Chapter 6 - Advanced Learning Techniques

"Tell me and I forget, teach me and I may remember, involve me and I learn."
– Benjamin Franklin

This chapter is a continuation of Chapter 3, where we introduced learning techniques. To recap, learning techniques can help us meet our model quality goals. Techniques like distillation and data augmentation improve the model quality, without increasing the footprint of the model (size, latency, etc). And as we have described earlier, some of these improved quality metrics can be traded off for a smaller footprint as desired.

Continuing with the theme of chapter 3, we will start this chapter by presenting self-supervised learning which has been instrumental in the success of natural language models like BERT. Self-Supervised learning helps models to quickly achieve impressive quality with a small number of labels.

As we described in chapter 3's 'Learning Techniques and Efficiency' section, labeling of training data is an expensive undertaking. Factoring in the costs of training human labelers on a given task, and then making sure that the labels are reliable, human labeling gets very expensive very quickly. Even after that it is likely that the model might not be able to capture the intricacies of your task well.

Self-Supervised learning helps to significantly improve the quality you can achieve while retaining the same labeling costs i.e., training *data-efficient* (specifically, label efficient) models. We will describe the general principles of Self-Supervised learning which are applicable to both language and vision. We will also demonstrate its efficacy through a colab.

Finally, we introduce miscellaneous techniques to help you improve your model's quality metrics without taking a hit on any of the footprint metrics. These techniques might get superseded by other better methods over time, but again our goal is to give you a gentle introduction to this area for you to be able to research and experiment with these and other similar ideas.

With that being said, let's jump right in.

Self-Supervised Learning

The vanilla supervised learning paradigm that we are familiar has two limitations when it comes to training a model for a new task:

1. Data Efficiency: It relies heavily on labeled data, and hence achieving a high performance on a new task requires a large number of labels.
2. Compute Efficiency: Training for new tasks requires new models to be trained from scratch. For models that share the same domain, it is likely that the first few layers learn similar features. Hence training new models from scratch for these tasks is likely wasteful.

Regarding the first limitation, we know that model quality can usually be naively improved by acquiring more labels (though the rate of improvement eventually plateaus). However, acquiring more labels through human effort is expensive, and is unlikely to scale to the level that we want for complex tasks. To achieve a reasonable quality on non-trivial tasks, the amount of labeled data required is large too. For the second limitation, training large models from scratch for every slightly different task is not efficient either. In many cases we might be limited by our training compute budget, so this approach is a non-starter.

While techniques like data-augmentation, distillation etc. as introduced in chapter 3 do help us achieve better quality with fewer labels and fewer training steps required for convergence, they do not alleviate the concerns completely. What should we do to get an order of magnitude savings for both?

Self-supervised learning (SSL) helps with learning *generalizable and robust representations* without the need of labeled examples. The focus is to ensure that these representations are learnt without focusing narrowly on a label, or the specific task at hand. Once the model learns these representations it can then be fine-tuned with a small number of labeled examples over a reasonable number of training epochs to do well on the given task. We will go into details of how this works shortly.

For now, let's assume that we have such a general model that works for natural language inputs. Then by definition the model should be able to encode the given text in a sequence of embeddings such that there is some semantic relationship preserved between pieces of text that are related. “A very happy birthday” and “Happy birthday to you” should have similar representations and they should be close in the embedding space because they convey the same meaning. However, “The croissant was too sweet” should have a much different representation that is far from both the former sentences.

Now notice **how such a model would be useful across many different tasks**. If the representations generated are indeed generalizable and robust (i.e., nothing ties them to a specific task and minor changes in the input don’t significantly change the output), then we can simply add a few additional layers (known as the prediction head), use the appropriate loss function, and train the model with the labeled data for the task at hand. We can keep the

original model weights frozen, or let them be trainable. Such models are referred to as *pre-trained models* in literature.

The crux is that the amount of data needed for the downstream task in this process is much less as compared to the amount of data we would have needed if we were training a task specific model from scratch. One such task is the Microsoft Research Paraphrase Corpus[1] where the model needs to predict if a pair of sentences are semantically equivalent. The dataset has only 5800 labeled examples of pairs, which would be incredibly small for this task if we were training a model using just the labeled data. However, with such a general model our hope is that we can use these limited number of labeled examples for fine-tuning since the model already knows the general concepts about language, and use the same model across many tasks.

Model reuse by itself also is a powerful attribute of this scheme, and lends itself to compute efficiency since only have to train the model on a small number of examples, saving training time compute too.

A Typical Self-Supervised Learning Recipe

We can break-down common self-supervised learning into two broad steps:

1. **Pre-training**: This step teaches the model about the world it is operating in (language, vision, multimodal) through certain tasks which ensure that the model learns general representations of the inputs. Pre-training is data-efficient since we end up saving on the number of labels required to achieve the desired model quality on our task.

2. **Fine-tuning**: This step adapts a pre-trained network to a specific task. Fine-tuning is compute efficient since we reuse the same base model for all the tasks that operate in the same domain as the pre-training task.

Refer to figure 6-1 for an illustration of pre-training and fine-tuning stages. In the figure we demonstrate pre-training with a large unlabeled dataset of animal images. The pre-trained model is then fine-tuned for downstream tasks, for example object detection for tigers, segmentation for pets etc., where the labeled data might be sparse.

[1] Dolan, William B. and Chris Brockett. "Automatically Constructing a Corpus of Sentential Paraphrases." ACL Anthology, 2005, aclanthology.org/I05-5002.

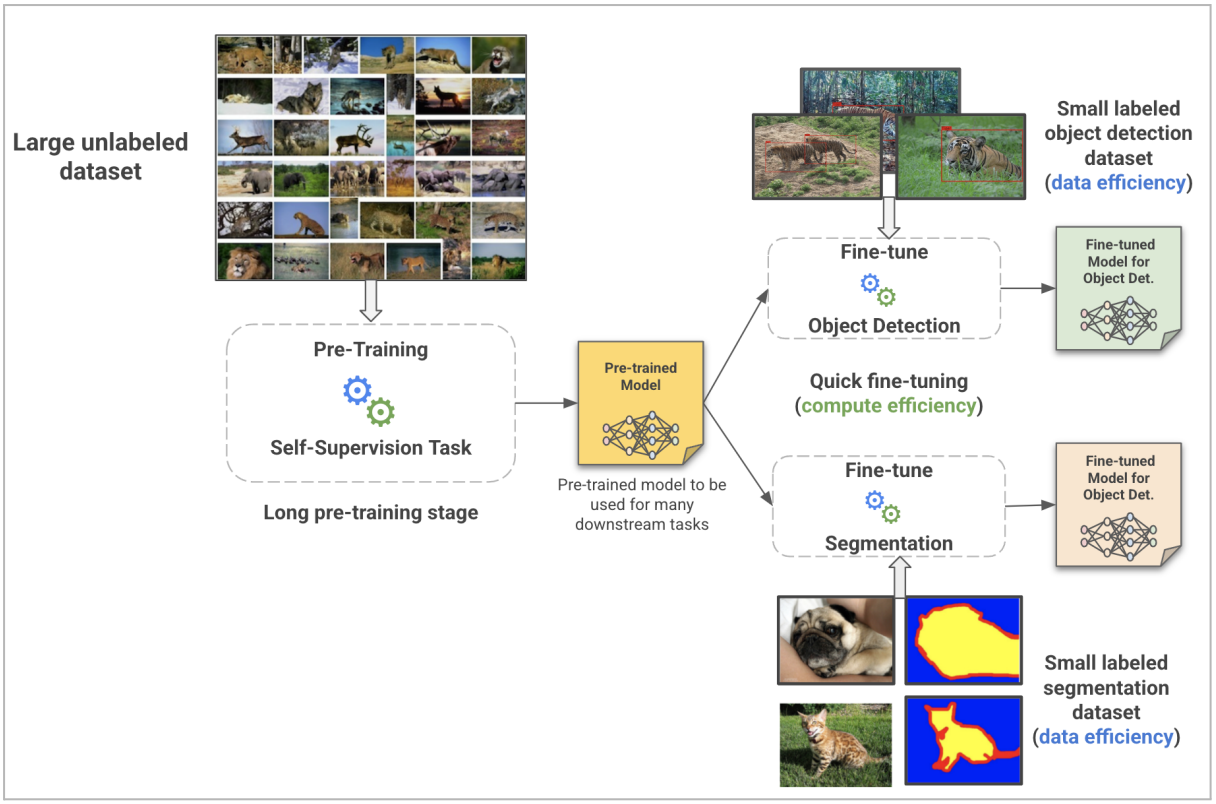


Figure 6-1: Pre-training and fine-tuning stages. With an example of a large unlabeled animal images dataset which is used for pre-training. The pre-trained model is then used for fine-tuning for downstream tasks.

Let's go over both the stages, in detail.

Pre-Training With Unlabeled Data

The first question to answer is how should we come up with these pre-trained models that generalize well? An important step is selecting the domain specific *pretext tasks* that can help the general models to capture the relationships between inputs.

In such pretext tasks, typically, the model pretends that a part/structure of the input is missing and it learns to predict the missing bit. It is similar to solving an almost finished jigsaw puzzle which has just a couple of open spots. Looking at it, we could tell what the final few pieces would look like. A pretext task requires the model to develop some level of understanding of the input, but it is not unsolvable or intractable. See figure 6-2 for a general theme that these tasks follow.

If you consider x to be a sequence that you can create from your unlabeled dataset, a few simple pretext tasks can be to predict the last element (future) from the previous elements (past), or the other way around. Again to re-emphasize we are just pretending that the data is missing for the sake of the pretext task. This works well for domains like natural language where your data will be a sequence of tokens.

You can extend the analogy to x being a tensor of rank n, and hide part of the input and train the model to predict it.

