# 트랜스포머를 활용한 자연어처리

8장

2024.01.23
염정훈

# 목차

# 의도 탐지

의도탐지 : 다양한 자연어 텍스트를 사전에 정의된 일련의 행동이나 의도로 분류하는 것

Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in Paris and I need a 15 passenger van

# 벤치마크 클래스

트랜스포머를 제품환경에 배포하기위한 조건

**모델성능** : 오류가 발생했을 때, 손실 비용이 크거나, 수백만개의 샘플에서 추론을 할때,
모델 지표가 향상되면 큰 이득을 얻을수 있다.

**레이턴시** : 대량의 트래픽을 처리하는 황경에서 고려한다.

**메모리** : 모바일과 에지장치에서 중요한 역할을 한다. 강력한 클라우드 서버에 접속하지 않고
예측을 만들어야 한다.

# 벤치마크 클래스 코드

```python
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):
        """PerformanceBenchmark.compute_accuracy() 메서드를 오버라이드합니다"""
        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"테스트 세트 정확도 - {accuracy['accuracy']:.3f}")
        return accuracy

    def compute_size(self):
        """PerformanceBenchmark.compute_size() 메서드를 오버라이드합니다"""
        state_dict = self.pipeline.model.state_dict()
        tmp_path = Path("model.pt")
        torch.save(state_dict, tmp_path)
        # 메가바이트 단위로 크기를 계산합니다
        size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
        # 임시 파일을 삭제합니다
        tmp_path.unlink()
        print(f"모델 크기 (MB) - {size_mb:.2f}")
        return {"size_mb": size_mb}
```

```python
def time_pipeline(self, query="What is the pin number for my account?"):
    """PerformanceBenchmark.time_pipeline() 메서드를 오버라이드합니다d"""
    latencies = []
    # 워밍업
    for _ in range(10):
        _ = self.pipeline(query)
    # 실행 측정
    for _ in range(100):
        start_time = perf_counter()
        _ = self.pipeline(query)
        latency = perf_counter() - start_time
        latencies.append(latency)
    # 통계 계산
    time_avg_ms = 1000 * np.mean(latencies)
    time_std_ms = 1000 * np.std(latencies)
    print(f"평균 레이턴시 (ms) - {time_avg_ms:.2f} +뻐- {time_std_ms:.2f}")
    return {"time_avg_ms": time_avg_ms, "time_std_ms": time_std_ms}

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
```

