Chapter 1

Transfer learning and adaptation

Note: Unfinished. To be added: prompting, domain adaptation figures, knowledge distillation. *Draft chapter from Torralba, Isola, Freeman*

A common criticism of current deep learning systems is that they are data hungry. To learn a new concept like "is this a dog?" may require showing a deep net thousands of labeled examples of that concept. Humans, on the other hand, can learn to recognize a new kind of animal after seeing it just once, an ability known as **one-shot learning**.

Humans can do this because we have extensive prior knowledge we bring to bear to accelerate the learning of new concepts. Deep nets are data hungry when they are trained from *scratch*. But they can actually be quite data efficient if we give them appropriate prior knowledge and the means to use their priors to accelerate learning. Transfer learning deals with how to use prior knowledge to solve new learning problems.

Transfer learning is an alternative to the ideas we saw last chapter, where we simply trained on a broad distribution of data so that more test queries happen to be things we have encountered before. Because it is usually impossible to train on *all* queries we might encounter, we often need to rely on transfer learning to adapt to new kinds of queries.

Transfer learning algorithms involve two parts:

1. What we transfer
2. What we adapt

Method	What is transferred	What is adapted
Prompting	Mapping	Inputs
Domain adaptation	Mapping	Inputs
Finetuning	Mapping init	Mapping
Distillation	Targets	Mapping
Linear probes	Mapping	Outputs

Table 1.1: Different kinds of adaptation.

1.1 Finetuning

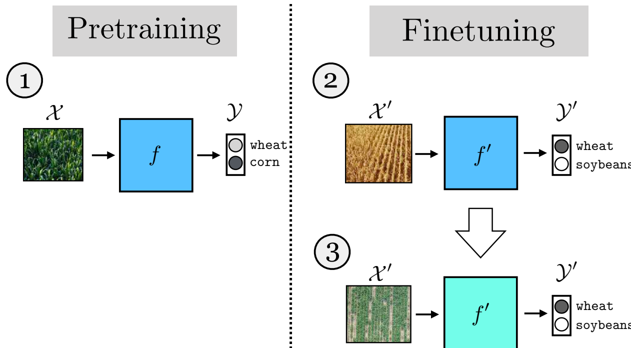
There are many ways to do transfer learning. We will start with perhaps the simplest: when you encounter a new task, just keep on learning *as if nothing happened*.

Suppose you are working at a farm and the drones that water the plants need to be able to recognize whether the crop is wheat or corn. Using the machinery we have learned so far, you know what to do: gather a big dataset of aerial photos of the crops and label each as either wheat or corn; then, train your favorite classifier to perform the mapping $f_\theta : \mathcal{X} \rightarrow \{\texttt{wheat}, \texttt{corn}\}$. It works! The crop yield is plentiful. Now a new season rolls

around and it turns out the value of corn has plummeted. You decide to plant soybeans instead. So you need a new classifier, what should you do?

One option is to train a new classifier from scratch, this time on images of either wheat or soybeans. But you already know how to recognize wheat – you still have last year's classifier f_θ. We would like to make use of f_θ to accelerate the learning of this year's new classifier. **Finetuning** consists of initialializing a new classifier with the parameter vector from last year's model, and then training it as usual. In other words, finetuning is making small (fine) adjustments (tuning) to the parameters of your model to adapt it to do something slightly different than what it did before. Or even a lot different.

Here is a diagram of the basic algorithm for finetuning:



By convention, we call the first phase of training "pretraining" and the second stage "finetuning", since often pretraining involves extensive training "in the factory", on a big dataset and thousands of processors, whereas usually finetuning refers to small updates done "in the wild" or on-device, with far fewer computational and data resources.

There are three stages: 1) **pretrain** $f : \mathcal{X} \rightarrow \mathcal{Y}$, 2) initialize $f' = f$, 3) **finetune** $f' : \mathcal{X}' \rightarrow \mathcal{Y}'$. The full algorithm is written below, with f and f' indicated as just a continually learning f_θ with different iterates of θ as learning progresses:

Algorithm 1: Training one model, then finetuning to produce a second model; using gradient descent

