Transfer Learning in NLP
NLPL Winter School

Thomas Wolf - HuggingFace Inc.

Overview

- ❏ Session 1: Transfer Learning - Pretraining and representations
- ❏ **Session 2: Transfer Learning - Adaptation and downstream tasks**
- ❏ Session 3: Transfer Learning - Limitations, open-questions, future directions

Sebastian Ruder

Matthew Peters

Swabha Swayamdipta

Many slides are adapted from **a Tutorial on Transfer Learning in NLP** I gave at NAACL 2019 with my amazing collaborators

Transfer Learning in NLP
NLPL Winter School
Session 2

Transfer Learning in NLP

Follow along with the tutorial:

- ❑ Colab: https://tinyurl.com/NAACLTransferColab
- ❑ Code: https://tinyurl.com/NAACLTransferCode

Agenda

[1] Introduction

[2] Pretraining

[3] What's in a representation?

[4] Adaptation

[5] Downstream

[6] Open Problems

4. Adaptation

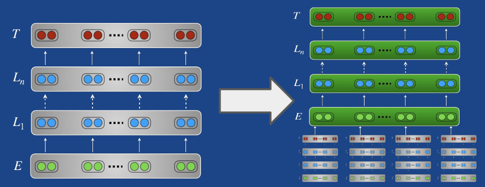
Image credit: Ben Didier

4 – How to adapt the pretrained model

Several orthogonal directions we can make decisions on:

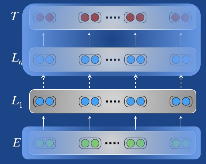
1. **Architectural** modifications?

   How much to change the pretrained model architecture for adaptation



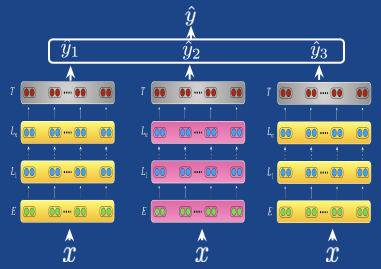
2. **Optimization** schemes?

   Which weights to train during adaptation and following what schedule



3. More **signal:** Weak supervision, Multi-tasking & Ensembling

   How to get more supervision signal for the target task



4.1 – Architecture

Two general options:

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

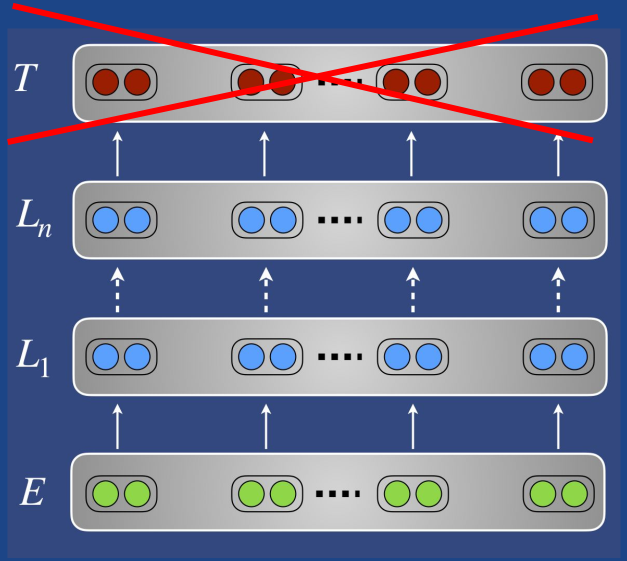
B. **Modify** pretrained model internal architecture:
Initialize encoder-decoders, task-specific modifications, adapters

Image credit: Darmawansyah

4.1.A – Architecture: Keep model **unchanged**

General workflow:

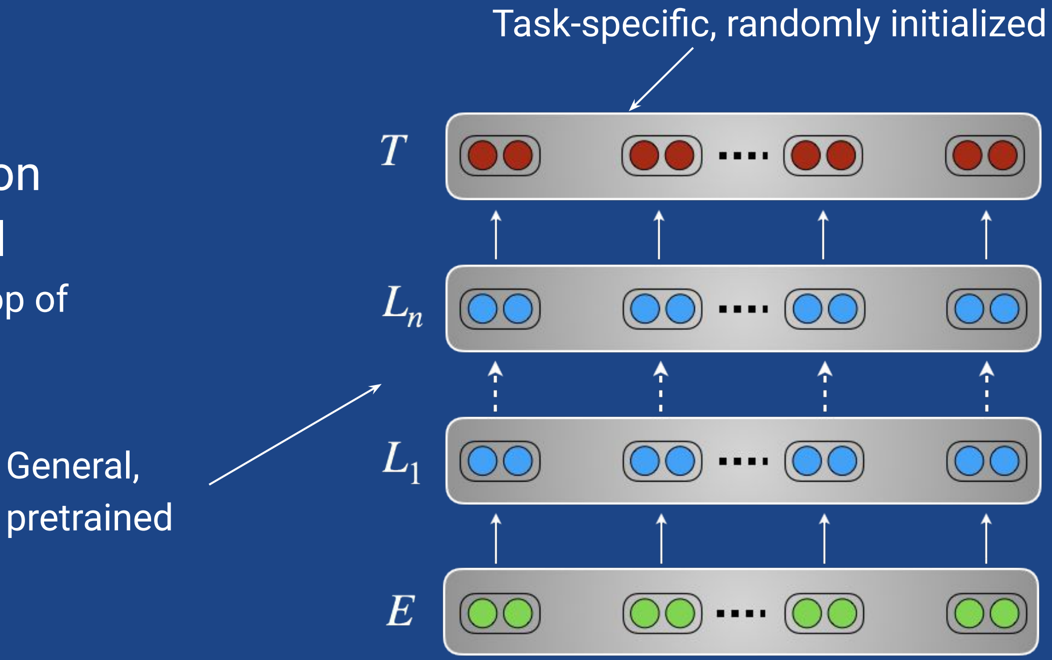
1. **Remove pretraining task head** if not useful for target task
   a. **Example**: remove softmax classifier from pretrained LM
   b. **Not always needed**: some adaptation schemes re-use the pretraining objective/task, e.g. for **multi-task learning**



4.1.A – Architecture: Keep model **unchanged**

General workflow:

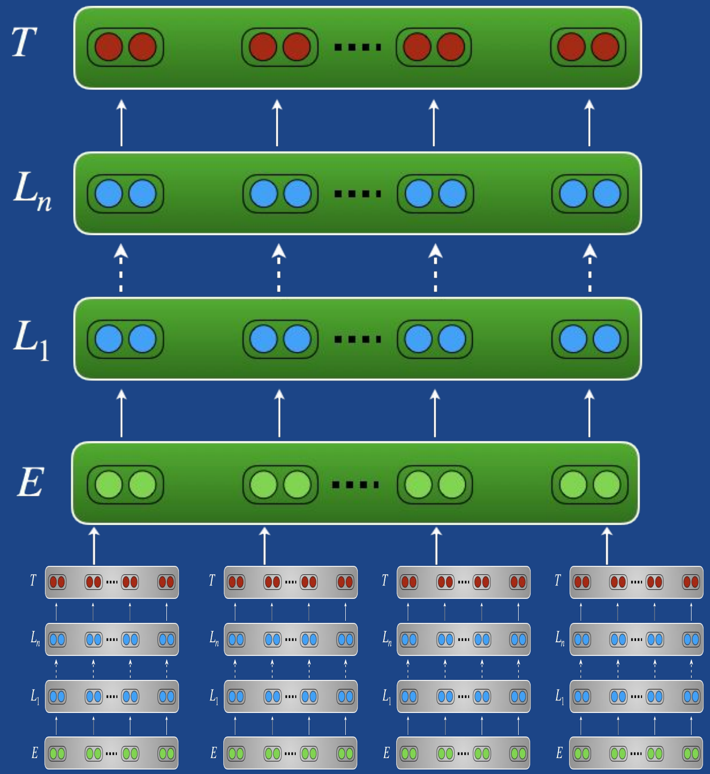
2. Add target task-specific layers on top/bottom of pretrained model
    a. **Simple**: adding linear layer(s) on top of the pretrained model



4.1.A – Architecture: Keep model **unchanged**

General workflow:

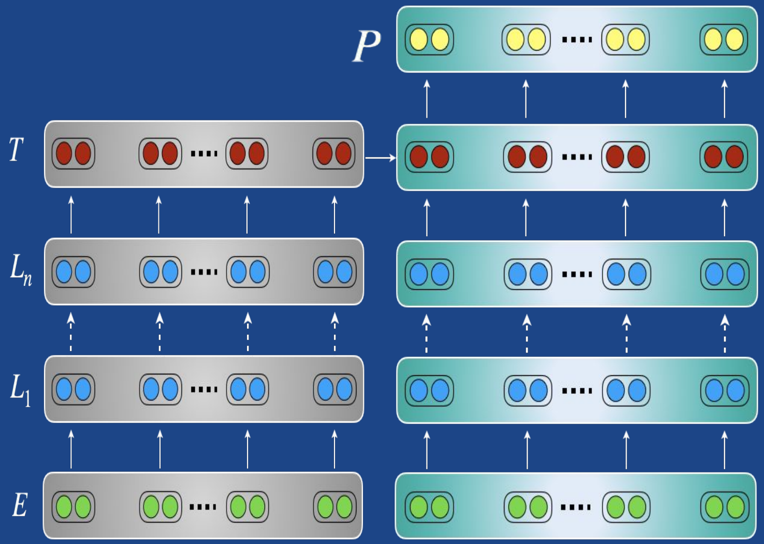
2. Add target task-specific layers on top/bottom of pretrained model
    a. **Simple**: adding linear layer(s) on top of the pretrained model
    b. **More complex**: model output as input for a separate model
    c. Often beneficial when target task requires **interactions** that are not available in pretrained embedding



4.1.B – Architecture: Modifying model internals

Various reasons:

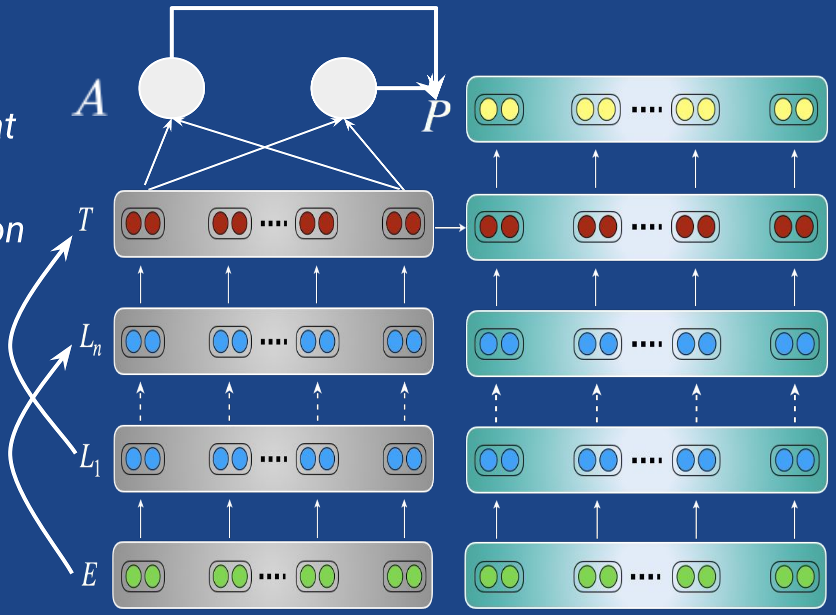
1. Adapting to a **structurally different** target task
   a. *Ex: Pretraining with a <u>single</u> input sequence (ex: language modeling) but adapting to a task with <u>several</u> input sequences (ex: translation, conditional generation...)*
   b. *Use the pretrained model weights to initialize as much as possible of a structurally different target task model*
   c. *Ex: Use monolingual LMs to initialize encoder and decoder parameters for MT ([Ramachandran et al., EMNLP 2017](); [Lample & Conneau, 2019]())*



4.1.B – Architecture: Modifying model internals

Various reasons:

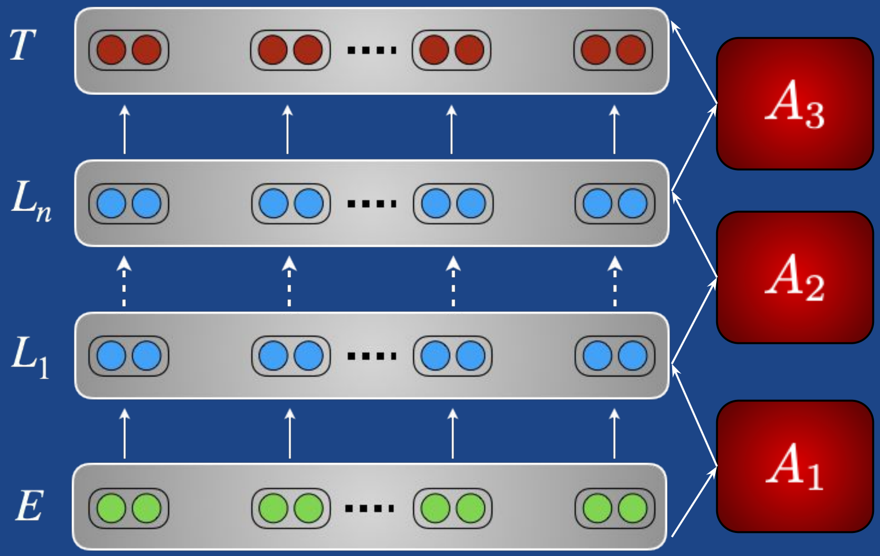
2. Task-specific **modifications**
   a. *Provide pretrained model with capabilities that are useful for the target task*
   b. *Ex: Adding skip/residual connections, attention (Ramachandran et al., EMNLP 2017)*



4.1.B – Architecture: Modifying model internals

Various reasons:

3. Using **less parameters** for adaptation:
   a. Less parameters to fine-tune
   b. Can be **very useful** given the increasing size of model parameters
   c. Ex: add bottleneck modules ("adapters") between the layers of the pretrained model (Rebuffi et al., NIPS 2017; CVPR 2018)



4.1.B – Architecture: Modifying model internals

Adapters

- ❑ Commonly connected with a **residual connection** in parallel to an existing layer
- ❑ Most effective when placed at **every layer** (smaller effect at bottom layers)
- ❑ **Different operations** (convolutions, self-attention) possible
- ❑ Particularly suitable for modular architectures like Transformers ([Houlsby et al., ICML 2019](); [Stickland and Murray, ICML 2019]())

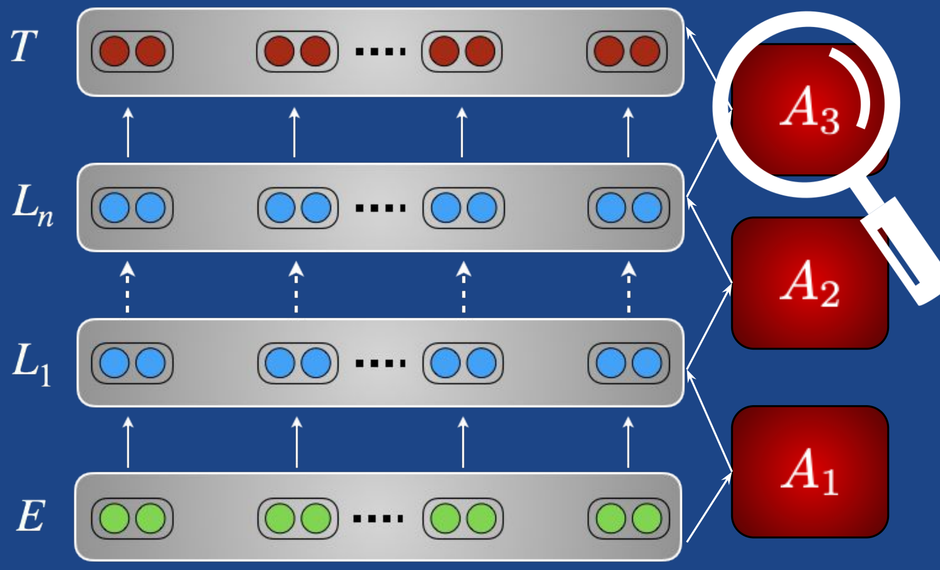
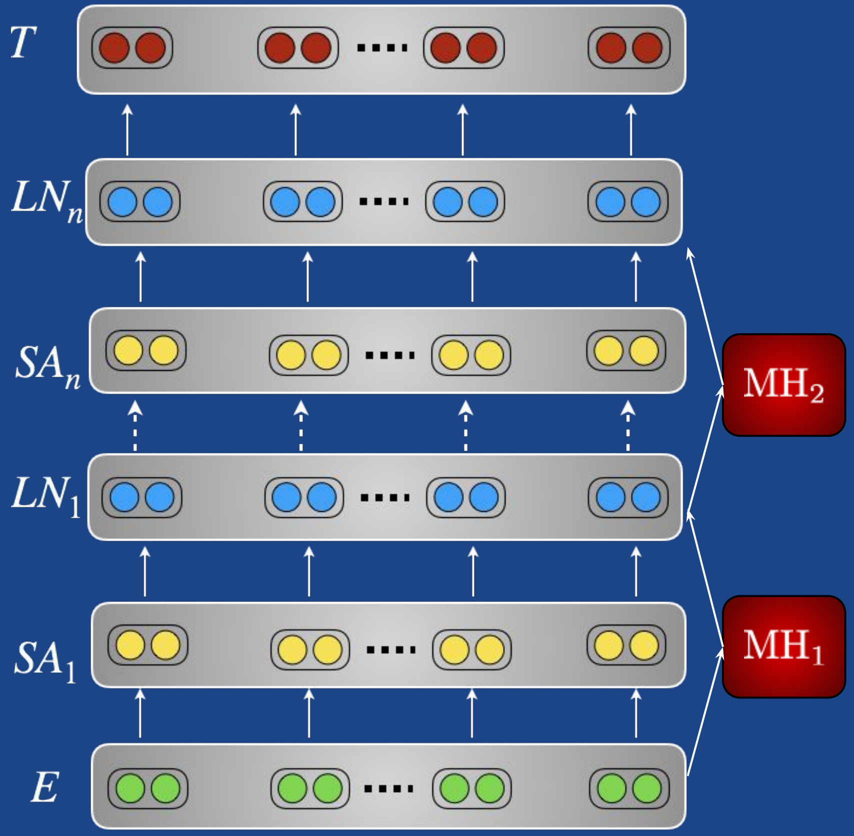


Image credit: Caique Lima

4.1.B – Architecture: Modifying model internals

Adapters (Stickland & Murray, ICML 2019)

- Multi-head attention (**MH**; shared across layers) is used in parallel with self-attention (**SA**) layer of BERT
- Both are added together and fed into a layer-norm (**LN**)



Hands-on #2:
Adapting our pretrained model

Image credit: Chanaky

Hands-on: Model adaptation

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*
- Reminder — material is here:
  - Colab http://tiny.cc/NAACLTransferColab ➪ code of the following slides
  - Code http://tiny.cc/NAACLTransferCode ➪ same code in a repo

Hands-on: Model adaptation

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:

  NUM, LOC, HUM, DESC, ENTY, ABBR

  Ex:

  ★ How did serfdom develop in and then leave Russia ? —> *DESC*

  ★ What films featured the character Popeye Doyle ? —> *ENTY*

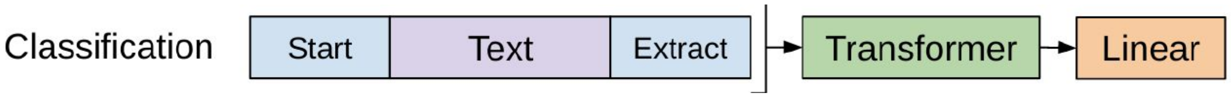
	Model	Test
TREC-6	CoVe (McCann et al., 2017)	4.2
	TBCNN (Mou et al., 2015)	4.0
	LSTM-CNN (Zhou et al., 2016)	3.9
	ULMFiT (ours)	**3.6**

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

(Howard and Ruder, ACL 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

Fine-tuning hyper-parameters:

– 6 classes in TREC-6

– Use fine tuning hyper parameters from Radford et al., 2018:

- learning rate from 6.5e-5 to 0.0
- fine-tune for 3 epochs

```
AdaptationConfig = namedtuple('AdaptationConfig',
  field_names="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")
```

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,
- extract a validation set.

