Training and fine-tuning

Model classes in 🤗 Transformers are designed to be compatible with native PyTorch and TensorFlow 2 and can be used seemlessly with either. In this quickstart, we will show how to fine-tune (or train from scratch) a model using the standard training tools available in either framework. We will also show how to use our included `Trainer()` class which handles much of the complexity of training for you.

This guide assume that you are already familiar with loading and use our models for inference; otherwise, see the task summary. We also assume that you are familiar with training deep neural networks in either PyTorch or TF2, and focus specifically on the nuances and tools for training models in 🤗 Transformers.

Sections:

Fine-tuning in native PyTorch

Model classes in 🤗 Transformers that don't begin with `TF` are PyTorch Modules, meaning that you can use them just as you would any model in PyTorch for both inference and optimization.

Let's consider the common task of fine-tuning a masked language model like BERT on a sequence classification dataset. When we instantiate a model with `from_pretrained()`, the model configuration and pre-trained weights of the specified model are used to initialize the model. The library also includes a number of task-specific final layers or 'heads' whose weights are instantiated randomly when not present in the specified pre-trained model. For example, instantiating a model with `BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)` will create a BERT model instance with encoder weights copied from the `bert-base-uncased` model and a randomly initialized sequence classification head on top of the encoder with an output size of 2. Models are initialized in `eval` mode by default. We can call `model.train()` to put it in train mode.

```
from transformers import BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', return_dict=True)
model.train()
```

This is useful because it allows us to make use of the pre-trained BERT encoder and easily train it on whatever sequence classification dataset we choose. We can use any PyTorch optimizer, but our library also provides the `AdamW()` optimizer which implements gradient bias correction as well as weight decay.

```
from transformers import AdamW
optimizer = AdamW(model.parameters(), lr=1e-5)
```

The optimizer allows us to apply different hyperpameters for specific parameter groups. For example, we can apply weight decay to all parameters other than bias and layer normalization terms:

```
no_decay = ['bias', 'LayerNorm.weight']
optimizer_grouped_parameters = [
    {'params': [p for n, p in model.named_parameters() if not any(nd in n for nd in no_decay)], 'wei
    {'params': [p for n, p in model.named_parameters() if any(nd in n for nd in no_decay)], 'weight_
]
optimizer = AdamW(optimizer_grouped_parameters, lr=1e-5)
```

Now we can set up a simple dummy training batch using `__call__()`. This returns a `BatchEncoding()` instance which prepares everything we might need to pass to the model.

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
text_batch = ["I love Pixar.", "I don't care for Pixar."]
encoding = tokenizer(text_batch, return_tensors='pt', padding=True, truncation=True)
input_ids = encoding['input_ids']
attention_mask = encoding['attention_mask']
```

When we call a classification model with the `labels` argument, the first returned element is the Cross Entropy loss between the predictions and the passed labels. Having already set up our optimizer, we can then do a backwards pass and update the weights:

```
labels = torch.tensor([1,0]).unsqueeze(0)
outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
loss = outputs.loss
loss.backward()
optimizer.step()
```

Alternatively, you can just get the logits and calculate the loss yourself. The following is equivalent to the previous example:

```
from torch.nn import functional as F
labels = torch.tensor([1,0]).unsqueeze(0)
outputs = model(input_ids, attention_mask=attention_mask)
loss = F.cross_entropy(labels, outputs.logitd)
loss.backward()
optimizer.step()
```

Of course, you can train on GPU by calling `to('cuda')` on the model and inputs as usual.

We also provide a few learning rate scheduling tools. With the following, we can set up a scheduler which warms up for `num_warmup_steps` and then linearly decays to 0 by the end of training.

```
from transformers import get_linear_schedule_with_warmup
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps, num_train_steps)
```

Then all we have to do is call `scheduler.step()` after `optimizer.step()`.

