Giriş

Bu bölümde bir modeli kullanmak ve bir model yaratmak detaylı olarak incelenecektir. Herhangi bir checkpoint'ten bir model başlatmak için `AutoModel` sınıfı kullanılacaktır.

AutoModel ve benzer sınıflar özünde kütüphanedeki modeller için birer wrapper'dır. Bu wrapper'lar, verilen checkpoint'e göre modelin mimarisi için bir çıkarımda bulunabilirler ve bu mimaride bir model oluştururlar.

Fakat kullanılacak model tipi biliniyorsa mimariyi tanımlayan sınıf direkt olarak kullanılabilir. Yazının devamında, bütün bunların BERT için nasıl işlediği anlatılacaktır.

Transformer Oluşturmak

Bir BERT modeli oluşturmak için yapılması gereken ilk şey bir `config` objesi oluşturmaktır. Bu obje, modeli build ederken kullanılacak birçok özellik içerir.

In [ ]:
```
from transformers import BertConfig, BertModel

# Building the config
config = BertConfig()

# Building the model from the config
model = BertModel(config)
print(config)
```

```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.26.1",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

Yukarıdaki bütün özellikler henüz anlatılmadı fakat bazılarını hatırlamakta fayda var. `hidden_size`, hidden state vektörünün boyutunu tanımlar. `num_hidden_layers` parametresi, transformer modelinin sahip olduğu layer sayısını tanımlar.

Different loading methods

Default configuration ile model üretmek, modeli random değerler ile başlatır. Model bu şekilde kullanılabilir ama çıktısı hiçbir anlama gelmeyecektirir, bu yüzden eğitilmesi lazımdır. Ancak daha önce tartışıldığı gibi, bu fazlaca efor demektir. Bu yüzden önceden eğitilmiş modellerden faydalanmak gerekir.

In [ ]:
```
from transformers import BertConfig, BertModel
```

```
config = BertConfig()
model = BertModel(config)

# Model is randomly initialized!
```

Daha önceden eğitilmiş bir modeli yüklemek oldukça kolaydır:

In [ ]:
```
from transformers import BertModel
model = BertModel.from_pretrained("bert-base-cased")
```

```
Some weights of the model checkpoint at bert-base-cased were not used when initializing
BertModel: ['cls.predictions.bias', 'cls.seq_relationship.weight', 'cls.predictions.tran
sform.LayerNorm.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationsh
ip.bias', 'cls.predictions.decoder.weight', 'cls.predictions.transform.dense.weight', 'c
ls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model trai
ned on another task or with another architecture (e.g. initializing a BertForSequenceCla
ssification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model
that you expect to be exactly identical (initializing a BertForSequenceClassification mo
del from a BertForSequenceClassification model).
```

Daha önceden gördüğümüz gibi, BertModel ve AutoModel ile yer değiştirebilir. Checkpoint agnostik kod üretebilmek için bundan sonra AutoModel ile devam edeceğiz. Bu sayede eğer kod bir checkpoint için çalışıyorsa, diğer checkpointler için de çalışacaktır. Dahası, farklı mimariler için bile çalışmaya devam edecektir (task aynı olduğu sürece).

Yukarıdaki modelde, BertConfig sınıfını kullanmadık. Bunun yerine pretrained modeli yükledik. Bu model checkpoint'i BERT author'ları tarafından oluşturulmuştur ve detayları model card sayfasında bulunabilir.

Yukarıdaki modelde, checkpoint'in ağırlıklarını yüklemiştir bu yüzden kullanıma hazırdır; bir train işlemine gerek yoktur. Fakat istenirse fine-tuning de mümkündür.

Model ağırlıkları indirilip `~/.cache/huggingface/transformers` yolunda saklanır. Bu yüzden modeli sonradan yüklemek istediğinizde süreç daha hızlı sonlanacaktır. Bu yolu değiştirmek kolaydır, bunun için dökümantasyonu inceleyebilirsiniz.

Saving models

Model kaydetmek, yüklemek kadar basittir ve `model.save_pretrained()` metodu ile gerçekleştirilebilir. Bu işlem sonucunda iki adet dosya diskinize kaydedilecektir:

- config.json
- pytorch_model.bin

`config.json`, mimariyi inşa etmek için gerekli bilgileri ve metadata içerir. `pytorch_model.bin`, modelin ağırlıklarını saklar, `state_dictionary` olarak da bilinir.

Using a Transformer model for inference

Bu aşamada modeli kullanmaya başlayabiliriz. Fakat transformer modelleri sadece nümerik girdiler ile çalışabilir (tokenizer tarafından üretilen nümerik girdiler). Tokenizer'lar hakkında calısmaya başlamadan önce modelin kabul ettiği girdileri inceleyelim.

Tokenizer'lar, modelin üzerinde çalıştığı backend'e göre inputları farklı tiplere otomatik olarak cast edebilir. Örneğin elimizde aşağıdaki sequence'lar olsun:

```
sequences = ["Hello!", "Cool.", "Nice!"]
```
Encode edilmiş çıktı şuna benzeyecektir:

```
encoded_sequences = [
    [101, 7592, 999, 102],
```

```
    [101, 4658, 1012, 102],
    [101, 3835, 999, 102],
]
```

Bu listelerden oluşan bir listedir. Tensör'ler sadece dikdörtgen şekilde olabilir, matrisler gibi. Elimizdeki encoded_sequence uygun formda olduğundan bunu bir tensor'e dönüştürmek kolaydır:

```
import torch

model_inputs = torch.tensor(encoded_sequences)
```

Üretilen bu tensor'leri modeller ile kullanmak oldukça kolaydır. Aşağıdaki kod modelin çıktılarını üretmek için kullanılabilir:

```
output = model(model_inputs)
```

Model birçok argüman kabul edebilir ancak sadece ID'ler gereklidir. Alabileceği argümanlar ve görevleri ileride tartışılacaktır. İlerlemeden önce tokenizer'lerin detaylı bir şekilde incelenmesinde fayda vardır.