I	love	Mom	'	s	cooking	[CLS]
I	love	you	too	!	[CLS]	
No	way	[CLS]				
This	is	the	one	[CLS]		
Yes	[CLS]					

```
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)
```

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)

```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)
        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=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
```

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

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

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

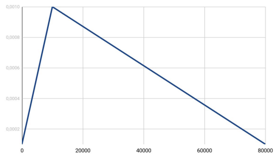
Hands-on: Model adaptation

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*

Schedule:
* *linearly increasing to lr*
* *linearly decreasing to 0.0*




```python
optimizer = torch.optim.Adam(adaptation_model.parameters(), lr=adapt_args.lr)

# 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 [seq 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 [seq 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)

# Attache 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, {'mymodel': adaptation_model})
torch.save(args, os.path.join(adapt_args.log_dir, 'fine_tuning_args.bin'))
```

Hands-on: Model adaptation – Results

We can now fine-tune our model on TREC:

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

```
Epoch [1/3]  [307/307] 100%| , loss=3.85e-01 [01:10<00:00]
Validation Epoch: 1 Error rate: 9.174311926605505
Epoch [2/3]  [307/307] 100%| , loss=1.73e-01 [01:10<00:00]
Validation Epoch: 2 Error rate: 5.871559633027523
Epoch [3/3]  [307/307] 100%| , loss=9.63e-02 [01:10<00:00]
Validation Epoch: 3 Error rate: 5.688073394495408
<ignite.engine.engine.State at 0x7ff4c8b385f8>
```

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

```
Test Results - Error rate: 3.600
```

	Model	Test
TREC-6	CoVe (McCann et al., 2017)	4.2
	TBCNN (Mou et al., 2015)	4.0
	LSTM-CNN (Zhou et al., 2016)	3.9
	ULMFiT (ours)	**3.6**

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

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

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.

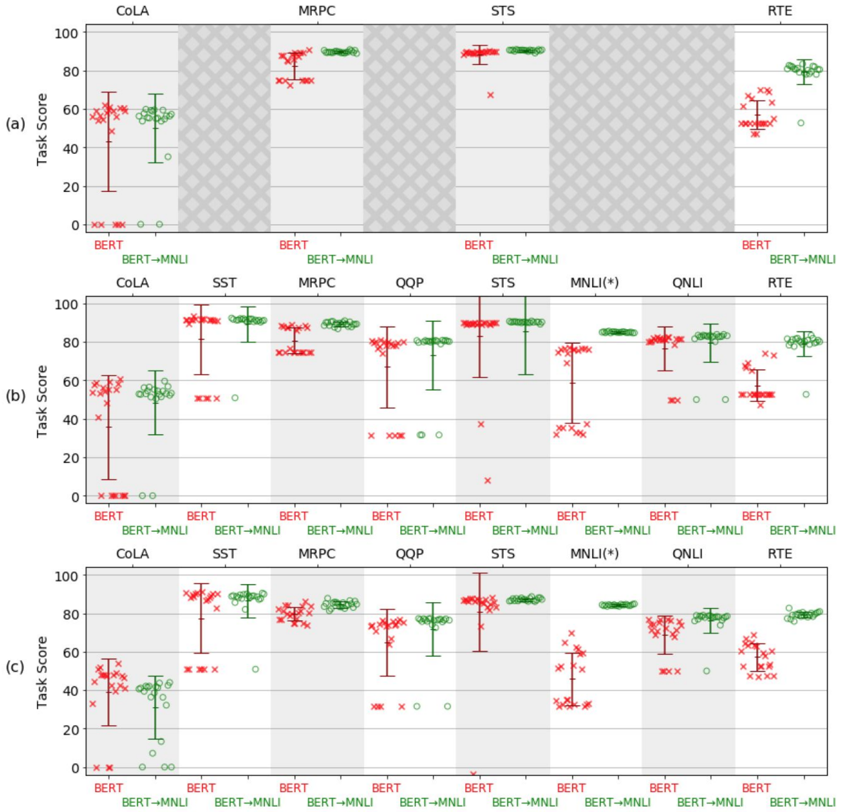


Figure 1: Distribution of task scores across 20 random restarts for BERT, and BERT with intermediary fine-tuning on MNLI. Each cross represents a single run. Error lines show mean±1std. (a) Fine-tuned on all data, for tasks with <10k training examples. (b) Fine-tuned on no more than 5k examples for each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

Phang et al., 2018

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. Choose **how and when** to update the weights
From top to bottom, gradual unfreezing, discriminative fine-tuning

C. Consider **practical trade-offs**
Space and time complexity, performance

Image credit: ProSymbols, purplestudio, Markus, Alfredo

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

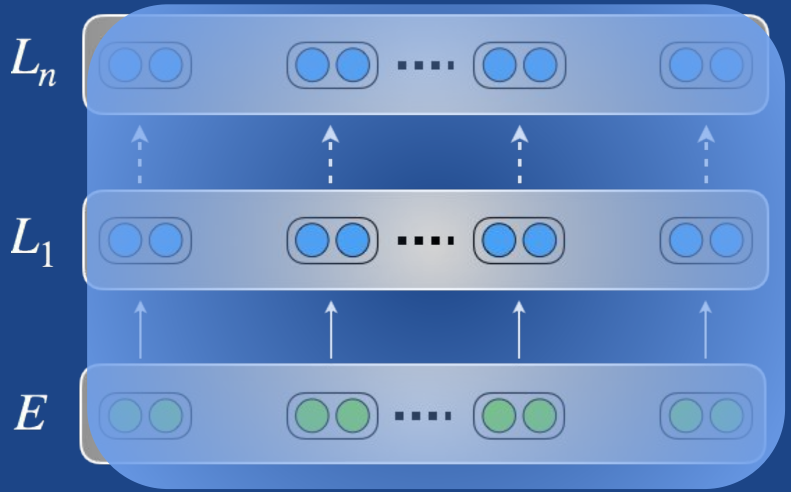
Image credit: purplestudio

4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Feature extraction:

- ❑ Weights are **frozen**

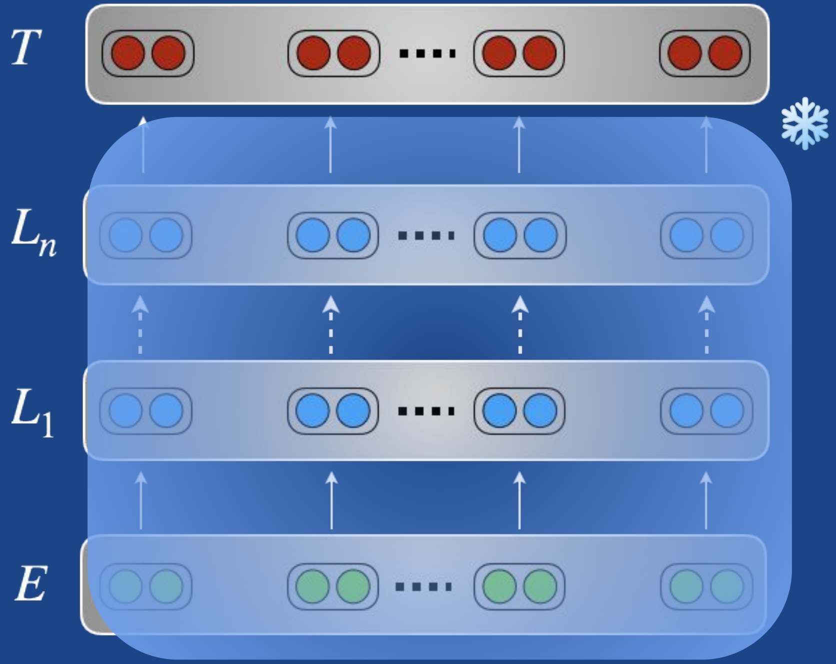


4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Feature extraction:

- ❑ Weights are **frozen**
- ❑ A **linear classifier** is trained on top of the pretrained representations

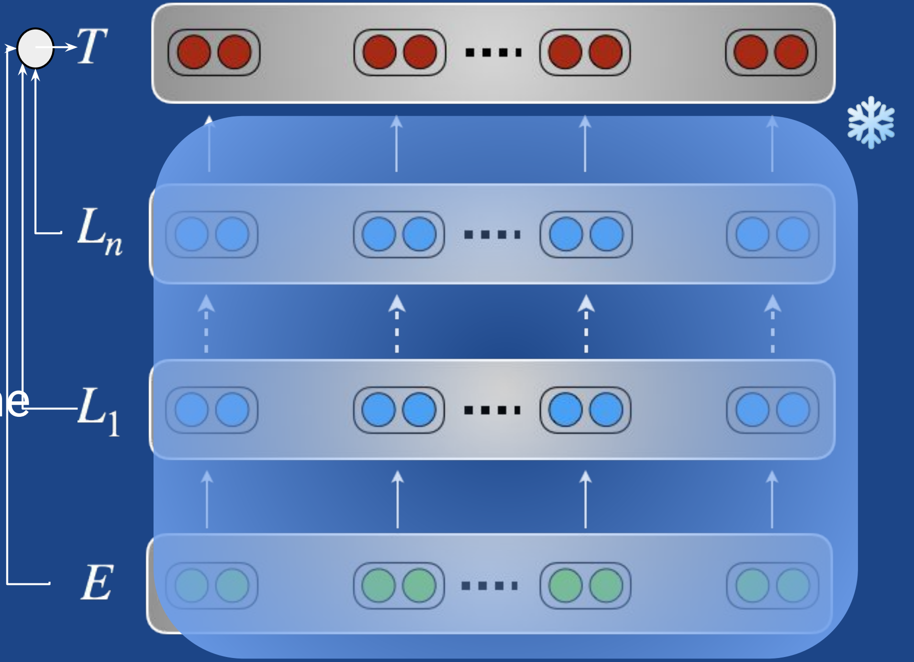


4.2.A – Optimization: Which weights?

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]())

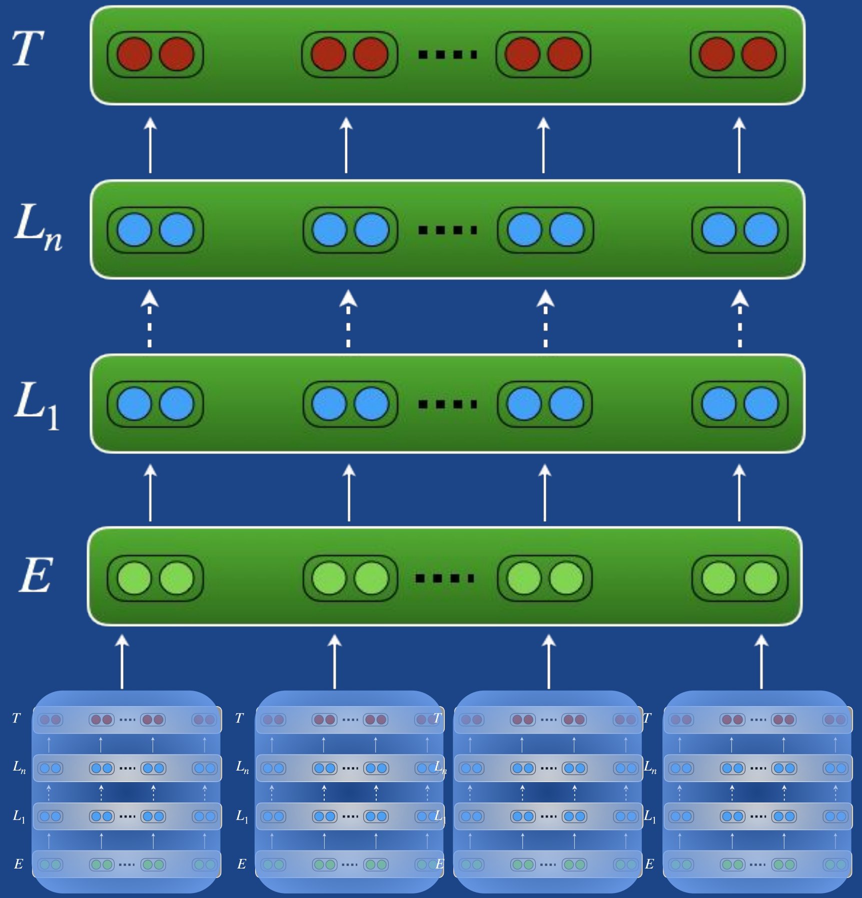


4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Feature extraction:

- ❏ Alternatively, pretrained representations are **used as features** in downstream model

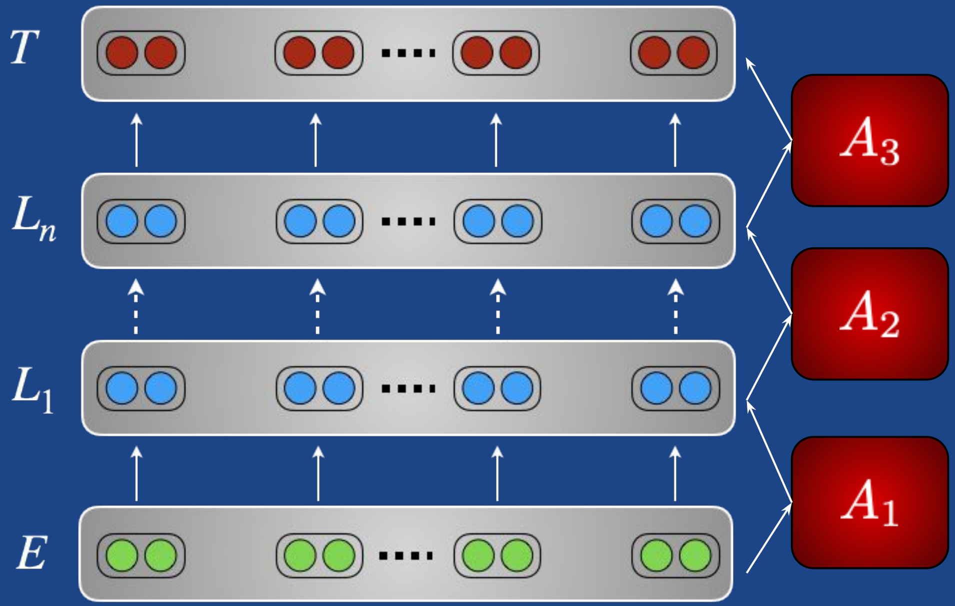


4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Adapters

- Task-specific modules that are added **in between** existing layers

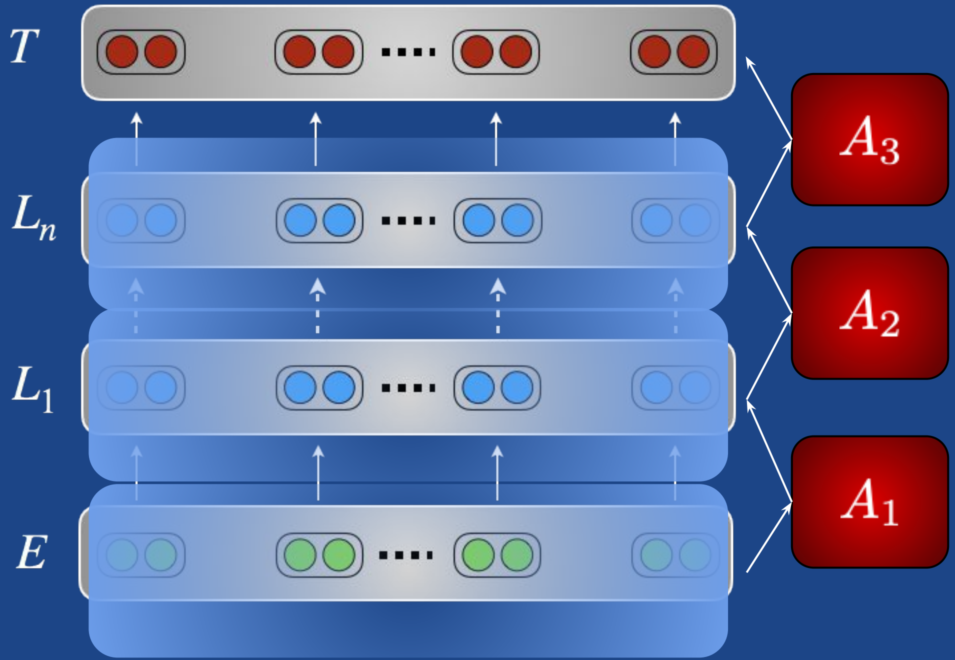


4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Adapters

- ❑ Task-specific modules that are added **in between** existing layers
- ❑ Only adapters are trained

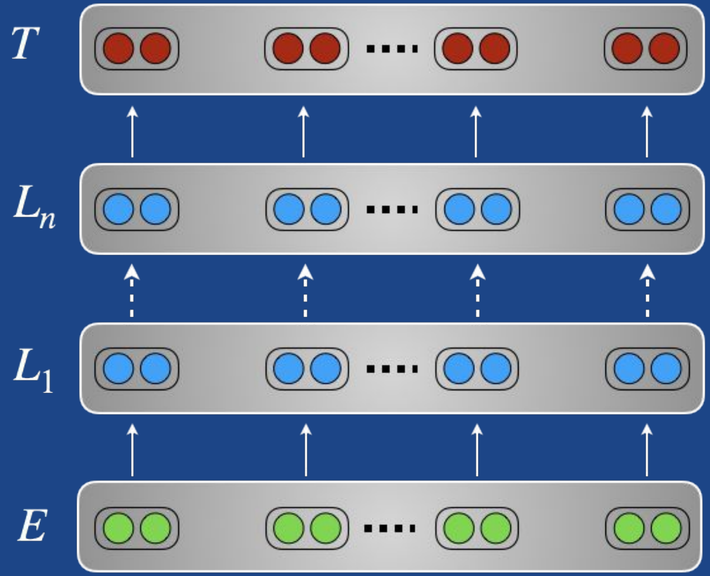


4.2.A – Optimization: Which weights?

