a prediction? Low-latency inference is crucial in real-time applications, particularly those operating at scale. For instance, Stack Overflow required a

responsive classifier to instantly flag problematic comments without disrupting user flow.

- **Memory Efficiency**
  How do we serve models like GPT-2 or T5, which span billions of parameters and demand significant compute resources? Memory becomes a critical constraint, especially when deploying to mobile devices or edge environments, where access to high-performance cloud infrastructure is limited or non-existent.

Why Benchmarking Matters: Key Takeaways

Failing to balance performance, latency, and memory constraints in Transformer deployment can result in:

- **Degraded User Experience**
  Slow, unresponsive models frustrate users and diminish product value.
- **Unnecessary Infrastructure Costs**
  Running large models on cloud servers for minimal traffic can lead to **excessive compute bills** and poor resource utilization.

The Solution: Build a Targeted Benchmark

To address these challenges, we'll design a **lightweight benchmarking framework** that:

- **Evaluates core constraints** (performance, latency, memory)
- **Operates over a defined pipeline and test set**
- **Lays the groundwork** for applying model optimization techniques like quantization, pruning, and distillation

This begins with a simple, extensible benchmarking class — the foundation for systematic performance profiling and compression experiments.

```
class PerformanceBenchmark:
 def __init__(self, pipeline, dataset, optim_type="BERT baseline"):
 self.pipeline = pipeline
 self.dataset = dataset
 self.optim_type = optim_type
 def compute_accuracy(self):
 pass
 def compute_size(self):
 pass
 def time_pipeline(self):
 pass
def run_benchmark(self):
 metrics = {}
 metrics[self.optim_type] = self.compute_size()
 metrics[self.optim_type].update(self.time_pipeline())
 metrics[self.optim_type].update(self.compute_accuracy())
 return metrics
```

Measuring Model Accuracy with Real Data

With our benchmarking skeleton in place, it's time to bring it to life by computing the **model's accuracy** on a representative test set.

To do this, we'll use the **CLINC150 dataset** — a widely used benchmark for intent classification tasks. This dataset was also used to fine-tune our baseline Transformer model, making it an ideal starting point for evaluation.

```
from datasets import load_dataset
clinc = load_dataset("clinc_oos", "plus")
clinc
DatasetDict({
 train: Dataset({
 features: ['text', 'intent'],
 num_rows: 15250
 })
 validation: Dataset({
 features: ['text', 'intent'],
 num_rows: 3100
 })
```

```
 test: Dataset({
 features: ['text', 'intent'],
 num_rows: 5500
 })
})
```

Understanding the CLINC150 Dataset Structure

Each entry in the **CLINC150** dataset contains:

- A **user query** (stored in the `text` field)
- Its corresponding **intent label** (stored in the `intent` field)

For benchmarking purposes, we'll focus on the **test split**, as it best simulates real-world usage. To get a sense of the data format, let's inspect a sample from the test set:

```
Understanding the CLINC150 Dataset Structure
Each entry in the CLINC150 dataset contains:

A user query (stored in the text field)

Its corresponding intent label (stored in the intent field)

For benchmarking purposes, we'll focus on the test split, as it best simulates real-world usage. To get a sense of the data format, let's inspect a sample from the test set:
```

The intents are provided as IDs, but we can easily get the mapping to strings (and vice versa) by accessing the Dataset.features attribute:

```
intents = clinc["test"].features["intent"]
intents.int2str(clinc["test"][42]["intent"])
'transfer'
```

Now that we have a basic understanding of the contents in the CLINC150 dataset, let's implement the compute_accuracy function.

```
from datasets import load_metric
accuracy_score = load_metric('accuracy')
accuracy_score
Metric(name: "accuracy", features: {'predictions': Value(dtype='int32',
 > id=None), 'references': Value(dtype='int32', id=None)}, usage: """
Args:
 predictions: Predicted labels, as returned by a model.
 references: Ground truth labels.
 normalize: If False, return the number of correctly classified samples.
 Otherwise, return the fraction of correctly classified samples.
 sample_weight: Sample weights.
Returns:
 accuracy: Accuracy score.
""", stored examples: 0)
```

To evaluate our model's performance, we'll use the **accuracy** metric — but it requires both the predicted and true labels to be represented as **integer IDs**.

Here's the process:

1. **Generate predictions** using a pre-trained pipeline on the `text` field.
2. **Convert predicted labels to integer IDs** using `ClassLabel.str2int()`, which maps string class names to their corresponding numerical indices.
3. **Collect all predictions and ground-truth labels** into separate lists.
4. **Compute accuracy** by passing both lists to the metric function.

Let's integrate this logic into our `PerformanceBenchmark` class to automate the process:

```
ef 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
PerformanceBenchmark.compute_accuracy = compute_accuracy
```

To understand the memory footprint of our model, we'll serialize it to disk and measure its size. PyTorch provides a convenient method for this using `torch.save`, which relies on Python's built-in `pickle` module. It can be used to persist everything from models and tensors to plain Python objects.