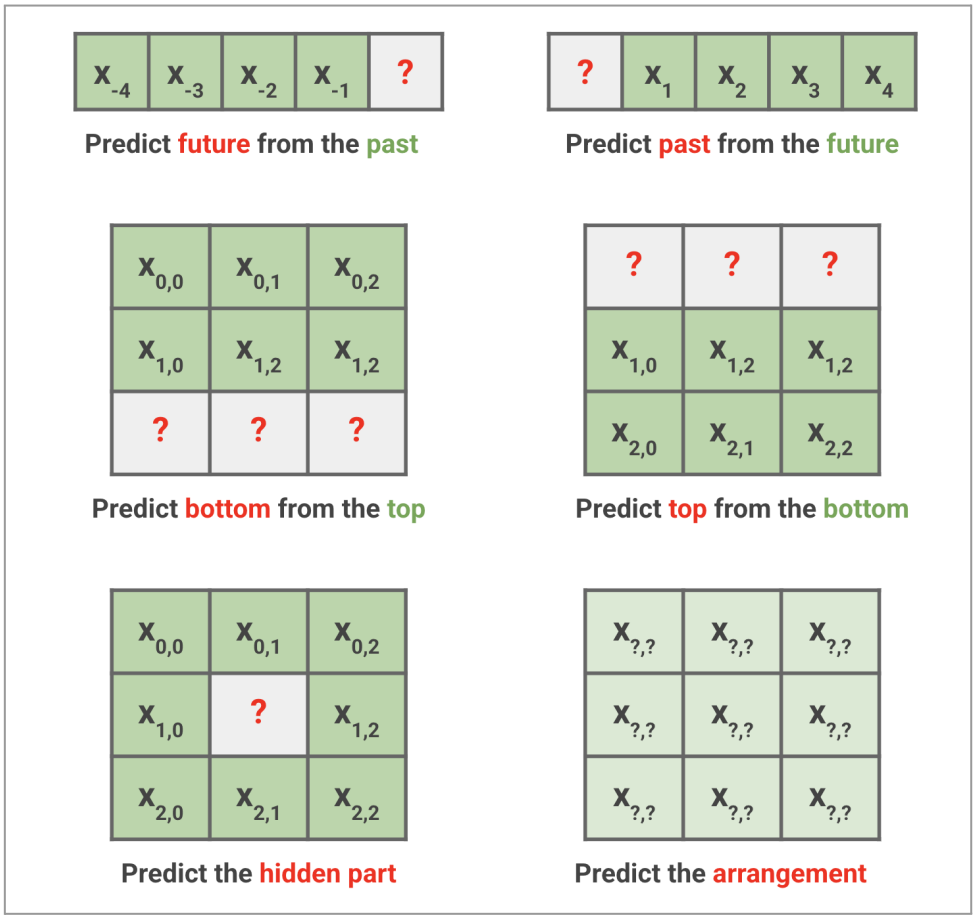


Figure 6-2: General theme of pretext tasks. If x is a tensor that you have picked from your unlabeled dataset, pretext tasks can be created by pretending that a part of the input is hidden, and the pretext task will predict the hidden part. You can also play around with the arrangement of the input, and make the model predict the right order of the elements of x.

The next question is where do we get the data for creating these tasks though? Since for each input, we can create the output by masking some part of the input itself, there is no need for any sort of human intervention for labeling. Therefore, we can simply use e-books, Wikipedia and other sources for NLU related models, and web images & videos for computer vision models. We can then construct the final dataset for the pretext task by simply masking the input as discussed, and the output is the part that we masked out or the original input.

Once we have pre-trained our model on one or a combination of pretext tasks, the prediction head (the final output layers which are specific to the pretext tasks) is removed to obtain a model that generates an embedding or sequence of embeddings for the given input (the general representation that we mentioned earlier).

One of the most famous pre-trained models is BERT, which was followed by many other variants like RoBERTa, ALBERT, ELECTRA, etc. For BERT, figure 6-3, the pretext tasks are as follows:

1. Masked Language Model (MLM): 15% of the tokens in the given sentence are masked and the model needs to predict the masked token.
2. Next Sentence Prediction (NSP): The second task is, given two sentences A and B, predict if B follows A.

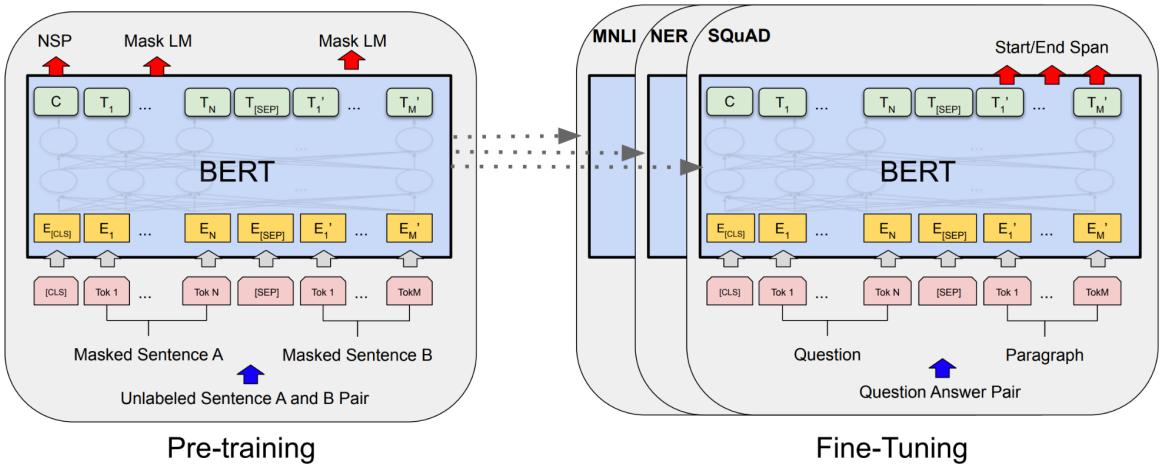


Figure 6-3: Pre-training and Fine-tuning steps for BERT. Source: Develin et al.

For BERT, the pre-training loss is the mean of the losses for the above two tasks.

Similar to NLU, the pretext tasks in vision have been used to train models that learn general representations. Doersch et al.[2] extract two patches from a training example and train the model to predict their relative position in the image (refer to figure 6-4 (a)). They demonstrate that using a network pre-trained in this fashion improves the quality of the final object detection task, as compared to randomly initializing the network. Similarly, another task is to predict the degree of rotation for a given rotated image[3] (refer figure 6-4 (b)). The authors report that the network trained in a self-supervised manner this way can be fine-tuned to perform nearly as well as a fully supervised network.

[2] Doersch, Carl, et al. "Unsupervised Visual Representation Learning by Context Prediction." arXiv, 19 May. 2015, doi:10.48550/arXiv.1505.05192.
[3] Gidaris, Spyros, et al. "Unsupervised Representation Learning by Predicting Image Rotations." arXiv, 21 Mar. 2018, doi:10.48550/arXiv.1803.07728.

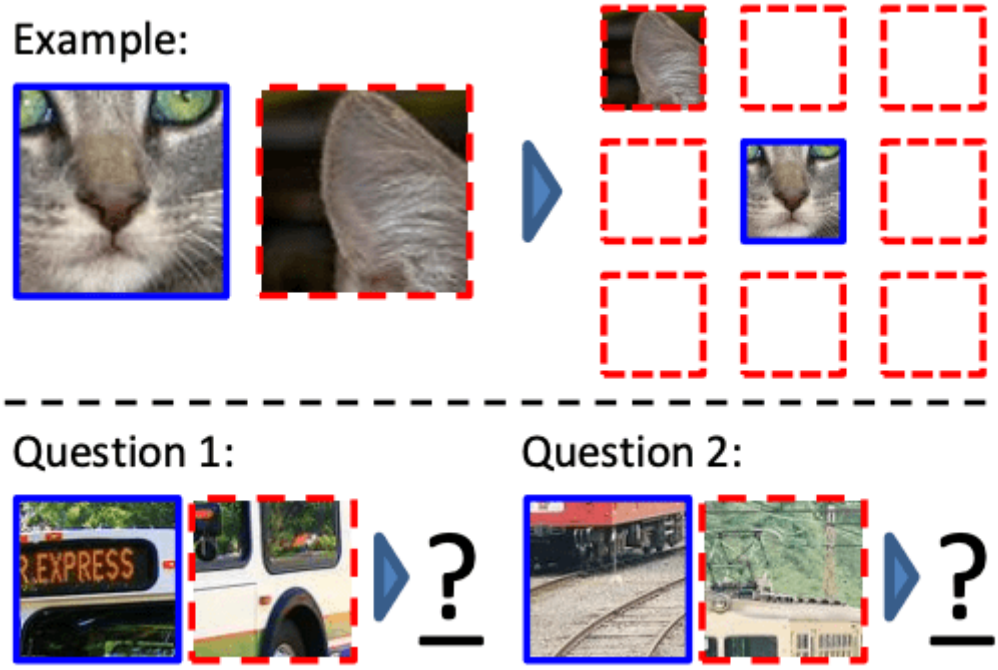


Figure 6-4 (a): Detecting relative order of patches derived from the same image.

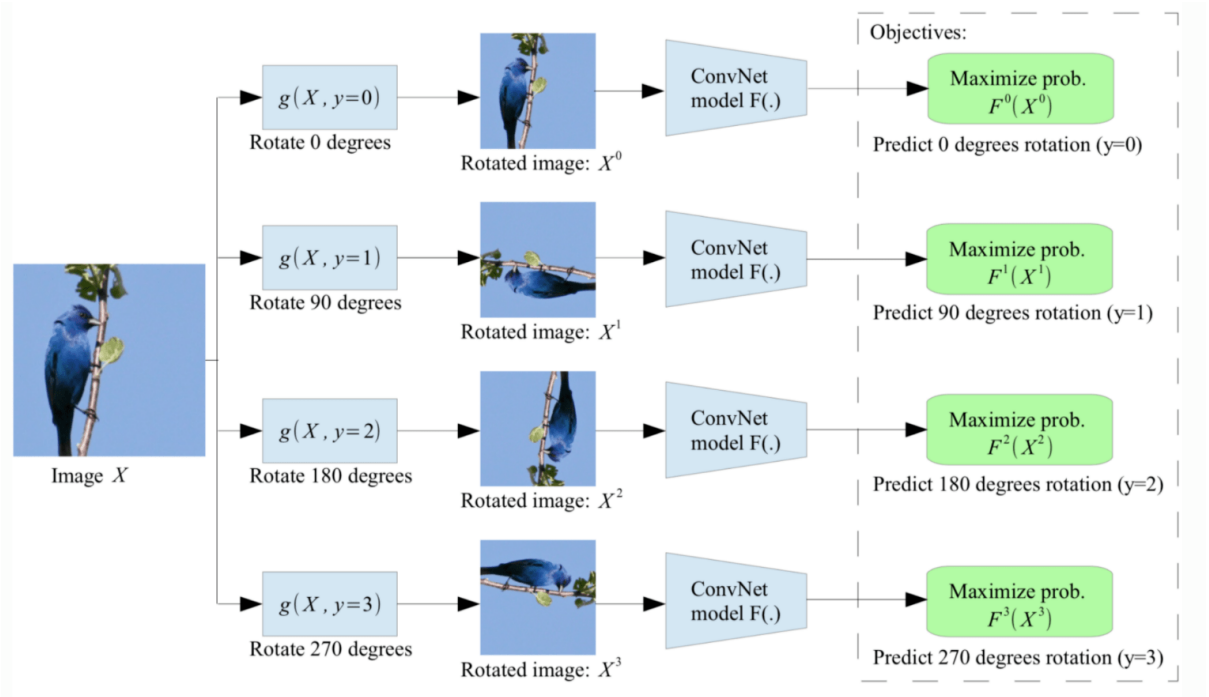


Figure 6-4 (b): Predicting the degree of rotation of a given image.

Figure 6-4: Pretext tasks for vision problems. Source: Doersch et al., Gidaris et al..

Once the general model is ready, we can fine-tune it for a specific task. The next section discusses it in detail.

Fine Tuning On Labeled Data

The next step in using the pre-trained models is to fine-tune them which is quite straightforward because these models have learnt generic representations of the input domain that transfer well across specific tasks in that domain. They can be adapted to solve the target task by:

1. Adding a new prediction head to the pre-trained model which can translate the general representations to the task specific output dimensions while also providing the necessary learning capacity to adapt to the new task.
2. Training (fine-tuning) the model with the new prediction head on the labeled data for our task. This can be done either by freezing the layers obtained from pre-training and just training the prediction head, or training the entire model.