Yes, change the pretrained weights!

Fine-tuning:

- ❏ Pretrained weights are used as **initialization** for parameters of the downstream model
- ❏ The **whole pretrained architecture** is trained during the adaptation phase



Hands-on #3:
Using Adapters and freezing

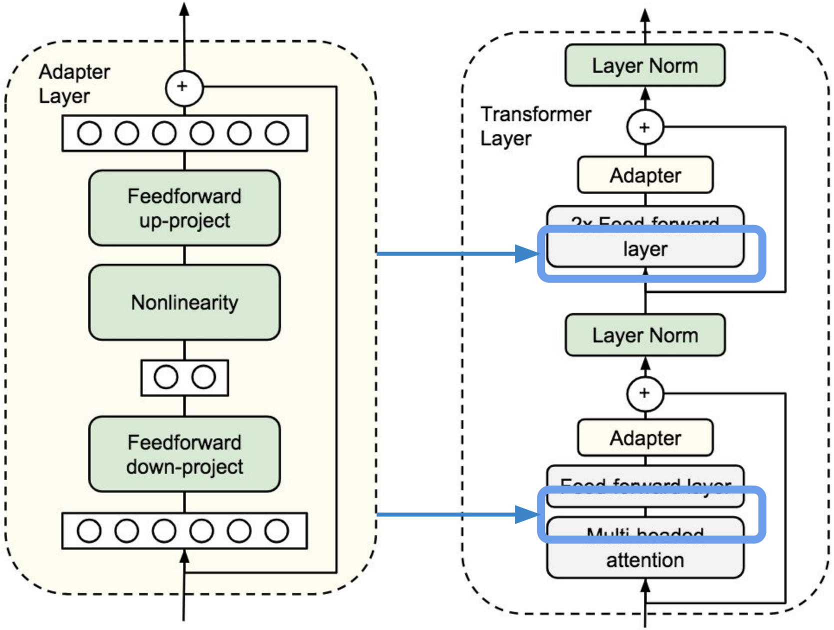
Image credit: Chanaky

Hands-on: Model adaptation

Second adaptation scheme: Using Adapters

- Modifications:
  - **add Adapters** inside the backbone model: *Linear* ➪ *ReLU* ➪ *Linear* *with a skip-connection*
- As previously:
  - add a linear layer on top
  - use an additional embedding (classification token) at the bottom

We will **only** *train the adapters, the added linear layer and the embeddings*. The other parameters of the model will be **frozen**.



Houlsby et al., ICML 2019

Hands-on: Model adaptation

Let's adapt our model architecture

Inherit from our pretrained model to have all the modules.

Add the adapter modules:
Bottleneck layers with 2 linear layers and a non-linear activation function (ReLU)

Hidden dimension is small: e.g. 32, 64, 256

The Adapters are inserted inside skip-connections after:
- ❑ the attention module
- ❑ the feed-forward module

```python
class TransformerWithAdapters(Transformer):
    def __init__(self, adapters_dim, embed_dim, hidden_dim, num_embeddings, num_max_positions,
                 num_heads, num_layers, dropout, causal):
        """ Transformer with adapters (small bottleneck layers) """
        super().__init__(embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers,
                         dropout, causal)
        self.adapters_1 = nn.ModuleList()
        self.adapters_2 = nn.ModuleList()
        for _ in range(num_layers):

            self.adapters_1.append(nn.Sequential(nn.Linear(embed_dim, adapters_dim),
                                                 nn.ReLU(),
                                                 nn.Linear(adapters_dim, embed_dim)))

            self.adapters_2.append(nn.Sequential(nn.Linear(embed_dim, adapters_dim),
                                                 nn.ReLU(),
                                                 nn.Linear(adapters_dim, embed_dim)))

    def forward(self, x, padding_mask=None):
        """ x has shape [seq length, batch], padding_mask has shape [batch, seq length] """
        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, adapter_1, layer_norm_2, feed_forward, adapter_2)\
                         in zip(self.layer_norms_1, self.attentions,    self.adapters_1,
                                self.layer_norms_2, self.feed_forwards, self.adapters_2):
            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)

            x = adapter_1(x) + x  # Add an adapter with a skip-connection after attention module

            h = x + h

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

            x = adapter_2(x) + x  # Add an adapter with a skip-connection after feed-forward module

            h = x + h
        return h
```

Hands-on: Model adaptation

Now we need to freeze the portions of our model we don't want to train.

We just indicate that no gradient is needed for the frozen parameters by setting *param.requires_grad* to *False* for the frozen parameters:

```
for name, param in adaptation_model.named_parameters():
    if 'embeddings' not in name and 'classification' not in name and 'adapters_1' not in name and 'adapters_2' not in name:

        param.detach_()
        param.requires_grad = False

    else:
        param.requires_grad = True

full_parameters = sum(p.numel() for p in adaptation_model.parameters())
trained_parameters = sum(p.numel() for p in adaptation_model.parameters() if p.requires_grad)

print(f"We will train {trained_parameters:3e} parameters out of {full_parameters:3e},"
      f" i.e. {100 * trained_parameters/full_parameters:.2f}%")
```

```
We will train 1.284961e+07 parameters out of 5.125265e+07, i.e. 25.07%
```

In our case we will train 25% of the parameters. The model is small & deep (many adapters) and we need to train the embeddings so the ratio stay quite high. For a larger model this ratio would be a lot lower.

Hands-on: Model adaptation

We use a hidden dimension of 32 for the adapters and a learning rate ten times higher for the fine-tuning (we have added quite a lot of newly initialized parameters to train from scratch).

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

```
Epoch [1/3]   [307/307] 100%| , loss=2.04e-01 [01:00<00:00]
Validation Epoch: 1 Error rate: 9.174311926605505
Epoch [2/3]   [307/307] 100%| , loss=8.40e-02 [00:57<00:00]
Validation Epoch: 2 Error rate: 7.522935779816509
Epoch [3/3]   [307/307] 100%| , loss=4.83e-02 [01:00<00:00]
Validation Epoch: 3 Error rate: 7.522935779816509
<ignite.engine.engine.State at 0x7ff4c60fd710>
```

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

```
Test Results - Error rate: 4.000
```

Results similar to full-fine-tuning case with advantage of training only 25% of the full model parameters.

For a small 50M parameters model this method is overkill ➪ for 300M–1.5B parameters models.

4.2.B – Optimization: What schedule?

We have decided which weights to update, but in which order and how should be update them?

Motivation: We want to **avoid overwriting useful pretrained information** and **maximize positive transfer**.

Related concept: **Catastrophic forgetting (McCloskey & Cohen, 1989; French, 1999)**
When a model forgets the task it was originally trained on.

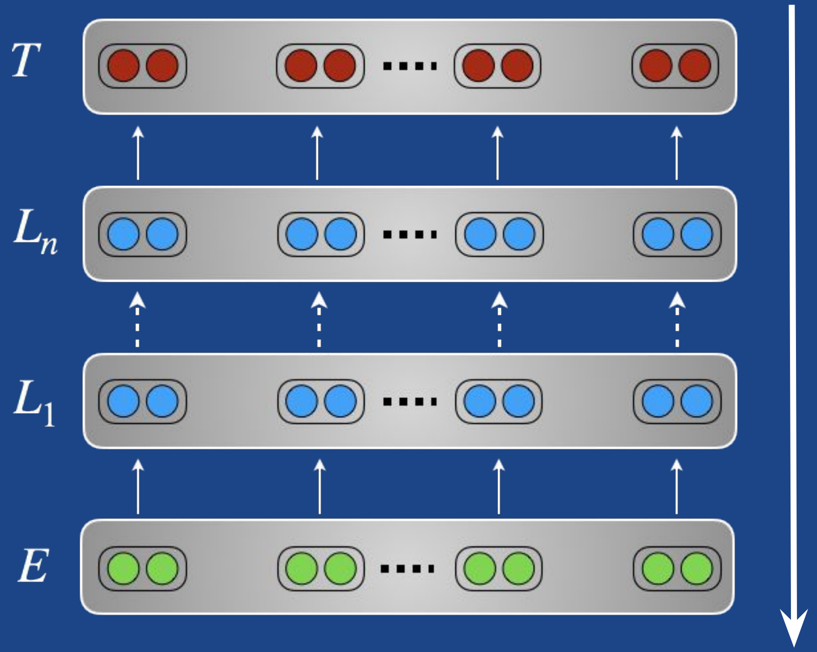
Image credit: Markus

4.2.B – Optimization: What schedule?

A guiding principle:

Update from top-to-bottom

- ❑ Progressively in **time**: freezing
- ❑ Progressively in **intensity**: Varying the learning rates
- ❑ Progressively vs. the **pretrained model**: Regularization

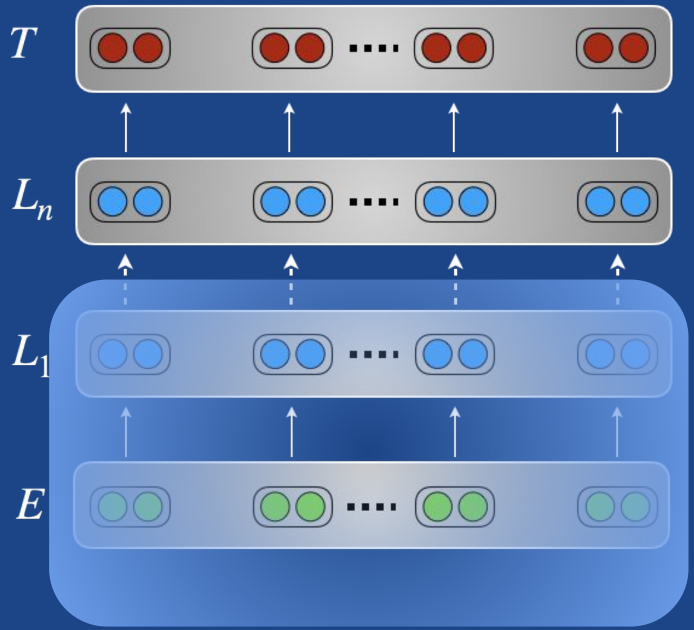


4.2.B – Optimization: Freezing

Main intuition: Training all layers at the same time on **data of a different distribution and task** may lead to instability and poor solutions.

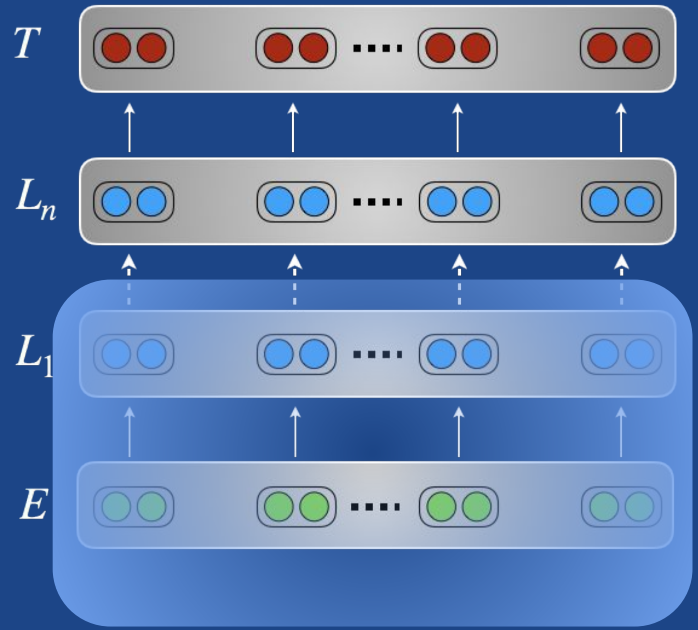
Solution: **Train layers individually** to give them time to adapt to new task and data.

Goes back to layer-wise training of early deep neural networks ([Hinton et al., 2006]; [Bengio et al., 2007]).



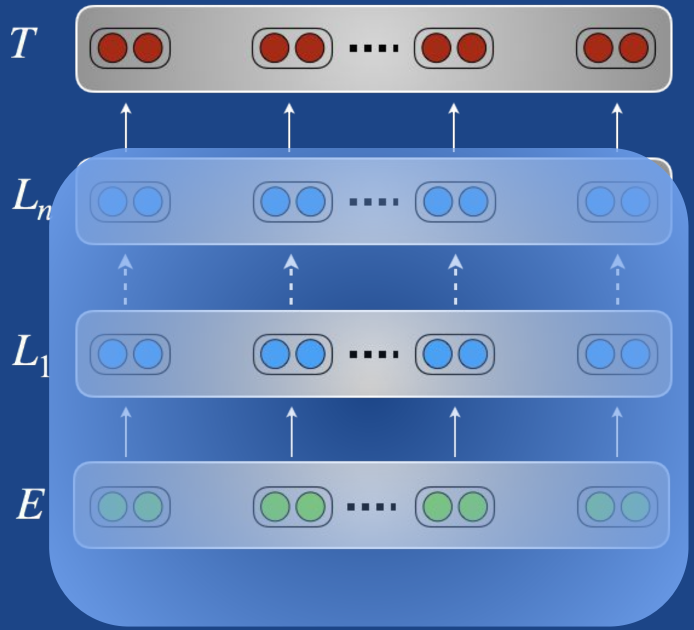
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)



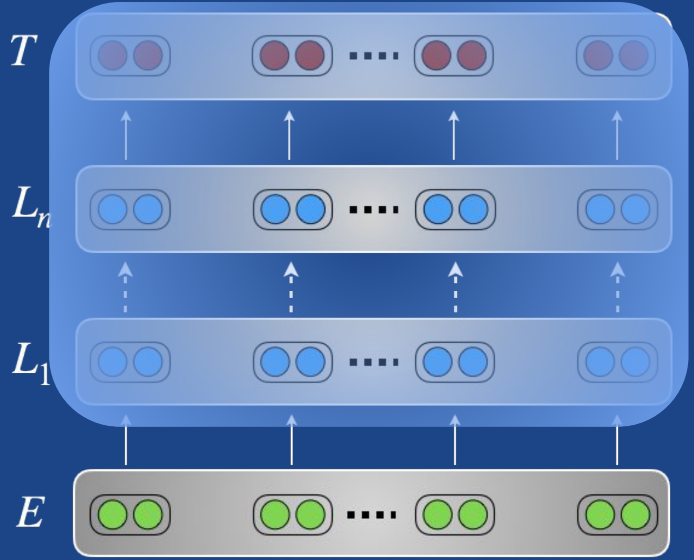
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer



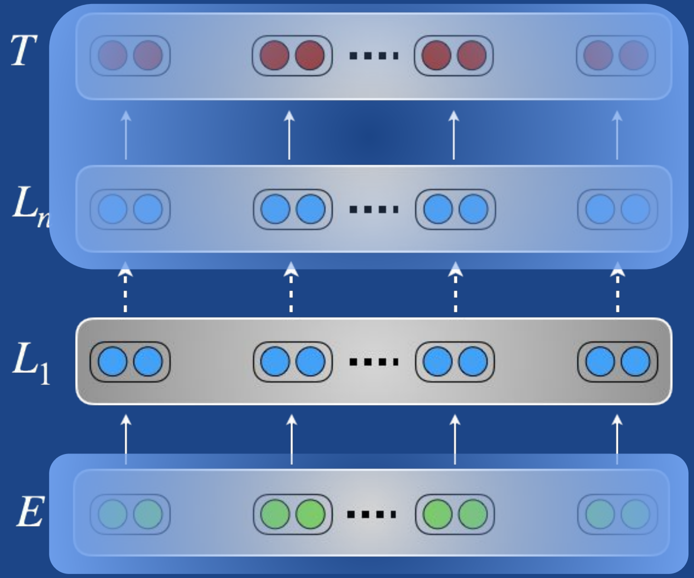
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time



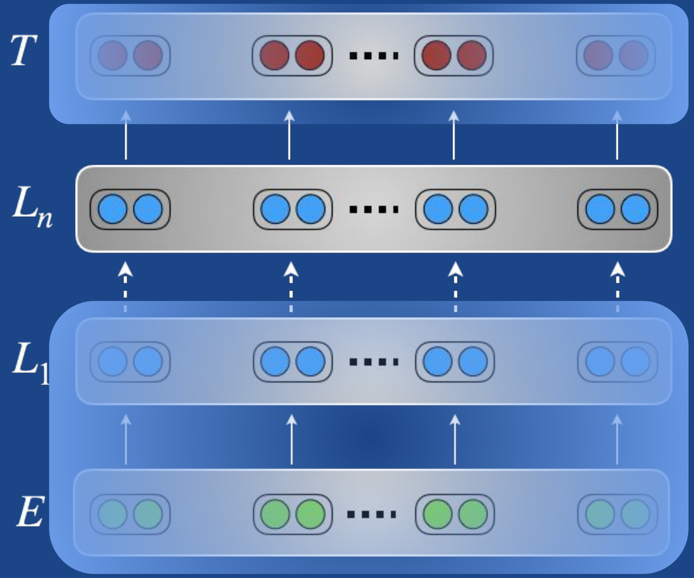
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time



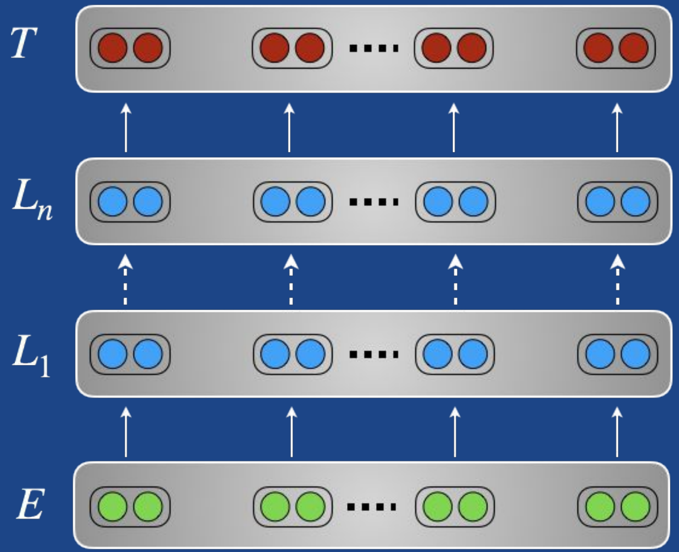
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time