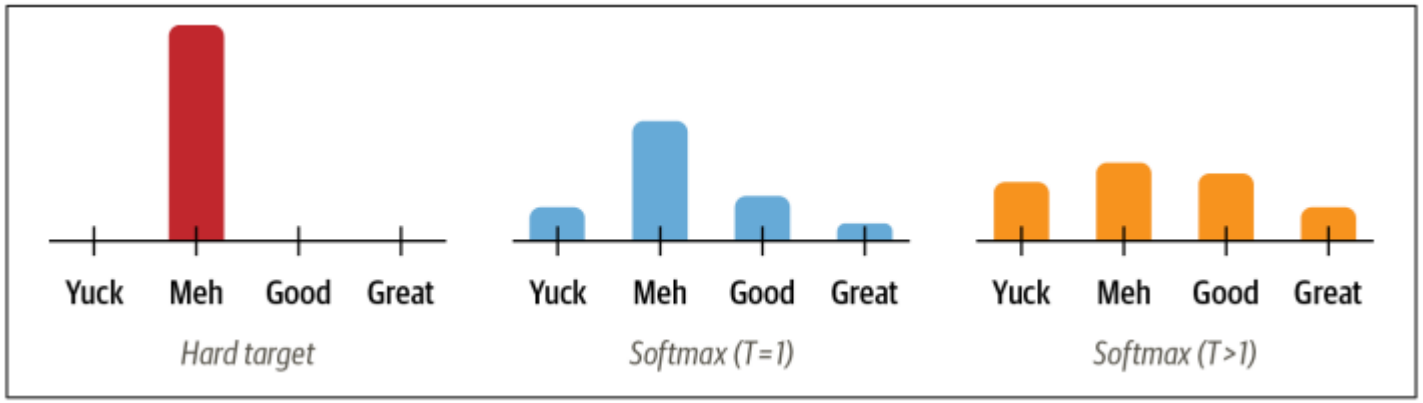
# 지식 정제

지식정제 : 작은 스튜던트 모델을 훈련하는 방법으로, 느리고 크지만 성능이 더 높은 티처의 동작을 모방하도록 작은 스튜던트 모델을 훈련하는 것

$$\mathbf{z}(x) = [z_1(x), ..., z_N(x)]$$

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

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

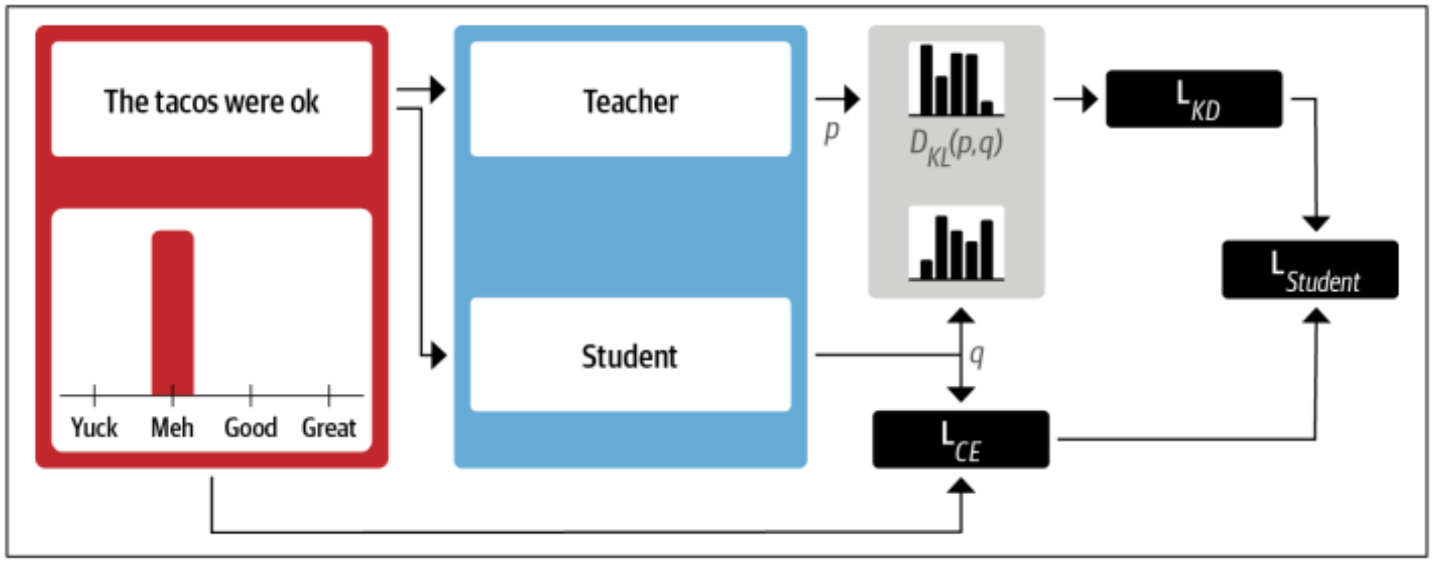


# 쿨백-라이블러(KL) 발산

$$D_{KL}(p,q) = \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)}$$

$$L_{KD} = T^2 D_{KL}$$

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



# 지식 정제 트레이너

지식 정제를 구현하기 위해  Trainer 클래스에 추가해야 하는 것
- α : 정제 손실의 상대적 가중치를 제어한다.
- T : 레이블의 확률 분포를 얼마나 완만하게 할지 조절한다.

- 미세 튜닝한 티처 모델

- 크로스 엔트로피와 지식 정제 손실을 연결한 새로운 손실함수

```python
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
```

```python
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, return_outputs=False):
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        inputs = inputs.to(device)
        outputs_stu = model(**inputs)
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits

        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits

        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))
        loss = self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
        return (loss, outputs_stu) if return_outputs else loss
```

# 스튜던트 선택

스튜던트로 사전 훈련된 언어 모델을 선택할 때, 레이턴시와 메모리 사용량을 줄이기위해 스튜던트로 작은 모델을 골라야한다.