Refer to figure 6-5 for a visualization of creating a fine-tuning a pre-trained model on a downstream task.

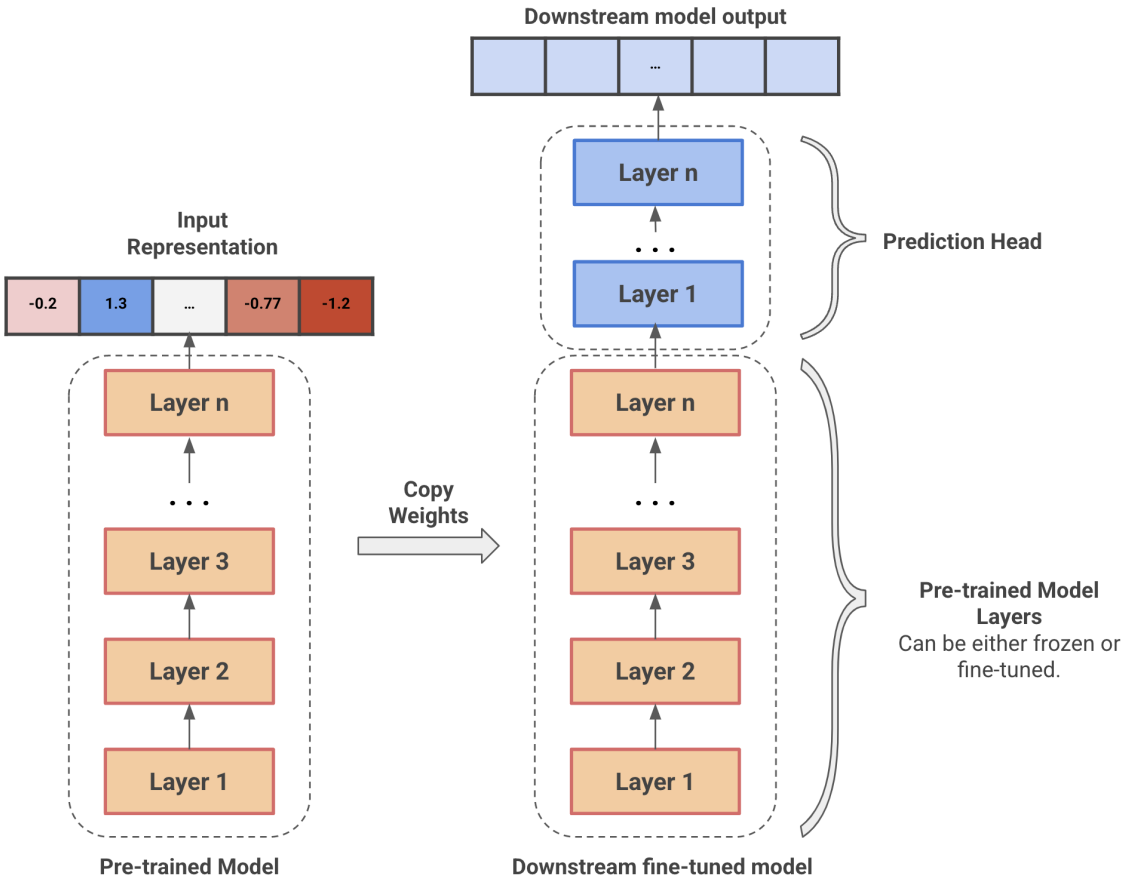


Figure 6-5: Fine-tuning a pre-trained model on a downstream task can be done by adding a prediction head to a pre-trained model, and then either keeping the pre-trained model layers frozen or letting it be fine-tuned.

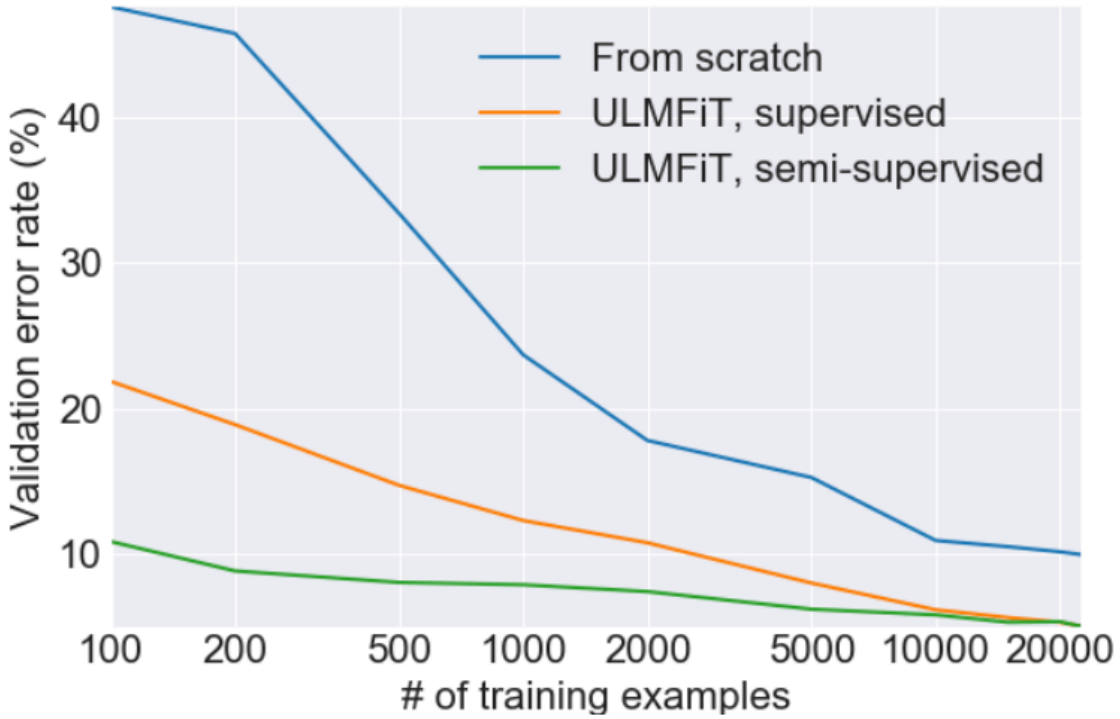
Since the labeled data we provided to the model in this fine-tuning stage is not being used for learning rudimentary features, but rather how to map the high-level representations it learned in the pretraining stage to solving our new task. Thus, the number of labeled data

examples to achieve the same quality is going to be a small fraction of what would have been required if we were training the model from scratch.

Efficiency Gains

From the lens of efficiency, using pre-trained models and fine-tuning them on a small labeled dataset helps make our model training *data-efficient* and *compute-efficient*. Pre-training + Fine-tuning helps the models converge faster, attain similar or better quality for the same amount of labeled data when compared to training from scratch, etc.

ULMFiT (Howard et al.[4]) pioneered the idea of training a general purpose language model, where a model learns to solve the pretext task of predicting the next word in a given sentence, without the need of an associated label. The authors found that using a large corpus of preprocessed unlabeled data such as the WikiText-103[5] dataset was a good choice for the pre-training step. It was sufficient for the model to learn the general properties of the English language. The authors also found that fine-tuning such a pre-trained model for a binary classification problem (IMDb dataset) required only 100 labeled examples ($\approx 10\times$ less labeled examples otherwise). If we add a middle-step of pre-training using unlabeled data from the same target dataset, the authors report needing $\approx 20\times$ fewer labeled examples. Refer to figure 6-6 for a comparison between the error obtained by training from scratch v/s using pre-training strategies.



[4] Howard, Jeremy and Sebastian Ruder. "Universal Language Model Fine-tuning for Text Classification." arXiv, 18 Jan. 2018, doi:10.48550/arXiv.1801.06146.
[5] WikiText-103 dataset is derived from English Wikipedia pages.

Figure 6-6: Validation error w.r.t. number of training examples for different training methods on IMDb (from scratch, ULMFiT supervised: pre-training with WikiText-103 and fine-tuning using labeled data, ULMFit semi-supervised: pre-training with WikiText-103 as well as unlabeled data from the target dataset and fine-tuning with labeled data). Source: Howard et al.

The pre-trained model can then be used for classification or seq2seq tasks by adding additional layers on top of the last hidden layer as demonstrated by Howard et al..

As we mentioned in the previous section, this idea of pre-training followed by fine-tuning is also used in BERT (Devlin et al.), and other related models like GPT, RoBERTa, T5, etc. At the time of its publishing, BERT beat the state-of-the-art on eleven NLU tasks.

A critical point to note is that we get the data-efficiency by spending extra training compute during the pre-training phase and pre-training BERT-like models is not cheap. The original paper reports BERT-Base requiring 4 Cloud TPU Pods (4 chips each, total 16 chips) over 4 days for a total of 1,536 TPU hours. Each Cloud TPU chip is priced at $3.22 / hr[6], which means the training would take ~ 1536 * 3.22 = $4,945.92. BERT-Large requires 16 Cloud TPU Pods for 4 days, which turns out to be 6,144 TPU hours and $19,783.68 to train.

Other pre-trained models can be a couple of orders of magnitude more expensive to pre-train. GPT-3 takes 355 GPU years, which comes out to be 3.1 Million GPU hours, which even with the cheapest GPU available on GCP (K80 GPU @ $0.45 / hr[7]) costs $1.39 Million. These calculations also exclude the compute spent in all the intermediate steps in getting to the final model such as experiments with architectures, hyper-parameter tuning, and model performance debugging.

However, since the pre-trained model is intended to be generalizable across many downstream tasks, the cost of pre-training can be amortized amongst these tasks. BERT has been used across a large number of NLU tasks. Say we consider the economics of training the BERT-Base model which costs ~ $5K to train. If using the pre-trained BERT saves $50 in training and data labeling costs for any given downstream application (which is very reasonable), we only need to achieve that saving across 100 applications before it becomes profitable to pre-train BERT-Base rather than train each application from scratch. In this case BERT models have been downloaded tens of thousands of times just from the official Tensorflow Hub repository[8].

Similarly models like GPT-3, T5, etc. have the capability to be *few-shot learners*. This means that they can be shown a few example inputs and outputs to solve a new task. GPT-3 is a transformer model that only has the decoder (input is a sequence of tokens, and the output is a sequence of tokens too). It excels in natural language generation and hence has been

---

[6] Cloud TPU pricing source: https://cloud.google.com/tpu/pricing. Numbers reported from October 2022.

[7] GPU pricing source: https://cloud.google.com/compute/gpus-pricing. Numbers reported from October 2022.

[8] BERT model on Tensorflow-Hub: https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4

extensively used for such tasks. Refer to figure 6-7 for an example where the large language model was trained to perform sentiment detection by showing it a few examples of the task.