```
loss.backward()
optimizer.step()
scheduler.step()
```

We highly recommend using `Trainer()`, discussed below, which conveniently handles the moving parts of training 🤗 Transformers models with features like mixed precision and easy tensorboard logging.

Freezing the encoder

In some cases, you might be interested in keeping the weights of the pre-trained encoder frozen and optimizing only the weights of the head layers. To do so, simply set the `requires_grad` attribute to `False` on the encoder parameters, which can be accessed with the `base_model` submodule on any task-specific model in the library:

```
for param in model.base_model.parameters():
    param.requires_grad = False
```

Fine-tuning in native TensorFlow 2

Models can also be trained natively in TensorFlow 2. Just as with PyTorch, TensorFlow models can be instantiated with `from_pretrained()` to load the weights of the encoder from a pretrained model.

```
from transformers import TFBertForSequenceClassification
model = TFBertForSequenceClassification.from_pretrained('bert-base-uncased')
```

Let's use `tensorflow_datasets` to load in the MRPC dataset from GLUE. We can then use our built-in `glue_convert_examples_to_features()` to tokenize MRPC and convert it to a TensorFlow `Dataset` object. Note that tokenizers are framework-agnostic, so there is no need to prepend `TF` to the pretrained tokenizer name.

```
from transformers import BertTokenizer, glue_convert_examples_to_features
import tensorflow as tf
import tensorflow_datasets as tfds
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
data = tfds.load('glue/mrpc')
train_dataset = glue_convert_examples_to_features(data['train'], tokenizer, max_length=128, task='mr
train_dataset = train_dataset.shuffle(100).batch(32).repeat(2)
```

The model can then be compiled and trained as any Keras model:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer=optimizer, loss=loss)
model.fit(train_dataset, epochs=2, steps_per_epoch=115)
```

With the tight interoperability between TensorFlow and PyTorch models, you can even save the model and then reload it as a PyTorch model (or vice-versa):

```
from transformers import BertForSequenceClassification
model.save_pretrained('./my_mrpc_model/')
pytorch_model = BertForSequenceClassification.from_pretrained('./my_mrpc_model/', from_tf=True)
```

Trainer

We also provide a simple but feature-complete training and evaluation interface through `Trainer()` and `TFTrainer()`. You can train, fine-tune, and evaluate any 🤗 Transformers model with a wide range of training options and with built-in features like logging, gradient accumulation, and mixed precision.

PyTorch TensorFlow

```
from transformers import BertForSequenceClassification, Trainer, TrainingArguments

model = BertForSequenceClassification.from_pretrained("bert-large-uncased")

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total # of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
)

trainer = Trainer(
    model=model,                         # the instantiated 🤗 Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=test_dataset            # evaluation dataset
)
```

Now simply call `trainer.train()` to train and `trainer.evaluate()` to evaluate. You can use your own module as well, but the first argument returned from `forward` must be the loss which you wish to optimize.

`Trainer()` uses a built-in default function to collate batches and prepare them to be fed into the model. If needed, you can also use the `data_collator` argument to pass your own collator function which takes in the data in the format provided by your dataset and returns a batch ready to be fed into the model. Note that `TFTrainer()` expects the passed datasets to be dataset objects from `tensorflow_datasets`.

To calculate additional metrics in addition to the loss, you can also define your own `compute_metrics` function and pass it to the trainer.

```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average='binary')
    acc = accuracy_score(labels, preds)
    return {
        'accuracy': acc,
        'f1': f1,
        'precision': precision,
        'recall': recall
    }
```

Finally, you can view the results, including any calculated metrics, by launching tensorboard in your specified `logging_dir` directory.

Additional resources

- A lightweight colab demo which uses `Trainer` for IMDb sentiment classification.
- 🤗 Transformers Examples including scripts for training and fine-tuning on GLUE, SQuAD, and several other tasks.
- How to train a language model, a detailed colab notebook which uses `Trainer` to train a masked language model from scratch on Esperanto.
- 🤗 Transformers Notebooks which contain dozens of example notebooks from the community for training and using 🤗 Transformers on a variety of tasks.