When saving a model in PyTorch, the **recommended approach** is to save its `state_dict` — a dictionary containing all the learnable parameters (such as weights and biases) for each layer of the model.

Let's inspect what's inside the `state_dict` of our baseline Transformer model:

```
list(pipe.model.state_dict().items())[42]
('bert.encoder.layer.2.attention.self.value.weight',
 tensor([[-1.0526e-02, -3.2215e-02, 2.2097e-02, ..., -6.0953e-03,
 4.6521e-03, 2.9844e-02],
 [-1.4964e-02, -1.0915e-02, 5.2396e-04, ..., 3.2047e-05,
 -2.6890e-02, -2.1943e-02],
 [-2.9640e-02, -3.7842e-03, -1.2582e-02, ..., -1.0917e-02,
 3.1152e-02, -9.7786e-03],
 ...,
 [-1.5116e-02, -3.3226e-02, 4.2063e-02, ..., -5.2652e-03,
 1.1093e-02, 2.9703e-03],
 [-3.6809e-02, 5.6848e-02, -2.6544e-02, ..., -4.0114e-02,
 6.7487e-03, 1.0511e-03],
 [-2.4961e-02, 1.4747e-03, -5.4
```

So if we save our model with

```
torch.save(model.state_dict(), PATH)
```

We can use Python's `pathlib` module to measure its size.
Specifically, `Path(PATH).stat().st_size` returns the file size in **bytes**.

Let's integrate this into a `compute_size()` method inside our `PerformanceBenchmark` class to automate the process:

```
import torch
from pathlib import Path
def compute_size(self):
 state_dict = self.pipeline.model.state_dict()
 tmp_path = Path("model.pt")
 torch.save(state_dict, tmp_path)
 # Calculate size in megabytes
 size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
 # Delete temporary file
 tmp_path.unlink()
 print(f"Model size (MB) - {size_mb:.2f}")
 return {"size_mb": size_mb}
PerformanceBenchmark.compute_size = compute_size
```

To complete our benchmark, we'll measure **inference latency** — the time it takes for the model to process a single input and return a predicted intent. This gives us an estimate of real-world responsiveness, especially important for production systems requiring real-time predictions.

In this context, **latency** includes all processing steps within the pipeline, including tokenization and model inference. While tokenization is extremely fast (typically ~1000× faster than inference), it's still part of the end-to-end process, so we include it for completeness.

To accurately measure execution time, we'll use Python's `time.perf_counter()`, which offers high-resolution timing and is preferred over `time.time()` for performance benchmarking.

We can use perf_counter to time our pipeline by passing our test query and calculating the time difference in milliseconds between the start and end:

```
from time import perf_counter
for _ in range(3):
 start_time = perf_counter()
 _ = pipe(query)
 latency = perf_counter() - start_time
 print(f"Latency (ms) - {1000 * latency:.3f}")
Latency (ms) - 64.923
Latency (ms) - 47.636
Latency (ms) - 47.344
```

Latency can vary significantly between runs, especially for small inputs or under inconsistent system load. Timing a **single pass** through the pipeline often results in noisy measurements due to background processes, CPU throttling, or just-in-time (JIT) compilation effects.

To mitigate this and get more **reliable latency estimates**, we take the following approach:

1. **Warm up the CPU**: Run a few initial inferences to stabilize the runtime environment.
2. **Repeat measurements**: Perform inference over many samples to gather a distribution of latencies.
3. **Report mean and standard deviation**: These statistical values give a more robust view of typical latency and its variability.

Here's how this logic can be implemented in the `PerformanceBenchmark` class:

```
import numpy as np
def time_pipeline(self, query="What is the pin number for my account?"):
 latencies = []
 # Warmup
 for _ in range(10):
 _ = self.pipeline(query)
 # Timed run
 for _ in range(100):
 start_time = perf_counter()
 _ = self.pipeline(query)
 latency = perf_counter() - start_time
 latencies.append(latency)
 # Compute run statistics
 time_avg_ms = 1000 * np.mean(latencies)
 time_std_ms = 1000 * np.std(latencies)
 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}
PerformanceBenchmark.time_pipeline = time_pipeline
```

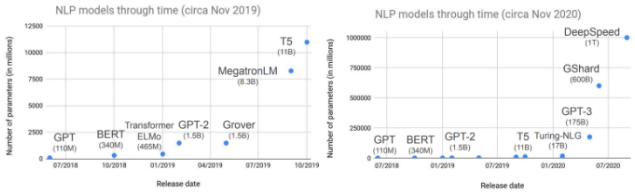
Benchmarking Our Baseline Model

we'll collect the results in the perf_metrics dictionary to keep track of each model's performance:

```
pb = PerformanceBenchmark(pipe, clinc["test"])
perf_metrics = pb.run_benchmark()
Model size (MB) - 418.17
Average latency (ms) - 46.05 +\- 10.13
Accuracy on test set - 0.867
```

Scaling Intelligence: Knowledge Distillation for Efficient Model Deployment

Knowledge distillation is a general-purpose method for training a smaller student model to mimic the behavior of a slower, larger, but better performing teacher.



