Transfer Learning

March 4th, 2021
UW DATA 598
Transfer Learning in Natural Language Processing

June 2, 2019
NAACL-HLT 2019

Slides adapted from…

Sebastian Ruder
Matthew Peters
Swabha Swayamdipta
Thomas Wolf
What is transfer learning?

(a) Traditional Machine Learning

(b) Transfer Learning

Pan and Yang (2010)
Focus: Natural Language Processing

Goal: provide broad overview of methods in transfer learning
   focusing on the most empirically successful methods *in NLP (as of 2019)*

Demo:
   Transfer learning from language model to a text classification task in NLP
Why transfer learning?
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
- Many tasks share common knowledge about data
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
- Many tasks share common knowledge about data
  - In NLP, tasks share linguistic representations, structural similarities, etc.
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
- Many tasks share common knowledge about data
  - In NLP, tasks share linguistic representations, structural similarities, etc.
- Tasks can inform each other
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
- Many tasks share common knowledge about data
  - In NLP, tasks share linguistic representations, structural similarities, etc.
- Tasks can inform each other
  - NLP: semantics are shared in tasks such as QA, sentiment classification etc.
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
- Many tasks share common knowledge about data
  - In NLP, tasks share linguistic representations, structural similarities, etc.
- Tasks can inform each other
  - NLP: semantics are shared in tasks such as QA, sentiment classification etc.
- Empirically, transfer learning has resulted in state-of-the-art performance
Why transfer learning?

- Annotated data is rare, make use of as much supervision as available.
- Many tasks share common knowledge about data
  - In NLP, tasks share linguistic representations, structural similarities, etc.
- Tasks can inform each other
  - NLP: semantics are shared in tasks such as QA, sentiment classification etc.
- Empirically, transfer learning has resulted in state-of-the-art performance
  - for many supervised NLP tasks (e.g. classification, information extraction, Q&A, etc.)
Why transfer learning (in NLP)? Empirically...

Performance on Named Entity Recognition (NER) on CoNLL-2003 (English) over time

- Florian et al., 2003: 88.76
- Ando and Zhang, 2005 co- and self-supervision: 89.31
- Collobert et al. 2011: 89.59
- Passos et al. 2014: 90.05
- Chiu and Nichols 2015: 90.69
- LSTM-CRF: 90.94
- LM-LSTM-CRF: 91.24
- Yang et al.: 91.26
- BERT Base: 92.4
- BERT Large: 92.8
- Cross-view + Multi-Task: 92.61
- Flair embeddings: 93.09
- BILSTM-CRF + ELMo: 92.22
- CNN Large + fine-tune: 93.5
Types of transfer learning in NLP

- **Transductive transfer learning**
  - Same task; labeled data only in source domain
  - Different domains

- **Inductive transfer learning**
  - Different tasks; labeled data in target domain
  - Tasks learned sequentially

- **Multi-task learning**
  - Tasks learned simultaneously

- **Cross-lingual learning**
  - Different languages

- **Sequential transfer learning**

---

Ruder (2019)
Types of transfer learning in NLP

- **Transfer learning**
  - **Transductive transfer learning**
    - Same task; labeled data only in source domain
    - Different domains
  - **Inductive transfer learning**
    - Different tasks; labeled data in target domain
    - Tasks learned simultaneously
  - **Sequential transfer learning**
    - Tasks learned sequentially
  - **Multi-task learning**
    - Tasks learned simultaneously
  - **Cross-lingual learning**
    - Different languages
  - **Domain adaptation**
    - Different domains

**We will focus on this**

Ruder (2019)
Agenda

[1] Introduction

[2] Pretraining

[3] What’s in a representation?

[4] Adaptation
1. Introduction
Sequential transfer learning

Learn on one task / dataset, then transfer to another task / dataset

- word2vec
- GloVe
- skip-thought
- InferSent
- ELMo
- ULMFiT
- GPT
- BERT

Pretraining → Adaptation

- classification
- sequence labeling
- Q&A
- ....
Pretraining tasks and datasets

- Unlabeled data and self-supervision
- Supervised pretraining
Pretraining tasks and datasets

- Unlabeled data and self-supervision
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.

- Supervised pretraining
Pretraining tasks and datasets

- Unlabeled data and self-supervision
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
  - Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model

- Supervised pretraining
Unlabeled data and self-supervision

- Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
- Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model
- Focus on efficient algorithms to make use of plentiful data

Supervised pretraining
Pretraining tasks and datasets

- **Unlabeled data and self-supervision**
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
  - Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model
  - Focus on efficient algorithms to make use of plentiful data

- **Supervised pretraining**
  - Very common in vision, less in NLP due to lack of large supervised datasets
Pretraining tasks and datasets

- Unlabeled data and self-supervision
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
  - Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model
  - Focus on efficient algorithms to make use of plentiful data

- Supervised pretraining
  - Very common in vision, less in NLP due to lack of large supervised datasets
  - Machine translation
Pretraining tasks and datasets

- **Unlabeled data and self-supervision**
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
  - Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model
  - Focus on efficient algorithms to make use of plentiful data

- **Supervised pretraining**
  - Very common in vision, less in NLP due to lack of large supervised datasets
  - Machine translation
  - NLI for sentence representations
Pretraining tasks and datasets

- **Unlabeled data and self-supervision**
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
  - Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model
  - Focus on efficient algorithms to make use of plentiful data

- **Supervised pretraining**
  - Very common in vision, less in NLP due to lack of large supervised datasets
  - Machine translation
  - NLI for sentence representations
  - Task-specific—transfer from one Q&A dataset to another
Concrete example—word vectors

Word embedding methods (e.g. word2vec) learn one vector per word:

\[
\text{cat} = [0.1, -0.2, 0.4, \ldots] \\
\text{dog} = [0.2, -0.1, 0.7, \ldots]
\]
Concrete example—word vectors

Word embedding methods (e.g. word2vec) learn one vector per word:

cat = [0.1, -0.2, 0.4, ...]
dog = [0.2, -0.1, 0.7, ...]
Concrete example—word vectors

Word embedding methods (e.g. word2vec) learn one vector per word:

cat = [0.1, -0.2, 0.4, ...]
dog = [0.2, -0.1, 0.7, ...]

I love my cat and dog . }-> “positive"
Major Themes
Major themes: From words to words-in-context

Word vectors

cats = [0.2, -0.3, ...]
dogs = [0.4, -0.5, ...]
Major themes: From words to words-in-context

Word vectors

cats = [0.2, -0.3, …]
dogs = [0.4, -0.5, …]

Sentence / doc vectors

We have two cats.

It’s raining cats and dogs.

[0.8, 0.9, …]
Major themes: From words to words-in-context

Word vectors

cats = [0.2, -0.3, …]
dogs = [0.4, -0.5, …]

Sentence / doc vectors

We have two cats.
[0.8, 0.9, …]

It’s raining cats and dogs.
[-1.2, 0.0, …]

Word-in-context vectors

We have two cats.
[1.2, -0.3, …]

It’s raining cats and dogs.
[0.8, 0.9, …]
[-0.4, 0.9, …]
Major themes: LM pretraining
Many successful pretraining approaches are based on language modeling.
Many successful pretraining approaches are based on language modeling. Informally, a LM learns $P_{θ}(text)$ or $P_{θ}(text | some other text)$.
Many successful pretraining approaches are based on language modeling.

Informally, a LM learns $P_θ(text)$ or $P_θ(text \mid some\ other\ text)$.

Doesn’t require human annotation.
Major themes: LM pretraining

- Many successful pretraining approaches are based on language modeling
- Informally, a LM learns $P_e(\text{text})$ or $P_e(\text{text} \mid \text{some other text})$
- Doesn’t require human annotation
- Many languages have enough text to learn high capacity model
Major themes: LM pretraining

- Many successful pretraining approaches are based on language modeling
- Informally, a LM learns \( P_\theta(text) \) or \( P_\theta(text \mid \text{some other text}) \)
- Doesn’t require human annotation
- Many languages have enough text to learn high capacity model
- Versatile—can learn both sentence and word representations with a variety of objective functions
Major themes: pretraining vs target task

Choice of pretraining and target tasks are coupled
Choice of pretraining and target tasks are coupled

- Sentence / document representations not useful for word level predictions
Choice of pretraining and target tasks are coupled

- Sentence / document representations not useful for word level predictions
- Word vectors can be pooled across contexts, but often outperformed by other methods.
Major themes: pretraining vs target task

Choice of pretraining and target tasks are coupled

- Sentence / document representations not useful for word level predictions
- Word vectors can be pooled across contexts, but often outperformed by other methods
- In contextual word vectors, bidirectional context important
Major themes: pretraining vs target task

Choice of pretraining and target tasks are coupled

- Sentence / document representations not useful for word level predictions
- Word vectors can be pooled across contexts, but often outperformed by other methods
- In contextual word vectors, bidirectional context important

In general:

- Similar pretraining and target tasks $\rightarrow$ best results
Agenda

[1] Introduction

[2] Pretraining


[4] Adaptation
2. Pretraining
Overview

- Language model pretraining
- Word vectors (types)
- Contextual word vectors (tokens)
- Self-supervised and Supervised pretraining
We have three dogs.
LM pretraining

word2vec, Mikolov et al (2013)

ELMo, Peters et al. 2018, ULMFiT (Howard & Ruder 2018), GPT (Radford et al. 2018)
We have a MASK and three dogs

LM pretraining

word2vec, Mikolov et al (2013)

ELMo, Peters et al. 2018, ULMFiT (Howard & Ruder 2018), GPT (Radford et al. 2018)

BERT, Devlin et al 2019
Word vectors
Why embed words?
Why embed words?

- Embeddings are themselves parameters—can be learned
Why embed words?

- Embeddings are themselves parameters—can be learned
- Sharing representations across tasks
Why embed words?

- Embeddings are themselves parameters—can be learned
- Sharing representations across tasks
- Lower dimensional space
  - Better for computation—difficult to handle sparse vectors.
word2vec

Efficient algorithm + large scale training $\rightarrow$ high quality word vectors

(Mikolov et al., 2013)

See also:
- Pennington et al. (2014): GloVe
- Bojanowski et al. (2017): fastText
Contextual word vectors
Contextual word vectors - Motivation

Word vectors compress all contexts into a \textit{single vector}.

Nearest neighbor GloVe vectors to “\textit{play}”

<table>
<thead>
<tr>
<th>VERB</th>
<th>NOUN</th>
<th>ADJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>playing</td>
<td>game</td>
<td>multiplayer</td>
</tr>
<tr>
<td>played</td>
<td>games</td>
<td></td>
</tr>
<tr>
<td></td>
<td>players</td>
<td></td>
</tr>
<tr>
<td></td>
<td>football</td>
<td></td>
</tr>
</tbody>
</table>
Contextual word vectors - Motivation

Word vectors compress all contexts into a single vector

Nearest neighbor GloVe vectors to “play”

**VERB**
- playing
- played

**NOUN**
- game
- games
- players
- football

**ADJ**
- multiplayer

??
- play
- (theatrical)
- Play
Contextual word vectors - Key Idea

✧ Instead of learning one vector per word type, learn a vector that depends on context

\[ f(\text{play} \mid \text{The kids play a game in the park.}) \]

\[ \neq \]

\[ f(\text{play} \mid \text{The Broadway play premiered yesterday.}) \]

✧ Many approaches based on language models.