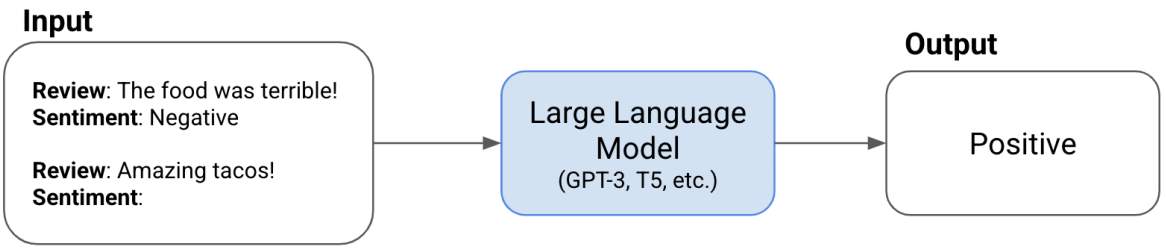


Figure 6-7: An example of few-shot learning with a large language model.

One of the prominent deployment of such models is the GitHub's Copilot software[9] where GPT-3 is used for auto-completing code snippets with an IDE. End-users can also use GPT-3 API[10] to build their own applications. Given the large number of possible uses for such models, the high costs of pre-training get spread over the number of applications using it.

Project: Using Pre-trained Language Models for News Classification

That was a lot of talk without any code. Our overarching goal with self-supervised learning is to be more efficient in the number of labels we need, and the number of training epochs we need to achieve our desired model quality. In this project we will demonstrate that self-supervised models provide both those efficiency gains.

We will work on the AGNews dataset (the same that we used in chapter 4) for text classification using a pre-trained BERT model, and demonstrate better quality and faster convergence than a BERT model that is trained from scratch. The code for this project is available **here** as a Jupyter notebook.

We will not be explicitly demonstrating pre-training BERT, since as we described earlier training them can be expensive as well as time-consuming. However, as initial pointers you can refer to this guide for pre-training BERT in Keras, and this guide for some optimizations to make it efficient. Also consider going through the work by Izsak et al.[11] which presents a collection of tweaks to achieve BERT-like quality but with a budget of 24 GPU hours.

Getting back to this project, we will be training using Google's TPUs this time.TPUs make it faster and cheaper to train large models, and are also available for free in Kaggle and Google Colab (apart from the paid service on Google Cloud). We will be talking more about TPUs in Chapter 10. For now, you can follow our lead. You can also adapt the code to run on GPUs if you desire.

We start by loading the dataset and ensuring that the dataset is placed on the CPU (and not the TPU).

---

[9] GitHub Copilot: https://github.com/features/copilot
[10] OpenAI GPT-3 API https://openai.com/api/
[11] Izsak, Peter, et al. "How to Train BERT with an Academic Budget." ACL Anthology, Nov. 2021, pp. 10644-52, doi:10.18653/v1/2021.emnlp-main.831.

```
import tensorflow_datasets as tfds

with tf.device('/job:localhost'):
   ds = tfds.load('ag_news_subset', try_gcs=True, shuffle_files=True,
                  batch_size=-1)
   train_dataset = tf.data.Dataset.from_tensor_slices(ds['train'])
   test_dataset = tf.data.Dataset.from_tensor_slices(ds['test'])
```

As usual, we will start off by creating our training and test datasets.

```
BATCH_SIZE = 256
batched_train =
train_dataset.shuffle(train_dataset.cardinality()).batch(BATCH_SIZE)
batched_test = test_dataset.shuffle(test_dataset.cardinality()).batch(BATCH_SIZE)
```

We will import the `tensorflow_text` library so that we can use the BERT model which relies on certain tensorflow ops.

```
import os

# tensorflow_text is required so that we can use certain ops used in our model.
import tensorflow_text as tf_text
```

Next we will import the `tensorflow_hub` library so that we can import pre-trained BERT models directly from Tensorflow Hub.

```
import tensorflow_hub as hub

# This line is required, once again, so that the TPU doesn't complain about the
# weights of the TF Hub models being on local storage.
os.environ['TFHUB_MODEL_LOAD_FORMAT'] = 'UNCOMPRESSED'
```

We first start by importing the BERT pre-processing model as a keras layer that converts input text into sequences of numeric identifiers. This is the same step as the tokenization that we performed in chapter 4 with the embeddings. The numeric identifiers are indices into the embedding tables in the pre-trained model. We will use this pre-processing layer to tokenize our training and test datasets.

```
# Check out the TF hub website for more preprocessors
preprocessor = hub.KerasLayer(
   'https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3')

# Create the final datasets that the BERT model will work on.
train_ds = batched_train.map(
   lambda x: (preprocessor(x['description']),
              tf.expand_dims(x['label'], axis=-1)))

test_ds = batched_test.map(
```

```
    lambda x: (preprocessor(x['description']),
               tf.expand_dims(x['label'], axis=-1)))
```

Now we will specify the two BERT encoders that we will use as encoders in this codelab, as a dictionary. The key being the lookup identifier for the BERT variant, and the value being the respective path of that variant in TF Hub. For the purpose of this codelab we will use the BERT-Small and BERT-Base variants.

```
BERT_ENCODERS = {
   # Recommended, because it is fast and has same interface as base BERT
   'bert-small':
"https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-2_H-128_A-2/2",
   'bert-base': 'https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4',
}
```

In this codelab we would like to demonstrate the performance of a model that we train from scratch, and doesn't have the pre-trained weights v/s a pre-trained model. Unfortunately when using TFHub, we can't only ask for the model and not its weights. Therefore, we will also add a helper method that lets us approximate a randomly initialized model by shuffling the weights.

```
import numpy as np

def shuffle_weights(model, weights=None):
   """Shuffle the weights in the given model.
     This is a fast approximation of re-initializing the model weights.
   """
   if weights is None:
      weights = model.get_weights()
   weights = [np.random.permutation(w.flat).reshape(w.shape) for w in weights]
   model.set_weights(weights)
```

Next up, we will define our model that uses the pre-trained BERT as an encoder. Notice how we add a few dense layers after the output of the BERT model. This helps adapt the complete model and be fine-tuned on our task.

```
def get_bert_model(
   encoder_size,
   learning_rate=2e-5,
   keep_tfhub_weights=True,
   num_classes=4):
 """Create a BERT classifier."""
 with tpu_strategy.scope():
   # Prepare the inputs.
   bert_inputs = dict(
       input_word_ids=tf.keras.layers.Input(
           shape=(None,), dtype=tf.int32, name='input_word_ids'),
       input_mask=tf.keras.layers.Input(
           shape=(None,), dtype=tf.int32, name='input_mask'),
```

```
        input_type_ids=tf.keras.layers.Input(
            shape=(None,), dtype=tf.int32, name='input_type_ids'),
    )

    # Create the encoder layer from TF Hub.
    bert_encoder = hub.KerasLayer(
        BERT_ENCODERS.get(encoder_size, 'bert-small'),
        trainable=True,
        name='bert_encoder')

    # Collect the output.
    output = bert_encoder(bert_inputs)['pooled_output']

    # Add a few dense layers + non-linearities in the end as a projection head
    output = tf.keras.layers.Dense(200, activation='relu')(output)
    output = tf.keras.layers.Dense(100, activation='relu')(output)
    output = tf.keras.layers.Dense(50, activation='relu')(output)
    output = tf.keras.layers.Dense(num_classes, activation=None)(output)
    output = tf.keras.layers.Activation('softmax')(output)
    bert_classifier = tf.keras.Model(bert_inputs, output)

    bert_classifier.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    if not keep_tfhub_weights:
       shuffle_weights(bert_classifier)
  return bert_classifier
```

Let's invoke the training with the BERT-Small model architecture, but not its weights (we will set the `keep_tfhub_weights parameter` to False).

```
bert_small_from_scratch_classifier = get_bert_model(
   'bert-small', learning_rate=1e-4, keep_tfhub_weights=False)
bert_small_from_scratch_classifier_history =
bert_small_from_scratch_classifier.fit(
   x=train_ds, epochs=4, validation_data=test_ds)

Epoch 1/4
469/469 [==============================] - 51s 59ms/step - loss: 0.7432 - accuracy:
0.6898 - val_loss: 0.3910 - val_accuracy: 0.8642
Epoch 2/4
469/469 [==============================] - 16s 33ms/step - loss: 0.3552 - accuracy:
0.8770 - val_loss: 0.3252 - val_accuracy: 0.8879
Epoch 3/4
469/469 [==============================] - 16s 34ms/step - loss: 0.2856 - accuracy:
0.9022 - val_loss: 0.2925 - val_accuracy: 0.9000
Epoch 4/4
```

```
469/469 [==============================] - 16s 33ms/step - loss: 0.2481 - accuracy:
0.9151 - val_loss: 0.2870 - val_accuracy: 0.9033
```

To compare with the scenario when we do use the BERT-Small weights, we start another run where `keep_tfhub_weights` is set to True.

```
bert_small_pretrained_classifier = get_bert_model(
   'bert-small', learning_rate=8e-5, keep_tfhub_weights=True)
bert_small_pretrained_classifier_history = bert_small_pretrained_classifier.fit(
   x=train_ds, epochs=4, validation_data=test_ds)

Epoch 1/4
469/469 [==============================] - 41s 54ms/step - loss: 0.4249 - accuracy:
0.8590 - val_loss: 0.2892 - val_accuracy: 0.9021
Epoch 2/4
469/469 [==============================] - 16s 34ms/step - loss: 0.2663 - accuracy:
0.9088 - val_loss: 0.2616 - val_accuracy: 0.9079
Epoch 3/4
469/469 [==============================] - 16s 33ms/step - loss: 0.2256 - accuracy:
0.9217 - val_loss: 0.2522 - val_accuracy: 0.9134
Epoch 4/4
469/469 [==============================] - 16s 34ms/step - loss: 0.1943 - accuracy:
0.9324 - val_loss: 0.2477 - val_accuracy: 0.9159
```

As is clear, the BERT-Small model with the pretrained weights clearly outperforms the model which is trained from scratch. The former achieves an accuracy of 91.59% while the latter achieves 90.33%. Refer to figure 6-8.

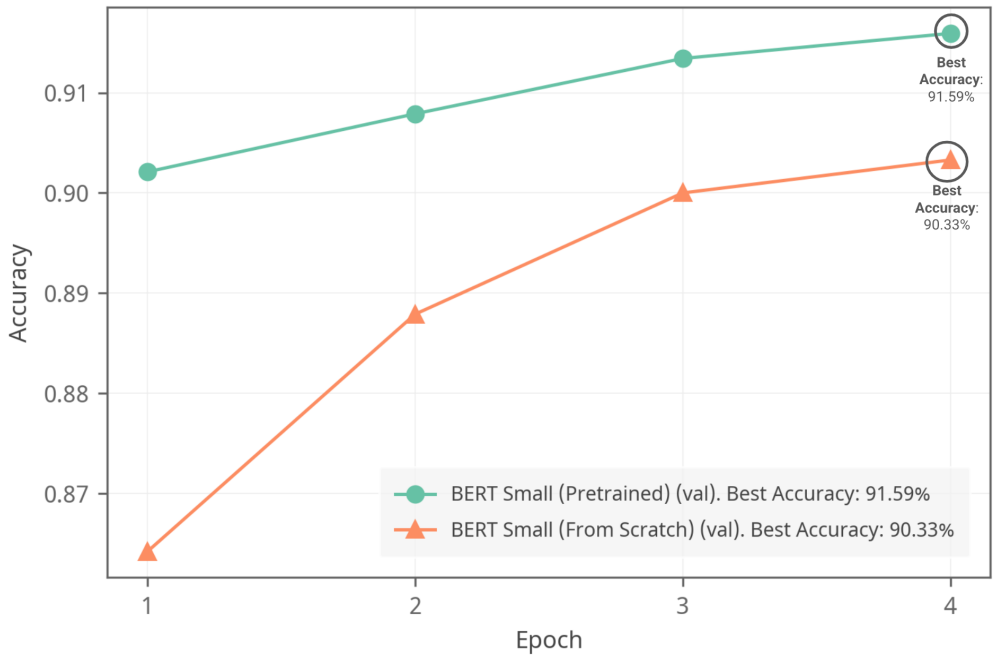


Figure 6-8: Comparison between the accuracies achieved by BERT-Small models when using and not-using the pre-trained model weights.