Parameter counts of several recent pretrained language models

Knowledge Distillation for Efficient Fine-Tuning

Knowledge distillation is a powerful technique used during the fine-tuning phase of supervised learning, where a larger, well-trained "teacher" model transfers its learned behavior to a smaller "student" model. The goal isn't just to replicate performance — but to **transfer nuanced, learned insights** often invisible in ground truth labels.

🔢 Mathematical Mechanism Behind Distillation

1. **Logits Generation:**
   Input sequence x is passed to the teacher, generating raw prediction scores: z(x)=[z1(x),z2(x),...,zN(x)]

2. **Softmax with Temperature Scaling:**
   Traditional softmax:

$$\frac{\exp\left(z_i\left(x\right)\right)}{\sum_j \exp\left(z_i\left(x\right)\right)},$$

However, this often leads to **peaked distributions** with little information gain.

$$p_i\left(x\right) = \frac{\exp\left(z_i\left(x\right)/T\right)}{\sum_j \exp\left(z_i\left(x\right)/T\right)}.$$

Improved softmax with temperature T:

☑ **Higher T⇒** Softer distributions
☑ **More informative** about class relationships and decision boundaries

⚖️ Loss Function: Balancing Accuracy with Insight

- **Student's Soft Predictions:** qi(x)
- **KL Divergence Loss (Knowledge Distillation Loss):**

$$L_{KD} = T^2 D_{KL} = T^2 \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)},$$

- The factor T^2 normalizes gradient magnitudes.
- **Total Student Loss:**

$$L_{\text{student}} = \alpha L_{CE} + (1-\alpha) L_{KD},$$

🧠 Inference Phase

At inference time, temperature T is reset to **1** to restore standard prediction confidence.

Knowledge Distillation During Pretraining: Building Smarter, Smaller Models

While knowledge distillation is often used during fine-tuning, it's equally effective during **pretraining** — allowing the creation of compact, general-purpose models that are faster and more efficient.

How It Works in Pretraining

- A **large pretrained teacher** (e.g., BERT) transfers its understanding of **masked language modeling (MLM)** to a **smaller student**.
- The student learns not only from the original MLM objective but also from the **behavioral patterns** and **representations** of the teacher.

DistilBERT Loss Function

In the **DistilBERT** architecture, the total loss combines three components:

$$L_{\text{DistilBERT}} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}\,.$$

Real-World Application

Since we already have a **fine-tuned BERT-base model**, we can now:

- Use it as a **teacher** to guide a smaller student model.
- Implement a custom **Trainer** that integrates both **cross-entropy** and **distillation losses**.

This approach not only accelerates inference time but also reduces resource usage — without compromising too much on performance.

Building a Knowledge Distillation Trainer in PyTorch

To implement **knowledge distillation** in a fine-tuning setup, we extend the Hugging Face `Trainer` class with additional components that allow a student model to learn from a pretrained teacher model.

Key Components to Add

1. **Hyperparameters:**

- `alpha (α)`: Balances between cross-entropy and distillation loss (default = 0.5).
- `temperature (T)`: Softens logits for smoother probability distributions (default = 2.0).

2.Teacher Model:

- A **fine-tuned BERT-base** model serves as the teacher from which the student will learn.

3.Custom Loss Function:

- Combines **cross-entropy loss** (for ground truth labels) with **KL divergence** (for mimicking teacher outputs).

Step-by-Step Code Implementation

1. Custom Training Arguments

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

2. Custom Trainer with Distillation Logic

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

```
class DistillationTrainer(Trainer):
    def __init__(self, *args, teacher_model=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.teacher_model = teacher_model

    def compute_loss(self, model, inputs):
        outputs_stu = model(**inputs)
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits

        # Forward pass through teacher (no gradients)
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits

        # Compute KL divergence-based distillation loss
        loss_fct = nn.KLDivLoss(reduction="batchmean")
        loss_kd = self.args.temperature ** 2 * loss_fct(
            F.log_softmax(logits_stu / self.args.temperature, dim=-1),
            F.softmax(logits_tea / self.args.temperature, dim=-1)
        )

        # Weighted sum of losses
        return self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
```

How It Works: Behind the Scenes

- **Teacher Predictions:** No gradients are computed; it's a fixed model.
- **Soft Logits:** Student logits are passed through `log_softmax`, teacher logits through `softmax`.
- **KL Divergence:** Measures how closely the student mimics the teacher's softened predictions.
- **Loss Blending:** Final loss = `α * CrossEntropy + (1 - α) * DistillationLoss`.