✧ We’ll only look at a few.
Pretraining Tasks
Supervised Pretraining: CoVe

(McCann et al., NeurIPS 2017)
Supervised Pretraining: CoVe

Pretrain bidirectional encoder with MT supervision, extract LSTM states

(McCann et al, NeurIPS 2017)
Supervised Pretraining: CoVe

Pretrain bidirectional encoder with MT supervision, extract LSTM states

Adding CoVe with GloVe gives improvements for classification, NLI, Q&A

(McCann et al, NeurIPS 2017)
Self-supervised Pretraining: GPT

(Radford et al., 2018)
Self-supervised Pretraining: GPT

Pretrain large 12-layer left-to-right Transformer Language Model.

(Radford et al., 2018)
Self-supervised Pretraining: GPT

Pretrain large 12-layer **left-to-right** Transformer Language Model.

[More on Transformers in coming slides]

(Radford et al., 2018)
Self-supervised Pretraining: GPT

Pretrain large 12-layer left-to-right Transformer Language Model.

[More on Transformers in coming slides]

Finetuning for sentence classification, sentence pair classification and multiple choice question- answer classification gave state-of-the-art results for 9 tasks.

(Radford et al., 2018)

<table>
<thead>
<tr>
<th>Method</th>
<th>MNLI-m</th>
<th>MNLI-mm</th>
<th>SNLI</th>
<th>SciTail</th>
<th>QNLI</th>
<th>RTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESIM + ELMo [44] (5x)</td>
<td>-</td>
<td>-</td>
<td>89.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CAFE [58] (5x)</td>
<td>80.2</td>
<td>79.0</td>
<td>89.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stochastic Answer Network [35] (3x)</td>
<td>80.6</td>
<td>80.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CAFE [58]</td>
<td>78.7</td>
<td>77.9</td>
<td>88.5</td>
<td>83.3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GenSen [64]</td>
<td>71.4</td>
<td>71.3</td>
<td>-</td>
<td>-</td>
<td>82.3</td>
<td>59.2</td>
</tr>
<tr>
<td>Multi-task BiLSTM + Attn [64]</td>
<td>72.2</td>
<td>72.1</td>
<td>-</td>
<td>-</td>
<td>82.1</td>
<td>61.7</td>
</tr>
<tr>
<td>Finetuned Transformer LM (ours)</td>
<td>82.1</td>
<td>81.4</td>
<td>89.9</td>
<td>88.3</td>
<td>88.1</td>
<td>56.0</td>
</tr>
</tbody>
</table>
Self-supervised Pretraining: GPT

Pretrain large 12-layer left-to-right Transformer Language Model.

[More on Transformers in coming slides]

Finetuning for sentence classification, sentence pair classification and multiple choice question-answer classification gave state-of-the-art results for 9 tasks.

More variants of GPT: 2 and 3!

(Radford et al., 2018)
Self-supervised Pretraining: BERT

BERT pretrains both sentence and contextual word representations, using **masked LM** and **next sentence prediction**. BERT-large has 340M parameters, 24 layers!

(Devlin et al. 2019)
Self-supervised Pretraining: BERT

BERT pretrains both sentence and contextual word representations, using masked LM and next sentence prediction. BERT-large has 340M parameters, 24 layers!

(Devlin et al. 2019)
Why does language modeling work so well?
Language modeling is a very difficult task, even for humans.
Why does language modeling work so well?

- Language modeling is a very difficult task, even for humans.
- Language models are expected to compress any possible context into a vector that generalizes over possible completions.
Why does language modeling work so well?

- Language modeling is a very difficult task, even for humans.
- Language models are expected to compress any possible context into a vector that generalizes over possible completions.
  - “They walked down the street to ???”
Why does language modeling work so well?

- Language modeling is a very difficult task, even for humans.
- Language models are expected to compress any possible context into a vector that generalizes over possible completions.
  - “They walked down the street to ????”
- To have any chance at solving this task, a model is forced to learn syntax, semantics, encode facts about the world, etc.
Why does language modeling work so well?

- Language modeling is a very difficult task, even for humans.
- Language models are expected to compress any possible context into a vector that generalizes over possible completions.
  - “They walked down the street to ???”
- To have any chance at solving this task, a model is forced to learn syntax, semantics, encode facts about the world, etc.
- Given enough data, a huge model, and enough compute, can do a reasonable job!
Why does language modeling work so well?

- Language modeling is a very difficult task, even for humans.
- Language models are expected to compress any possible context into a vector that generalizes over possible completions.
  - “They walked down the street to ???”
- To have any chance at solving this task, a model is forced to learn syntax, semantics, encode facts about the world, etc.
- Given enough data, a huge model, and enough compute, can do a reasonable job!
- Empirically works better than translation: “Language Modeling Teaches You More Syntax than Translation Does” (Zhang et al. 2018)
Hands-on #1:
Pretraining a Transformer Language Model
Hands-on: Overview
Hands-on: Overview

Current developments in Transfer Learning combine new approaches for training schemes (sequential training) as well as models (transformers) can look intimidating and complex.
Hands-on: Overview

Current developments in Transfer Learning combine new approaches for training schemes (sequential training) as well as models (transformers) → can look intimidating and complex.

Goals:

- Let’s make these recent works “uncool” i.e. as accessible as possible
- Expose all the details in a simple, concise and self-contained code-base
- Show that transfer learning can be simple (less hand-engineering) & fast (pretrained model)
Hands-on: Overview

Current developments in Transfer Learning combine new approaches for training schemes (sequential training) as well as models (transformers) \(\Rightarrow\) can look intimidating and complex

- **Goals:**
  - Let’s make these recent works “uncool” i.e. as accessible as possible
  - Expose all the details in a simple, concise and self-contained code-base
  - Show that transfer learning can be simple (less hand-engineering) & fast (pretrained model)

- **Plan**
  - Build a GPT-2 / BERT model
  - Pretrain it on a rather large corpus with ~100M words
  - Adapt it for a target task (question categorization) to get SOTA performances
Our core model will be a Transformer (Vaswani et al., 2017). Large-scale transformer architectures (GPT-2, BERT, XLM…) are very similar to each other and consist of:
Our core model will be a Transformer (Vaswani et al., 2017). Large-scale transformer architectures (GPT-2, BERT, XLM…) are very similar to each other and consist of:

- summing words and position embeddings
- applying a succession of transformer blocks with:
  - layer normalisation
  - a self-attention module
  - dropout and a residual connection
- another layer normalisation
- a feed-forward module with one hidden layer and a non-linearity: Linear $\Rightarrow$ Non-Linear Activation $\Rightarrow$ Linear
- dropout and a residual connection

Illustration from (Child et al., 2019)
Hands-on pre-training

Our core model will be a Transformer (Vaswani et al., 2017). Large-scale transformer architectures (GPT-2, BERT, XLM…) are very similar to each other and consist of:

- summing words and position embeddings
- applying a succession of transformer blocks with:
  - layer normalisation
  - a self-attention module
  - dropout and a residual connection
- another layer normalisation
- a feed-forward module with one hidden layer and a non-linearity: Linear ⇒ Non-Linear Activation ⇒ Linear
- dropout and a residual connection

Main differences between GPT/GPT-2/BERT are the objective functions:
- causal language modeling for GPT
- masked language modeling for BERT (+ next sentence prediction)

Illustration from (Child et al., 2019)
Hands-on pre-training

Let’s code the backbone of our model!

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()
        for _ in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                     nn.ReLU(),
                                                     nn.Linear(hidden_dim, embed_dim)))
            self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
            self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):
        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.triu(attn_mask, diagonal=1)
        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions,
                                                                        self.layer_norms_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)
            h = x + h
            h = layer_norm_2(h)
            x = feed_forward(h)
            h = x + h

        return h
```
Let’s code the backbone of our model!

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()

        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()

        for _ in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                     nn.ReLU(),
                                                     nn.Linear(hidden_dim, embed_dim)))
            self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
            self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):

        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.triu(attn_mask, diagonal=1)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions, self.layer_norms_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)
            h = x + h
            h = layer_norm_2(h)
            x = feed_forward(h)
            x = self.dropout(x)
            h = x + h

        return h
```
Hands-on pre-training

Let’s code the backbone of our model!

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norns_1, self.layer_norns_2 = nn.ModuleList(), nn.ModuleList()

        for n in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                     nn.ReLU(),
                                                     nn.Linear(hidden_dim, embed_dim)))

            self.layer_norns_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
            self.layer_norns_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):

        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.triu(attn_mask, diagonal=1)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norns_1, self.attentions,
                                                                       self.layer_norns_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)
            h = x + h

            h = layer_norm_2(h)
            x = feed_forward(h)
            x = self.dropout(x)
            h = x + h

        return h
```
Hands-on pre-training

Let’s code the backbone of our model!

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()
        for _ in range(num_layers):
            self.attentions.append(MultiheadAttention(embed_dim, num_heads, dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                     nn.ReLU(),
                                                     nn.Linear(hidden_dim, embed_dim)))
            self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
            self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):
        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.triu(attn_mask, diagonal=0)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions, self.layer_norms_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)
            h = x + h

            h = layer_norm_2(h)
            x = feed_forward(h)
            h = x + h

        return h
```
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()
        for _ in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, embed_dim)))

            self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
            self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):
        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.tril(attn_mask, diagonal=1)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions, self.layer_norms_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)
            h = x + h
            h = layer_norm_2(h)
            x = feed_forward(h)
            h = x + h

        return h
Hands-on pre-training

Two attention masks?

- **padding_mask** masks the padding tokens. It is specific to each sample in the batch:

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()

        for _ in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                     nn.ReLU(),
                                                     nn.Linear(hidden_dim, embed_dim)))

        self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
        self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):

        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.triu(attn_mask, diagonal=1)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions, self.layer_norms_2, self.feed_forwards):

            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)
            h = x + h

            h = layer_norm_2(h)
            x = feed_forward(h)
            x = self.dropout(x)
            h = x + h

        return h
```
Hands-on pre-training

Two attention masks?

- **padding_mask** masks the padding tokens. It is specific to each sample in the batch:

<table>
<thead>
<tr>
<th>I</th>
<th>love</th>
<th>Mom</th>
<th>s</th>
<th>cooking</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>love</td>
<td>you</td>
<td>too</td>
<td>!</td>
</tr>
<tr>
<td>No</td>
<td>way</td>
<td>This</td>
<td>the</td>
<td>shit</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **attn_mask** is the same for all samples in the batch. It masks the previous tokens for causal transformers:

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.tokens_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()

        for _ in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                     nn.ReLU(),
                                                     nn.Linear(hidden_dim, embed_dim)))

        self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
        self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):
        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)

        h = self.tokens_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions, self.layer_norms_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)

            x = feed_forward(x)
            h = x + h

        return h
```
To pretrain our model, we need to add a few elements: a head, a loss and initialize weights.

```python
class TransformerWithLMHead(nn.Module):
    def __init__(self, config):
        """ Transformer with a language modeling head on top (tied weights) ""
        super().__init__()
        self.config = config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings, config.num_max_positions, config.num_heads, config.num_layers, config.dropout, causal=not config.mlm)

        self.lm_head = nn.Linear(config.embed_dim, config.num_embeddings, bias=False)
        self.apply(self.init_weights)
        self.tie_weights()

    def tie_weights(self):
        self.lm_head.weight = self.transformer.tokens_embeddings.weight

    def init_weights(self, module):
        """ initialize weights - nn.MultiheadAttention is already initialized by PyTorch (xavier) ""
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, labels=None, padding_mask=None):
        """ x has shape [seq length, batch], padding_mask has shape [batch, seq length] ""
        hidden_states = self.transformer(x, padding_mask)
        logits = self.lm_head(hidden_states)

        if labels is not None:
            shift_logits = logits[:-1] if self.transformer.causal else logits
            shift_labels = labels[1:] if self.transformer.causal else labels
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
            return logits, loss

        return logits
```
To pretrain our model, we need to add a few elements: a head, a loss and initialize weights.