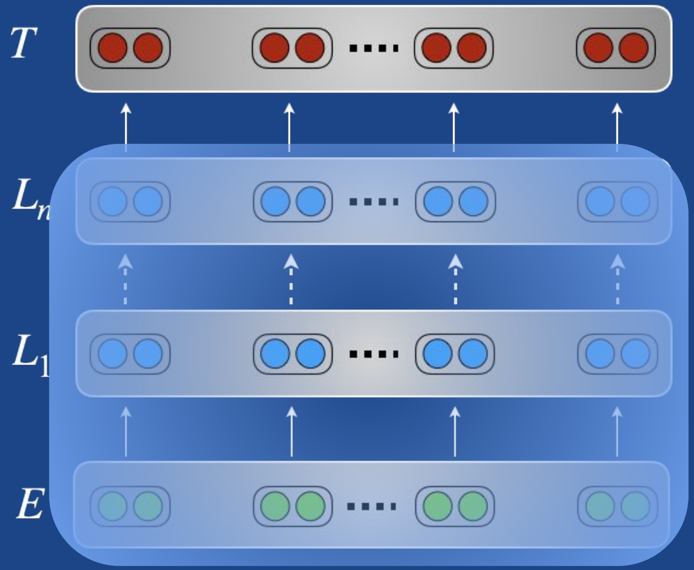
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time
  3. Train all layers



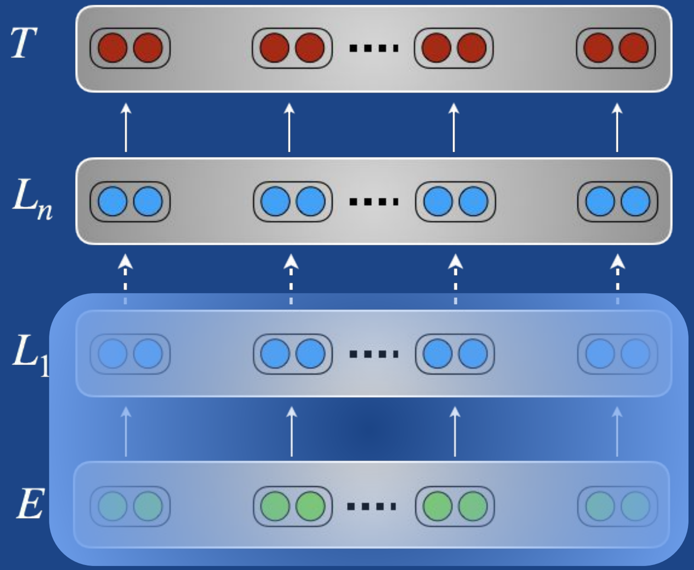
4.2.B – Optimization: Freezing

- ❑ Freezing all but the top layer (Long et al., ICML 2015)
- ❑ Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- ❑ Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another



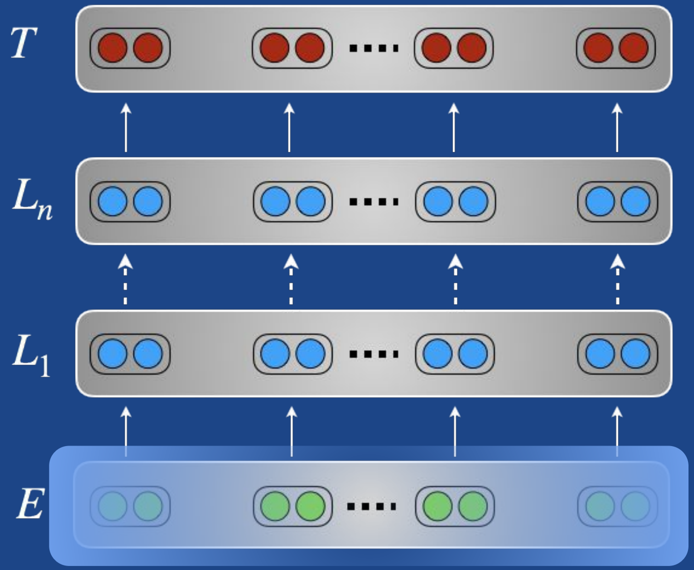
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another



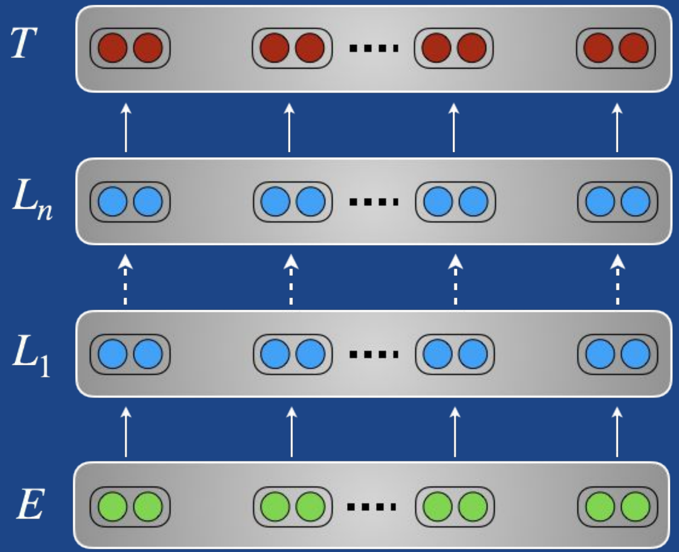
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another



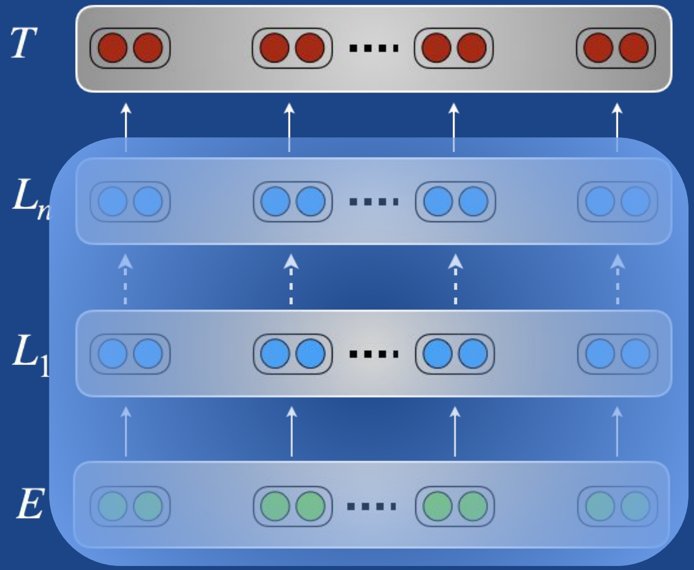
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another



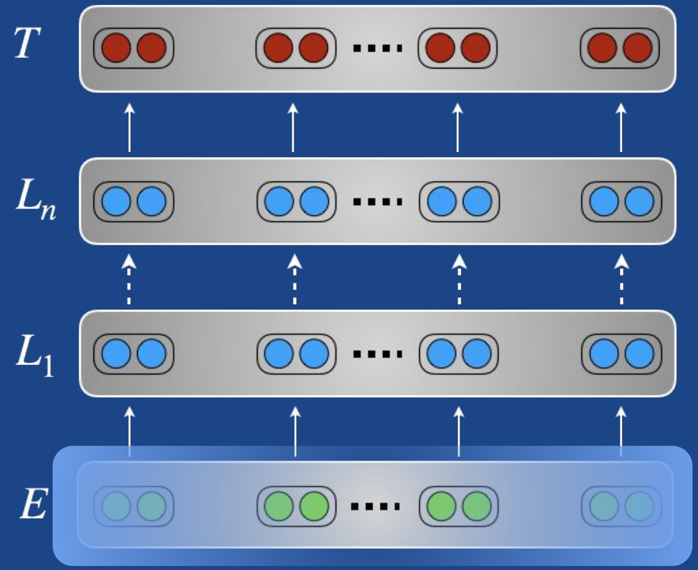
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning
  1. Fine-tune additional parameters for n epochs



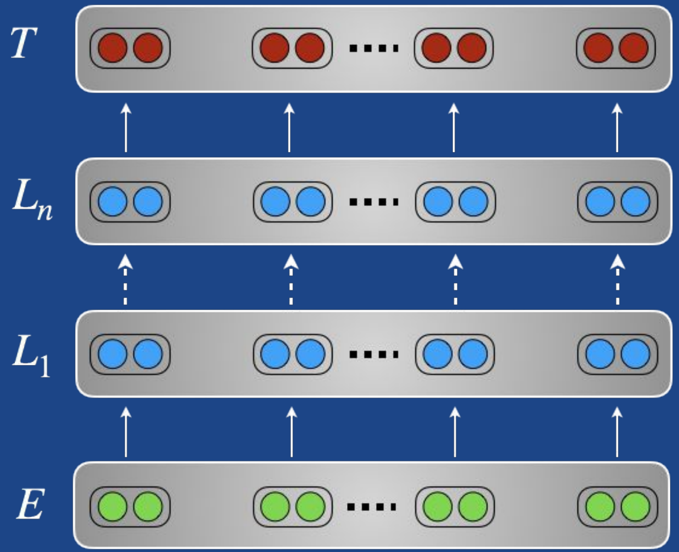
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning
  1. Fine-tune additional parameters for n epochs
  2. Fine-tune pretrained parameters without embedding layer for k epochs



4.2.B – Optimization: Freezing

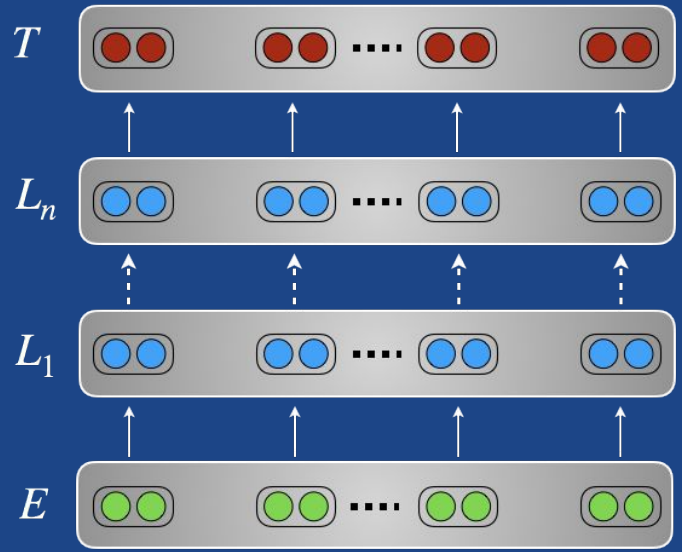
- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning
    1. Fine-tune additional parameters for n epochs
    2. Fine-tune pretrained parameters without embedding layer for k epochs
    3. Train all layers until convergence



4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning

Commonality: **Train all parameters jointly** in the end



Hands-on #4:
Using gradual unfreezing

Image credit: Chanaky

Hands-on: Adaptation

Gradual unfreezing is similar to our previous freezing process.
We start by freezing all the model except the newly added parameters:

```
for name, param in adaptation_model.named_parameters():
    if 'embeddings' not in name and 'classification' not in name:
        param.detach_()
        param.requires_grad = False

    else:
        param.requires_grad = True

full_parameters = sum(p.numel() for p in adaptation_model.parameters())
trained_parameters = sum(p.numel() for p in adaptation_model.parameters() if p.requires_grad)

print(f"We will start by training {trained_parameters:3e} parameters out of {full_parameters:3e},"
      f" i.e. {100 * trained_parameters/full_parameters:.2f}%")
```

```
We will start by training 1.199579e+07 parameters out of 5.039883e+07, i.e. 23.80%
```

We then gradually unfreeze an additional block along the training so that we train the full model at the end:

Unfreezing interval

Find index of layer to unfreeze

Name pattern matching

```
import re
# We will unfreeze blocks regularly along the training: one block every `unfreezing_interval` step
unfreezing_interval = int(len(train_loader) * adapt_args.n_epochs / (args.num_layers + 1))

@trainer.on(Events.ITERATION_COMPLETED)
def unfreeze_layer_if_needed(engine):
    if engine.state.iteration % unfreezing_interval == 0:
        # Which layer should we unfreeze now
        unfreezing_index = args.num_layers - (engine.state.iteration // unfreezing_interval)

        # Let's unfreeze it
        unfreezed = []
        for name, param in adaptation_model.named_parameters():
            if re.match(r"transformer\.[^\.]*\." + str(unfreezing_index) + r"\.", name):
                unfreezed.append(name)
                param.require_grad = True
        print(f"Unfreezing block {unfreezing_index} with {unfreezed}")
```

Hands-on: Adaptation

Gradual unfreezing has not been investigated in details for Transformer models
- ➩ no specific hyper-parameters advocated in the literature

Residual connections may have an impact on the method
- ➩ should probably adapt LSTM hyper-parameters

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

```
Epoch [1/3]    [307/307] 100%| , loss=7.56e-02 [00:57<00:00]
Unfreezing block 15 with ['transformer.attentions.15.in_proj_weight', 'transformer.attentions.15.in_proj_bias
Unfreezing block 14 with ['transformer.attentions.14.in_proj_weight', 'transformer.attentions.14.in_proj_bias
Unfreezing block 13 with ['transformer.attentions.13.in_proj_weight', 'transformer.attentions.13.in_proj_bias
Unfreezing block 12 with ['transformer.attentions.12.in_proj_weight', 'transformer.attentions.12.in_proj_bias
Unfreezing block 11 with ['transformer.attentions.11.in_proj_weight', 'transformer.attentions.11.in_proj_bias
Validation Epoch: 1 Error rate: 7.706422018348624
Epoch [2/3]    [307/307] 100%| , loss=2.27e-02 [00:59<00:00]
Unfreezing block 10 with ['transformer.attentions.10.in_proj_weight', 'transformer.attentions.10.in_proj_bias
Unfreezing block 9 with ['transformer.attentions.9.in_proj_weight', 'transformer.attentions.9.in_proj_bias',
Unfreezing block 8 with ['transformer.attentions.8.in_proj_weight', 'transformer.attentions.8.in_proj_bias',
Unfreezing block 7 with ['transformer.attentions.7.in_proj_weight', 'transformer.attentions.7.in_proj_bias',
Unfreezing block 6 with ['transformer.attentions.6.in_proj_weight', 'transformer.attentions.6.in_proj_bias',
Unfreezing block 5 with ['transformer.attentions.5.in_proj_weight', 'transformer.attentions.5.in_proj_bias',
Validation Epoch: 2 Error rate: 6.788990825688068
Epoch [3/3]    [307/307] 100%| , loss=5.05e-03 [00:56<00:00]
Unfreezing block 4 with ['transformer.attentions.4.in_proj_weight', 'transformer.attentions.4.in_proj_bias',
Unfreezing block 3 with ['transformer.attentions.3.in_proj_weight', 'transformer.attentions.3.in_proj_bias',
Unfreezing block 2 with ['transformer.attentions.2.in_proj_weight', 'transformer.attentions.2.in_proj_bias',
Unfreezing block 1 with ['transformer.attentions.1.in_proj_weight', 'transformer.attentions.1.in_proj_bias',
Unfreezing block 0 with ['transformer.attentions.0.in_proj_weight', 'transformer.attentions.0.in_proj_bias',
Unfreezing block -1 with []
Validation Epoch: 3 Error rate: 7.339449541284404
<ignite.engine.engine.State at 0x7ff4c61999e8>
```

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

```
Test Results - Error rate: 5.200
```

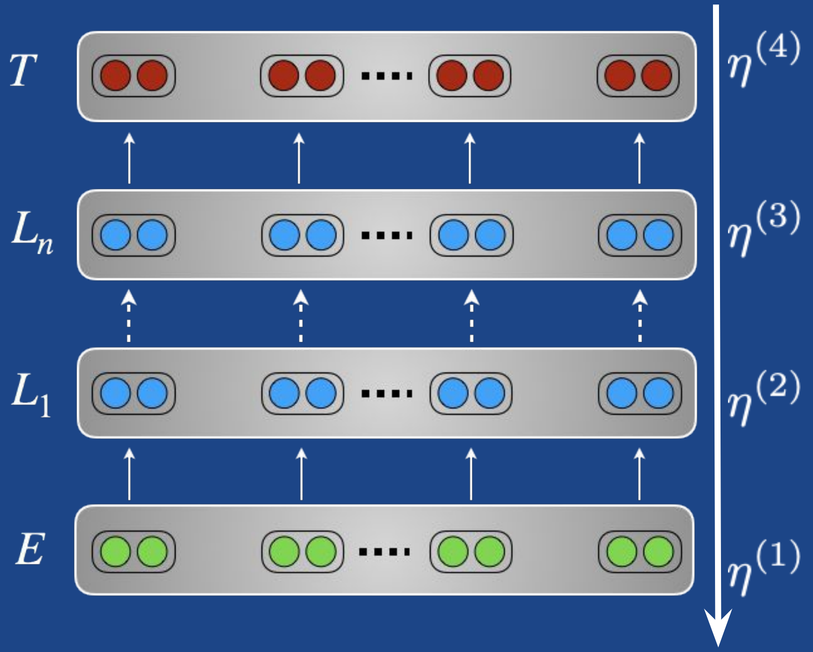
We show simple experiments in the Colab. Better hyper-parameters settings can probably be found.

4.2.B – Optimization: Learning rates

Main idea: Use **lower learning rates** to **avoid overwriting** useful information.

Where and when?