We repeated the same process for BERT-Base and noticed a similar effect. Using a pre-trained BERT-Base model achieves a best accuracy of 93.97%, while using the same architecture but not the pre-trained model achieves a best accuracy of 90.07%. Refer to figure 6-9.

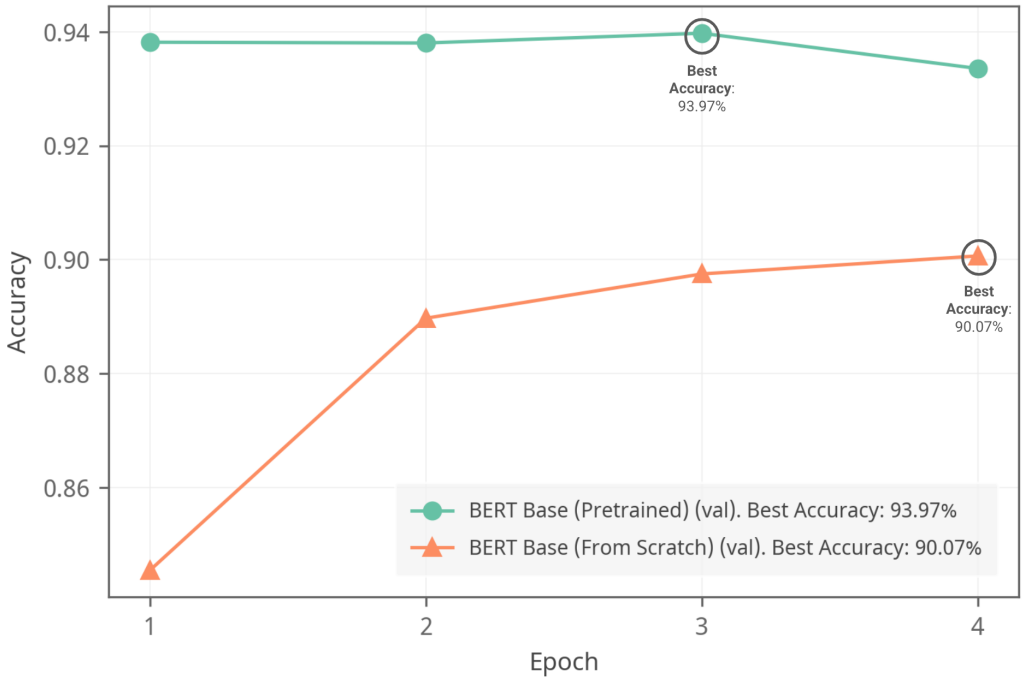


Figure 6-9: Comparison between the accuracies achieved by BERT-Base models when using and not-using the pre-trained model weights.

Thus, we have demonstrated that using pre-trained models is:

1. Data-Efficient: When using pre-trained BERT variants in both cases, we achieve a significant performance boost when compared to not using the pre-trained weights, while using the same number of labels.
2. Compute-Efficient: In both cases, when using the pre-trained model the accuracy was higher after the very first epoch, than the best accuracy achieved over four epochs of training without using the pre-trained weights.

Contrastive Learning

Another common theme in self-supervised learning is contrastive learning. Similar to the other self-supervised learning techniques, the goal is to learn generalizable representations of the inputs in the domain of interest. What's unique about contrastive learning is that we create pairs of inputs, and ask the model to contrast between them by generating the representations for both of them.

These pairs can be either a positive pair - where both the inputs are semantically similar, or a negative pair with both the inputs that are semantically dissimilar. The model is expected to generate hidden representations that are similar for positive pairs, and dissimilar for negative

pairs. We can use a metric like cosine similarity to enforce this similarity / dissimilarity of representations. Once the model is trained to a low enough test loss, we can then adapt it to a downstream task in the same domain just as we did in the previous subsections.

It is easy to create the negative pairs: one can simply pick two random inputs from a diverse enough domain and they are likely to be dissimilar. How do we go about creating positive pairs?

One example of such a recipe is the SimCLR framework[12,13] (refer to Figure 6-10). SimCLR creates positive pairs by using different data augmentations on the same input x. The resulting inputs x_i and x_j are passed through the 'encoder network' which is represented by the function $f(\cdot)$ and generates h_i and h_j, the respective hidden representations of the two inputs, as presented in figure 6-10.

In this scenario, we would expect to directly optimize for similarity between h_i and h_j, but the authors found that it was better to add a small neural network referred to as the 'projection head' (represented by the function $g(\cdot)$ to first project the hidden representations into a lower dimensional space to obtain z_i and z_j. The loss function would then enforce agreement between z_i and z_j.

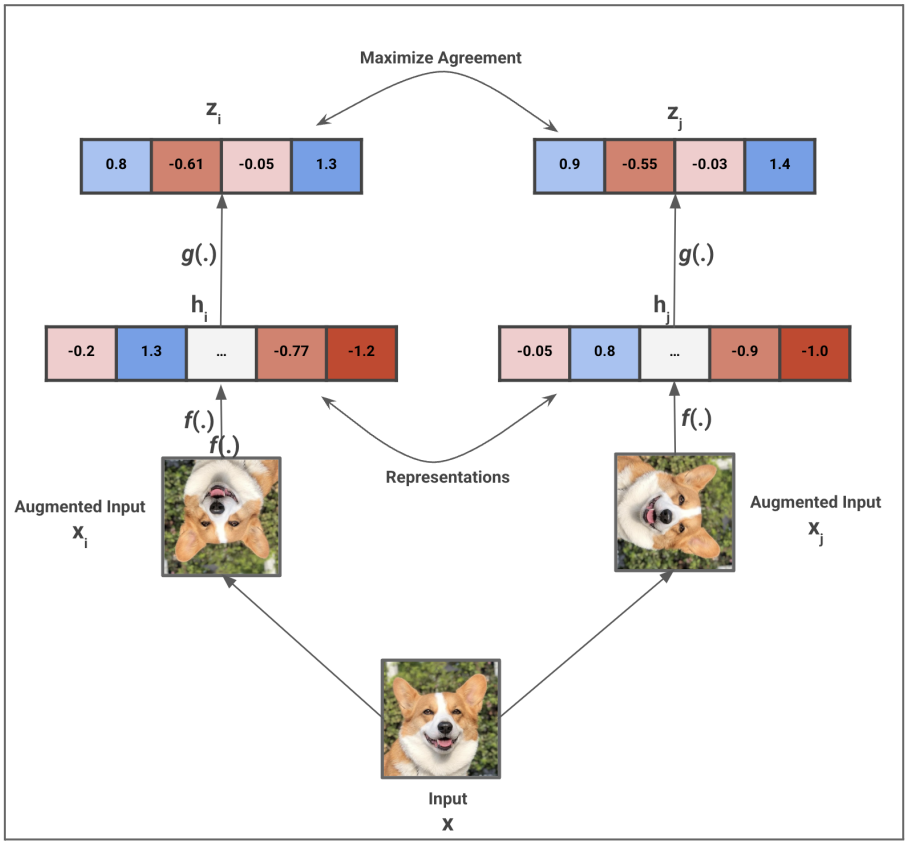


Figure 6-10: Contrastive learning as implemented in the SimCLR framework. The input x is augmented to generate two views, x_i and x_j. Using the shared encoder $f(\cdot)$, hidden

[12] Chen, Ting, et al. "A Simple Framework for Contrastive Learning of Visual Representations." arXiv, 13 Feb. 2020, doi:10.48550/arXiv.2002.05709.
[13] Chen, Ting, et al. "Big Self-Supervised Models are Strong Semi-Supervised Learners." arXiv, 17 June 2020, doi:10.48550/arXiv.2006.10029.

representations h_i and h_j are generated. These are then projected down using a projection head $g(.)$ to generate z_i and z_j. The model is then trained to maximize agreement between z_i and z_j.

Negative pairs are created using all the other pairs, and the loss to be minimized is a variant of the cross-entropy loss. We would refer you to the SimCLR paper for more details about the chosen loss functions and other alternatives considered.

Once the desired test loss is achieved, the projection head ($h(.)$) is no longer needed since its only purpose was to help maximize the agreement between the representations of the positive pairs. The encoder network ($g(.)$) is sufficient for usage in downstream tasks, and the hidden representation $h = f(x)$ for a given input x can then be used for the new task.

As is the case with other self-supervised learning techniques, our goal here is to demonstrate better generalization and a more label efficient method. Towards that end, the authors report a top-1 accuracy of 73.9% on ImageNet when fine-tuning with only 1% labels (13 labels per class). The SimCLR fine-tuned checkpoint with ResNet-50 as the encoder architecture also achieved a better accuracy on ImageNet with only 10% labels, when compared to a ResNet-50 that was trained from scratch with all the labels.

These are impressive results, and demonstrate how self-supervised learning techniques can help significantly outperform the naive baselines in terms of quality and label-efficiency. As an example, achieving > 70% accuracy on ImageNet with only 13 labels per class is a hard task, because ImageNet is a 1000-way classification problem. Therefore you should consider fine-tuning existing contrastive learning checkpoints wherever applicable (similar domain, data-augmentation techniques are valid for your usecase, etc.).

In case existing checkpoints are not directly useful, we hope that this section also provides you with a gentle introduction to self-supervised learning such that you can take a stab at adapting these techniques for your usecase. Now we will spend the rest of the chapter going through a wide variety of learning techniques. All aboard!

Bag of Assorted Learning Techniques

So far we have introduced broad themes under learning techniques. In this section, we will go over several relatively simpler techniques that you can experiment with in your model training. It can be worth your time to keep up to date on similar techniques that are presented in new research papers, since they can help improve your baseline models even if they were presented in the context of different model architectures and hyperparameters.

For example, the paper titled: *ResNet Strikes Back* by Wightman et al.[14] demonstrates improvement in the accuracy achieved by the ResNet-50 model from 75.2% to 80.4% without using any additional data or a teacher model. Even though ResNet-50 was introduced back in 2015, updating it with newer learning techniques improved the accuracy significantly without having to change anything in the architecture. Similarly the paper by He et al.[15] demonstrates multiple percentage points of accuracy improvements in EfficientNet through various learning techniques.

Let's pause to think about the significance of these results. It is kind of like receiving upgrades to your computer, except that these new upgrades don't make it slower because your hardware is older, but actually make it perform better than it used to. How nifty!