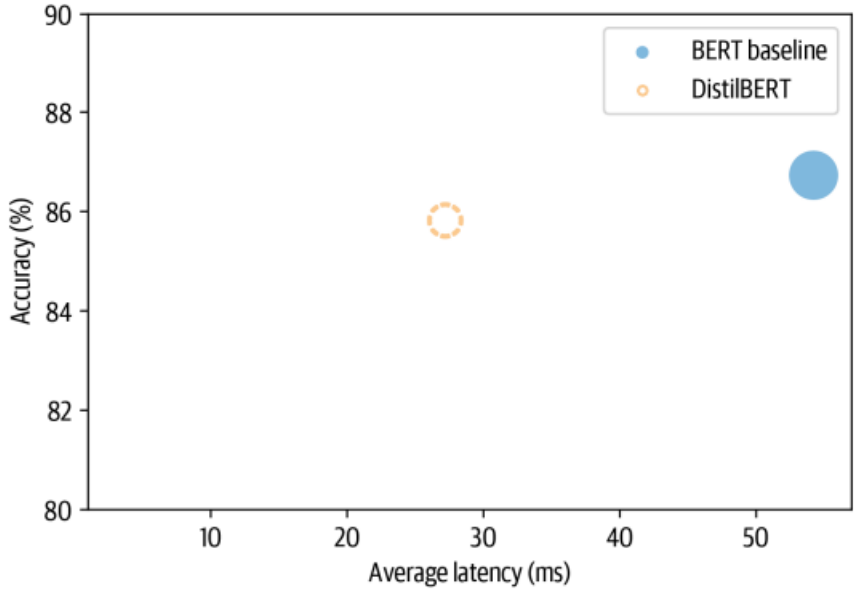
```
from transformers import AutoTokenizer

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

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

clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=["text"])
clinc_enc = clinc_enc.rename_column("intent", "labels")
```

```
batch_size = 48

finetuned_ckpt = "distilbert-base-uncased-finetuned-clinc"
student_training_args = DistillationTrainingArguments(
    output_dir=finetuned_ckpt, 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)
```

```
teacher_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
teacher_model = (AutoModelForSequenceClassification
                 .from_pretrained(teacher_ckpt, num_labels=num_labels)
                 .to(device))

distilbert_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)

distilbert_trainer.train()
```

| Epoch | Training Loss | Validation Loss | Accuracy |
|---|---|---|---|
| 1 | 4.293100 | 3.277564 | 0.728710 |
| 2 | 2.615900 | 1.864696 | 0.835806 |
| 3 | 1.540800 | 1.152417 | 0.897742 |
| 4 | 1.009900 | 0.854747 | 0.912903 |
| 5 | 0.794000 | 0.773027 | 0.916129 |

```
Model size (MB) - 255.89
Average latency (ms) - 27.53 +\- 0.60
Accuracy on test set - 0.858
```



# 옵투나

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

distilled_ckpt = "distilbert-base-uncased-distilled-clinc"
student_training_args.output_dir = distilled_ckpt

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 |
|---|---|---|---|
| 1 | 0.9031 | 0.574540 | 0.736452 |
| 2 | 0.4481 | 0.285621 | 0.874839 |
| 3 | 0.2528 | 0.179766 | 0.918710 |
| 4 | 0.1760 | 0.139828 | 0.929355 |
| 5 | 0.1416 | 0.121053 | 0.934839 |
| 6 | 0.1243 | 0.111640 | 0.934839 |
| 7 | 0.1133 | 0.106174 | 0.937742 |
| 8 | 0.1075 | 0.103526 | 0.938710 |
| 9 | 0.1039 | 0.101432 | 0.938065 |
| 10 | 0.1018 | 0.100493 | 0.939355 |

```
Model size (MB) - 255.89
Average latency (ms) - 25.96 +\- 1.63
Accuracy on test set - 0.868
```