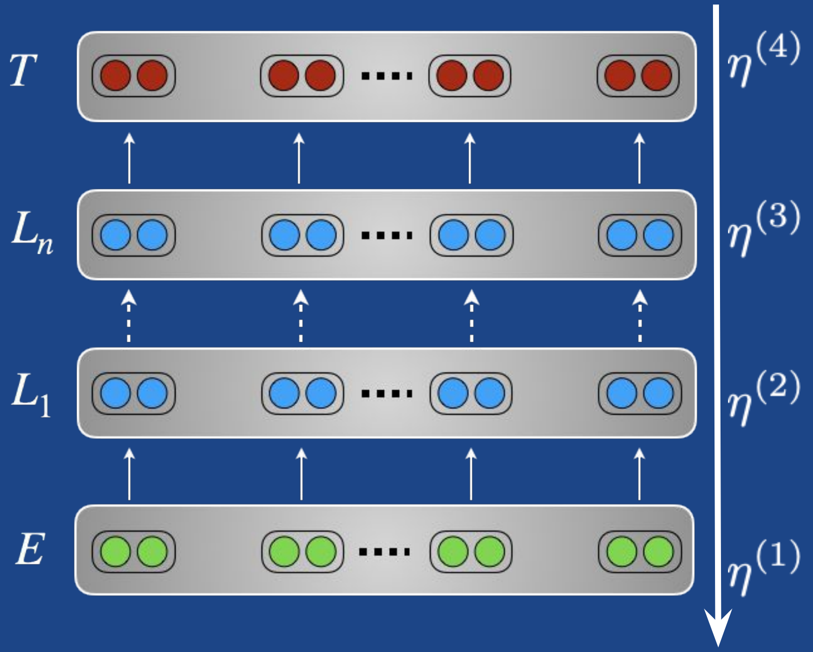
- ❏ **Lower layers** (capture general information)
- ❏ **Early** in training (model still needs to adapt to target distribution)
- ❏ **Late** in training (model is close to convergence)



4.2.B – Optimization: Learning rates

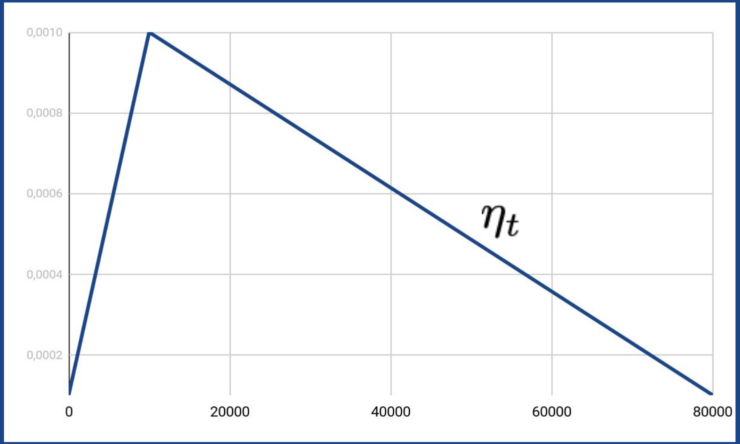
- Discriminative fine-tuning ([Howard & Ruder, ACL 2018]())
  - Lower layers capture general information
    → Use lower learning rates for lower layers

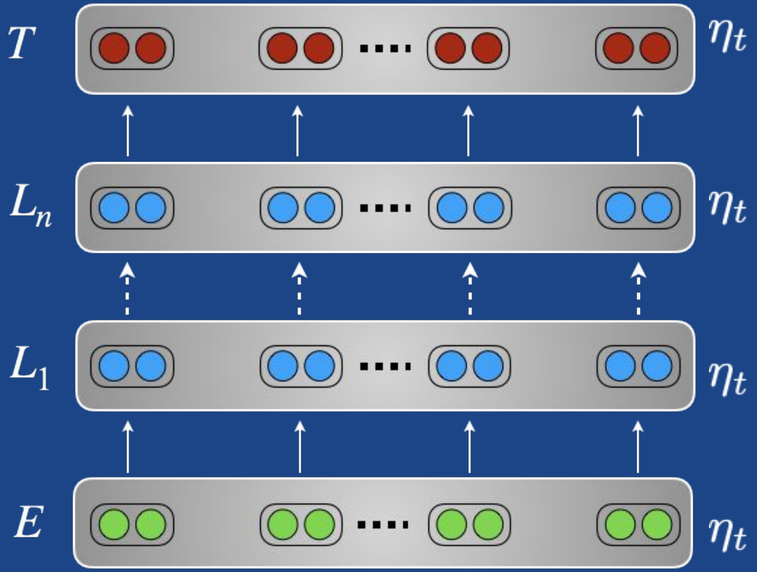
$$\eta^{(i)} = \eta \times d_f^{-i}$$



4.2.B – Optimization: Learning rates

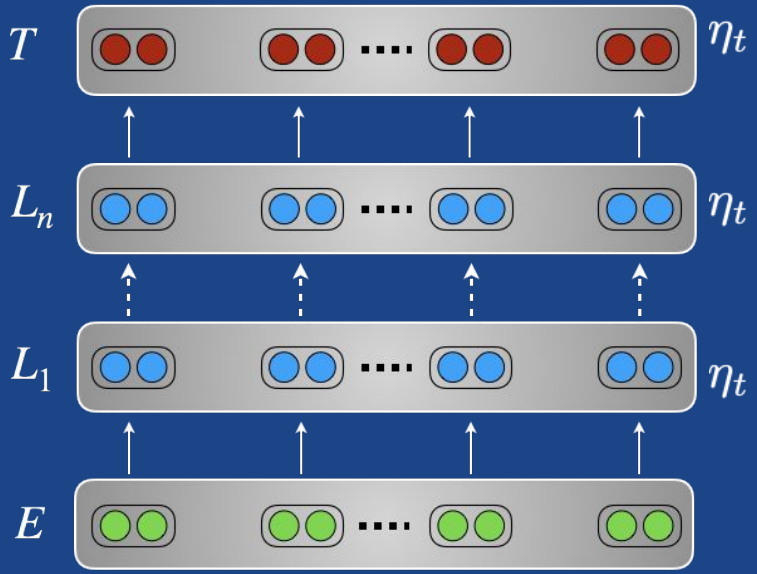
- Discriminative fine-tuning
- Triangular learning rates ([Howard & Ruder, ACL 2018]())
  - Quickly move to a suitable region, then slowly converge over time





4.2.B – Optimization: Learning rates

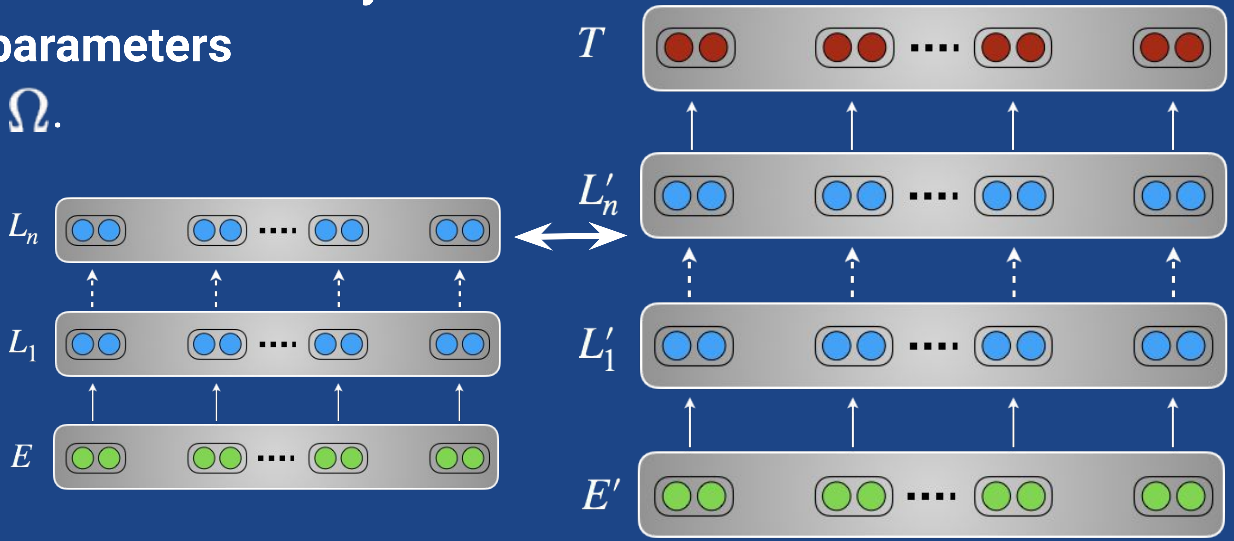
- Discriminative fine-tuning
- Triangular learning rates (Howard & Ruder, ACL 2018)
  - Quickly move to a suitable region, then slowly converge over time
  - Also known as “learning rate warm-up”
  - Used e.g. in Transformer (Vaswani et al., NIPS 2017) and Transformer-based methods (BERT, GPT)
  - Facilitates optimization; easier to escape suboptimal local minima



η_t

4.2.B – Optimization: Regularization

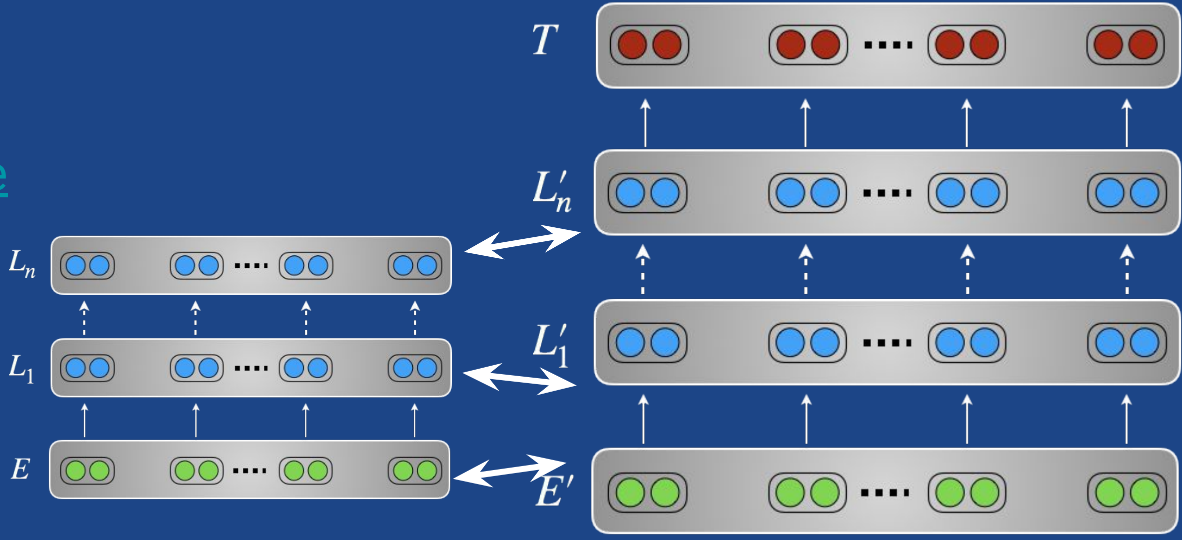
Main idea: minimize catastrophic forgetting by encouraging target model parameters to **stay close to pretrained model parameters** using a regularization term Ω.



4.2.B – Optimization: Regularization

- *Simple method*:
  Regularize new parameters **not to deviate too much from pretrained ones** (Wiese et al., CoNLL 2017):

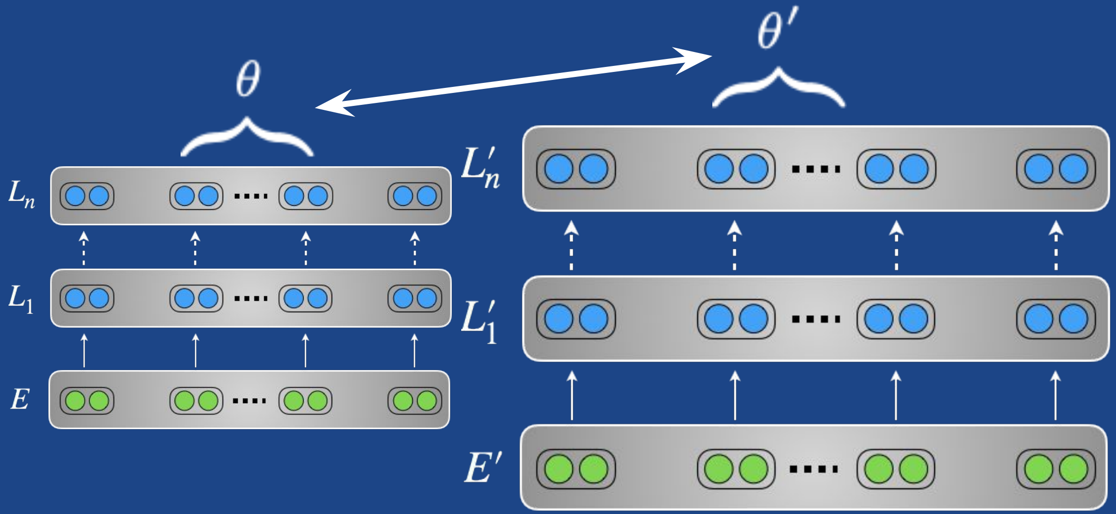
$$\Omega = \Sigma_1 \|L_i - L_i'\|_2$$



4.2.B – Optimization: Regularization

- *More advanced* (elastic weight consolidation; **EWC**): Focus on parameters θ that are **important for the pretrained task** based on the Fisher information matrix F ([Kirkpatrick et al., PNAS 2017](#)):

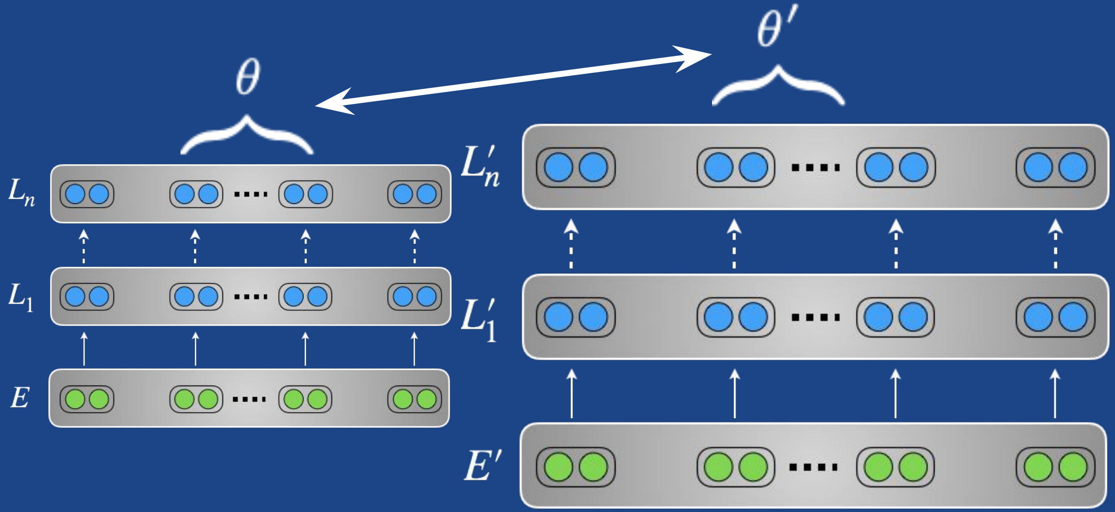
  $$\Omega = \sum_i \frac{\lambda}{2} F_i(\theta'_i - \theta_i)^2$$



4.2.B – Optimization: Regularization

EWC has downsides in continual learning:

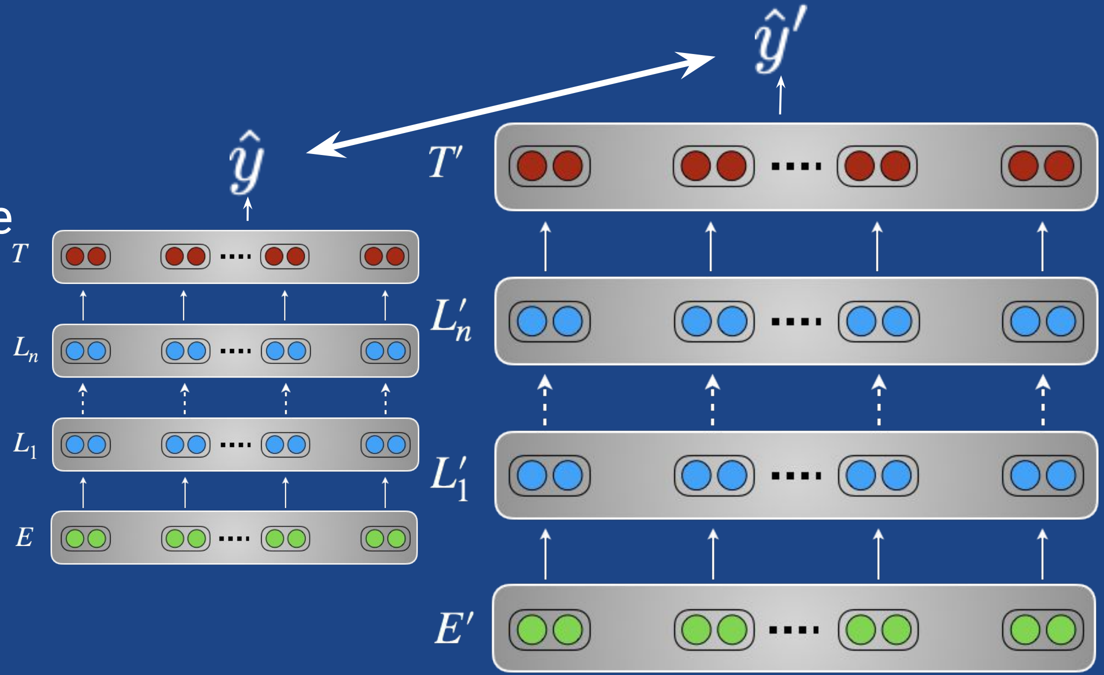
- ❏ May over-constrain parameters
- ❏ Computational cost is linear in the number of tasks ([Schwarz et al., ICML 2018](#))



4.2.B – Optimization: Regularization

- If tasks are similar, we may also encourage source and target predictions to be close based on cross-entropy, similar to distillation:

$$\Omega = \mathcal{H}(\hat{y}, \hat{y}')$$

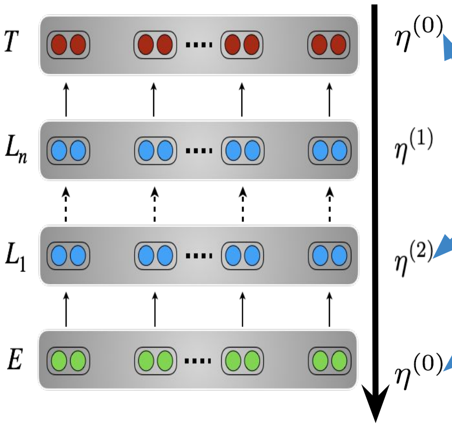


Hands-on #5:
Using discriminative learning

Image credit: Chanaky

Hands-on: Model adaptation

Discriminative learning rate can be implemented using two steps in our example:
First we organize the parameters of the various layers in labelled parameters groups in the optimizer:




```python
import re

# Build parameters groups by layer, numbered from the top ['1', '2', ..., '15']
parameter_groups = []
for i in range(args.num_layers):
    name_pattern = r"transformer\.[^\.]*\." + str(i) + r"\."
    group = {'name': str(args.num_layers - i),
             'params': [p for n, p in adaptation_model.named_parameters() if re.match(name_pattern, n)]}
    parameter_groups.append(group)

# Add the rest of the parameters (embeddings and classification layer) in a group labeled '0'
name_pattern = r"transformer\.[^\.]*\.\d*\."
group = {'name': '0',
         'params': [p for n, p in adaptation_model.named_parameters() if not re.match(name_pattern, n)]}
parameter_groups.append(group)

# Sanity check that we still have the same number of parameters
assert sum(p.numel() for g in parameter_groups for p in g['params'])\
    == sum(p.numel() for p in adaptation_model.parameters())

optimizer = torch.optim.Adam(parameter_groups, lr=adapt_args.lr)
```