In some cases you can borrow insights from the papers we mentioned, but in other cases you might have to experiment and figure out the learning techniques and their right combinations that work well for your model training setup. With that being said, let's jump to how label smoothing can help us avoid overfitting.

Label Smoothing

Label smoothing is a regularization method that helps reduce the overfitting we might see with our models where the model predicts a label over-confidently. Consider the case where we have a model and solving a multi-class classification problem with K classes. The ground-truth labels will either be the index of the correct class for that given example, or a one-hot vector of size K with y_k = 1.0, where k is the index of the correct class.

The final layer of a model trained for such a task will be of size K (representing the K logits) and followed by the softmax activation, which as you may know look as follows:

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

$\hat{y}_i$ denotes the model's output probability associated with the i-th class and $\mathbf{z}$ is the logit vector for the given input. Since the denominator is the same for all $\hat{y}_i$, the probabilities add up to 1:

$$\sum_{i=1}^{K} \hat{y}_i = 1.0$$

Upon training the model, the loss function will ensure that $\hat{y}_k$ (probability assigned by the model to the right class) is pushed as close to 1.0 since the label for the k-th class is 1, and all the other labels are 0. However, note that as the training progresses this sharp-focus on reducing the loss by getting the $\hat{y}_i$ as close to 1.0 as possible, ensures that z_k continues to

[14] Wightman, Ross, et al. "ResNet strikes back: An improved training procedure in timm." arXiv, 1 Oct. 2021, doi:10.48550/arXiv.2110.00476.
[15] He, Tong, et al. "Bag of Tricks for Image Classification with Convolutional Neural Networks." arXiv, 4 Dec. 2018, doi:10.48550/arXiv.1812.01187.

increase to a large positive value, while all other z_j in the denominator continue to decrease to a large negative value. Over many epochs, this leads to overfitting on the given training dataset and poor generalization.
A remedy was proposed in the Inception paper[16] in the form of label-smoothing. The idea is to introduce a tiny amount of scalar noise in each correct label (y_k), denoted by ϵ and a noise of $\frac{\epsilon}{K}$ in every other y_i. ϵ is in $[0, 1]$.

Formally, prior to label smoothing the probability $\hat{y_i}$ for the i-th class is given by:

$$\hat{y_i} = \begin{cases} 1.0 & i = k, \\ 0.0 \end{cases}$$

After, label smoothing it is defined as follows:

$$\hat{y_i} = \begin{cases} 1.0 - \epsilon & i = k, \\ \frac{\epsilon}{K} & \text{otherwise} \end{cases}$$

ϵ can be used to control the noise. If it is too small, it might not have any effect. If it is too high, the distribution might become way too noisy for the model to learn anything. You should treat label smoothing as yet another regularization technique. In fact this paper[17] goes into details about when label smoothing helps. The original Inception paper reported a 0.2% increase in top-1 and top-5 accuracies on ImageNet, with $\epsilon = 0.1$. Given that ImageNet is a 1000-way classification problem, a 0.2% jump is significant.

Label smoothing is easy to implement on your own. However, various frameworks support it through their cross entropy loss function implementation. For example, Tensorflow provides a parameter to set the ϵ via the `label_smoothing` parameter in the `CategoricalCrossentropy` loss function, which you can easily set in your experiments.

Yet another way of improving generalization is to allow the model to learn concepts in the order of their difficulty. Curriculum learning shows us how.

Curriculum Learning

We know from experiments and machine learning theory that increasing the size of the dataset typically helps improve quality (though the rate of improvement eventually plateaus). However, not all training examples are created equal, and you might find that the model learns a subset of training data earlier in the training than the rest. Training examples might have different levels of *hardness* depending on how informative the features are.

---

[16] Szegedy, Christian, et al. "Rethinking the Inception Architecture for Computer Vision." arXiv, 2 Dec. 2015, doi:10.48550/arXiv.1512.00567.
[17] Lukasik, Michal, et al. "ICML'20: Proceedings of the 37th International Conference on Machine Learning." Does label smoothing mitigate label noise? JMLR.org, 13 July 2020, pp. 6448-58, doi:10.5555/3524938.3525536.

In curriculum learning, we start to train the model with the easiest examples and gradually allow harder examples as the training progresses. This has been empirically shown to improve model quality. In fact, the name *curriculum learning* is borrowed from how teaching in a classroom follows a curriculum where the basic and easier to grasp fundamentals are taught first, followed by incrementally more difficult concepts that build upon previous lessons.

The intuition behind this is the theory of Continuation Methods (CM)[18] which is a known approach for optimizing non-convex functions, where multiple local minima might exist. Typical deep learning objective functions are non-convex too, and directly working with these functions might lead to the optimizer getting stuck in a local minima. Continuation methods start with a *smoothened approximation* of the original objective function, and then gradually reduce that smoothening to reach the original non-convex function.

Bengio et al.[19] demonstrate this through experiments, where they start with *easier* examples and slowly add in progressively *harder* examples which leads to improvement in the model quality when compared to a model that was trained without curriculum learning. Training the model with only the easy examples in the beginning makes the loss function smoother , and as we add the harder examples we start to move towards the original loss landscape which might have many local minima.

The authors see an improvement in model quality when using curriculum learning to train the model for both a toy dataset where the goal is shape recognition, as well as for training a language model over wikipedia's english data where the goal was to predict the next word. For both, the difficulty of the dataset was gradually increased over time and compared with a non-curriculum learning strategy where the entire dataset was available for training from the beginning.

For example, in the language model the curriculum learning strategy starts by training the model over training examples where the target word is amongst the top K words in the vocabulary. The value of K is gradually increased as the training progresses. The intuition here is that training examples where the word to be predicted is commonly used would be easier to learn. The authors report a statistically significant difference between the final error in the word prediction curriculum and non-curriculum strategies.

How do we go about implementing curriculum learning ourselves? We need two things here:

Collecting examples of varied difficulty

To start off, we need a way to find easier examples to start training the model with. For this, one way is to come up with a scoring function s that is a heuristic for the hardness of the given example. For any two given examples z_i and z_j, if $s(z_i) < s(z_j)$, then the example z_j is harder to learn than the example z_i. Some examples of these heuristics can be:

---

[18] Allgower, Eugene L. and Kurt Georg. Numerical Continuation Methods. Springer, link.springer.com/book/10.1007/978-3-642-61257-2.

[19] Bengio, Yoshua, et al. "ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning." Curriculum learning. Association for Computing Machinery, 14 June 2009, pp. 41-48, doi:10.1145/1553374.1553380.

1. The epoch at which the model first learns to correctly predict the output for a given example. If the model learns to predict an example correctly in the early epochs, it is possibly easier than others.
2. Average of the loss value for the given example over the training run.
3. The number of times the example is learnt correctly and forgotten during the training. This could indicate examples that are harder to learn.
4. Examples containing rare features would be difficult for the model in the beginning.

Alternatively, we can construct examples of varied difficulty ourselves. Some examples are below:

1. For image classification problems, we can start with images that are resized to smaller dimensions. The model would learn coarse-grained features for these images. As the training progresses, the image size can be gradually increased. This was proposed in the FastAI course[20].
2. For natural language problems (or for that matter any problem with sequential data), we can consider heuristics like vocabulary frequency (sequences with rare tokens are likely harder as shown in the language model task by Bengio et al.), sequence length (shorter sequences are likely easier), etc. For example Li et al.[21] proposed training GPT-3 like models with a technique called sequence length warmup where the model training starts with the input truncated to a limit which is gradually increased as the training progresses. The authors reported a 10x data saving and 17x training time savings with this technique.

Pacing the training example difficulty

Next, we need a *pacing* function to tune the difficulty level of the examples during training. We sort the training examples (z) in ascending order of $s(z_i)$, where s is the scoring function. Until the *t*-th epoch, we use the first $p(t)$ fraction of examples from the sorted training set. If we train for a total of n epochs, then $p(n)$ should be 1.0 to ensure that the entire dataset is used towards the end.

The pacing function starts with a fixed value $p(0)$. It is gradually ramped up linearly or exponentially to reach a value of 1.0 in the final epoch. It controls when and how much complexity you want to introduce in the training.

Figure 6-12 shows multiple examples of pacing functions. The x-axis is the training iteration i.e. the variable t described above, and the y-axis is the fraction of data that is enabled from the sorted training set $p(t)$. The dotted pacing line shows a pacing function that starts with a fixed fraction of the data sorted by the scores, and at some iteration starts training with all the data. The solid and dashed lines show fixed and varied exponential pacing functions.

---

[20] Fast.AI Course: https://github.com/fastai/fastbook/blob/780b76bef3127ce5b64f8230fce60e915a7e0735/07_sizing_and_tta.ipynb

[21] Li, Conglong, et al. "The Stability-Efficiency Dilemma: Investigating Sequence Length Warmup for Training GPT Models." arXiv, 13 Aug. 2021, doi:10.48550/arXiv.2108.06084.

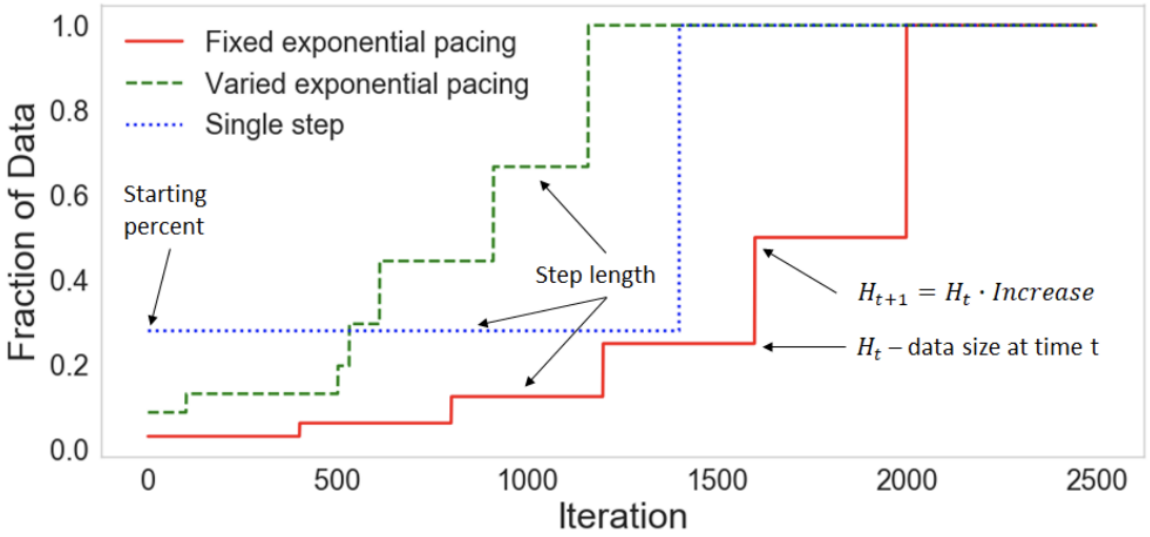


Figure 6-12: Examples of various pacing functions used for curriculum learning. Each pacing function describes the schedule of enabling the training dataset sorted by increasing hardness.