Choosing a Good Student Initialization

First we'll need to tokenize and encode our queries, so let's instantiate the tokenizer from DistilBERT and create a simple function to take care of the preprocessing:

```
student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_ckpt)
def tokenize_text(batch, tokenizer):
 return tokenizer(batch["text"], truncation=True)
clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=["text"],
 fn_kwargs={"tokenizer": student_tokenizer})
clinc_enc.rename_column_("intent", "labels")
```

Here we've removed the text column since we no longer need it and we've also used the fn_kwargs argument to specify which tokenizer should be used in the tokenize_text function. We've also renamed the intent column to labels so it can be automatically detected by the trainer. Now that our texts are processed, the next thing to do is instantiate DistilBERT for fine-tuning. Since we will be doing multiple runs with the trainer, we'll use a function to initialize the model with each new run:

```
import torch
from transformers import AutoConfig
num_labels = intents.num_classes
id2label = bert_model.config.id2label
label2id = bert_model.config.label2id
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
student_config = (AutoConfig
 .from_pretrained(student_ckpt, num_labels=num_labels,
 id2label=id2label, label2id=label2id))
def student_init():
 return (AutoModelForSequenceClassification
 .from_pretrained(student_ckpt, config=student_config).to(device))
```

We need to define the metrics to track during training,

```
f compute_metrics(pred):
 predictions, labels = pred
```

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

Finally, we just need to define the training arguments. To warm-up, we'll set $\alpha = 1$ to see how well DistilBERT performs without any signal from the teacher:

```
batch_size = 48
student_training_args = DistillationTrainingArguments(
 output_dir="checkpoints", 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)
```

Next we load the teacher model, instantiate the trainer and start fine-tuning:

```
teacher_checkpoint = "lewtun/bert-base-uncased-finetuned-clinc"
teacher_model = (AutoModelForSequenceClassification
 .from_pretrained(teacher_checkpoint, num_labels=num_labels)
 .to(device))
distil_trainer = DistillationTrainer(model_init=student_init,
 teacher_model=teacher_model, args=student_training_args,
 train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
 compute_metrics=compute_metrics, tokenizer=student_tokenizer)
distil_trainer.train();
```

Epoch	Training Loss	Validation Loss	Accuracy	Runtime	Samples Per Second
1	4.309400	3.318003	0.702903	0.970700	3193.619000
2	2.659500	1.904174	0.843871	0.979200	3165.834000
3	1.573200	1.178305	0.894194	0.988600	3135.649000
4	1.026700	0.873162	0.911613	0.987400	3139.536000
5	0.805600	0.785567	0.917742	1.019300	3041.436000

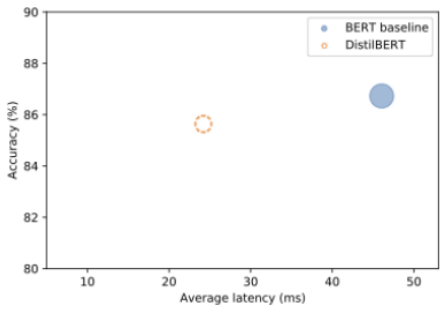
Wrap it in a TextClassificationPipeline and run it through our performance benchmark:

```
pipe = TextClassificationPipeline(
 model=distil_trainer.model.to("cpu"), tokenizer=distil_trainer.tokenizer)
optim_type = "DistilBERT"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())

Model size (MB) - 255.89
Average latency (ms) - 24.13 +\- 10.06
Accuracy on test set - 0.856
```

Create a scatter plot of the accuracy against the latency, with the radius of each point corresponding to the size of the model.

```
import pandas as pd
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]
 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])
 plt.ylim(80,90)
 plt.xlim(5, 53)
 plt.ylabel("Accuracy (%)")
 plt.xlabel("Average latency (ms)")
 plt.show()
plot_metrics(perf_metrics, optim_type)
```



Tuning Distillation Hyperparameters with Optuna

Optuna treats hyperparameter tuning as an **optimization problem**. It defines an **objective function**, then runs multiple **trials** to minimize or maximize it.

Rosenbrock's Banana Function:

A classic benchmark in optimization:

$$f(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2$$

- **Global minimum** at: (x,y)=(1,1)
- Named for its **curved, banana-shaped contours**
- Simple in theory but **challenging to converge** to the true minimum

Now, let's apply a similar approach to **optimize knowledge distillation parameters** in a Hugging Face `Trainer`.

Define the Hyperparameter Space

```
def hp_space(trial):
    return {
```

```
        "num_train_epochs": trial.suggest_int("num_train_epochs", 5, 10),
        "alpha": trial.suggest_float("alpha", 0, 1),
        "temperature": trial.suggest_int("temperature", 2, 20)
    }
```

Run the Hyperparameter Search

```
best_run = distil_trainer.hyperparameter_search(
    n_trials=20,
    direction="maximize",
    hp_space=hp_space
)
```

- `direction="maximize"` tells Optuna to seek **higher accuracy**.
- `best_run` contains the best trial's configuration and performance.

Sample Output

```
print(best_run)
# BestRun(run_id='4', objective=3080.87,
#   hyperparameters={'num_train_epochs': 8, 'alpha': 0.31, 'temperature': 16})
```

💡 **Insight:** The selected **α = 0.31** indicates that **most of the learning signal** came from **knowledge distillation**, rather than the ground truth labels.