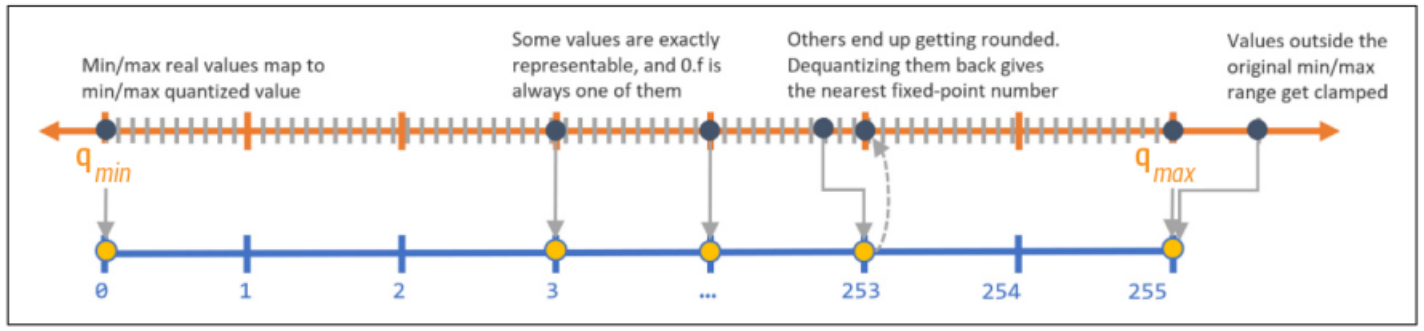
# 양자화

양자화 방식은 계산량을 줄이는 대신, 가중치와 활성화 출력을 32비트 부동소수점(FP32)가 아닌 8비트정수(INT8)같이 정밀도가 낮은 데이터 타입으로 표현해 계산을 효율적으로 하는 것이다.

양자화는 부동 소수점 숫자 f를 이산화 할 수 있다.
원래 범위 $[f_{\max}, f_{\min}]$ 에서 작은 범위 $[q_{\max}, q_{\min}]$ 로 매핑하고 모든 값을 선형적으로 분포시킨다.

$$f = \left(\frac{f_{\max} - f_{\min}}{q_{\max} - q_{\min}}\right)(q - Z) = S(q - Z)$$

여기서의 S는 양의 부동 소수점 숫자고, 상수 Z는 q와 동일한 타입이며, 영점이라고 부른다.
고정 소수점 숫자를 역양자화해 부동 소수점 숫자로 되돌려야 해서, 아핀 변환 이라고 한다.



# 양자화

모델에 있는 모든 연산에서 정밀도를 바꾸면 모델의 계산그래프에 변동이 생기고 성능에 나쁜영향을 끼친다.
심층 신경망에서의 양자화 방법

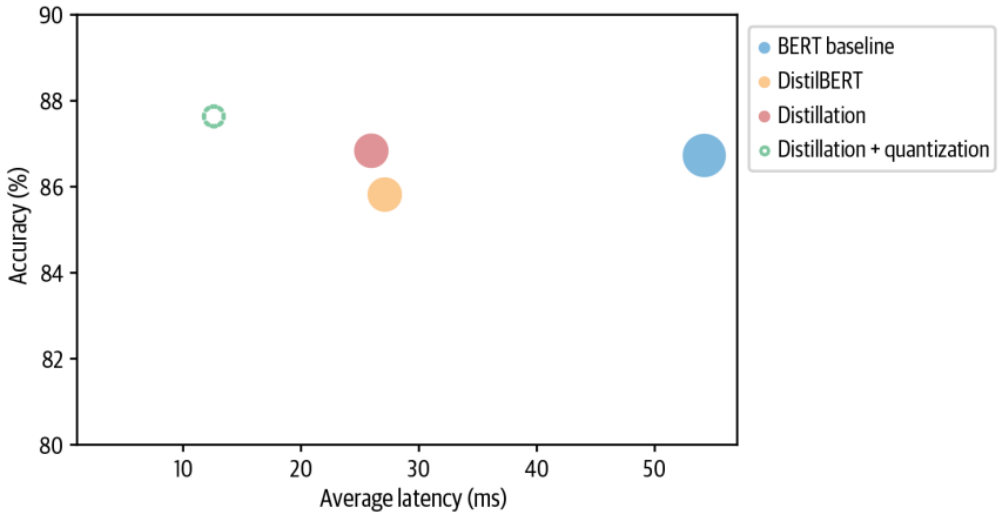
1. 동적 양자화 : 동적 양자화를 사용할 때는 훈련 도중에 아무것도 바뀌지 않고 추론 과정에만 적응된다.
다른 양자화 방법과 마찬가지로 모델 가중치가 추론 전에 INT8로 변환된다.

2. 정적 양자화 : 즉석에서 활성화를 양자화하지 않고 양자화 체계를 사전에 계산해 부동 소수점 변환을 피한다.
정적 양자화는 추론에 앞서 활성화 패턴을 관찰해 수행하고, 이상적인 양자화 체계를 계산해 저장한다.
INT8과 FP32간의 전환을 피하고 계산 속도를 높일 수 있다.