We add these elements with a pretraining model encapsulating our model.
Hands-on pre-training

To pretrain our model, we need to add a few elements: a **head**, a **loss** and initialize **weights**.

We add these elements with a pretraining model encapsulating our model.

1. **A pretraining head** on top of our core model: we choose a language modeling head with tied weights.

```python
class TransformerWithLMHead(nn.Module):
    def __init__(self, config):
        """ Transformer with a language modeling head on top (tied weights) ""
        super().__init__()
        self.config = config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings,
                                        config.num_max_positions, config.num_heads, config.num_layers,
                                        config.dropout, causal=not config.mlm)

        self.lm_head = nn.Linear(config.embed_dim, config.num_embeddings, bias=False)
        self.apply(self.init_weights)
        self.tie_weights()

    def tie_weights(self):
        self.lm_head.weight = self.transformer.tokens_embeddings.weight

    def init_weights(self, module):
        """ initialize weights - nn.MultiheadAttention is already initialized by PyTorch (xavier) ""
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if isinstance(module, nn.Linear) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, labels=None, padding_mask=None):
        """ x has shape [seq length, batch], padding_mask has shape [batch, seq length] ""
        hidden_states = self.transformer(x, padding_mask)
        logits = self.lm_head(hidden_states)

        if labels is not None:
            shift_logits = logits[:-1] if self.transformer.causal else logits
            shift_labels = labels[:-1] if self.transformer.causal else labels
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
            return logits, loss

        return logits
```
To pretrain our model, we need to add a few elements: a **head**, a **loss** and **initialize weights**.

We add these elements with a pretraining model encapsulating our model.

1. **A pretraining head** on top of our core model: we choose a language modeling head with tied weights.

2. **Initialize** the weights.
Hands-on pre-training

To pretrain our model, we need to add a few elements: a head, a loss and initialize weights.

We add these elements with a pretraining model encapsulating our model.

1. **A pretraining head** on top of our core model: we choose a language modeling head with tied weights

2. **Initialize** the weights

3. **Define a loss function**: we choose a cross-entropy loss on current (or next) token predictions
Hands-on pre-training

Now let’s take care of our data and configuration

```python
from pytorch_pretrained_bert import BertTokenizer, cached_path
tokenizer = BertTokenizer.from_pretrained('bert-base-cased', do_lower_case=False)

from collections import namedtuple

Config = namedtuple('Config',
    field_names="embed_dim, hidden_dim, num_max_positions, num_embeddings, num_heads, num_layers,"
    "dropout, initializer_range, batch_size, lr, max_norm, n_epochs, n_warmup,"
    "mlm, gradient_accumulation_steps, device, log_dir, dataset_cache")
args = Config(410, 2100, 256, len(tokenizer.vocab), 10, 16, 0.1, 0.02, 16, 2.5e-4, 1.0, 50, 1000, False, 4, "cuda" if torch.cuda.is_available() else "cpu", "/", "/dataset_cache.bin")

dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/wikitext-103/"
    "wikitext-103-train-tokenized-bert.bin")
datasets = torch.load(dataset_file)

# Convert our encoded dataset to torch.tensors and reshape in blocks of the transformer's input length
for split_name in ['train', 'valid']:
tensor = torch.tensor(datasets[split_name], dtype=torch.long)
num_sequences = (tensor.size(0) // args.num_max_positions) * args.num_max_positions
datasets[split_name] = tensor.narrow(0, 0, num_sequences).view(-1, args.num_max_positions)

model = TransformerWithLMPHead(args).to(args.device)
 optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```
We'll use a pre-defined open vocabulary tokenizer: BERT's model cased tokenizer.

Now let's take care of our data and configuration

```python
from pytorch_pretrained_bert import BertTokenizer, cached_path
tokenizer = BertTokenizer.from_pretrained('bert-base-cased', do_lower_case=False)
```

```python
from collections import namedtuple

Config = namedtuple('Config',
    field_names="embed_dim, hidden_dim, num_max_positions, num_embeddings       , num_heads, num_layers,"
    "dropout, initializer_range, batch_size, lr, max_norm, n_epochs, n_warmup,"
    "mlm, gradient_accumulation_steps, device, log_dir, dataset_cache"
)

args = Config(410, 2100, 256, len(tokenizer.vocab), 10, 16,
    0.1, 0.02, 16, len(tokenizer.vocab), 10, 50, 1000,
    False, 4, "cuda" if torch.cuda.is_available() else "cpu", "/" , "/dataset_cache.bin")
```

```python
dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/wikitext-103/"
    "wikitext-103-train-tokenized-bert.bin")
datasets = torch.load(dataset_file)

# Convert our encoded dataset to torch.tensors and reshape in blocks of the transformer's input length
for split_name in ['train', 'valid']:
tensor = torch.tensor(datasets[split_name], dtype=torch.long)
num_sequences = (tensor.size(0) // args.num_max_positions) * args.num_max_positions

datasets[split_name] = tensor.narrow(0, 0, num_sequences).view(-1, args.num_max_positions)
```

```python
model = TransformerWithLMHead(args.to(args.device)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```
Hands-on pre-training

We'll use a pre-defined open vocabulary tokenizer: BERT's model cased tokenizer.

Hyper-parameters taken from Dai et al., 2018 (Transformer-XL) $\Rightarrow$ ~50M parameters causal model.

Now let's take care of our data and configuration

```python
from pytorch_pretrained_bert import BertTokenizer, cached_path
tokenizer = BertTokenizer.from_pretrained('bert-base-cased', do_lower_case=False)
```

```python
from collections import namedtuple

Config = namedtuple('Config',
    field_names="embed_dim, hidden_dim, num_max_positions, num_embeddings
    , num_heads, num_layers,"
    "dropout, initializer_range, batch_size, lr, max_norm, n_epochs, n_warmup,"
    "mlm, gradient_accumulation_steps, device, log_dir, dataset_cache")
args = Config(410, 2100, 256, len(tokenizer.vocab), 10, 16, 0.1, 0.02, 16, 2.5e-4, 1.0, 50, 1000, False, 4, "cuda" if torch.cuda.is_available() else "cpu", "./", "./dataset_cache.bin")
```

```python
dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/wikitext-103/
    "wikitext-103-train-tokenized-bert.bin")
datasets = torch.load(dataset_file)

# Convert our encoded dataset to torch.tensors and reshape in blocks of the transformer's input length
for split_name in ['train', 'valid'];
tensor = torch.tensor(datasets[split_name], dtype=torch.long)
num_sequences = (tensor.size(0) // args.num_max_positions) * args.num_max_positions
datasets[split_name] = tensor.narrow(0, 0, num_sequences).view(-1, args.num_max_positions)
```

```python
model = TransformerWithLMHead(args).to(args.device)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```
Hands-on pre-training

We'll use a pre-defined open vocabulary tokenizer: BERT’s model cased tokenizer.

Hyper-parameters taken from Dai et al., 2018 (Transformer-XL) $\Rightarrow$ ~50M parameters causal model.

Now let’s take care of our data and configuration

```python
from pytorch_pretrained_bert import BertTokenizer, cached_path
tokenizer = BertTokenizer.from_pretrained( 'bert-base-cased', do_lower_case=False)
```

```python
from collections import namedtuple

Config = namedtuple('Config',
    field_names= "embed_dim, hidden_dim, num_max_positions, num_embeddings, num_heads, num_layers,"
    "dropout, initializer_range, batch_size, lr, max_norm, n_epochs, n_warmup,"
    "mlm, gradient_accumulation_steps, device, log_dir, dataset_cache")

args = Config(410, 2100, 256, len(tokenizer.vocab), 10, 16, 0.1, 0.02, 16, 2.5e-4, 1.0, 50, 1000, False, 4, "cuda" if torch.cuda.is_available() else "cpu", "/", "/dataset_cache.bin")
```

```python
dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/wikitext-103/"
                                 "wikitext-103-train-tokenized-bert.bin")

datasets = torch.load(dataset_file)

# Convert our encoded dataset to torch.tensors and reshape in blocks of the transformer's input length
for split_name in [ 'train', 'valid']:
tensor = torch.tensor(datasets[split_name], dtype=torch.long)
num_sequences = (tensor.size(0) // args.num_max_positions) * args.num_max_positions
datasets[split_name] = tensor.narrow(0, 0, num_sequences).view(-1, args.num_max_positions)
```

```python
model = TransformerWithLMPooler(args).to(args.device)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```
Hands-on pre-training

We'll use a pre-defined open vocabulary tokenizer: BERT’s model cased tokenizer.

Hyper-parameters taken from Dai et al., 2018 (Transformer-XL) ⇒ ~50M parameters causal model.

Now let’s take care of our data and configuration

```python
from pytorch_pretrained_bert import BertTokenizer, cached_path
tokenizer = BertTokenizer.from_pretrained('bert-base-cased', do_lower_case=False)
```

```python
from collections import namedtuple

Config = namedtuple('Config',
    field_names="embed_dim, hidden_dim, num_max_positions, num_embeddings",
    num_heads, num_layers,"
    "dropout, initializer_range, batch_size, lr, max_norm, n_epochs, n_warmup,
    "mlm, gradient_accumulation_steps, device, log_dir, dataset_cache")
args = Config(410, 2100, 256, len(tokenizer.vocab), 10, 0.1, 0.02, 16, 2.5e-4, 1.0, 50, 1000, False, 4, "cuda" if torch.cuda.is_available() else "cpu", "/", "/dataset_cache.bin")
```

```python
dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/wikitext-103/"
  
  "wikitext-103-train-tokenized-bert.bin")
datasets = torch.load(dataset_file)

# Convert our encoded dataset to torch.tensors and reshape in blocks of the transformer's input length for split_name in ['train', 'valid']:
tensor = torch.tensor(datasets[splith_name], dtype=torch.long)
num_sequences = (tensor.size(0) // args.num_max_positions) * args.num_max_positions
datasets[splith_name] = tensor.narrow(0, 0, num_sequences).view(-1, args.num_max_positions)
```

```python
model = TransformerWithLMHead(args).to(args.device)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```
Hands-on pre-training

And we’re done: let’s train!