Label smoothing and curriculum learning both help with better generalization and avoiding overfitting. Another way to overfit might be intrinsic to how we search for the loss minima in the objective function using today's optimizers. In the next subsection on sharpness aware minimization is an interesting peek into how tweaking the objective function can help with generalization.

Sharpness Aware Minimization

Neural networks are universal function approximators (i.e. given enough parameters, they can learn any function) and their objective functions are non-convex. What does this non-convexity mean? To recap, refer to the two plots in figure 6-13. Both are plots of functions in a single variable, with the variable x on the x-axis and $f(x)$ being the y-axis, and we are trying to find the minima for both. On the left is a convex function and it has a single valley or a minima. On the right is a non-convex function which has multiple valleys or local minimas, of which only one is the true global minima. In practice the optimizer does not know where the global minima lies, so it might get stuck in local minima / valleys when optimizing a non-convex function.

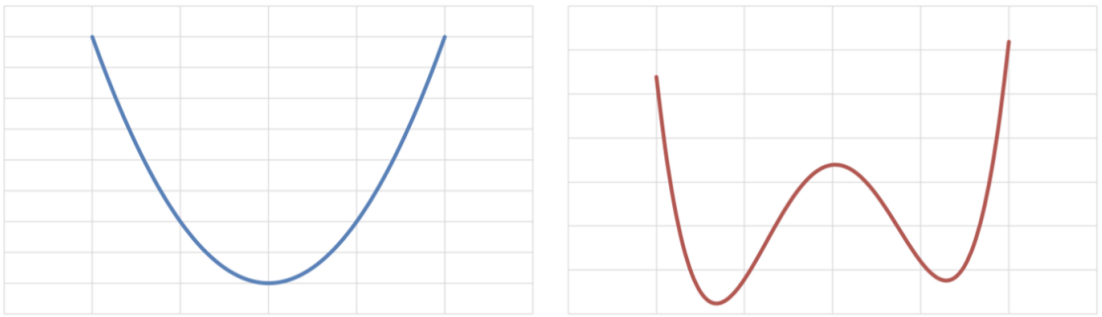


Figure 6-13: The graph on the left represents a convex objective function and the one on the right represents a non-convex objective function for a single variable.

Typically, the objective function (loss function) of a deep learning model is non-convex. One axis represents the loss values, and each trainable model parameter would be represented on a separate axis since changing any trainable model parameter would have an effect on the loss value. Since a deep learning model can have anywhere from thousands to billions of parameters, finding the exact global minima is intractable. That's why we use optimization algorithms that work for convex functions, with some additional help through techniques like momentum to help the optimizer escape the local minimas.

Sharpness-Aware Minimization (SAM)[22] is one such technique. It suggests that steep valleys in the objective function might just be the peculiarities of the training dataset, which might not be representative of the true distribution. Hence, a minima in these steeper valleys is less likely to generalize, leading to higher loss on the test dataset.

Therefore, the optimizer should prefer a flatter minima over a steeper minima. This idea is intuitively analogous to regularization where we prefer to find solutions with model parameters having smaller absolute values due to occam's razor[23]. Refer to figure 6-14 for a visualization of steep and flat local minimas. The left hand side image shows a loss landscape that has a sharp and steep minima because the loss suddenly drops to a very low value. On the right hand side shows the loss landscape that is relatively smooth and the neighborhood of the minima shows a gradual drop as compared to the left.

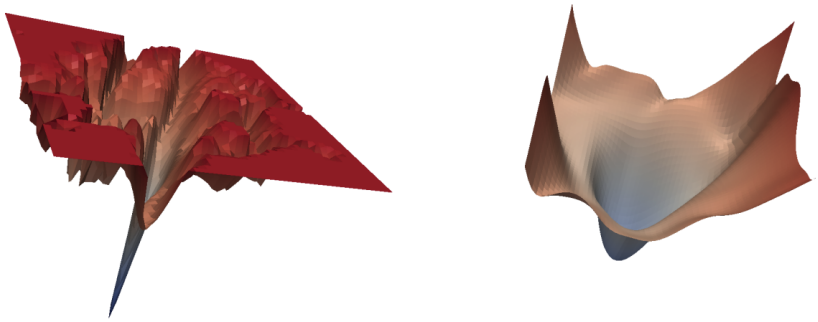


Figure 6-14: Partial visualization of a loss landscape with a steep local minima (left) and a relatively flatter local minima (right). Source: Forret et al.

SAM encourages the optimizer to find a minima where the neighborhood of that minima has low loss too, by using the SAM loss. If we denote the weights of a model by $\mathbf{w}$, and its loss function on a training dataset S as L_S. Then the SAM objective function is defined as:

$$L_S^{SAM}(\mathbf{w}) = \max_{||\boldsymbol{\epsilon}||_p < \rho} L_S(\mathbf{w} + \boldsymbol{\epsilon})$$

[22] Foret, Pierre, et al. "Sharpness-Aware Minimization for Efficiently Improving Generalization." arXiv, 3 Oct. 2020, doi:10.48550/arXiv.2010.01412.
[23] https://en.wikipedia.org/wiki/Occam%27s_razor

Let's break this down. Essentially, all this function is doing is checking how steep the neighborhood of $\mathbf{w}$ is, and penalizing for steepness. Visually it might be easier to confirm the relative steepness of the loss function if $\mathbf{w}$ is small, as we saw in figure 6-13.

Mathematically, we can do this by finding a perturbation vector ϵ which is the same shape as $\mathbf{w}$, such that $L_S(\mathbf{w} + \boldsymbol{\epsilon})$ is maximized to find the worst case loss in the neighborhood of w. To ensure that we are only looking not far from $\mathbf{w}$, we ensure that the p-th norm of $\boldsymbol{\epsilon}$ is less than ρ.

Let's assume ρ is set to a reasonable value and we have two possible minimas under consideration with the same loss, one with a steeper neighborhood $\mathbf{w}_{\text{steep}}$ and the other with a relatively flatter neighborhood $\mathbf{w}_{\text{flat}}$. Since the neighborhood of the former is steeper than the later by definition, we will find an $\boldsymbol{\epsilon}_{\text{steep}}$ such that $L_S^{SAM}(\mathbf{w}_{\text{steep}}) > L_S^{SAM}(\mathbf{w}_{\text{flat}})$. Thus, the optimizer would prefer the flatter minima with the new objective function.

The authors found that $p = 2$ worked best for them. Thus, the only hyper-parameter here is ρ which controls the range of the neighborhood that you want to explore, which should be set carefully since either a very small or very large value of ρ can make this method ineffectual.

The final objective to minimize (with a regularization term on $\mathbf{w}$) can be written as:

$$\min_{\mathbf{w}} L_S^{SAM}(\mathbf{w}) + \lambda ||\mathbf{w}||_2^2$$

The authors report significant improvement on many image classification tasks with different model architectures and data augmentation settings when using SAM. For instance, on the ImageNet task and the ResNet-152 model architecture trained over 400 epochs, SAM helps reduce the Top-1 error from 20.9% to 18.4%.

Tensorflow provides an easy way to try out SAM on your models, which returns a model that will now minimize the new SAM objective. The API looks as follows:

```
sam_model = tf.keras.models.experimental.SharpnessAwareMinimization(
    original_model,  # This is a TF / Keras model object that you have
                     # already built.
    rho=0.05         # An important hyper-parameter that controls the
                     # size of the neighborhood that you look into.
)
```

We hope that you can try out SAM on your own models, which may differ from the typical benchmark datasets and models used for comparing such techniques. Similarly, we might find that techniques like distillation might not be as helpful in certain settings. Subclass distillation in the next subsection can help us in some of these cases. Let's find out how.

Subclass Distillation

It can also be useful to revisit some of the other learning techniques in the context of the problem at hand. For instance, in chapter 3, we found that distillation was a very handy technique to improve our model's quality v/s footprint tradeoff. The motivation behind Subclass Distillation (Mueller et al.[24]) comes from the observation that during conventional knowledge distillation [Hinton et al. 2015] the amount of information that the student network receives from the teacher network scales linearly with the number of classes of the corresponding classification problem, assuming that the distillation temperature is set to a high enough value. On the other hand, when using one-hot labels the amount of information received by the model being trained scales logarithmically with the number of classes (one-hot labels with k classes provide $\log_2 k$ bits of information). Therefore, when the number of classes is small, distillation might not be much better than using one-hot labels. This is particularly an issue with binary classification problems, which are very common.

Subclass Distillation is a way of avoiding this issue by having the teacher 'invent' s subclasses for each of the original c classes, for a total of $c \times s$ new classes. The teacher is trained by:

1. Adding a 'subclass head' which generates s subclasses for each original class.
2. Using the original cross entropy loss where the probability of a class is calculated by summing up the probabilities of all its invented subclasses, and the ground-truth labels are used as is.
3. Finally, using an auxiliary loss which encourages the teacher model to distribute the probability mass across all the subclasses, otherwise it is possible to have a trivial solution where the model assigns almost all of the probability for a class to a single subclass.

The two losses can be combined using a hyper-parameter β which controls the weight of the auxiliary loss.

$$L_{\text{teacher}} = L_{\text{cross-entropy}} + \beta L_{\text{aux}}$$

In the above equation $L_{\text{cross-entropy}}$ denotes the loss function created using the original one-hot ground-truth labels, where the predictions of the teacher model over c classes are created by summing up the probabilities of subclasses for each class. L_{aux} denotes the auxiliary loss function that we mentioned above. We will refer you to the paper for a formal definition of L_{aux}.

Once the teacher is trained, it can then generate logits for $c \times s$ classes, instead of the original c classes. The student also has a subclass head to generate its predictions over the same $c \times s$ classes as the teacher. We now have a loss function L_{distill} on the student side, where the labels are the subclass logits from the teacher, and the predictions are the subclass logits from the student.

There is also a $L_{\text{cross-entropy}}$ loss function which is similar to the one used in the teacher with

[24] Müller, Rafael, et al. "Subclass Distillation." arXiv, 10 Feb. 2020, doi:10.48550/arXiv.2002.03936.

the original ground-truth labels, and the summing up of probabilities of subclasses to create the student's predictions. The student's final loss function is a combination of the two losses we mentioned above, and their individual contributions are controlled by the hyper-parameter α.

$$L_{\text{student}} = \alpha L_{\text{distill}} + (1 - \alpha) L_{\text{cross-entropy}}$$

Figure 6-15 illustrates the subclass distillation procedure, and how it compares with vanilla distillation.