3. 양자화를 고려한 훈련 : INT8 을 사용하는 대신 FP32를 반올림해 양자화 효과를 흉내낸다.
정방향 패스와 역방향 패스에서 모두 적용되며, 정적 양자화와 동적 양자화를 사용해 모델 성능을 향상시킨다.

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

```
Model size (MB) - 132.40
Average latency (ms) - 12.54 +\- 0.73
Accuracy on test set - 0.876
```



# ONNX, ONNX 런타임

ONNX는 다양한 프레임 워크에서 딥러닝 모델을 나타내기 위해 공통연산자와 공통 파일 포맷을 정의하는 공개 표준이다.

ONNX는 표준 연산자와 데이터 타입으로 그래프를 표현해, 프레임워크간 전환을 쉽게한다.
ORT는 연산자 융합과 상수폴딩같은 ONNX그래프를 최적화하는 도구를 제공한다.
여러 하드웨어에서 모델을 실행하도록 실행 공급자를 위한 인터페이스를 정의한다.

ORT를 실행하려면 정제된 모델을 ONNX포맷으로 변환해야 한다.

1. 하나의 파이프라인으로 모델을 초기화한다
2. ONNX가 계산 그래프를 기록하도록 플레이스홀더 입력으로 파이프라인을 실행한다.
3. 동적 시퀀스 길이를 처리하기 위해 동적인 축을 정의한다.
4. 네트워크 파라미터와 함께 이 그래프를 저장한다.

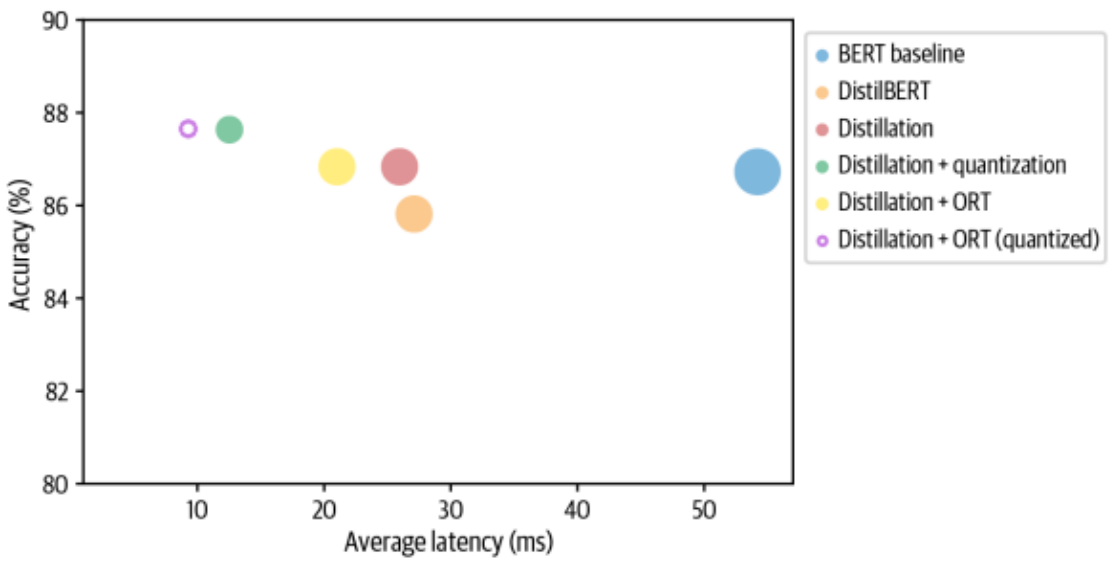
트랜스포머에서는 convert_graph_to_onnx.convert() 함수를 써서 이 과정을 간단하게 처리할 수 있다.

```
from transformers.convert_graph_to_onnx import convert
from onnxruntime import (GraphOptimizationLevel, InferenceSession,
                         SessionOptions)

model_ckpt = "distilbert-base-uncased-distilled-clinc"
onnx_model_path = Path("onnx/model.onnx")
convert(framework="pt", model=model_ckpt, tokenizer=tokenizer,
        output=onnx_model_path, opset=12, pipeline_name="text-classification")

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)
```