```python
import os
from torch.utils.data import DataLoader
from ignite.engine import Engine, Events
from ignite.metrics import RunningAverage
from ignite.contrib.handlers import ModelCheckpoint
from ignite.contrib.handlers import CosineAnnealingScheduler, create_lr_scheduler_with_warmup, ProgressBar

dataloader = DataLoader(datasets['train'], batch_size=args.batch_size, shuffle=True)

def update(engine, batch):
    model.train()
    batch = batch.transpose(0, 1).contiguous().to(args.device)  # to shape [seq length, batch]
    logits, loss = model(batch, labels=batch)
    loss = loss / args.gradient_accumulation_steps
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), args.max_norm)
    if engine.state.iteration % args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()

trainer = Engine(update)

# Add progressbar with loss
RunningAverage(output_transform=lambda x: x).attach(trainer, 'loss')
ProgressBar(persist=True).attach(trainer, metric_names=['loss'])

# Learning rate scheduler: linearly warm-up to lr and then decrease the learning rate to zero with cosine
# scheduler = CosineAnnealingScheduler(optimizer, 'lr', args.lr, 0.0, len(dataloader) * args.n_epochs)
# scheduler = create_lr_scheduler_with_warmup(scheduler, 0.0, args.lr, args.n_epochs)
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Save checkpoints and training config
checkpoint_handler = ModelCheckpoint(args.log_dir, 'checkpoint', save_interval=1, n_saved=5)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, {'mymodel': model})
torch.save(args, os.path.join(args.log_dir, 'training_args.bin'))

trainer.run(train_dataloader, max_epochs=args.n_epochs)
```

***
Epoch [1/50] [365/28874] 1% | loss=2.30e+00 [03:43<5:42:22]
```
Hands-on pre-training

And we’re done: let’s train!

```python
import os
from torch.utils.data import DataLoader
from ignite.engine import Engine, Events
from ignite.metrics import RunningAverage
from ignite.contrib.handlers import ModelCheckpoint

# Define training function

def update(engine, batch):
    model.train()
    batch = batch.transpose(0, 1).contiguous().to(args.device)  # to shape [seq length, batch]
    logits, loss = model(batch, labels=batch)
    loss = loss / args.gradient_accumulation_steps
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), args.max_norm)
    if engine.state.iteration % args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()

trainer = Engine(update)

# Add progress bar with loss
RunningAverage(output_transform=lambda x: x).attach(trainer, "loss")
ProgressBar(persist=True).attach(trainer, metric_names=['loss'])

# Learning rate scheduler: linearly warm-up to lr and then decrease the learning rate to zero with cosine
# cos_scheduler = CosineAnnealingScheduler(optimizer, 'lr', args.lr, 0.0, len(dataloader) * args.n_epochs)
# scheduler = create_lr_scheduler_with_warmup(cos_scheduler, 0.0, args.lr, args.n_warmup)
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Save checkpoints and training config
checkpoint_handler = ModelCheckpoint(args.log_dir, 'checkpoint', save_interval=1, n_saved=5)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, {'mymodel': model})
torch.save(args, os.path.join(args.log_dir, 'training_args.bin'))

trainer.run(train_dataloader, max_epochs=args.n_epochs)
```

***

Epoch [1/50] | [365/28874] 1% |  loss=2.30e+00 [03:43<4:52:22]
Hands-on pre-training

And we’re done: let’s train!

```python
import os
from torch.utils.data import DataLoader
from ignite.engine import Engine, Events
from ignite.metrics import RunningAverage
from ignite.contrib.handlers import CosineAnnealingScheduler, create_lr_scheduler_with_warmup, ProgressBar

dataloader = DataLoader(datasets['train'], batch_size=args.batch_size, shuffle=True)

# Define training function
def update(engine, batch):
    model.train()
    batch = batch.transpose(0, 1).contiguous().to(args.device)  # to shape [seq length, batch]
    logits, loss = model(batch, labels=batch)
    loss = loss / args.gradient_accumulation_steps
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), args.max_norm)
    if engine.state.iteration % args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()

trainer = Engine(update)

# Add progressbar with loss
RunningAverage(output_transform=lambda x: x).attach(trainer, 'loss')
ProgressBar(persist=True).attach(trainer, metric_names=['loss'])

# Learning rate scheduler: linearly warm-up to lr and then decrease the learning rate to zero with cosine
# scheduler = create_lr_scheduler_with_warmup(optimizer, 'lr', args.lr, 0.0, len(dataloader) * args.n_epochs)
# trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Save checkpoints and training config
checkpoint_handler = ModelCheckpoint(args.log_dir, 'checkpoint', save_interval=1, n_saved=5)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, {'mymodel': model})
torch.save(args, os.path.join(args.log_dir, 'training_args.bin'))

trainer.run(train_dataloader, max_epochs=args.n_epochs)
```

Go!
On pretraining

- **Intensive**: in our case 5h–20h on 8 V100 GPUs (few days w. 1 V100) to reach a good perplexity
  ⇒ share your pretrained models
- **Robust to the choice of hyper-parameters** (apart from needing a warm-up for transformers)
- Language modeling is a hard task, your model should **not have enough capacity to overfit** if your dataset is large enough ⇒ you can just start the training and let it run.
- **Masked-language modeling**: typically 2-4 times slower to train than causal LM
  We only mask 15% of the tokens ⇒ smaller signal
First model:
- **exactly the one** we built together ⇒ a 50M parameters causal Transformer
- Trained 15h on 8 V100
- Reached a **word-level perplexity of 29** on wikitext-103 validation set (quite competitive)

Second model:
- Same model but trained with a **masked-language modeling** objective (see the repo)
- Trained 30h on 8 V100
- Reached a “masked-word” perplexity of 8.3 on wikitext-103 validation set
Agenda

[1] Introduction
[2] Pretraining
[4] Adaptation
3. What is in a Representation?
Why care about what is in a representation?
Why care about what is in a representation?

- Alternative to Extrinsic evaluation with downstream tasks
  - Complex, diverse with task-specific quirks
Why care about what is in a representation?

- Alternative to Extrinsic evaluation with downstream tasks
  - Complex, diverse with task-specific quirks

- Measures language-awareness of representations
  - To generalize to other tasks, new inputs
  - As intermediates for possible improvements to pretraining
Why care about what is in a representation?

- Alternative to Extrinsic evaluation with downstream tasks
  - Complex, diverse with task-specific quirks

- Measures language-awareness of representations
  - To generalize to other tasks, new inputs
  - As intermediates for possible improvements to pretraining

- Interpretability!
  - Are we getting our results because of the right reasons?
  - Uncovering biases...
What to analyze?

\[ \mathcal{L} \]

\[ T \]

\[ L_n \]

\[ L_1 \]

\[ E \]
What to analyze?

- Embeddings
  - Word
  - Contextualized

\[ \mathcal{L} \]

\[ T \]

\[ L_n \]

\[ L_1 \]

\[ E \]
What to analyze?

- Embeddings
  - Word
  - Contextualized
- Network Activations
What to analyze?

- Embeddings
  - Word
  - Contextualized

- Network Activations

- Alterations:
  - Architecture
    - (RNN / Transformer)
What to analyze?

- **Embeddings**
  - Word
  - Contextualized

- **Network Activations**

- **Alterations:**
  - Architecture
    - (RNN / Transformer)
  - Layers
What to analyze?

- Embeddings
  - Word
  - Contextualized

- Network Activations

- Alterations:
  - Architecture
    - (RNN / Transformer)
  - Layers
  - Pretraining Objectives
Analysis Method 1: Visualization

Hold the embeddings / network activations static or **frozen**
Plotting embeddings *faithfully* into a lower dimensional (2D/3D) space
- t-SNE [van der Maaten & Hinton, 2008](https://l2s.ens-cachan.fr/~vanessional/Publications/visualizing.pdf)
- PCA projections
Plotting embeddings **faithfully** into a lower dimensional (2D/3D) space
- t-SNE *van der Maaten & Hinton, 2008*
- PCA projections

Image: **Tensorflow**
Visualizing Embedding Geometries

- Plotting embeddings **faithfully** into a lower dimensional (2D/3D) space
  - t-SNE *van der Maaten & Hinton, 2008*
  - PCA projections

- Visualizing word analogies *Mikolov et al., 2013*
  - Spatial relations
  - $w_{\text{king}} - w_{\text{man}} + w_{\text{woman}} \sim w_{\text{queen}}$

*Pennington et al., 2014*
Plotting embeddings **faithfully** into a lower dimensional (2D/3D) space
- t-SNE *van der Maaten & Hinton, 2008*
- PCA projections

Visualizing word analogies *Mikolov et al., 2013*
- Spatial relations
  - $w_{\text{king}} - w_{\text{man}} + w_{\text{woman}} \sim w_{\text{queen}}$

High-level view of lexical semantics
- Only a limited number of examples
- Connection to other tasks is unclear *Goldberg, 2017*
Neuron activation values correlate with features / labels
Neuron activation values correlate with features / labels

Radford et al., 2017
Neuron activation values correlate with features / labels

Visualizing Neuron Activations

Radford et al., 2017

Karpathy et al., 2016
Visualizing Neuron Activations

- Neuron activation values correlate with features / labels

- Indicates learning of recognizable features
  - How to select which neuron? Hard to scale!
  - Interpretable ≠ Important (Morcos et al., 2018)

Cell that is sensitive to the depth of an expression:

```c
#define CONFIG_AUDITSYSCALL

static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
```

Figure: Distribution of neuron values for negative and positive reviews (Radford et al., 2017)

Figure: Code snippet for auditing a match class (Karpathy et al., 2016)
Popular in machine translation, or other seq2seq architectures:

- **Alignment** between words of source and target.
- Long-distance word-word **dependencies** (intra-sentence attention)

---

**Vaswani et al., 2017**
Visualizing Attention Weights

- Popular in machine translation, or other seq2seq architectures:
  - Alignment between words of source and target.
  - Long-distance word-word dependencies (intra-sentence attention)

- Sheds light on architectures

---

Vaswani et al., 2017
Visualizing Attention Weights

- Popular in machine translation, or other seq2seq architectures:
  - **Alignment** between words of source and target.
  - Long-distance word-word **dependencies** (intra-sentence attention)

- Sheds light on architectures
  - Having sophisticated attention mechanisms can be a good thing!
Popular in machine translation, or other seq2seq architectures:

- **Alignment** between words of source and target.
- Long-distance word-word **dependencies** (intra-sentence attention)

Sheds light on architectures

- Having sophisticated attention mechanisms can be a good thing!
- Layer-specific (layer 5 / layer 6 in fig.)
Popular in machine translation, or other seq2seq architectures:

- **Alignment** between words of source and target.
- Long-distance word-word **dependencies** (intra-sentence attention)

Sheds light on architectures

- Having sophisticated attention mechanisms can be a good thing!
- Layer-specific (layer 5 / layer 6 in fig.)

Interpretation can be tricky

- Few examples only - cherry picking?
- Robust **corpus-wide** trends? Next!

---

**Vaswani et al., 2017**
Analysis Method 2: Behavioral Probes

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - number agreement in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - number agreement in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences

Kuncoro et al., 2018

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - number agreement in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences
- RNNs outperform other non-neural baselines.

Kuncoro et al., 2018

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - *number agreement* in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences
- RNNs outperform other non-neural baselines.

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - **number agreement** in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences

- RNNs outperform other non-neural baselines.

- Performance improves when trained explicitly with syntax ([Kuncoro et al. 2018](#))

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - number agreement in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences
- RNNs outperform other non-neural baselines.
- Performance improves when trained explicitly with syntax (Kuncoro et al. 2018)

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018

Kuncoro et al. 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - number agreement in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences

- RNNs outperform other non-neural baselines.