We can then compute the learning rate of each group depending on its label (at each training iteration):

$$\eta^i = \eta \times {d_f}^{-i}$$

Hyper-parameter

```python
@trainer.on(Events.ITERATION_STARTED)
def update_layer_learning_rates(engine):
    for param_group in optimizer.param_groups:
        layer_index = int(param_group["name"])
        param_group["lr"] = param_group["lr"] / (adapt_args.decreasing_factor ** layer_index)
```

4.2.C – 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**

Image credit: Alfredo

4.2.C – Optimization trade-offs: Space

Task-specific modifications

Feature extraction **Adapters** **Fine-tuning**

Many ⟷ Few

Additional parameters

Feature extraction **Adapters** **Fine-tuning**

Many ⟷ Few

Parameter reuse

Feature extraction **Adapters** **Fine-tuning**

All ⟷ None

4.2.C – Optimization trade-offs: Time

Training time

Feature extraction **Adapters** **Fine-tuning**

Slow ⟷ Fast

4.2.C – 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
- Anecdotally, Transformers are easier to fine-tune (less sensitive to hyper-parameters) than LSTMs

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

4.3 – Getting more signal

The target task is often a **low-resource** task. We can often improve the performance of transfer learning by combining a diverse set of signals:

A. From **fine-tuning** a single model on a single adaptation task....
The Basic: fine-tuning the model with a simple classification objective

B. ... to **gathering signal** from other datasets and related tasks ...
Fine-tuning with Weak Supervision, Multi-tasking and Sequential Adaptation

C. ... to **ensembling** models
Combining the predictions of several fine-tuned models

Image credit: Naveen

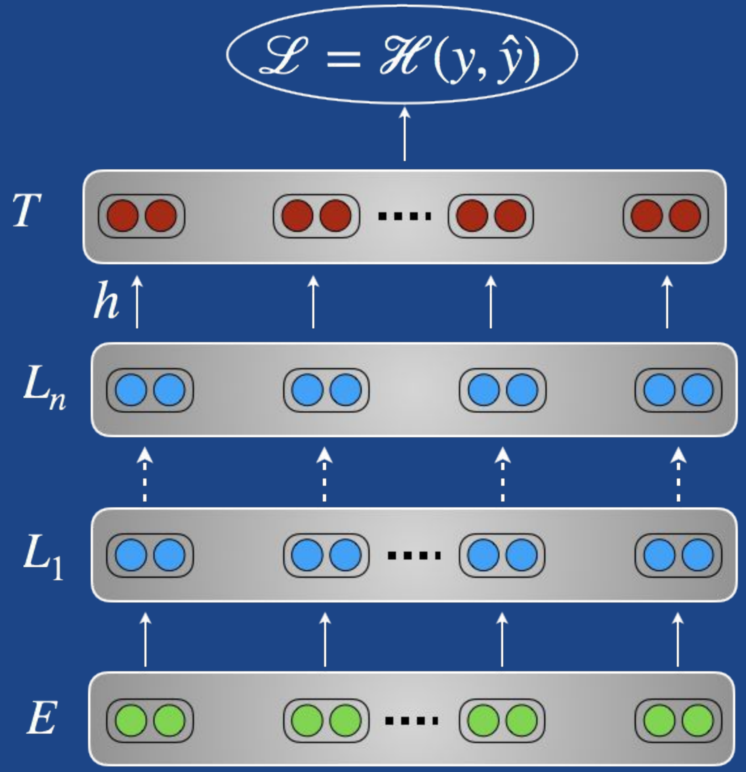
4.3.A – Getting more signal: Basic fine-tuning

Simple example of fine-tuning on a text classification task:

A. Extract a single fixed-length vector from the model:
   hidden state of first/last token or mean/max of hidden-states

B. Project to the classification space with an additional classifier

C. Train with a classification objective



4.3.B – Getting more signal: Related datasets/tasks

A. Sequential adaptation
   Intermediate fine-tuning on related datasets and tasks

B. Multi-task fine-tuning with related tasks
   Such as NLI tasks in GLUE

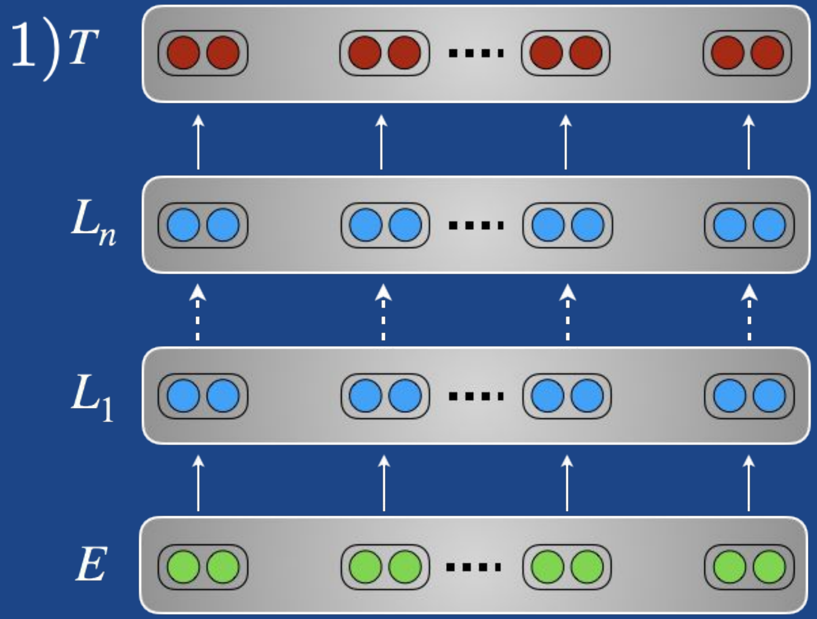
C. Dataset Slicing
   When the model consistently underperforms on particular slices of the data

D. Semi-supervised learning
   Use unlabelled data to improve model consistency

4.3.B – Getting more signal: Sequential adaptation

Fine-tuning on related high-resource dataset

1. Fine-tune model on related task with more data

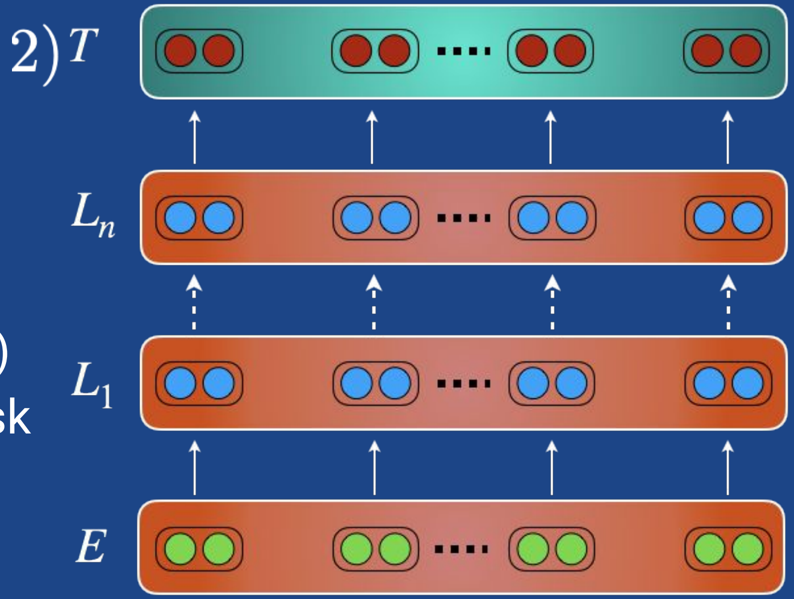


4.3.B – Getting more signal: Sequential adaptation

Fine-tuning on related high-resource dataset

1. Fine-tune model on related task with more data
2. Fine-tune model on target task

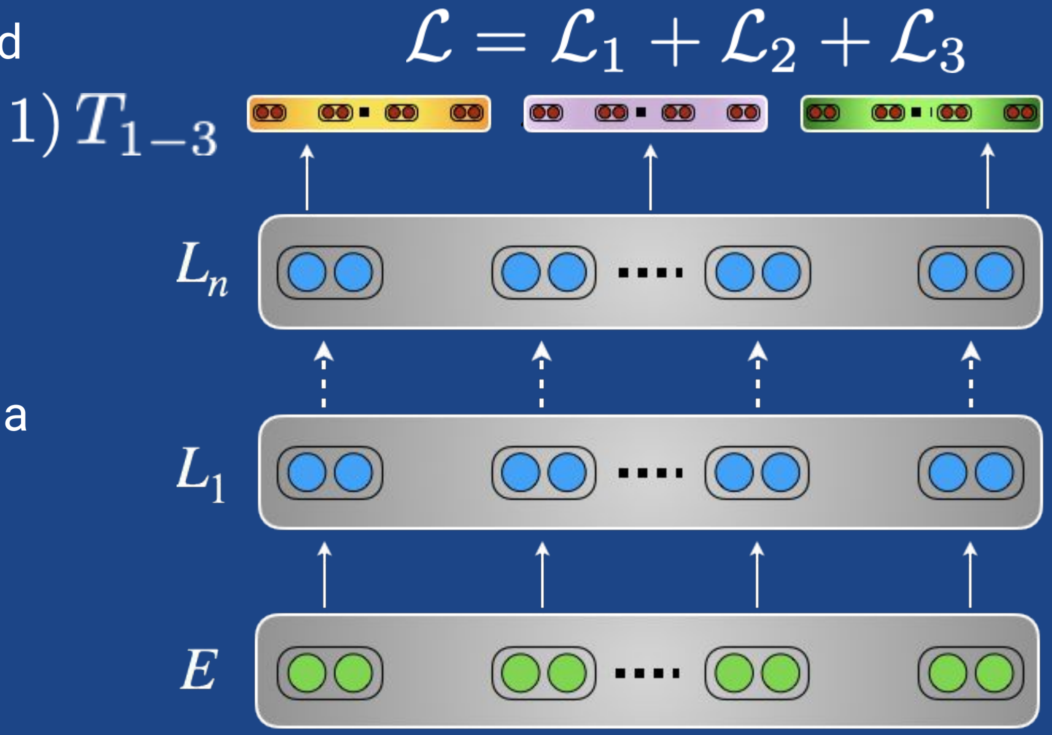
- ❏ Helps particularly for tasks with limited data and similar tasks ([Phang et al., 2018](#))
- ❏ Improves sample complexity on target task ([Yogatama et al., 2019](#))



4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model jointly on related tasks

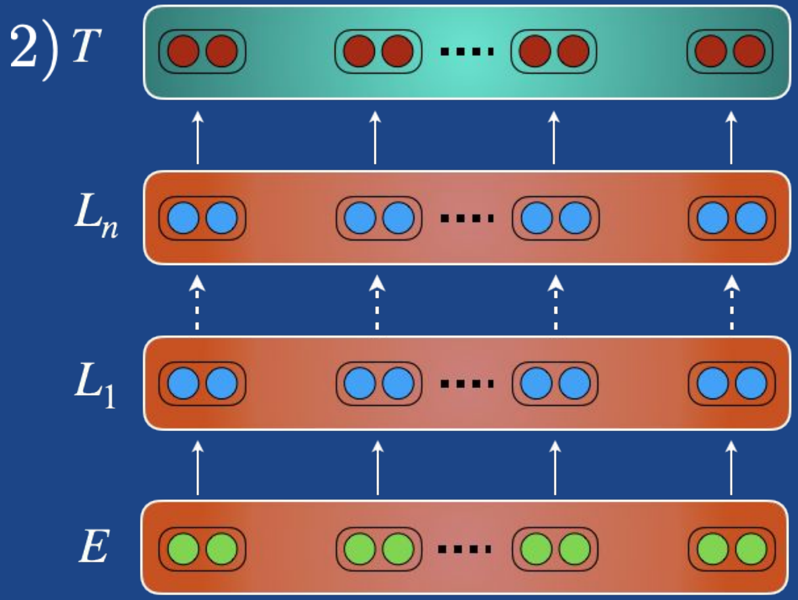
- ❏ For each optimization step, sample a task and a batch for training.
- ❏ Train via multi-task learning for a couple of epochs.



4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model jointly on related tasks

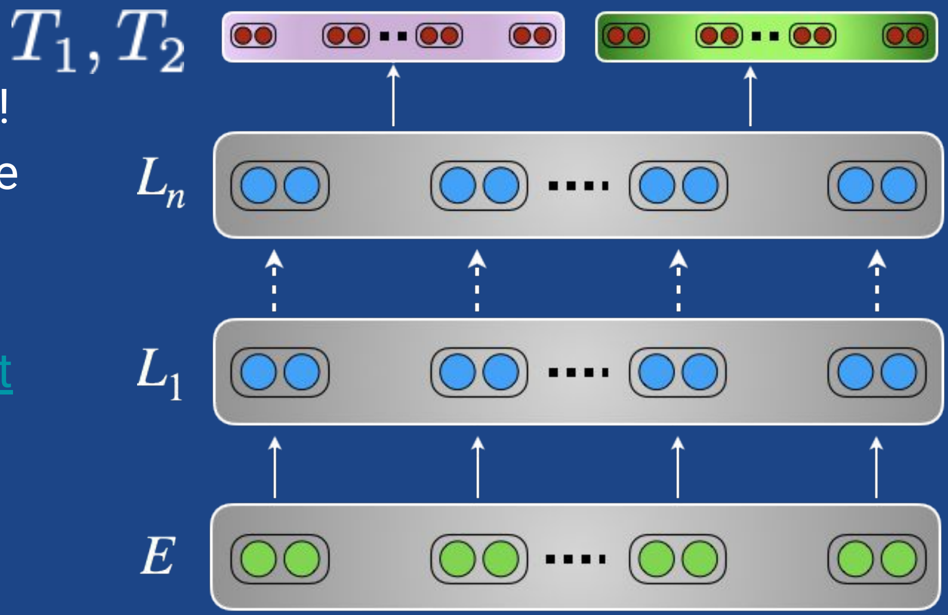
- For each optimization step, sample a task and a batch for training.
- Train via multi-task learning for a couple of epochs.
- Fine-tune on the target task only for a few epochs at the end.



4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model with an unsupervised auxiliary task

$$\mathcal{L} = \mathcal{L}_1 + \lambda \mathcal{L}_2$$



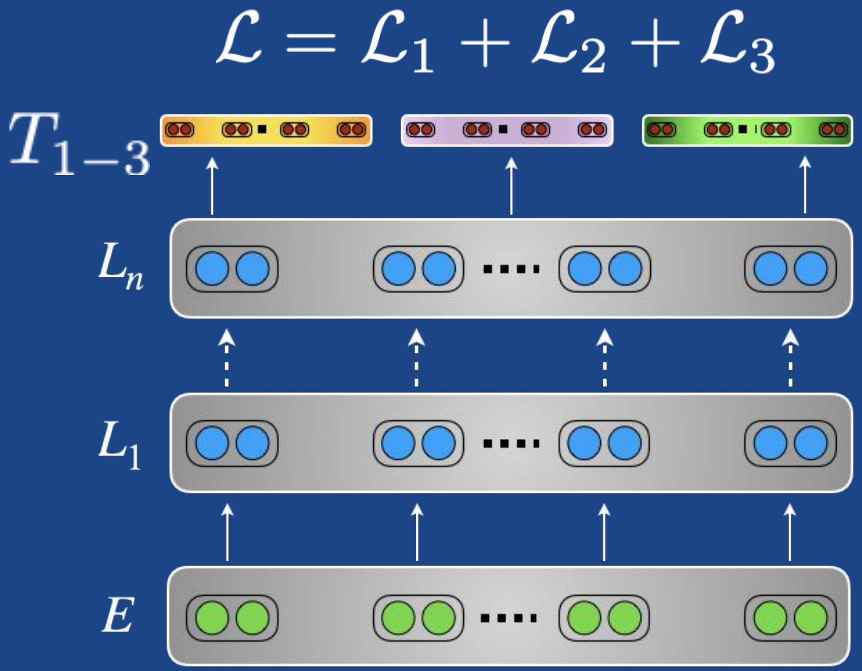
- ❏ Language modelling is a related task!
- ❏ Fine-tuning the LM helps adapting the pretrained parameters to the target dataset.
- ❏ Helps even without pretraining (Rei et al., ACL 2017)
- ❏ Can optionally anneal ratio λ (Chronopoulou et al., NAACL 2019)
- ❏ Used as a separate step in ULMFiT

4.3.B – Getting more signal: Dataset slicing

Use auxiliary heads that are trained **only on particular subsets** of the data

- ❑ Analyze errors of the model
- ❑ Use heuristics to automatically identify challenging subsets of the training data
- ❑ Train auxiliary heads jointly with main head

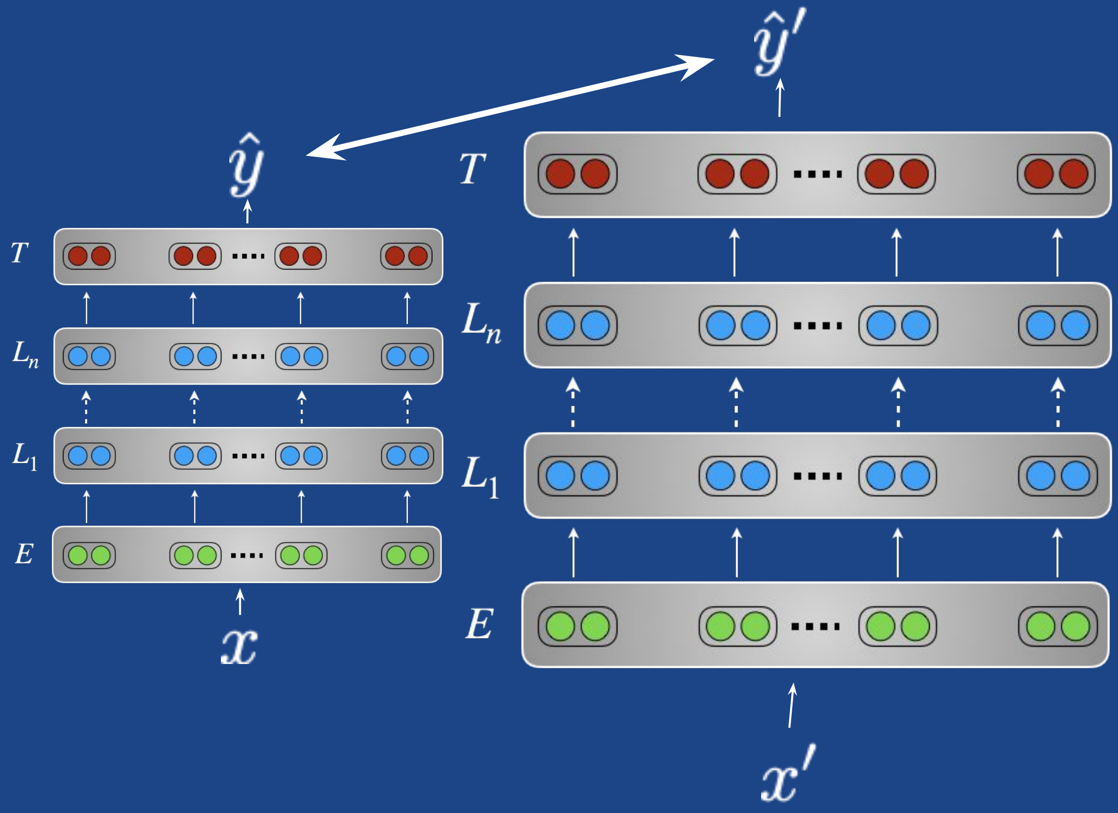
See also [Massive Multi-task Learning with Snorkel MeTaL]()