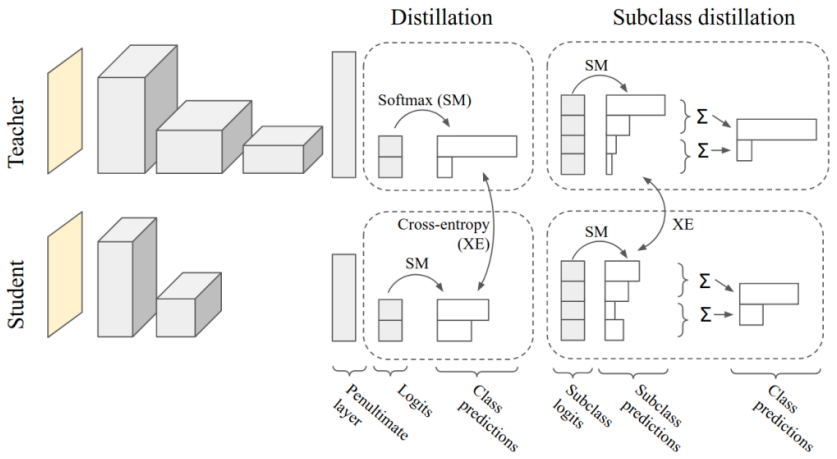


Figure 6-15: Comparison between vanilla distillation and subclass distillation.

The authors demonstrate faster convergence as well as improved accuracy on binary classification tasks, when compared to conventional distillation. For example on the natively binary classification task like Criteo's ad click prediction problem where the subclass technique achieves the same accuracy as a student with vanilla distillation but with fewer training iteration steps. When trained on a synthetic dataset of CIFAR-2x5 (the original 10 classes grouped into two super classes), the method achieved 2.1% better accuracy than vanilla distillation.

We will now go over stochastic depth, a technique which can be useful if you are training very deep networks.

Stochastic Depth

Deep networks with hundreds of layers such as ResNets have led to significant decrease in error rate on academic datasets, but they also suffer from the problem of *vanishing gradients*. As the number of layers increases, the gradient has to backpropagate from the output layer, through a large number of intermediate layers.

For the purpose of explanation, consider a simple neural network with an input x, and n layers each with a single neuron of scalar weight w_i, and bias b_i. Let y_i denote the output of the i-th layer.

If we assume no activations are applied for simplicity, then we can compute the output of the i-th layer as follows:

$$y_i = y_{i-1}w_i + b_i$$

With $y_0 = x$ in the base case. If we have a loss function L that depends on y_n, then we can calculate $\frac{\partial L}{\partial y_n}$. Our final goal is to calculate the partial derivative of the loss function with respect to the weight of the i-th layer, $\frac{\partial L}{\partial w_i}$, which is the gradient for that layer's weight.

Let's start by using the chain rule, to compute the partial derivative of the loss function with respect to y_{n-1} as follows:

$$\frac{\partial L}{\partial y_{n-1}} = \frac{\partial L}{\partial y_n}.\frac{\partial y_n}{\partial y_{n-1}}$$

And from the definition of y_i, we can calculate $\frac{\partial y_n}{\partial y_{n-1}}$ which is simply w_n.

$$\frac{\partial L}{\partial y_{n-1}} = \frac{\partial L}{\partial y_n}.w_n$$

More generally, we can calculate $\frac{\partial L}{\partial y_i}$, and from that $\frac{\partial L}{\partial w_i}$ using the chain rule again.

$$\frac{\partial L}{\partial y_i} = \frac{\partial L}{\partial y_n}.(w_n w_{n-1} w_{n-2} \dots w_{i+1})$$
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y_n}.(w_n w_{n-1} w_{n-2} \dots w_{i+1}).y_i$$

As you can see, if the network has a large number of layers and the weights[25] have small values, the product of the weights can become infinitesimally small. This will lead to $\frac{\partial L}{\partial w_i}$ to become infinitesimally small too. Note that we are not accounting for activation functions like tanh which further worsen the problem.

Skip connections as introduced in the ResNet architecture is one step towards solving this problem by creating 'residual blocks' (refer to figure 6-16 for an illustration).

Let the output of the l-th residual block be denoted by H_l. For the l-th residual block, we start with the input from the previous residual block, H_{l-1}. This input is branched into two, in one branch (the lower branch in the figure) we pass it through an identity function denoted

[25] Typically, the weights are normally distributed with mean = 0 and a small variance.

by id. In the other branch (the upper branch in the figure) we pass it through f_l is a sequence of multiple convolutional, batch-norm, ReLU layers. The output of both the branches is added together, and passed through a non-linearity (ReLU).

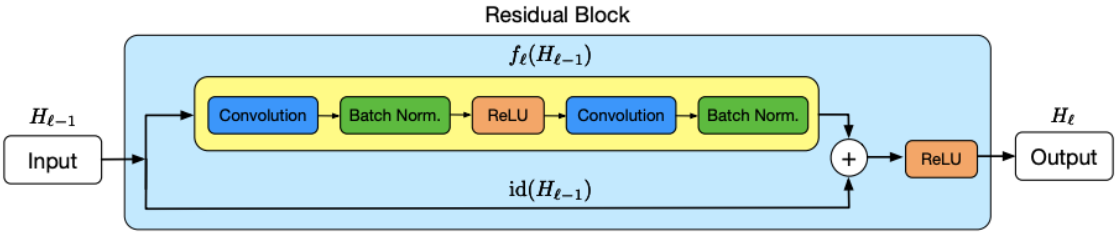


Figure 6-16: A residual block in ResNet.

Formally, it is computed as follows:

$$H_l = \mathrm{ReLU}(f_l(H_{l-1}) + \mathrm{id}(H_{l-1})).$$

Note how in the residual block, the output of the previous layer (H_{l-1}) skips the layers represented by the function f_l. The stochastic depth idea takes this one step further by probabilistically dropping a residual block with a probability p_l. The goal is to keep this probability high (less likelihood of dropping the given block) in the earlier layers, and low for later layers (higher likelihood of dropping the given block). This is because the earlier layers are known to compute lower-level features that have a bigger impact on the final output than the later layers, so we want to retain these features with a relatively higher probability.

Concretely, we reformulate the definition of H_l during training to as follows, after accounting for this probabilistic dropping:

$$H_l = \mathrm{ReLU}(b_l f_l(H_{l-1}) + \mathrm{id}(H_{l-1}))$$

Here b_l is a bernoulli random variable, which is 1.0 with a probability p_l and, 0.0 with a probability $1 - p_l$. Thus, during training the l-th block will act as an identity function with probability $1 - p_l$. We retain the original behavior during inference.

Finally, p_l is calculated using the following formula:

$$p_l = 1 - \frac{l}{L}(1 - p_L)$$

where L is the total number of layers, and p_L is the desired survival probability of the final layer. p_L is typically set to 0.5. This enables a smooth linear decay of p_l from nearly 1.0 in the first layer, to 0.5 in the final layer. Refer to figure 6-17 for an illustration.

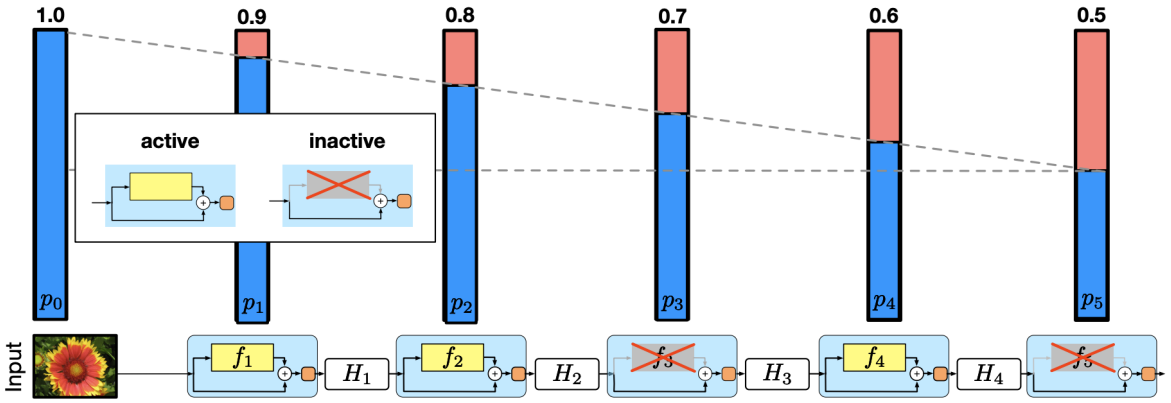


Figure 6-17: Probability of survival (p_l) of the l-th residual block for a network with five blocks and the final probability ($p_L = 0.5$).

Under these conditions, the *expected network depth* during training reduces to $(3L - 1)/4$. By expected network depth we informally mean the number of blocks that are enabled in expectation. For example, in a ResNet with L = 54 blocks, the expected number of blocks during training is $\approx$ 40. Although, during inference, we run the full network as usual with some minor recalibration of outputs as mentioned in the paper.

In terms of efficiency benefits, stochastic depth helps speed up training by approximately 25% when $p_L = 0.5$ as per the authors. It also helps improve test accuracy by acting as a regularizer. Each residual block is active only during a fraction p_l of the training steps, which creates an ensemble-like effect where each training step activates a different combination of the full network. Ensembles are known to help improve generalization, but explicitly creating ensembles is expensive.

The authors report 18% test error reduction on CIFAR-10 and 9.23% reduction in the test error on the CIFAR-100 dataset. The latter result was achieved with a 1202 layer ResNet which earlier overfit on the training dataset, while also being ~ 25% faster to train. Overall, stochastic depth seems like a promising approach for avoiding overfitting in deep networks, which also brings training efficiencies.

We have now come to the end of this chapter, and we are sure that there was a lot of content to absorb, so let's review some takeaways.

Summary

In this second chapter on learning techniques, we tried to provide depth by describing self-supervised learning in detail, and breadth by briefly introducing a collection of other simple techniques that you can incorporate in your model training. We explored self-supervised learning with an accompanying project demonstrating its data-efficiency and compute-efficiency properties. Self-supervised learning is present and used in models across many domains now. It has become a part of training large language and vision models, without the need of very large and expensive human labeled datasets tailored for a

specific usecase. Instead, we can now rely on practically unbounded unlabeled data, which is relatively easier to obtain for the usual domains of interest.

This makes it way easier and cheaper to achieve higher quality models with scant labeled data. In fact very large models like GPT-3 are few-shot learners, in that they can be shown a couple of examples of the task to be solved, and they can use those examples to *learn* how to solve that given task without the need for the model weights to be updated.

We also went over a collection of a few other learning techniques that you can incorporate in your regular model training. The goal was to provide an introduction to the broad themes that you can explore, even if these individual techniques are replaced by superior methods in the future.

For instance, label smoothing helps avoid overconfident predictions and hence overfitting. Curriculum learning tries to train a model by increasing the difficulty of the examples incrementally. Sharpness Aware Minimization tries to alter the optimization path by explicitly looking for minima which are flatter, and hence likely to generalize better.

Subclass distillation is a twist on the original distillation recipe, but focused on the problem of smaller numbers of classes, where the benefit from distillation might be diluted. Finally we presented stochastic depth which helps avoid the vanishing gradient problem that plagues really deep networks. Often you might be able to combine multiple of these and other similar techniques to improve your model quality, but it is not possible to recommend any single or a combination of techniques that work well together. For instance, training the teacher with label smoothing has been shown to hurt distillation[26].

As always, we recommend that to build an intuition for what works better and when, you should go ahead and try these ideas with both academic datasets which are easier to play with, or your own model and datasets which are more representative of the real world.

[26] Müller, Rafael, et al. "When Does Label Smoothing Help?" arXiv, 6 June 2019, doi:10.48550/arXiv.1906.02629.