- Performance improves when trained explicitly with syntax (Kuncoro et al. 2018)

- Probe: Might be vulnerable to co-occurrence biases

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - **number agreement** in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences

- RNNs outperform other non-neural baselines.

- Performance improves when trained explicitly with syntax ([Kuncoro et al. 2018](#))

- Probe: Might be vulnerable to co-occurrence biases
  - “dogs in the neighborhood bark(s)”

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
RNN-based language models (RNN-based)
- number agreement in subject-verb dependencies
- For natural and nonce/ungrammatical sentences
- LM perplexity differences

RNNs outperform other non-neural baselines.

Performance improves when trained explicitly with syntax (Kuncoro et al. 2018)

Probe: Might be vulnerable to co-occurrence biases
- “dogs in the neighborhood bark(s)”
- Nonce sentences might be too different from original...

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018

Kuncoro et al. 2018
Analysis Method 3: Classifier Probes

Hold the embeddings / network activations static and train a simple supervised model on top
Analysis Method 3: Classifier Probes

Hold the embeddings / network activations static and 

train a **simple supervised** model on top

Probe classification task (Linear / MLP)
Probing Surface-level Features

Zhang et al. 2018; Liu et al., 2018; Conneau et al., 2018
Probing Surface-level Features

- Given a sentence, predict properties such as:
  - Length
  - Is a word in the sentence?

- Given a word in a sentence predict properties such as:
  - Previously seen words, contrast with language model
  - Position of word in the sentence

Zhang et al. 2018; Liu et al., 2018; Conneau et al., 2018
Given a sentence, predict properties such as:
- Length
- Is a word in the sentence?

Given a word in a sentence predict properties such as:
- Previously seen words, contrast with language model
- Position of word in the sentence

Checks ability to memorize

Zhang et al. 2018; Liu et al., 2018; Conneau et al., 2018
Given a sentence, predict properties such as
- Length
- Is a word in the sentence?

Given a word in a sentence predict properties such as:
- Previously seen words, contrast with language model
- Position of word in the sentence

Checks ability to memorize
- Well-trained, richer architectures tend to fare better

Zhang et al. 2018; Liu et al., 2018; Conneau et al., 2018
Probing Surface-level Features

- Given a sentence, predict properties such as:
  - Length
  - Is a word in the sentence?

- Given a word in a sentence predict properties such as:
  - Previously seen words, contrast with language model
  - Position of word in the sentence

- Checks ability to memorize
  - Well-trained, richer architectures tend to fare better
  - Training on linguistic data memorizes better

Zhang et al. 2018; Liu et al., 2018; Conneau et al., 2018
Probing: Layers of the network
RNN layers: General linguistic properties
- Lowest layers: morphology
- Middle layers: syntax
- Highest layers: Task-specific semantics

Transformer layers:
- Different trends for different tasks; middle-heavy
- Also see Tenney et. al., 2019
Probing: Layers of the network

- **RNN layers:** General linguistic properties
  - Lowest layers: **morphology**
  - Middle layers: **syntax**
  - Highest layers: Task-specific **semantics**

- **Transformer layers:**
  - Different trends for different tasks; **middle-heavy**
  - Also see Tenney et. al., 2019

---

Fig. from Liu et al. (NAACL 2019)
Language modeling **outperforms** other unsupervised and supervised objectives.
- Machine Translation
- Dependency Parsing
- Skip-thought

Low-resource settings (size of training data) might result in opposite trends.

Zhang et al., 2018; Blevins et al., 2018; Liu et al., 2019;
Representations are **predictive** of certain linguistic phenomena:

- **Alignments** in translation, Linguistic features (e.g. syntactic **hierarchies**)

**What have we learnt so far?**
Representations are **predictive** of certain linguistic phenomena:
- **Alignments** in translation, Linguistic features (e.g. syntactic **hierarchies**)

Network architectures determine what is in a representation
- Syntax and BERT Transformer ([Tenney et al., 2019; Goldberg, 2019](#))
- Different layer-wise trends across architectures
Open questions about probes

- What information should a good probe look for?
  - Probing a probe!
Open questions about probes

- What information should a good probe look for?
  - Probing a probe!

- What does probing performance tell us?
  - Hard to synthesize results across a variety of baselines...
Open questions about probes

- What information should a good probe look for?
  - Probing a probe!

- What does probing performance tell us?
  - Hard to synthesize results across a variety of baselines...

- Can introduce some complexity in itself
  - linear or non-linear classification.
  - behavioral: design of input sentences
Open questions about probes

- What information should a good probe look for?
  - Probing a probe!

- What does probing performance tell us?
  - Hard to synthesize results across a variety of baselines...

- Can introduce some complexity in itself
  - linear or non-linear classification.
  - behavioral: design of input sentences

- Should we be using probes as evaluation metrics?
  - might defeat the purpose...
- Progressively erase or mask network components
  - Word embedding dimensions
  - Hidden units
  - Input - words / phrases
Progressively erase or mask network components

- Word embedding dimensions
- Hidden units
- Input - words / phrases

Figure 5: Heatmap of word importance (computed using Eq. 1) in sentiment analysis.

Li et al., 2016
So, what is in a representation?
So, what is in a representation?

- Depends on how you look at it!
  - **Visualization:**
    - bird’s eye view
    - few samples -- might call to mind cherry-picking
  - **Probes:**
    - discover corpus-wide specific properties
    - may introduce own biases...
  - **Network ablations:**
    - great for improving modeling,
    - could be task specific
So, what is in a representation?

- Depends on how you look at it!
  - **Visualization:**
    - bird’s eye view
    - few samples -- might call to mind cherry-picking
  - **Probes:**
    - discover corpus-wide **specific** properties
    - may introduce own biases...
  - **Network ablations:**
    - great for **improving modeling**, could be task specific

- Analysis methods as tools to aid model development!
Introduction

Pretraining

What's in a representation?

Adaptation
4. Adaptation
4 – How to adapt the pretrained model

Several orthogonal directions we can make decisions on:
Several orthogonal directions we can make decisions on:

1. **Architectural** modifications?

   *How much to change the pretrained model architecture for adaptation*
4 – How to adapt the pretrained model

Several orthogonal directions we can make decisions on:

1. **Architectural** modifications?
   
   *How much to change the pretrained model architecture for adaptation*

2. **Optimization** schemes?
   
   *Which weights to train during adaptation and following what schedule*
4.1 – Architecture

Two general options:
4.1 – Architecture

Two general options:

A. **Keep** pretrained model **internals unchanged**: 
   *Add classifiers on top, embeddings at the bottom, use outputs as features*
4.1 – Architecture

Two general options:

A. **Keep** pretrained model **internals unchanged**:  
*Add classifiers on top, embeddings at the bottom, use outputs as features*

B. **Modify** pretrained model internal architecture:  
*Initialize encoder-decoders, task-specific modifications, adapters*
4.1 – Architecture

Two general options:

A. **Keep** pretrained model **internals unchanged**: 
   *Add classifiers on top, embeddings at the bottom, use outputs as features*

B. **Modify** pretrained model internal architecture:
   *Initialize encoder-decoders, task-specific modifications, adapters*
4.1.A – Architecture: Keep model unchanged

General workflow:
4.1.A – Architecture: Keep model unchanged

General workflow:

1. **Remove pretraining task head** if not useful for target task
   a. **Example**: remove softmax classifier from pretrained LM
   b. **Not always needed**: some adaptation schemes re-use the pretraining objective/task, e.g. for multi-task learning
4.1.A – Architecture: Keep model unchanged

General workflow:

Also known as finetuning*
4.1.A – Architecture: Keep model unchanged

General workflow:

2. Add target task-specific layers on top/bottom of pretrained model
   a. Simple: adding linear layer(s) on top of the pretrained model

Also known as finetuning*
4.1.A – Architecture: Keep model unchanged

General workflow:

2. Add target task-specific layers on top/bottom of pretrained model
   a. Simple: adding linear layer(s) on top of the pretrained model

Also known as finetuning*
4.1.A – Architecture: Keep model unchanged

General workflow:

2. Add target task-specific layers on top/bottom of pretrained model
   a. Simple: adding linear layer(s) on top of the pretrained model

Also known as finetuning*
Hands-on #2: Adapting our pretrained model
Let’s see how a simple fine-tuning scheme can be implemented with our pretrained model:

- **Plan**
  - **Start from our Transformer language model**
  - **Adapt the model to a target task:**
    - *keep the model core unchanged, load the pretrained weights*
    - *add a linear layer on top, newly initialized*
    - *use additional embeddings at the bottom, newly initialized*
Adaptation task

- We select a text classification task as the downstream task

- TREC-6: The Text REtrieval Conference (TREC) Question Classification (Li et al., COLING 2002)

- TREC consists of open-domain, fact-based questions divided into broad semantic categories contains 5500 labeled training questions & 500 testing questions with 6 labels: 
  
  NUMERIC, LOCATION, HUMAN, DESCRIPTION, ENTITY, ABBREVIATION
Adaptation task

- We select a text classification task as the downstream task
- TREC-6: The Text REtrieval Conference (TREC) Question Classification (Li et al., COLING 2002)
- TREC consists of open-domain, fact-based questions divided into broad semantic categories contains 5500 labeled training questions & 500 testing questions with 6 labels: NUMERIC, LOCATION, HUMAN, DESCRIPTION, ENTITY, ABBREVIATION

Ex:

- ★ How did serfdom develop in and then leave Russia? —> DESCRIPTION
- ★ What films featured the character Popeye Doyle? —> ENTITY
Adaptation task

- We select a text classification task as the downstream task
- TREC-6: The Text REtrieval Conference (TREC) Question Classification (Li et al., COLING 2002)
- TREC consists of open-domain, fact-based questions divided into broad semantic categories contains 5500 labeled training questions & 500 testing questions with 6 labels: NUMERIC, LOCATION, HUMAN, DESCRIPTION, ENTITY, ABBREVIATION

Ex:

- ★ How did serfdom develop in and then leave Russia? -> DESCRIPTION
- ★ What films featured the character Popeye Doyle? -> ENTITY

<table>
<thead>
<tr>
<th>Model</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoVe (McCann et al., 2017)</td>
<td>4.2</td>
</tr>
<tr>
<td>TBCNN (Mou et al., 2015)</td>
<td>4.0</td>
</tr>
<tr>
<td>LSTM-CNN (Zhou et al., 2016)</td>
<td>3.9</td>
</tr>
<tr>
<td>ULMFiT (ours)</td>
<td><strong>3.6</strong></td>
</tr>
</tbody>
</table>

Transfer learning models shine on this type of low-resource task

(Howard and Ruder, ACL 2018)
First adaptation scheme
First adaptation scheme

Classification

Start | Text | Extract

Transformer → Linear

(Radford et al., 2018)
First adaptation scheme

- Modifications:
  - Keep model internals unchanged
  - Add a linear layer on top
  - Add an additional embedding (classification token) at the bottom

(Radford et al., 2018)
Hands-on: Model adaptation

First adaptation scheme

- **Modifications:**
  - Keep model internals unchanged
  - Add a linear layer on top
  - Add an additional embedding (classification token) at the bottom