Apply Best Hyperparameters & Retrain

```
for k, v in best_run.hyperparameters.items():
    setattr(distil_trainer.args, k, v)

distil_trainer.train()
```

Epoch	Training Loss	Validation Loss	Accuracy	Runtime	Samples Per Second
1	1.608300	2.977128	0.714516	0.981500	3158.565000
2	0.904200	1.566405	0.877419	0.981000	3159.929000
3	0.509100	0.881892	0.915806	0.988000	3137.623000
4	0.317700	0.594229	0.932581	1.024500	3025.740000
5	0.230200	0.475622	0.934839	0.995800	3113.162000
6	0.189800	0.419630	0.939032	1.014300	3056.174000
7	0.170300	0.394079	0.943226	1.012700	3061.031000
8	0.161400	0.386891	0.942258	1.009100	3072.173000

Save the model for future use:

```
distil_trainer.save_model("models/distilbert-base-uncased-distilled-clinc")
```

Benchmarking Our Distilled Model

Create a pipeline and redo our benchmark to see how we perform on the test set:

```
pipe = TextClassificationPipeline(
 model=distil_trainer.model.to("cpu"), tokenizer=distil_trainer.tokenizer)
optim_type = "Distillation"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
Model size (MB) - 255.89
Average latency (ms) - 24.58 +\- 7.66
Accuracy on test set - 0.871
```

Accelerating Transformers with Quantization

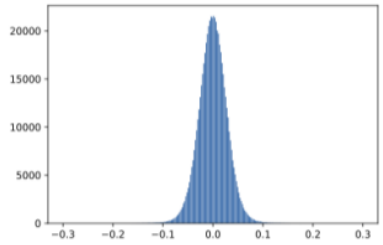
While **knowledge distillation** reduces model size by training a smaller student model, **quantization** improves efficiency by reducing the precision of computations — typically from 32-bit floats (FP32) to 8-bit integers (INT8). This results in:

- **Smaller model size**
- **Faster inference**
- **Minimal accuracy loss**

Visualizing Weight Distributions for Quantization

Transformer weights often lie in a narrow range, making them ideal for INT8 quantization.

```
import matplotlib.pyplot as plt
weights =
bert_model.state_dict()["bert.encoder.layer.0.attention.output.dense.weight"]
plt.hist(weights.flatten().numpy(), bins=250, range=(-0.3, 0.3));
```



If most values are in [−0.1, 0.1], we can safely quantize them to INT8 (−128 to 127) with minimal loss.

Manual Quantization Example

Step 1: Calculate Scale and Zero-Point

```
zero_point = 0
scale = (weights.max() - weights.min()) / (127 - (-128))
```

Step 2 : Quantize the Tensor

```
(weights / scale + zero_point).clamp(-128, 127).round().char()
[[ 2, -1, 1, ..., -2, -6, 9],
 [ 7, 2, -4, ..., -3, 5, -3],
 [-15, -8, 5, ..., 3, 0, -2],
 ...,
 [ 11, -1, 12, ..., -2, 0, -3],
 [ -2, -6, -13, ..., 11, -3, -10],
 [-12, 5, -3, ..., 7, -3, -1]], dtype=torch.int8)
```

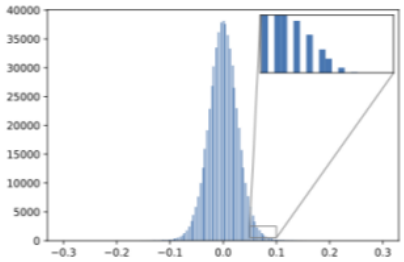
Step 3: Using PyTorch's API

```
from torch import quantize_per_tensor
quantized_weights = quantize_per_tensor(weights, scale, zero_point, torch.qint8)
quantized_weights.int_repr()
([[ 2, -1, 1, ..., -2, -6, 9],
 [ 7, 2, -4, ..., -3, 5, -3],
 [-15, -8, 5, ..., 3, 0, -2],
 ...,
 [ 11, -1, 12, ..., -2, 0, -3],
 [ -2, -6, -13, ..., 11, -3, -10],
 [-12, 5, -3, ..., 7, -3, -1]], dtype=torch.int8)
```

If we dequantize this tensor, we can visualize the frequency distribution to see the effect that rounding has had on our original values:

```
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes,mark_inset
# Create histogram
fig, ax = plt.subplots()
ax.hist(quantized_weights.dequantize().flatten().numpy(),
 bins=250, range=(-0.3,0.3));
# Create zoom inset
```

```
axins = zoomed_inset_axes(ax, 5, loc='upper right')
axins.hist(quantized_weights.dequantize().flatten().numpy(),
 bins=250, range=(-0.3,0.3));
x1, x2, y1, y2 = 0.05, 0.1, 500, 2500
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)
axins.axes.xaxis.set_visible(False)
axins.axes.yaxis.set_visible(False)
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
plt.show()
```



This shows very clearly the discretization that's induced by only mapping some of the weight values precisely and rounding the rest. To round out our little analysis, let's compare how long it takes to compute the multiplication of two weight tensors with FP32 and INT8 values. For the FP32 tensors we can multiply them using PyTorch's nifty @ operator:

```
%%timeit
weights @ weights
```