4.3.B – Getting more signal: Semi-supervised learning

Can be used to make model predictions **more consistent using unlabelled data**

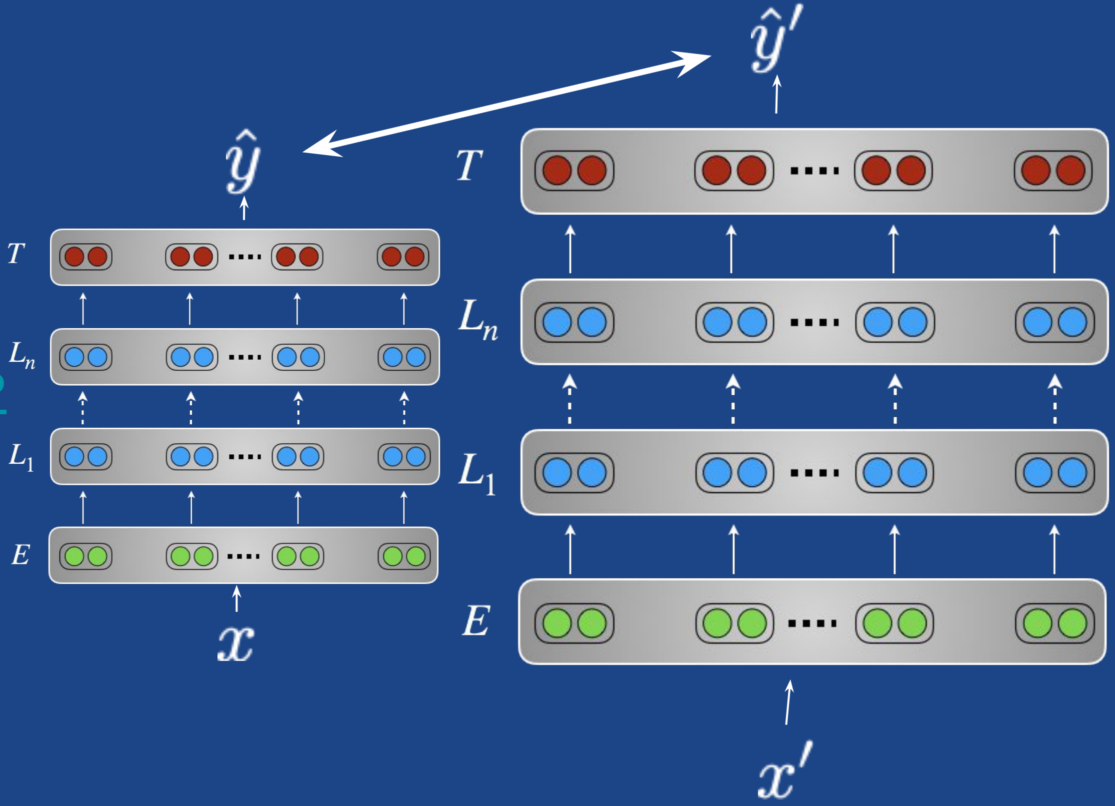
- Main idea: Minimize distance between predictions on original input x and perturbed input x'



4.3.B – Getting more signal: Semi-supervised learning

Can be used to make model predictions **more consistent using unlabelled data**

- Perturbation can be noise, masking (Clark et al., EMNLP 2018), data augmentation, e.g. back-translation (Xie et al., 2019)



4.3.C – Getting more signal: Ensembling

Reaching the state-of-the-art by ensembling independently fine-tuned models

- **Ensembling** models
  Combining the predictions of models fine-tuned with various hyper-parameters

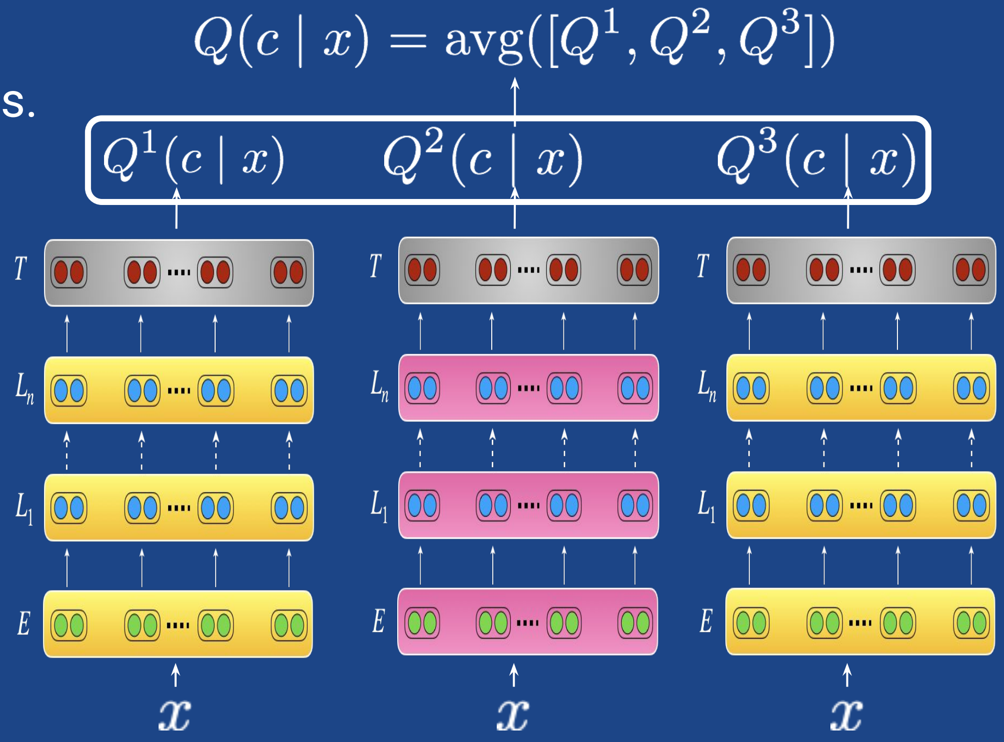
- **Knowledge distillation**
  Distill an ensemble of fine-tuned models in a single smaller model

4.3.C – Getting more signal: Ensembling

Combining the predictions of models fine-tuned with various hyper-parameters.

Model fine-tuned...

- ❑ on different tasks
- ❑ on different dataset-splits
- ❑ with different parameters (dropout, initializations...)
- ❑ from variant of pre-trained models (e.g. cased/uncased)



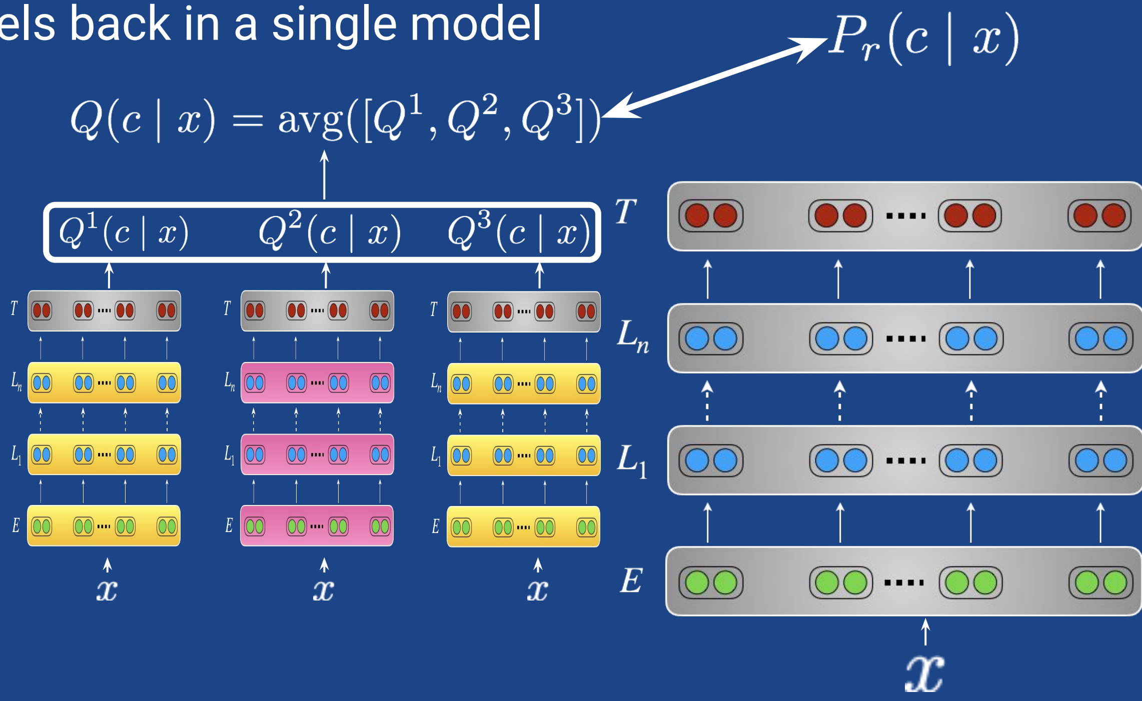
4.3.C – Getting more signal: Distilling

Distilling ensembles of large models back in a single model

- ❑ knowledge distillation: train a student model on soft targets produced by the teacher (the ensemble)

$$-\sum_{c} Q(c \mid X) \log(P_r(c \mid X))$$

- ❑ Relative probabilities of the teacher labels contain information about how the teacher generalizes



Hands-on #6:
Using multi-task learning

Image credit: Chanaky

Hands-on: Multi-task learning

Multitasking with a classification loss + language modeling loss.

Create **two heads**:
– language modeling head
– classification head

Total loss is a **weighted sum** of
– language modeling loss and
– classification loss

```python
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 [seq length, batch]

    _, losses = adaptation_model(inputs,
                                 clf_tokens_mask=(inputs == tokenizer.vocab['[CLS]']),
                                 clf_labels=labels,
                                 lm_labels=inputs,
                                 padding_mask=(batch == tokenizer.vocab['[PAD]']))

    clf_loss, lm_loss = losses
    loss = (adapt_args.clf_loss_coef * clf_loss
          + adapt_args.lm_loss_coef  * lm_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()
```

```python
class TransformerWithClfHeadAndLMHead(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,
                                       config.dropout, causal=not config.mlm)

        self.lm_head = nn.Linear(config.embed_dim, config.num_embeddings, bias=False)
        self.classification_head = nn.Linear(config.embed_dim, fine_tuning_config.num_classes)

        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):
        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, lm_labels=None, clf_labels=None, padding_mask=None):
        """ x and clf_tokens_mask have shape [seq length, batch] padding_mask has shape [batch, seq length] """
        hidden_states = self.transformer(x, padding_mask)

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

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

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

        if len(loss):
            return (lm_logits, clf_logits), loss

        return lm_logits, clf_logits
```

Hands-on: Multi-task learning

We use a coefficient of 1.0 for the classification loss and 0.5 for the language modeling loss and fine-tune a little longer (6 epochs instead of 3 epochs, the validation loss was still decreasing).

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

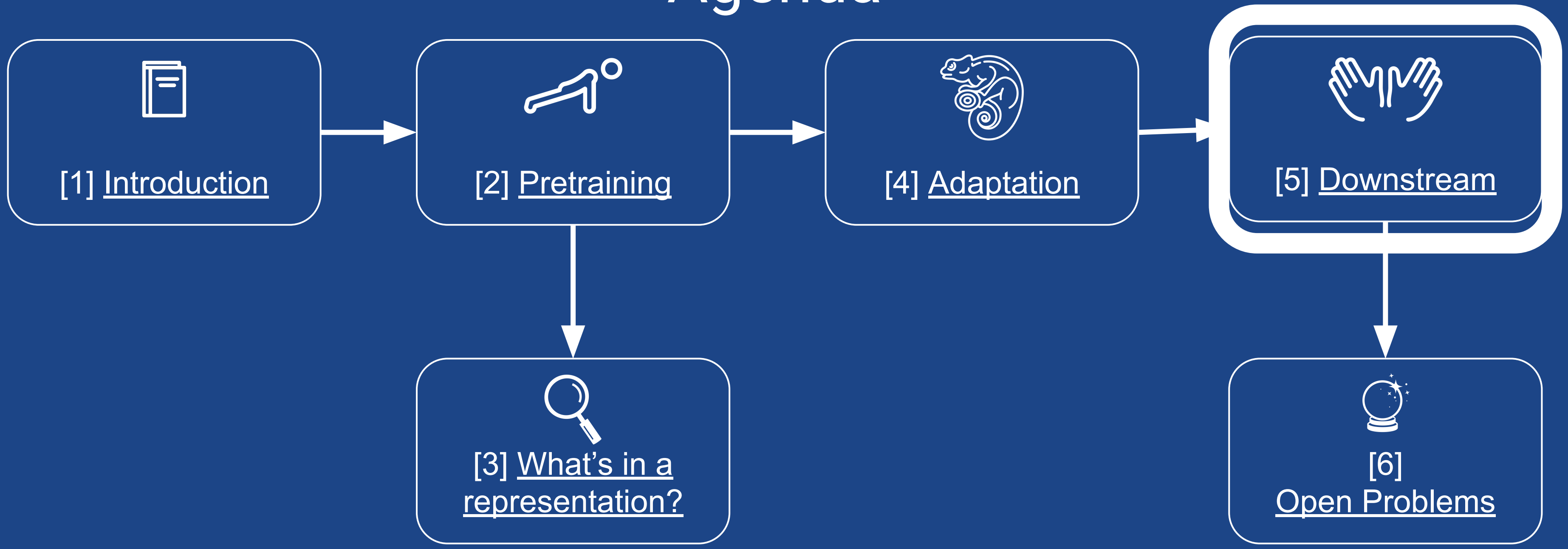
```
Epoch [1/6]  [307/307] 100%| , loss=1.07e+00 [01:21<00:00]
Validation Epoch: 1 Error rate: 9.357798165137611
Epoch [2/6]  [307/307] 100%| , loss=7.08e-01 [01:21<00:00]
Validation Epoch: 2 Error rate: 7.522935779816509
Epoch [3/6]  [307/307] 100%| , loss=5.46e-01 [01:22<00:00]
Validation Epoch: 3 Error rate: 5.688073394495408
Epoch [4/6]  [307/307] 100%| , loss=4.66e-01 [01:21<00:00]
Validation Epoch: 4 Error rate: 5.321100917431187
Epoch [5/6]  [307/307] 100%| , loss=4.22e-01 [01:21<00:00]
Validation Epoch: 5 Error rate: 5.688073394495408
Epoch [6/6]  [307/307] 100%| , loss=3.98e-01 [01:21<00:00]
Validation Epoch: 6 Error rate: 5.321100917431187
<ignite.engine.engine.State at 0x7ff4c9357e80>
```

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

```
Test Results - Error rate: 3.400
```

Multi-tasking helped us **improve** over single-task full-model fine-tuning!

Agenda


[1] Introduction
[2] Pretraining
[3] What's in a representation?
[4] Adaptation
[5] Downstream
[6] Open Problems

5. Downstream applications
Hands-on examples

Image credit: Fahmi

5. Downstream applications - Hands-on examples

In this section we will explore downstream applications and practical considerations along two orthogonal directions:

A. What are the various applications of transfer learning in NLP
Document/sequence classification, Token-level classification, Structured prediction and Language generation

B. How to leverage several frameworks & libraries for practical applications
Tensorflow, PyTorch, Keras and third-party libraries like fast.ai, HuggingFace...

Frameworks & libraries: practical considerations

- Pretraining large-scale models is costly
  Use open-source models
  Share your pretrained models
- Sharing/accessing pretrained models
  - **Hubs**: Tensorflow Hub, PyTorch Hub
  - **Author released** checkpoints: ex BERT, GPT...
  - **Third-party** libraries: AllenNLP, fast.ai, HuggingFace
- Design considerations
  - **Hubs/libraries**:
    - Simple to use but can be difficult to modify model internal architecture
  - **Author released checkpoints**:
    - More difficult to use but you have full control over the model internals

Consumption	**CO_2e (lbs)**
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11,023
American life, avg, 1 year	36,156
Car, avg incl. fuel, 1 lifetime	126,000
Training one model	
SOTA NLP model (tagging)	13
w/ tuning & experimentation	33,486
Transformer (large)	121
w/ neural architecture search	394,863

"Energy and Policy Considerations for Deep Learning in NLP" - Strubell, Ganesh, McCallum - ACL 2019

5. Downstream applications - Hands-on examples

A. Sequence and document level classification
Hands-on: Document level classification (fast.ai)

B. Token level classification
Hands-on: Question answering (Google BERT & Tensorflow/TF Hub)

C. Language generation
Hands-on: Dialog Generation (OpenAI GPT & HuggingFace/PyTorch Hub)

Icons credits: David, Susannanova, Flatart, ProSymbols

5.A – Sequence & document level classification

Transfer learning for document classification using the fast.ai library.

- Target task:
  IMDB: a binary sentiment classification dataset containing 25k highly polar movie reviews for training, 25k for testing and additional unlabeled data.
  http://ai.stanford.edu/~amaas/data/sentiment/
- Fast.ai has in particular:
  - a pre-trained English model available for download
  - a standardized data block API
  - easy access to standard datasets like IMDB
- Fast.ai is based on PyTorch

5.A – Document level classification using fast.ai

fast.ai gives access to many high-level API out-of-the-box for vision, text, tabular data and collaborative filtering.

The library is designed for speed of experimentation, e.g. by importing all necessary modules at once in interactive computing environments, like:

```
from fastai.text import *    # Quick access to NLP functionality
```

Fast.ai then comprises all the high level modules needed to quickly setup a transfer learning experiment.