- **Computation flow:**
  - Model input: the tokenized question with a classification token at the end
  - Extract the last hidden-state associated to the classification token
  - Pass the hidden-state in a linear layer and softmax to obtain class probabilities

(Radford et al., 2018)
Hands-on: Model adaptation

```
AdaptationConfig = namedtuple('AdaptationConfig',
                            'num_classes, dropout, initializer_range, batch_size, lr, max_norm, n_epochs,
                            'n_warmup, valid_set_prop, gradient_accumulation_steps, device,'
                            'log_dir, dataset_cache')
adapt_args = AdaptationConfig(6, 0.1, 0.02, 16, 6.5e-5, 1.0, 3,
                              10, 0.1, 1, "cuda" if torch.cuda.is_available() else "cpu",
                              "/" , "/dataset_cache.bin")

import random
from torch.utils.data import TensorDataset, random_split

dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/trec/
                          "trec-tokenized-bert.bin")
datasets = torch.load(dataset_file)

for split_name in ['train', 'test']:
    # Trim the samples to the transformer's input length minus 1 & add a classification token
datasets[split_name] = [x[:args.num_max_positions-1] + [tokenizer.vocab['[CLS]']]
                               for x in datasets[split_name]]

    # Pad the dataset to max length
    padding_length = max(len(x) for x in datasets[split_name])
datasets[split_name] = [x + [tokenizer.vocab['[PAD]']] * (padding_length - len(x))
                               for x in datasets[split_name]]

    # Convert to torch.Tensor and gather inputs and labels
tensor = torch.tensor(datasets[split_name], dtype=torch.long)
lables = torch.tensor(datasets[split_name + '_labels'], dtype=torch.long)
datasets[split_name] = TensorDataset(tensor, labels)

    # Create a validation dataset from a fraction of the training dataset
valid_size = int(adapt_args.valid_set_prop * len(datasets['train']))
train_size = len(datasets['train']) - valid_size
valid_dataset, train_dataset = random_split(datasets['train'], [valid_size, train_size])

train_loader = DataLoader(train_dataset, batch_size=adapt_args.batch_size, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=adapt_args.batch_size, shuffle=False)
test_loader = DataLoader(datasets['test'], batch_size=adapt_args.batch_size, shuffle=False)
```
Hands-on: Model adaptation

Fine-tuning hyper-parameters:
- 6 classes in TREC-6
- Other fine tuning hyper parameters from Radford et al., 2018

```python
import random
from torch.utils.data import TensorDataset, random_split

dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/trec/trec-tokenized-bert.bin")
datasets = torch.load(dataset_file)

for split_name in ['train', 'test']:
    # Trim the samples to the transformer's input length minus 1 & add a classification token
    datasets[split_name] = [x[:args.num_max_positions-1] + [tokenizer.vocab['[CLS]']] for x in datasets[split_name]]

    # Pad the dataset to max length
    padding_length = max(len(x) for x in datasets[split_name])
    datasets[split_name] = [x + [tokenizer.vocab['[PAD]']] * (padding_length - len(x)) for x in datasets[split_name]]

    # Convert to torch.Tensor and gather inputs and labels
    tensor = torch.tensor(datasets[split_name], dtype=torch.long)
    labels = torch.tensor(datasets[split_name + '_labels'], dtype=torch.long)
    datasets[split_name] = TensorDataset(tensor, labels)

    # Create a validation dataset from a fraction of the training dataset
    valid_size = int(adapt_args.valid_set_prop * len(datasets['train']))
    train_size = len(datasets['train']) - valid_size
    valid_dataset, train_dataset = random_split(datasets['train'], [valid_size, train_size])

    train_loader = DataLoader(train_dataset, batch_size=adapt_args.batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=adapt_args.batch_size, shuffle=False)
    test_loader = DataLoader(datasets['test'], batch_size=adapt_args.batch_size, shuffle=False)
```

AdaptationConfig = namedtuple('AdaptationConfig',
    'num_classes, dropout, initializer_range, batch_size, lr, max_norm, n_epochs,
    'n_warmup, valid_set_prop, gradient_accumulation_steps, device,
    'log_dir, dataset_cache')
adapt_args = AdaptationConfig(6, 0.1, 0.02, 16, 6.5e-5, 1.0, 3,
10, 0.1, 1, "cuda" if torch.cuda.is_available() else "cpu",
"./", "./dataset_cache.bin")
Hands-on: Model adaptation

Fine-tuning hyper-parameters:
- 6 classes in TREC-6
- Other fine tuning hyper parameters from Radford et al., 2018

Let’s load and prepare our dataset:

```python
import random
from torch.utils.data import TensorDataset, random_split

dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/trec/" "trec-tokenized-bert.bin")
datasets = torch.load(dataset_file)

for split_name in ['train', 'test']:
    # Trim the samples to the transformer’s input length minus 1 & add a classification token
datasets[split_name] = [x[:args.num_max_positions-1] + [tokenizer.vocab['[CLS]']] for x in datasets[split_name]]

    # Pad the dataset to max length
    padding_length = max(len(x) for x in datasets[split_name])
datasets[split_name] = [x + [tokenizer.vocab['[PAD]']] * (padding_length - len(x)) for x in datasets[split_name]]

    # Convert to torch.Tensor and gather inputs and labels
    tensor = torch.tensor(datasets[split_name], dtype=torch.long)
    labels = torch.tensor(datasets[split_name + '_labels'], dtype=torch.long)
datasets[split_name] = TensorDataset(tensor, labels)

    # Create a validation dataset from a fraction of the training dataset
    valid_size = int(adapt_args.valid_set_prop * len(datasets['train']))
    train_size = len(datasets['train']) - valid_size
    valid_dataset, train_dataset = random_split(datasets['train'], [valid_size, train_size])

    train_loader = DataLoader(train_dataset, batch_size=adapt_args.batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=adapt_args.batch_size, shuffle=False)
    test_loader = DataLoader(datasets['test'], batch_size=adapt_args.batch_size, shuffle=False)
```
Fine-tuning hyper-parameters:
- 6 classes in TREC-6
- Other fine tuning hyper parameters from Radford et al., 2018

Let’s load and prepare our dataset:
- trim to the transformer input size & add a classification token at the end of each sample,
Hands-on: Model adaptation

Fine-tuning hyper-parameters:
- 6 classes in TREC-6
- Other fine tuning hyper parameters from Radford et al., 2018

Let’s load and prepare our dataset:
- trim to the transformer input size
- add a classification token at the end of each sample,
- pad to the left,

<table>
<thead>
<tr>
<th>I</th>
<th>love</th>
<th>Mom</th>
<th>‘</th>
<th>s</th>
<th>cooking</th>
<th>[CLS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>love</td>
<td>you</td>
<td>too!</td>
<td></td>
<td>[CLS]</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>way</td>
<td>[CLS]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This is the one</td>
<td>[CLS]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>[CLS]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
import random
from torch.utils.data import TensorDataset, random_split

dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/trec/" "trec-tokenized-bert.bin")
datasets = torch.load(dataset_file)
for split_name in ['train', 'test']:
    # Trim the samples to the transformer's input length minus 1 & add a classification token
    datasets[split_name] = [x[:args.num_max_positions-1] + [tokenizer.vocab['[CLS]']] for x in datasets[split_name]]

    # Pad the dataset to max length
    padding_length = max(len(x) for x in datasets[split_name])
datasets[split_name] = [x + [tokenizer.vocab['[PAD]']] * (padding_length - len(x)) for x in datasets[split_name]]

    # Convert to torch.Tensor and gather inputs and labels
    tensor = torch.tensor(datasets[split_name], dtype=torch.long)
    labels = torch.tensor(datasets[split_name] + _labels, dtype=torch.long)
    datasets[split_name] = TensorDataset(tensor, labels)

    # Create a validation dataset from a fraction of the training dataset
    valid_size = int(adapt_args.valid_set_prop * len(datasets['train']))
    train_size = len(datasets['train']) - valid_size
don_valid_dataset, train_dataset = random_split(datasets['train'], [valid_size, train_size])

    train_loader = DataLoader(train_dataset, batch_size=adapt_args.batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=adapt_args.batch_size, shuffle=True)
    test_loader = DataLoader(datasets['test'], batch_size=adapt_args.batch_size, shuffle=False)
```
Hands-on: Model adaptation

Fine-tuning hyper-parameters:
- 6 classes in TREC-6
- Other fine tuning hyper parameters from Radford et al., 2018

Let’s load and prepare our dataset:
- trim to the transformer input size & add a classification token at the end of each sample,
- pad to the left,
- convert to tensors.
Hands-on: Model adaptation

Adapt our model architecture

```python
class TransformerWithClfHead(nn.Module):
    def __init__(self, config, fine_tuning_config):
        super().__init__()
        self.config = fine_tuning_config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings,
                                        config.num_max_positions, config.num_heads, config.num_layers,
                                        fine_tuning_config.dropout, causal=not config.mlm)

        self.classification_head = nn.Linear(config.embed_dim, fine_tuning_config.num_classes)

        self.apply(self.init_weights)

    def init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        elif isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, clf_tokens_mask, clf_labels=None, padding_mask=None):
        hidden_states = self.transformer(x, padding_mask)

        clf_tokens_states = (hidden_states * clf_tokens_mask.unsqueeze(-1).float()).sum(dim=0)
        clf_logits = self.classification_head(clf_tokens_states)

        if clf_labels is not None:
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(clf_logits.view(-1, clf_logits.size(-1)), clf_labels.view(-1))
            return clf_logits, loss

        return clf_logits
```

# If you have pretrained a model in the first section, you can use its weights
# state_dict = model.state_dict()

# Otherwise, just load pretrained model weights (and reload the training config as well)
state_dict = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/"
                                    "nascl-2019-tutorial/model_checkpoint.pth"), map_location='cpu')
args = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/"
                                "nascl-2019-tutorial/model_training_args.bin"))

adaptation_model = TransformerWithClfHead(config=args, fine_tuning_config=adapt_args).to(adapt_args.device)
incompatible_keys = adaptation_model.load_state_dict(state_dict, strict=False)
print(f"Parameters discarded from the pretrained model: {incompatible_keys.unexpected_keys}")
print(f"Parameters added in the adaptation model: {incompatible_keys.missing_keys}")
```

Parameters discarded from the pretrained model: ['lm_head.weight']
Parameters added in the adaptation model: ['classification_head.weight', 'classification_head.bias']
Hands-on: Model adaptation

Adapt our model architecture

Keep our pretrained model unchanged as the backbone.

class TransformerWithClfHead(nn.Module):
    def __init__(self, config, fine_tuning_config):
        super().__init__()
        self.config = fine_tuning_config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings,
                                        config.num_max_positions, config.num_heads, config.num_layers,
                                        fine_tuning_config.dropout, causal=not config.mlm)

        self.classification_head = nn.Linear(config.embed_dim, fine_tuning_config.num_classes)

        self.apply(self.init_weights)

    def init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, clf_tokens_mask, clf_labels=None, padding_mask=None):
        hidden_states = self.transformer(x, padding_mask)

        clf_tokens_states = (hidden_states * clf_tokens_mask.unsqueeze(-1).float()).sum(dim=1)

        clf_logits = self.classification_head(clf_tokens_states)

        if clf_labels is not None:
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(clf_logits.view(-1, clf_logits.size(-1)), clf_labels.view(-1))
            return clf_logits, loss
        return clf_logits

    # If you have pretrained a model in the first section, you can use its weights
    # state_dict = model.state_dict()

    # Otherwise, just load pretrained model weights (and reload the training config as well)
    state_dict = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/
                                          \"nasci-2019-tutorial/model_checkpoint.pth\", map_location='cpu')
args = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/
                                          \"nasci-2019-tutorial/model_training_args.bin\""))

adaptation_model = TransformerWithClfHead(config=args, fine_tuning_config=adapt_args).to(\(adapt_args.device
incompatible_keys = adaptation_model.load_state_dict(state_dict, strict=False)
print(f'Parameters discarded from the pretrained model: {incompatible_keys.unexpected_keys}')
print(f'Parameters added in the adaptation model: {incompatible_keys.missing_keys}

Parameters discarded from the pretrained model: ['lm_head.weight']
Parameters added in the adaptation model: ['classification_head.weight', 'classification_head.bias']
Adapt our model architecture
Keep our pretrained model unchanged as the backbone.
Replace the pre-training head (language modeling) with the classification head:

A linear layer, which takes as input the hidden-state of the [CLF] token (using a mask)
Adapt our model architecture

Keep our pretrained model unchanged as the backbone.