For the quantized tensors we need the QFunctional wrapper class so that we can perform operations with the special torch.qint8 data type:

```
from torch.nn.quantized import QFunctional
q_fn = QFunctional()
```

This class supports various elementary operations like addition and in our case we can time the multiplication of our quantized tensors as follows:

```
%%timeit
q_fn.mul(quantized_weights, quantized_weights)
107 µs ± 7.87 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Memory Comparison

```
import sys
sys.getsizeof(weights.storage()) / sys.getsizeof(quantized_weights.storage())
# ~4× smaller
```

🎯 Quantization Strategies

Method	When Used	Notes
Dynamic Quantization	Inference only	Simple, no training changes, quantize weights & activations on-the-fly
Static Quantization	Pre-inference	Uses calibration data, avoids float conversions during inference
Quantization Aware Training (QAT)	During training	Simulates quantization to minimize accuracy drop

Quantizing Transformers with PyTorch

```
from torch.quantization import quantize_dynamic
from transformers import AutoModelForSequenceClassification, AutoTokenizer

model_ckpt = "models/distilbert-base-uncased-distilled-clinc"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

model = AutoModelForSequenceClassification.from_pretrained(model_ckpt).to("cpu")

model_quantized = quantize_dynamic(model, {torch.nn.Linear}, dtype=torch.qint8)
```

This line:

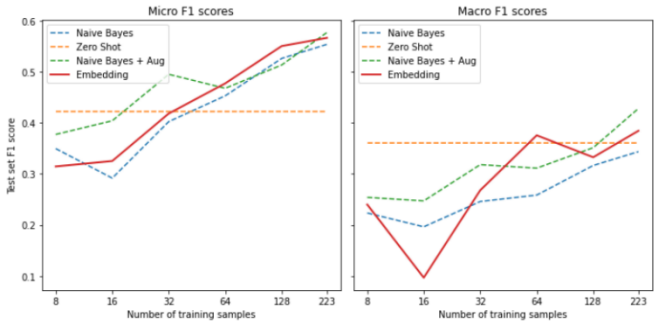
- Quantizes all `nn.Linear` layers.
- Uses INT8 arithmetic for faster inference.
- Maintains almost the same accuracy.

Benchmarking the Power of Quantized Models

With our model successfully quantized, it's time to put its performance to the test. We'll run a benchmark to evaluate both its speed and memory efficiency — crucial factors for deployment in resource-constrained environments.

Here's how we set up and execute the benchmark:

```
pipe = TextClassificationPipeline(model=model_quantized, tokenizer=tokenizer)
optim_type = "Distillation + quantization"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
plot_metrics(perf_metrics, optim_type)
```



Optimizing Inference with ONNX and the ONNX Runtime

With our distilled model already optimized and quantized, it's time to push the boundaries even further using the ONNX framework — a robust platform for deep learning model interoperability and high-performance inference.

ONNX (Open Neural Network Exchange) is an open standard that defines:

- A **common operator set** across frameworks
- A **unified file format** for model export/import
- A **computational graph representation** of neural networks

Thanks to ONNX, you can easily **export a PyTorch model** and import it into TensorFlow — or vice versa — enabling flexible deployment across ecosystems.

Setting up OpenMP environment variables for ONNX:

```
from psutil import cpu_count
%env OMP_NUM_THREADS={cpu_count()}
%env OMP_WAIT_POLICY=ACTIVE
env: OMP_NUM_THREADS=8
env: OMP_WAIT_POLICY=ACTIVE
```

Converting our distilled model to the ONNX format:

```
from transformers.convert_graph_to_onnx import convert
onnx_model_path = Path("onnx/model.onnx")
convert(framework="pt", model=model_ckpt, tokenizer=tokenizer,
 output=onnx_model_path, opset=12, pipeline_name="sentiment-analysis")
ONNX opset version set to: 12
Loading pipeline (model: models/distilbert-base-uncased-distilled-clinc,
 > tokenizer: PreTrainedTokenizerFast(name_or_path='models/distilbert-base-
 > uncased-distilled-clinc', vocab_size=30522, model_max_len=512, is_fast=True,
 > padding_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token':
 > '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token':
 > '[MASK]'}))
Creating folder onnx
Using framework PyTorch: 1.5.0
Found input input_ids with shape: {0: 'batch', 1: 'sequence'}
Found input attention_mask with shape: {0: 'batch', 1: 'sequence'}
Found output output_0 with shape: {0: 'batch'}
Ensuring inputs are in correct order
```

```
head_mask is not present in the generated input list.
Generated inputs order: ['input_ids', 'attention_mask']
```

ONNX uses operator sets to group together immutable operator specifications, so opset=12 corresponds to a specific version of the ONNX library. Now that we have our model saved.

We need to create and inference session to feed inputs to the model:

```
om onnxruntime import (GraphOptimizationLevel, InferenceSession,
 SessionOptions)
def create_model_for_provider(model_path, provider="CPUExecutionProvider"):
 options = SessionOptions()
 options.intra_op_num_threads = 1
 options.graph_optimization_level = GraphOptimizationLevel.ORT_ENABLE_ALL
 session = InferenceSession(str(model_path), options, providers=[provider])
 session.disable_fallback()
 return session
onnx_model = create_model_for_provider(onnx_model_path)
```

Testing this out with an example from the test set. Since the output from the convert function tells us that ONNX expects just the input_ids and attention_mask as inputs, we need to drop the label column from our sample:

```
inputs = clinc_enc["test"][:1]
del inputs["labels"]
logits_onnx = onnx_model.run(None, inputs)[0]
logits_onnx.shape


##(1, 151)
```

Get the predicted label by taking the argmax:

```
np.argmax(logits_onnx)

## adding  ground labels


clinc_enc["test"][0]["labels"]
```

We'll create our own class that mimics the core behaviour:

```
from scipy.special import softmax
class OnnxPipeline:
 def __init__(self, model, tokenizer):
 self.model = model
 self.tokenizer = tokenizer
 def __call__(self, query):
 model_inputs = self.tokenizer(query, return_tensors="pt")
 inputs_onnx = {k: v.cpu().detach().numpy()
 for k, v in model_inputs.items()}
 logits = self.model.run(None, inputs_onnx)[0][0, :]
 probs = softmax(logits)
 pred_idx = np.argmax(probs).item()
 return [{"label": intents.int2str(pred_idx), "score": probs[pred_idx]}]



##We can then test this on our simple query to see if we recover the car_rental
intent:

pipe = OnnxPipeline(onnx_model, tokenizer)
pipe(query)
[{'label': 'car_rental', 'score': 0.8440852}]
```

Benchmarking ONNX Models Efficiently

Now that our ONNX pipeline is functioning correctly, the next logical step is to **benchmark its performance**. To do this, we'll extend our existing `PerformanceBenchmark` class. Since the ONNX model is an instance

of `InferenceSession` (rather than a PyTorch `nn.Module`), it lacks attributes like `state_dict`, making `torch.save()` unusable for size calculation.

🔧 To address this, we'll **override only the `compute_size()` method** while reusing the existing implementations for `compute_accuracy()` and `time_pipeline()`.

Here's a streamlined way to handle ONNX model sizing:

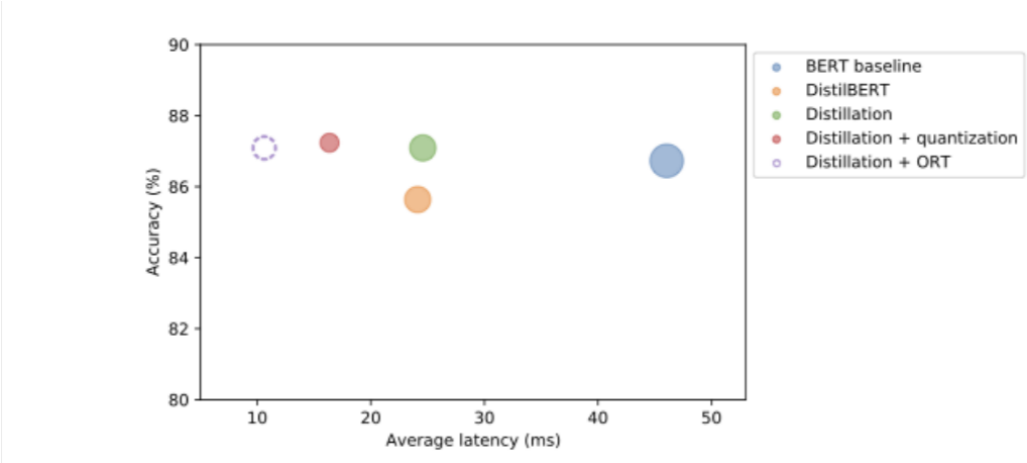
```
lass OnnxPerformanceBenchmark(PerformanceBenchmark):
 def __init__(self, *args, model_path, **kwargs):
 super().__init__(*args, **kwargs)
 self.model_path = model_path
 def compute_size(self):
 size_mb = Path(self.model_path).stat().st_size / (1024 * 1024)
 print(f"Model size (MB) - {size_mb:.2f}")
 return {"size_mb": size_mb}
```

With our new benchmark, let's see how our distilled model performs when converted to ONNX format:

```
optim_type = "Distillation + ORT"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
 model_path="onnx/model.onnx")
perf_metrics.update(pb.run_benchmark())

#Model size (MB) - 255.89
#Average latency (ms) - 10.54 +\- 2.20
#Accuracy on test set - 0.871

plot_metrics(perf_metrics, optim_type)
```



Optimizing Transformer Inference with ONNX Runtime

We've seen that ONNX Runtime (ORT) already provides strong out-of-the-box performance gains when converting a distilled Transformer model. But we can take this further by applying **Transformer-specific graph optimizations** using ORT's optimization toolkit.

Transformer-Specific Optimization

For Transformer architectures like DistilBERT, ONNX Runtime Tools offers advanced optimization tailored to models of type `bert`. To begin, we define a set of model-specific optimization options using `BertOptimizationOptions`:

```
from onnxruntime_tools.transformers.onnx_model_bert import BertOptimizationOptions

model_type = "bert"
opt_options = BertOptimizationOptions(model_type)
opt_options.enable_embed_layer_norm = False  # Improves model size compression
```

Disabling embedding layer norm fusion can yield better compression in some cases.

Next, we run the optimization pass:

```
from onnxruntime_tools import optimizer

opt_model = optimizer.optimize_model(
    "onnx/model.onnx",
    model_type=model_type,
    num_heads=12,
    hidden_size=768,
    optimization_options=opt_options
)

opt_model.save_model_to_file("onnx/model.opt.onnx")
```

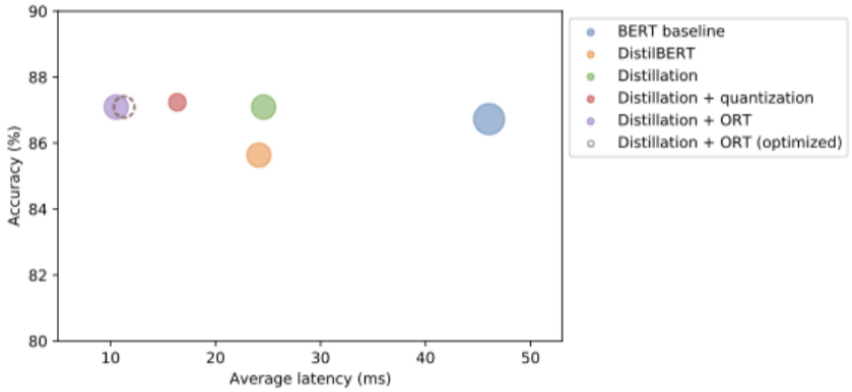
We provide the number of attention heads and hidden size from our DistilBERT model. Once optimized, we can run a performance benchmark:

```
onnx_model_opt = create_model_for_provider("onnx/model.opt.onnx")
pipe = OnnxPipeline(onnx_model_opt, tokenizer)
optim_type = "Distillation + ORT (optimized)"

pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
model_path="onnx/model.opt.onnx")
perf_metrics.update(pb.run_benchmark())

# Output
# Model size (MB) - 255.86
# Average latency (ms) - 11.22 ± 3.52
# Accuracy on test set - 0.871

plot_metrics(perf_metrics, optim_type)
```



🔍 **Insight**: Our original ONNX optimization was already near optimal — this BERT-specific pass did not yield major improvements in size or speed.

Adding Quantization into the Mix

To further reduce size and latency, we apply **dynamic quantization** using ONNX Runtime's quantization tools. Unlike PyTorch, which mainly quantizes `nn.Linear` layers, ORT can also quantize embedding layers, leading to better results.

```
from onnxruntime.quantization import quantize_dynamic, QuantType

model_input = "onnx/model.onnx"
model_output = "onnx/model.quant.onnx"

quantize_dynamic(model_input, model_output, weight_type=QuantType.QInt8)
```

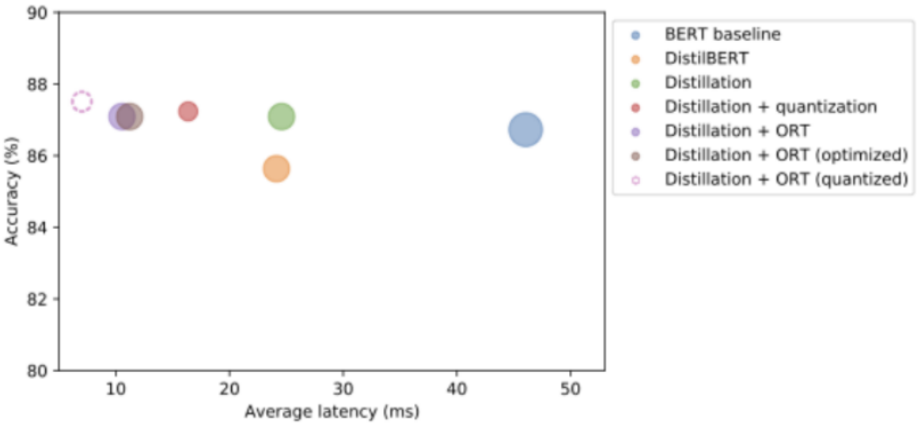
Now, let's benchmark the quantized ONNX model:

```
onnx_quantized_model = create_model_for_provider(model_output)
pipe = OnnxPipeline(onnx_quantized_model, tokenizer)
optim_type = "Distillation + ORT (quantized)"

pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
model_path=model_output)
perf_metrics.update(pb.run_benchmark())

# Output
# Model size (MB) - 185.71
# Average latency (ms) - 6.95 ± 4.75
# Accuracy on test set - 0.875

plot_metrics(perf_metrics, optim_type)
```



Result: ORT quantization cuts both size and latency by nearly **50%** compared to PyTorch quantization. Overall, this yields an impressive **7x speedup** over the original BERT baseline, with minimal accuracy loss.