Load IMDB dataset & inspect it.

```
path = untar_data(URLs.IMDB_SAMPLE)
print("Path:", path)
df = pd.read_csv(path/'texts.csv')
df.head()
```

Path: /root/.fastai/data/imdb_sample

	label	text	is_valid
0	negative	Un-bleeping-believable! Meg Ryan doesn't even ...	False
1	positive	This is a extremely well-made film. The acting...	False
2	negative	Every once in a long while a movie will come a...	False
3	positive	Name just says it all. I watched this movie wi...	False
4	negative	This movie succeeds at being one of the most u...	False

DataBunch for the language model and the classifier

```
data_lm = TextLMDataBunch.from_csv(path, 'texts.csv')
data_clas = TextClasDataBunch.from_csv(path, 'texts.csv',
                    vocab=data_lm.train_ds.vocab, bs=42)
```

Load an AWD-LSTM (Merity et al., 2017) pretrained on WikiText-103 & fine-tune it on IMDB using the language modeling loss.

```
moms = (0.8,0.7)
learn = language_model_learner(data_lm, AWD_LSTM)
learn.unfreeze()
learn.fit_one_cycle(4, slice(1e-2), moms=moms)
learn.save_encoder('enc')
```

epoch	train_loss	valid_loss	accuracy	time
0	4.723435	3.968737	0.283498	00:15
1	4.416326	3.874095	0.286878	00:15
2	4.148463	3.836543	0.290434	00:16
3	3.951989	3.828021	0.291311	00:16

5.A – Document level classification using fast.ai

Once we have a fine-tune language model (AWD-LSTM), we can create a text classifier by adding a classification head with:
– A layer to concatenate the final outputs of the RNN with the maximum and average of all the intermediate outputs (along the sequence length)
– Two blocks of *nn.BatchNorm1d* ⇨ *nn.Dropout* ⇨ *nn.Linear* ⇨ *nn.ReLU* with a hidden dimension of 50.

Now we fine-tune in two steps:

1. train the classification head only while keeping the language model frozen, and

2. fine-tune the whole architecture.

Colab: http://tiny.cc/NAACLTransferFastAiColab

```
learn = text_classifier_learner(data_clas, AWD_LSTM)
learn.load_encoder('enc')
learn.fit_one_cycle(4, moms=moms)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.663383	0.682115	0.572139	00:10
1	0.623683	0.609520	0.651741	00:10
2	0.597989	0.582999	0.666667	00:10
3	0.580533	0.555404	0.666667	00:09

```
learn.unfreeze()
learn.fit_one_cycle(8, slice(1e-5,1e-3), moms=moms)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.555569	0.557091	0.681592	00:20
1	0.566048	0.541689	0.721393	00:21
2	0.554564	0.543157	0.736318	00:20
3	0.556879	0.526971	0.756219	00:20
4	0.552898	0.522964	0.751244	00:19
5	0.541698	0.514611	0.756219	00:19
6	0.535575	0.514330	0.756219	00:19
7	0.529567	0.515582	0.746269	00:19

5.B – Token level classification: BERT & Tensorflow

Transfer learning for token level classification: Google's BERT in TensorFlow.

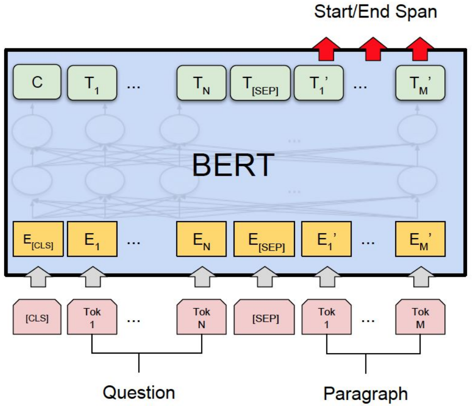
- Target task:
  SQuAD: a question answering dataset.
  https://rajpurkar.github.io/SQuAD-explorer/
- In this example we will directly use a Tensorflow checkpoint
  - Example: https://github.com/google-research/bert
  - We use the usual Tensorflow workflow: create model graph comprising the core model and the added/modified elements
  - Take care of variable assignments when loading the checkpoint

5.B – SQuAD with BERT & Tensorflow

Let's adapt BERT to the target task.

Keep our core model unchanged.

Replace the pre-training head (language modeling) with a classification head:
a linear projection layer to estimate 2 probabilities for each token:
– being the start of an answer
– being the end of an answer.




```
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids,
                 use_one_hot_embeddings):
  """Creates a classification model."""
  model = modeling.BertModel(
      config=bert_config,
      is_training=is_training,
      input_ids=input_ids,
      input_mask=input_mask,
      token_type_ids=segment_ids,
      use_one_hot_embeddings=use_one_hot_embeddings)

  final_hidden = model.get_sequence_output()

  final_hidden_shape = modeling.get_shape_list(final_hidden, expected_rank=3)
  batch_size = final_hidden_shape[0]
  seq_length = final_hidden_shape[1]
  hidden_size = final_hidden_shape[2]

  output_weights = tf.get_variable(
      "cls/squad/output_weights", [2, hidden_size],
      initializer=tf.truncated_normal_initializer(stddev=0.02))

  output_bias = tf.get_variable(
      "cls/squad/output_bias", [2], initializer=tf.zeros_initializer())

  final_hidden_matrix = tf.reshape(final_hidden,
                                   [batch_size * seq_length, hidden_size])
  logits = tf.matmul(final_hidden_matrix, output_weights, transpose_b=True)
  logits = tf.nn.bias_add(logits, output_bias)

  logits = tf.reshape(logits, [batch_size, seq_length, 2])
  logits = tf.transpose(logits, [2, 0, 1])

  unstacked_logits = tf.unstack(logits, axis=0)

  (start_logits, end_logits) = (unstacked_logits[0], unstacked_logits[1])

  return (start_logits, end_logits)
```

5.B – SQuAD with BERT & Tensorflow

Load our pretrained checkpoint

```
def get_assignment_map_from_checkpoint(tvars, init_checkpoint):
  """Compute the union of the current variables and checkpoint variables."""
  assignment_map = {}
  initialized_variable_names = {}

  name_to_variable = collections.OrderedDict()
  for var in tvars:
    name = var.name
    m = re.match("^(.*):\\d+$", name)
    if m is not None:
      name = m.group(1)
    name_to_variable[name] = var

  init_vars = tf.train.list_variables(init_checkpoint)

  assignment_map = collections.OrderedDict()
  for x in init_vars:
    (name, var) = (x[0], x[1])
    if name not in name_to_variable:
      continue
    assignment_map[name] = name
    initialized_variable_names[name] = 1
    initialized_variable_names[name + ":0"] = 1

  return (assignment_map, initialized_variable_names)
```

To load our checkpoint, we just need to setup an assignement_map from the variables of the checkpoint to the model variable, keeping only the variables in the model.

```
(start_logits, end_logits) = create_model(
    bert_config=bert_config,
    is_training=is_training,
    input_ids=input_ids,
    input_mask=input_mask,
    segment_ids=segment_ids,
    use_one_hot_embeddings=use_one_hot_embeddings)

tvars = tf.trainable_variables()

(assignment_map,
 initialized_variable_names) = get_assignment_map_from_checkpoint(tvars, init_checkpoint)

tf.train.init_from_checkpoint(init_checkpoint, assignment_map)
```

And we can use *tf.train.init_from_ckeckpoint*

5.B – SQuAD with BERT & Tensorflow

TensorFlow-Hub

Working directly with TensorFlow requires to have access to–and include in your code– the *full code of the pretrained model*.

TensorFlow Hub is a library for **sharing** machine learning models as *self-contained pieces of TensorFlow graph with their weights and assets.*

Modules are automatically downloaded and cached when instantiated.

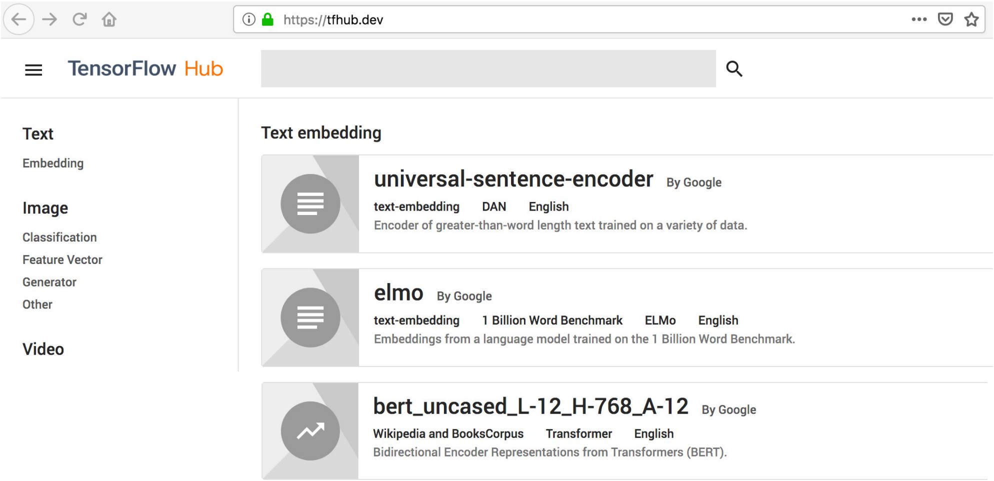
Each time a module *m* is called e.g. *y = m(x)*, it adds operations to the current TensorFlow graph to compute *y* from *x*.

```
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids,
                 use_one_hot_embeddings):
  """Creates a classification model."""

  model = modeling.BertModel(
      config=bert_config,
      is_training=is_training,
      input_ids=input_ids,
      input_mask=input_mask,
      token_type_ids=segment_ids,
      use_one_hot_embeddings=use_one_hot_embeddings)

  final_hidden = model.get_sequence_output()
```

```
!pip install "tensorflow_hub==0.4.0"

import tensorflow_hub as hub

def create_model(is_predicting, input_ids, input_mask, segment_        els,
                 num_labels):
  """Creates a classification model."""

  bert_module = hub.Module(
      BERT_MODEL_HUB,
      trainable=True)
  bert_inputs = dict(
      input_ids=input_ids,
      input_mask=input_mask,
      segment_ids=segment_ids)
  bert_outputs = bert_module(
      inputs=bert_inputs,
      signature="tokens",
      as_dict=True)

  # Use "pooled_output" for classification tasks on an entire sentence.
  # Use "sequence_outputs" for token-level output.
  final_hidden = bert_outputs["sequence_outputs"]
```

5.B – SQuAD with BERT & Tensorflow

Tensorflow Hub host a nice selection of pretrained models for NLP



Tensorflow Hub can also used with Keras exactly how we saw in the BERT example

The main limitations of Hubs are:

- ❏ No access to the source code of the model (*black-box*)
- ❏ Not possible to modify the internals of the model (*e.g. to add Adapters*)

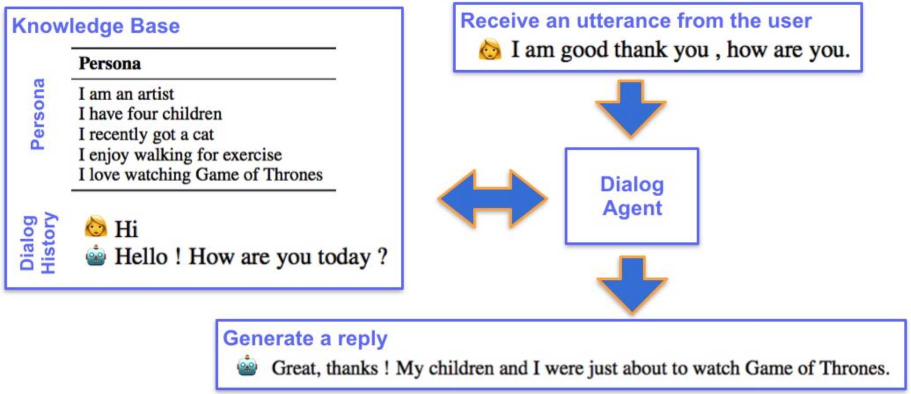
5.C – Language Generation: OpenAI GPT & PyTorch

Transfer learning for language generation: OpenAI GPT and HuggingFace library.

- ❏ Target task:
  ConvAI2 – The 2nd Conversational Intelligence Challenge for training and evaluating models for non-goal-oriented dialogue systems, i.e. chit-chat
  http://convai.io
- ❏ HuggingFace library of pretrained models
  - ❏ a repository of large scale pre-trained models with BERT, GPT, GPT-2, Transformer-XL
  - ❏ provide an easy way to download, instantiate and train pre-trained models in PyTorch
- ❏ HuggingFace's models are now also accessible using PyTorch Hub

5.C – Chit-chat with OpenAI GPT & PyTorch

A dialog generation task:



Language generation tasks are close to the language modeling pre-training objective, but:

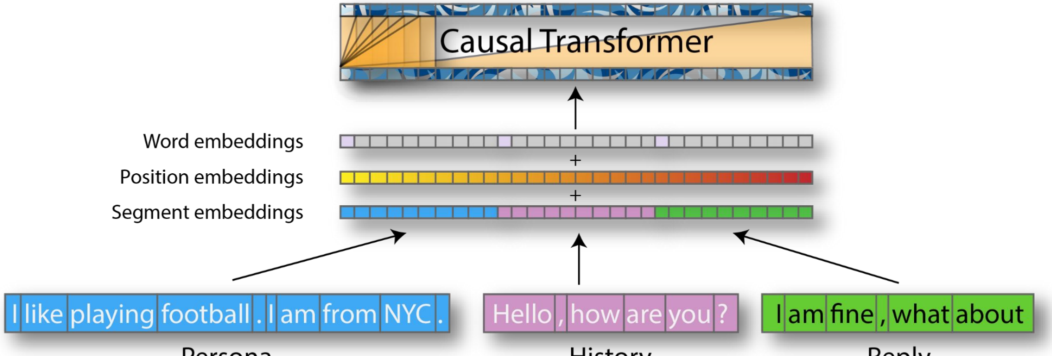
- ❏ Language modeling pre-training involves a single input: *a sequence of words.*
- ❏ In a dialog setting: several type of contexts are provided to generate an output sequence:
  - ❏ *knowledge base*: persona sentences,
  - ❏ *history of the dialog*: at least the last utterance from the user,
  - ❏ *tokens of the output sequence* that have already been generated.

How should we adapt the model?

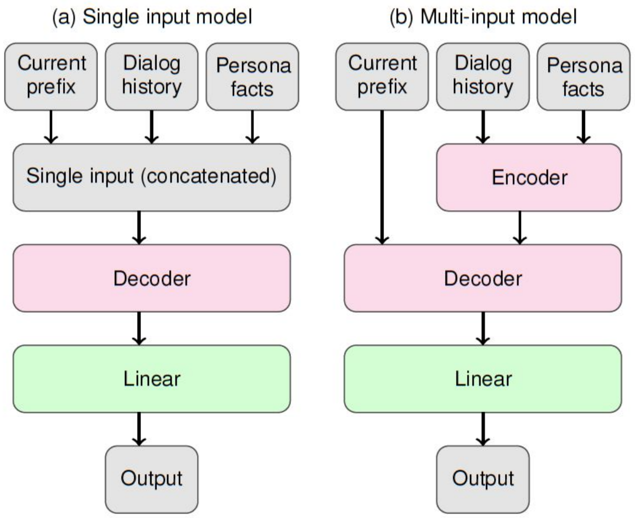
5.C – Chit-chat with OpenAI GPT & PyTorch

Several options:

- ❏ Duplicate the model to initialize an encoder-decoder structure
  e.g. Lample & Conneau, 2019
- ❏ Use a single model with concatenated inputs
  see e.g. Wolf et al., 2019, Khandelwal et al. 2019



Concatenate the various context separated by delimiters and add position and segment embeddings



Golovanov, Kurbanov, Nikolenko, Truskovskyi, Tselousov and Wolf, ACL 2019

5.C – Chit-chat with OpenAI GPT & PyTorch

Let's import pretrained versions of OpenAI GPT tokenizer and model.

```
from pytorch_pretrained_bert import OpenAIGPTLMHeadModel, OpenAIGPTTokenizer

model = OpenAIGPTLMHeadModel.from_pretrained('openai-gpt')
tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
```

And add a few new tokens to the vocabulary

```
# We use 5 special tokens: <bos>, <eos>, <speaker1>, <speaker2>, <pad>
# to indicate start/end of the input sequence, tokens from user/bot and padding
SPECIAL_TOKENS = ["<bos>", "<eos>", "<speaker1>", "<speaker2>", "<pad>"]

# Add these special tokens to the vocabulary and the embeddings of the model:
tokenizer.set_special_tokens(SPECIAL_TOKENS)
model.set_num_special_tokens(len(SPECIAL_TOKENS))
```

```
from itertools import chain

# Let's define our contexts and special tokens
persona_string = ["i like football", "i am from NYC"]
history_string = ["how are you ?", "pretty fine"]
reply_string = "great !"
bos, eos, speaker1, speaker2 = "<bos>", "<eos>", "<speaker1>", "<speaker2>"

persona = [tokenizer.tokenize(s) for s in persona_string]
history = [tokenizer.tokenize(s) for s in history_string]
reply = tokenizer.tokenize(reply_string)

def build_inputs(persona, history, reply):
    # Build our sequence by adding delimiters and concatenating
    sequence = [[bos] + list(chain(*persona))] + history + [reply + [eos]]
    sequence = [sequence[0]] + [ [speaker2 if (len(sequence)-i) % 2 else speaker1] + s
                                for i, s in enumerate(sequence[1:])]
    # Build our word, segments and position inputs from the sequence
    words = list(chain(*sequence))                          # word tokens
    segments = [speaker2 if i % 2 else speaker1             # segment tokens
                for i, s in enumerate(sequence) for _ in s]
    position = list(range(len(words)))                      # position tokens
    return words, segments, position, sequence

words, segments, position, sequence = build_inputs(persona, history, reply)

# Tokenize words and segments embeddings:
words = tokenizer.convert_tokens_to_ids(words)
segments = tokenizer.convert_tokens_to_ids(segments)
lm_targets = ([-1] * sum(len(s) for s in sequence[:-1])) \
             + [-1] + tokenizer.convert_tokens_to_ids(sequence[-1][1:])
```

Now most of the work is about preparing the inputs for the model.

We organize the contexts in segments

Add delimiter at the extremities of the segments

And build our word, position and segment inputs for the model.

Then train our model using the pretraining language modeling objective.

5.C – Chit-chat with OpenAI GPT & PyTorch

PyTorch Hub

Last Friday, the PyTorch team soft-launched a beta version of *PyTorch Hub*. Let's have a quick look.

- ❏ PyTorch Hub is based on **GitHub repositories**
- ❏ A model is shared by adding a *hubconf.py* script to the root of a GitHub repository
- ❏ Both **model definitions** and **pre-trained weights** can be shared
- ❏ More details: https://pytorch.org/hub and https://pytorch.org/docs/stable/hub.html

In our case, to use *torch.hub* instead of *pytorch-pretrained-bert*, we can simply call *torch.hub.load* with the path to *pytorch-pretrained-bert* GitHub repository:

```
import torch

tokenizer = torch.hub.load('huggingface/pytorch-pretrained-BERT', 'openAIGPTTokenizer', 'openai-gpt')
model = torch.hub.load('huggingface/pytorch-pretrained-BERT', 'openAIGPTLMHeadModel', 'openai-gpt')
```

PyTorch Hub will fetch the model from the *master branch* on *GitHub*. This means that you don't need to package your model (*pip*) & users will always access the most recent version (*master*).

That’s all for this time

Image credit: Andrejs Kirma