Replace the pre-training head (language modeling) with the classification head:

A linear layer, which takes as input the hidden-state of the [CLF] token (using a mask)

* Initialize all the weights of the model.
Hands-on: Model adaptation

Adapt our model architecture

Keep our pretrained model unchanged as the backbone.

Replace the pre-training head (language modeling) with the classification head:
A linear layer, which takes as input the hidden-state of the [CLF] token (using a mask)

* Initialize all the weights of the model.
* Reload common weights from the pretrained model.

```python
class TransformerWithClfHead(nn.Module):
    def __init__(self, config, fine_tuning_config):
        super().__init__()
        self.config = fine_tuning_config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings,
                                        config.num_max_positions, config.num_heads, config.num_layers,
                                        fine_tuning_config.dropout, causal=not config.mlm)

        self.classification_head = nn.Linear(config.embed_dim, fine_tuning_config.num_classes)

    def apply(self, init_weights):
        self.transformer.apply(init_weights)

    def load_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.configinitializer_range)
        elif isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, clf_tokens_mask, clf_labels=None, padding_mask=None):
        hidden_states = self.transformer(x, padding_mask)

        clf_tokens_states = (hidden_states * clf_tokens_mask.unsqueeze(-1).float()).sum(dim=0)
        clf_logits = self.classification_head(clf_tokens_states)

        if clf_labels is not None:
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(clf_logits.view(-1), clf_labels.view(-1))
            return clf_logits, loss
        return clf_logits
```

# If you have pretrained a model in the first section, you can use its weights
# state_dict = model.state_dict()

# Otherwise, just load pretrained model weights (and reload the training config as well)
state_dict = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/
"nascl-2019-tutorial/model_checkpoint.pth"), map_location='cpu')
args = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/
"nascl-2019-tutorial/model_training_args.bin"))

adaptation_model = TransformerWithClfHead(config=args, fine_tuning_config=adapt_args).to(adapt_args.device)
incompatible_keys = adaptation_model.load_state_dict(state_dict, strict=False)
print("Parameters discarded from the pretrained model:" + incompatible_keys.unexpected_keys")
print("Parameters added in the adaptation model:" + incompatible_keys.missing_keys")
```

Parameters discarded from the pretrained model: [‘lm_head.weight’]
Parameters added in the adaptation model: [‘classification_head.weight’, ‘classification_head.bias’]
Hands-on: Model adaptation

Our fine-tuning code:

```python
# Training function and trainer
def update(engine, batch):
    adaptation_model.train()
    batch, labels = (t.to(adapt_args.device) for t in batch)
    inputs = batch.transpose(0, 1).contiguous()  # to shape [seg length, batch]
    _, loss = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), clf_labels=labels, padding_mask=(batch == tokenizer.vocab('[PAD]')))
    loss = loss / adapt_args.gradient_accumulation_steps
    loss.backward()
    torch.nn.utils.clip_grad_norm_(adaptation_model.parameters(), adapt_args.max_norm)
    if engine.state.iteration % adapt_args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()
trainer = Engine(update)

# Evaluation function and evaluator (evaluator output is the input of the metrics)
def inference(engine, batch):
    adaptation_model.eval()
    with torch.no_grad():
        batch, labels = (t.to(adapt_args.device) for t in batch)
        inputs = batch.transpose(0, 1).contiguous()  # to shape [seg length, batch]
        clf_logits = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), padding_mask=(batch == tokenizer.vocab('[PAD]')))
        return clf_logits, labels
    evaluator = Engine(inference)

# Attach metric to evaluator & evaluation to trainer: evaluate on valid set after each epoch
Accuracy().attach(evaluator, "accuracy")
@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(engine):
    evaluator.run(valid_loader)
    print(f"Validation Epoch: {engine.state.epoch} Error rate: {100*(1 - evaluator.state.metrics["accuracy"])}%")

# Learning rate schedule: linearly warm-up to lr and then to zero
scheduler = PiecewiseLinear(optimizer, 'lr', [(0, 0.0), (adapt_args.n_warmup, adapt_args.lr),
                                               (len(train_loader)*adapt_args.n_epochs, 0.0)])
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Add progressbar with loss
RunningAverage(output_transform=lambda x: x).attach(trainer, 'loss')
ProgressBar(persist=True).attach(trainer, metric_names=["loss"])"
Hands-on: Model adaptation

Our fine-tuning code:

A simple training update function:
* prepare inputs: transpose and build padding & classification token masks
* we have options to clip and accumulate gradients

```python
# Training function and trainer
def update(engine, batch):
    adaptation_model.train()
    batch, labels = (t.to(adapt_args.device) for t in batch)
    inputs = batch.transpose(0, 1).contiguous()  # to shape [seg_length, batch]
    _, loss = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), clf_labels=labels,
                                padding_mask=(batch == tokenizer.vocab('[PAD]')))
    loss = loss / adapt_args.gradient_accumulation_steps
    loss.backward()
    torch.nn.utils.clip_grad_norm_(adaptation_model.parameters(), adapt_args.max_norm)
    if engine.state.iteration % adapt_args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()
trainer = Engine(update)
```

```python
# Evaluation function and evaluator (evaluator output is the input of the metrics)
def inference(engine, batch):
    adaptation_model.eval()
    with torch.no_grad():
        batch, labels = (t.to(adapt_args.device) for t in batch)
        inputs = batch.transpose(0, 1).contiguous()  # to shape [seg_length, batch]
        clf_logits = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), padding_mask=(batch == tokenizer.vocab('[PAD]')))
        return clf_logits, labels
    evaluator = Engine(inference)
```

# Attach metric to evaluator & evaluation to trainer; evaluate on valid set after each epoch
Accuracy().attach(evaluator, "accuracy")
@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(engine):
    evaluator.run(valid_loader)
    print(f"Validation Epoch: {engine.state.epoch} Error rate: (100*(1 - evaluator.state.metrics["accuracy"]))")

# Learning rate schedule: linearly warm-up to lr and then to zero
scheduler = PiecewiseLinear(optimizer, 'lr', [(0, 0.0), (adapt_args.n_warmup, adapt_args.lr),
                                             (len(train_loader)*adapt_args.n_epochs, 0.0)])
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Add progressbar with loss
RunningAverage(output_transform=lambda x: x).attach(trainer, 'loss')
ProgressBar(persist=True).attach(trainer, metric_names=['loss'])

# Save checkpoints and finetuning config
checkpoint_handler = ModelCheckpoint(adapt_args.log_dir, 'finetuning_checkpoint', save_interval=1, require_empty=False)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, {'symodel': adaptation_model})
torch.save(args, os.path.join(adapt_args.log_dir, 'fine_tuning_args.bin'))
```
Hands-on: Model adaptation

Our fine-tuning code:

A simple training update function:
- **prepare inputs:** transpose and build padding & classification token masks
- *we have options to clip and accumulate gradients*

We will evaluate on our validation and test sets:
- **validation:** after each epoch
- **test:** at the end

```python
# Training function and trainer
def update(engine, batch):
    adaptation_model.train()
    batch, labels = (t.to(adapt_args.device) for t in batch)
    inputs = batch.transpose(0, 1).contiguous() # to shape [seg length, batch]
    _, loss = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), clf_labels=labels,
                               padding_mask=(batch == tokenizer.vocab('[PAD]')))
    loss.backward()
    loss.backward(0)
    if engine.state.iteration & adapt_args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()
trainer = Engine(update)

# Evaluation function and evaluator (evaluator output is the input of the metrics)
def inference(engine, batch):
    adaptation_model.eval()
    with torch.no_grad():
        batch, labels = (t.to(adapt_args.device) for t in batch)
        clf_logits = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), padding_mask=(batch == tokenizer.vocab('[PAD]')))
        return clf_logits, labels
```

```python
# Attach metric to evaluator & evaluation to trainer: evaluate on valid set after each epoch
Accuracy().attach(evaluator, "accuracy")
trainer.on(Events.EPOCH_COMPLETED)

# Learning rate schedule: linearly warm-up to lr and then to zero
scheduler = PiecewiseLinear(optimizer, 'lr', [(0, 0.0), (adapt_args.n_warmup, adapt_args.lr)],
                            (len(train_loader)*adapt_args.n_epochs, 0.0))
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)
```

```python
# Save checkpoints and finetuning config
checkpoint_handler = ModelCheckpoint(adapt_args.log_dir, 'finetuning_checkpoint', save_interval=1, require_empty=False)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, ("symodel": adaptation_model))
torch.save(args, os.path.join(adapt_args.log_dir, 'fine_tuning_args.bin'))
```
We can now fine-tune our model on TREC:
We can now fine-tune our model on TREC:

```python
trainer.run(train_loader, max_epochs=adapt_args.n_epochs)
```

- **Epoch [1/3]**
  - [307/307] 100% error rate: 9.174311926605505, loss=3.85e-01 [01:10:00:00]

- **Validation Epoch: 1** Error rate: 9.174311926605505

- **Epoch [2/3]**
  - [307/307] 100% error rate: 5.8715569633027523, loss=1.73e-01 [01:10:00:00]

- **Validation Epoch: 2** Error rate: 5.8715569633027523

- **Epoch [3/3]**
  - [307/307] 100% error rate: 5.688073394495408, loss=9.63e-02 [01:10:00:00]

- **Validation Epoch: 3** Error rate: 5.688073394495408

```python
evaluator.run(test_loader)
print(f"Test Results - Error rate: {100*(1.00 - evaluator.state.metrics[\'accuracy\']):.3f}"")
```

- **Test Results - Error rate:** 3.600
We can now fine-tune our model on TREC:

```python
trainer.run(train_loader, max_epochs=adapt_args.n_epochs)
```

We are at the state-of-the-art (ULMFiT)

<table>
<thead>
<tr>
<th>Model</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoVe (McCann et al., 2017)</td>
<td>4.2</td>
</tr>
<tr>
<td>TBCNN (Mou et al., 2015)</td>
<td>4.0</td>
</tr>
<tr>
<td>LSTM-CNN (Zhou et al., 2016)</td>
<td>3.9</td>
</tr>
<tr>
<td>ULMFiT (ours)</td>
<td><strong>3.6</strong></td>
</tr>
</tbody>
</table>
We can now fine-tune our model on TREC:

```python
[50] trainer.run(train_loader, max_epochs=adapt_args.n_epochs)
```

- **Epoch [1/3]**
  - [307/307] 100% accuracy, loss=3.85e-01 [01:10+00:00]
  - Validation Epoch: 1 Error rate: 9.174311926605505

- **Epoch [2/3]**
  - [307/307] 100% accuracy, loss=1.73e-01 [01:10+00:00]
  - Validation Epoch: 2 Error rate: 5.871559633027523

- **Epoch [3/3]**
  - [307/307] 100% accuracy, loss=9.63e-02 [01:10+00:00]
  - Validation Epoch: 3 Error rate: 5.68807394495408

```python
<ignite.engine.engine.State at 0x7ff4e8b385f8>
```

```python
evaluator.run(test_loader)
print(f"Test Results - Error rate: {100*(1.00 - evaluator.state.metrics['accuracy']):.3f}"")
```

- **Test Results** - Error rate: 3.600

**Remarks:**
- The error rate goes down quickly! After one epoch we already have >90% accuracy.
  - Fine-tuning is highly **data efficient** in Transfer Learning
- We took our pre-training & fine-tuning hyper-parameters straight from the literature on related models.
  - Fine-tuning is often **robust** to the exact choice of hyper-parameters

We are at the state-of-the-art (ULMFiT)
Let’s conclude this hands-on with a few additional words on robustness & variance.
Let’s conclude this hands-on with a few additional words on robustness & variance.

- Large pretrained models (e.g. BERT large) are prone to degenerate performance when fine-tuned on tasks with small training sets.
Let’s conclude this hands-on with a few additional words on robustness & variance.

- Large pretrained models (e.g. BERT large) are prone to degenerate performance when fine-tuned on tasks with small training sets.
- Observed behavior is often “on-off”: it either works very well or doesn’t work at all.

Phang et al., 2018
Hands-on: Model adaptation – Results

Let’s conclude this hands-on with a few additional words on robustness & variance.

- Large pretrained models (e.g. BERT large) are prone to degenerate performance when fine-tuned on tasks with small training sets.
- Observed behavior is often “on-off”: it either works very well or doesn’t work at all.
- Understanding the conditions and causes of this behavior (models, adaptation schemes) is an open research question.

Phang et al., 2018

Figure 1: Distribution of task scores across 20 random restarts for BERT, and BERT with intermediary fine-tuning on MNLI. Each cross represents a single run. Error lines show mean±1std. (a) Fine-tuned on all data, 5k tasks with <10k training examples. (b) Fine-tuned on no more than 5k examples for each task. (c) Fine-tuned on no more than 1k examples for each task. (*) indicates that the intermediate task is the same as the target task.
4.2 – Optimization
4.2 – Optimization

Several directions when it comes to the optimization itself:
4.2 – Optimization

Several directions when it comes to the optimization itself:

A. Choose **which weights** we should update
   
   *Feature extraction, fine-tuning, adapters*
4.2 – Optimization

Several directions when it comes to the optimization itself:

A. Choose **which weights** we should update
   *Feature extraction, fine-tuning, adapters*

B. Consider **practical trade-offs**
   *Space and time complexity, performance*
4.2.A – Optimization: Which weights?

The main question: To tune or not to tune (the pretrained weights)?
4.2.A – Optimization: Which weights?

The main question: **To tune or not to tune (the pretrained weights)?**

A. **Do not change** pretrained weights

*Feature extraction, adapters*
4.2.A – Optimization: Which weights?

The main question: **To tune or not to tune (the pretrained weights)?**

A. **Do not change** pretrained weights
   - *Feature extraction, adapters*

B. **Change** pretrained weights
   - *Fine-tuning*
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Feature extraction:
- Weights are **frozen**
- A linear classifier is trained on top of the pretrained representations
- Don’t just use features of the top layer!
- Learn a linear combination of layers

(Peters et al., NAACL 2018, Ruder et al., AAAI 2019)
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Adapters

- Task-specific modules that are added *in between* existing layers
- Only adapters are trained
4.2.A – Optimization: Which weights?

Yes, change the pretrained weights!
4.2.A – Optimization: Which weights?

Yes, change the pretrained weights!

Fine-tuning:
4.2.A – Optimization: Which weights?

Yes, change the pretrained weights!

Fine-tuning:

- Pretrained weights are used as initialization for parameters of the downstream model.
4.2.A – Optimization: Which weights?

Yes, change the pretrained weights!

Fine-tuning:

- Pretrained weights are used as initialization for parameters of the downstream model
- The whole pretrained architecture is trained during the adaptation phase
4.2.B – Optimization: Trade-offs

Several trade-offs when choosing which weights to update:
4.2.B – Optimization: Trade-offs

Several trade-offs when choosing which weights to update:

A. **Space** complexity

*Task-specific modifications, additional parameters, parameter reuse*
Several trade-offs when choosing which weights to update:

A. **Space** complexity
   - *Task-specific modifications, additional parameters, parameter reuse*

B. **Time** complexity
   - *Training time*
4.2.B – Optimization: Trade-offs

Several trade-offs when choosing which weights to update:

A. **Space** complexity
   
   *Task-specific modifications, additional parameters, parameter reuse*

B. **Time** complexity
   
   *Training time*

C. **Performance**
4.2.B – Optimization trade-offs: Space

Task-specific modifications

Feature extraction
  Many

Adapters

Fine-tuning
  Few
4.2.B – Optimization trade-offs: Space

Task-specific modifications

Feature extraction

Many

Adapters

Fine-tuning

Few

Additional parameters

Feature extraction

Many

Adapters

Fine-tuning

Few
4.2.B – Optimization trade-offs: Space

Task-specific modifications

Feature extraction
Many

Adapters
Few

Fine-tuning

Additional parameters

Feature extraction
Many

Adapters
Few

Fine-tuning

Parameter reuse

Feature extraction
All

Adapters
None

Fine-tuning
4.2.B – Optimization trade-offs: Time

Training time

Feature extraction

Adapters

Fine-tuning

Slow

Fast
4.2.B – Optimization trade-offs: Performance
Rule of thumb: If task source and target tasks are dissimilar*, use feature extraction (Peters et al., 2019)

*dissimilar: certain capabilities (e.g. modelling inter-sentence relations) are beneficial for target task, but pretrained model lacks them
4.2.B – Optimization trade-offs: Performance

- Rule of thumb: If task source and target tasks are **dissimilar**, use feature extraction ([Peters et al., 2019](#)).
- Otherwise, feature extraction and fine-tuning often perform similar.

---

*dissimilar: certain capabilities (e.g. modelling inter-sentence relations) are beneficial for target task, but pretrained model lacks them.*
Rule of thumb: If task source and target tasks are **dissimilar***, use feature extraction ([Peters et al., 2019](#)). Otherwise, feature extraction and fine-tuning often perform similar. Fine-tuning BERT on textual similarity tasks works significantly better.

*dissimilar: certain capabilities (e.g. modelling inter-sentence relations) are beneficial for target task, but pre-trained model lacks them*
4.2.B – Optimization trade-offs: Performance

- Rule of thumb: If task source and target tasks are **dissimilar**, use feature extraction ([Peters et al., 2019](#))
- Otherwise, feature extraction and fine-tuning often perform similar
- Fine-tuning BERT on textual similarity tasks works significantly better
- Adapters achieve performance competitive with fine-tuning

*dissimilar: certain capabilities (e.g. modelling inter-sentence relations) are beneficial for target task, but pretrained model lacks them*
Rule of thumb: If task source and target tasks are dissimilar*, use feature extraction (Peters et al., 2019)

Otherwise, feature extraction and fine-tuning often perform similar

Fine-tuning BERT on textual similarity tasks works significantly better

Adapters achieve performance competitive with fine-tuning

Anecdotally, Transformers are easier to fine-tune (less sensitive to hyper-parameters) than recurrent neural nets (e.g. LSTMs)

*dissimilar: certain capabilities (e.g. modelling inter-sentence relations) are beneficial for target task, but pretrained model lacks them
In summary

[1] Introduction

[2] Pretraining


[4] Adaptation
Pretraining tasks
Pretraining tasks

More diverse self-supervised objectives
Pretraining tasks

More diverse self-supervised objectives

- computer vision

Example:

Sampling a patch and a neighbour and predicting their spatial configuration (Doersch et al., ICCV 2015)

Image colorization (Zhang et al., ECCV 2016)
Pretraining tasks

More diverse self-supervised objectives

- computer vision

- Self-supervision in language mostly based on word co-occurrence (Ando and Zhang, 2005) Instead, supervision on different levels of meaning

Example:

Sampling a patch and a neighbour and predicting their spatial configuration (Doersch et al., ICCV 2015)

Image colorization (Zhang et al., ECCV 2016)
Pretraining tasks

More diverse self-supervised objectives

- computer vision

- Self-supervision in language mostly based on word co-occurrence (Ando and Zhang, 2005) Instead, supervision on different levels of meaning
  - Discourse, document, sentence, etc.

Example:

Sampling a patch and a neighbour and predicting their spatial configuration (Doersch et al., ICCV 2015)

Image colorization (Zhang et al., ECCV 2016)
Pretraining tasks

More diverse self-supervised objectives

- computer vision

- Self-supervision in language mostly based on word co-occurrence (Ando and Zhang, 2005) Instead, supervision on different levels of meaning
  - Discourse, document, sentence, etc.
  - Using other signals, e.g. meta-data

Sampling a patch and a neighbour and predicting their spatial configuration (Doersch et al., ICCV 2015)

Image colorization (Zhang et al., ECCV 2016)
Pretraining tasks
Pretraining tasks

Need for grounded representations
Pretraining tasks

Need for grounded representations

- Limits of distributional hypothesis—difficult to learn certain types of information from raw text
  - Human reporting bias: not stating the obvious (Gordon and Van Durme, AKBC 2013)
  - Common sense isn’t written down
  - No grounding to other modalities
Pretraining tasks

Need for grounded representations

- Limits of distributional hypothesis—difficult to learn certain types of information from raw text
  - Human reporting bias: not stating the obvious (Gordon and Van Durme, AKBC 2013)
  - Common sense isn’t written down
  - No grounding to other modalities

- Possible solutions:
  - Incorporate other structured knowledge (e.g. knowledge bases like ERNIE, Zhang et al 2019)
  - Multimodal learning (e.g. with visual representations like VideoBERT, Sun et al. 2019)
  - Interactive/human-in-the-loop approaches (e.g. dialog, Hancock et al. 2018)
Continual learning

- Current transfer learning performs adaptation once.
- Ultimately, we’d like to have models that continue to retain and accumulate knowledge across many tasks (Yogatama et al., 2019).
- No distinction between pretraining and adaptation; just one stream of tasks.
- Main challenge towards this: Catastrophic forgetting.
Thank you!

Questions?

Email: swabhas@allenai.org
https://swabhs.com
@swabhz

Other Resources:
Colab
Full tutorial Video
Tutorial
Slides