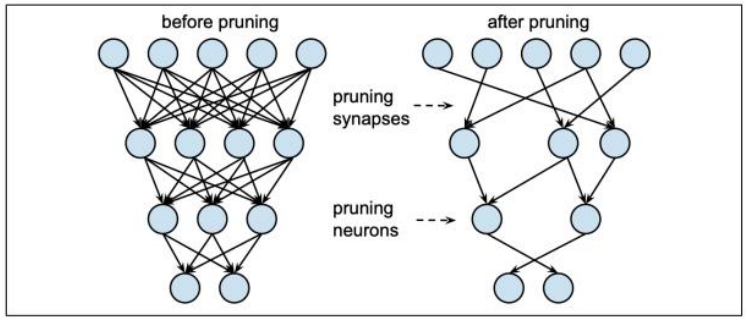
ONNX 모델이 text-classificataion 파이프라인과 호환되지 않아서 만든 새로운 파이프라인

```python
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]}]
```

```
Model size (MB) - 255.88
Average latency (ms) - 21.02 +\- 0.55
Accuracy on test set - 0.868

Model size (MB) - 64.20
Average latency (ms) - 9.24 +\- 0.29
Accuracy on test set - 0.877
```



# 가중치 가지치기




가지치기란 훈련하는 동안 가중치 연결을 점진적으로 제거해 모델을 희소하게 만드는 것이다.

$$\text{Top}_k(\mathbf{S})_{ij} = \begin{cases} 1 & \text{if } S_{ij} \text{ in top } k\,\% \\ 0 & \text{otherwise} \end{cases}$$

가중치 가지치기가 동작하는 방식은 중요도 점수 행렬 S 를 계산하고 상위 K% 의 가중치를 선택하는 것이다.

$$a_i = \sum_k W_{ik} M_{ik} x_k$$

# 절댓값 가지치기

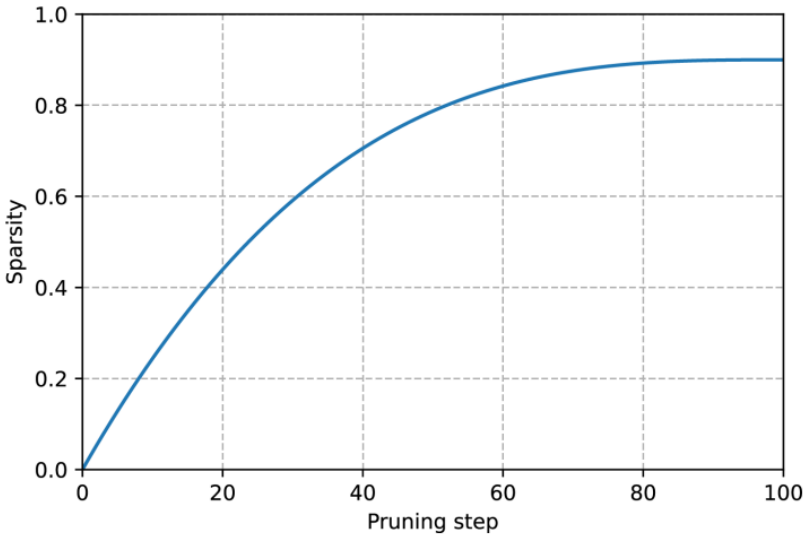
절댓값 가지치기는 가중치 절댓값 크기에 따라 점수 S 를 계산한다.

$$\mathbf{S} = \left( \mid W_{ij} \mid \right)_{1 \le j, j \le n}$$

그 다음 마스크를 만든다. $\mathbf{M} = \mathrm{Top}_k(\mathbf{S})$.

어떤 가중치가 중요한지 학습하도록 모델을 훈련하고 덜 중요한 가중치를 가지치기하는 식으로 적용한다.
원하는 희소성에 도달할 때 까지 희소한 모델을 훈련하고, 이것을 반복한다.

$$s_t = s_f + \left(s_i - s_f\right)\left(1 - \frac{t - t_0}{N\Delta t}\right)^3 \qquad \text{for } t \in \left\{t_0, t_0 + \Delta t, ..., t_0 + N\Delta t\right\}$$



# 이동 가지치기

이동 가지치기는 미세 튜닝하는 동안 점진적으로 가중치를 제거해 모델을 희소하게 만드는 것이다.
따라서 이동 가지치기는 점수를 가중치에서 바로 구하지 않고,
신경망의 다른 파라미터처럼 경사 하강법을 통해 학습한다.
이동 가지치기의 개념에 의하면 직관적으로 원점으로부터 가장 크게 이동하는 가중치가 가장 중요하다.