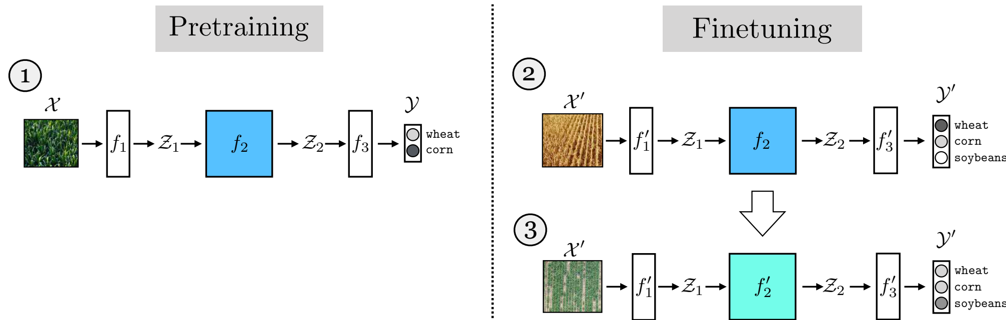
1 **Input:** initial parameter vector θ^0, data $\{x^{(i)}, y^{(i)}\}_{i=1}^N$, $\{x'^{(i)}, y'^{(i)}\}_{i=1}^M$, learning rates η_1 and η_2
2 **Output:** trained models f_{θ^N} and $f_{\theta^{N+M}}$
3 **Pretraining: for** $k = 1, \ldots, K_1$ **do**
4 $\quad J = \mathbb{E}_{x,y}[\mathcal{L}(f_{\theta^{k-1}}(x), y)]$
5 $\quad \theta^k \rightarrow \theta^{k-1} - \eta_1 \nabla_\theta J$
6 **Finetuning: for** $k = 1, \ldots, K_2$ **do**
7 $\quad J = \mathbb{E}_{x',y'}[\mathcal{L}(f_{\theta^{k-1+N}}(x'), y')]$
8 $\quad \theta^{k+N} \rightarrow \theta^{k-1+N} - \eta_2 \nabla_\theta J$

Finetuning is simple and works well. But there is one tricky bit we still need to deal with: what if the structure of f_θ is incompatible with the new problem we wish to finetune on?

For example, suppose f is a neural net and θ are the weights and biases of this net. Then the above algorithm will only work if the dimensionality of $\mathcal{X}$ matches the dimensionality of $\mathcal{X}'$, and the same for $\mathcal{Y}$ and $\mathcal{Y}'$. This is because, for our neural net model, f_θ has the same domain and range regardless of θ. What if we start with our net trained to classify between wheat and corn, and now want to finetune it to classify between wheat, corn, and soybeans? The dimensionality of the output has changed from two classes to three.

To handle this it is common to cut off the last layer of the network and replace it with a new final layer that outputs the correct dimensionality. If the input dimensionality changes, you can do the same with the first layer of the network. Here's what this looks like:

To be precise, let $f : \mathcal{X} \rightarrow \mathcal{Y}$ decompose as $f = f_1 \circ f_2 \circ f_3$, with $f_1 : \mathcal{X} \rightarrow \mathcal{Z}_1$, $f_2 : \mathcal{Z}_1 \rightarrow \mathcal{Z}_2$, and $f_3 : \mathcal{Z}_2 \rightarrow \mathcal{Y}$. Now we wish to learn a function $f' : \mathcal{X}' \rightarrow \mathcal{Y}'$, that decomposes as $f' = f_1' \circ f_2' \circ f_3'$, with $f_1' : \mathcal{X}' \rightarrow \mathcal{Z}_1$, $f_2 : \mathcal{Z}_1 \rightarrow \mathcal{Z}_2$, and $f_3 : \mathcal{Z}_2 \rightarrow \mathcal{Y}'$. The



finetuning approach here would be to first learn f, then, to learn f', we initialize f_2' with the parameters of f_2 and initialize f_1' and f_3' randomly (i.e. train these from scratch).

Transfer learning algorithms are differentiated largely based on the settings in which they apply. Do we keep around the data for task 1 when we move on to task 2? When we train the model for task 1, do we know what task 2 will be? And so forth.

1.2 Linear probes

As discussed above, a model can be adapted by changing some or all of its parameters, or by adding entirely new modules to the computation graph. One important kind of new module is a "read out" module that takes some features from the computation graph as input and produces a new prediction as output. When these modules are linear, they are called **linear probes**. These modules are relatively lightweight (few parameters) and can be optimized with tools from linear optimization (where there are often closed form solutions). Because of this, linear probes are very popular as a way of repurposing a neural net to perform a new task. These modules are also useful as a way of assessing what kind of knowledge each feature map in the original network represents, in the sense that if a feature map can linearly classify some kind of attribute of the data, then that feature maps "knows" something about that attribute. In this usage, linear probes are probing the knowledge represented in some layer of a neural net.

1.3 Knowledge distillation

Knowledge distillation transfers *prediction targets* from one learning problem to another. This is useful whenever the second learner is somehow less privileged than the first.

1.4 Prompting

Prompting edits the input data to solve a new problem.

1.5 Generative data

Generative data transfers knowledge about the input data to accelerate future problem solving on this same kind of data.

1.6 Domain adaptation

Often, the data we train on comes from a different distribution, or domain, than the data we will encounter at test time. For example, maybe we have trained a recognition system on Internet photos and now we want to deploy the system on a self-driving car. The imagery the

car sees looks very different than everyday photos you might find on the Internet. **Domain adaptation** refers to methods for adapting the training domain to be more like the test domain, or vice versa.

The first option is to adapt the test data, $\{x^{(i)}_{\texttt{test}}, y^{(i)}_{\texttt{test}}\}_{i=1}^N$ to look more like the training data. Then f should work well on both the training data and the test data. Or we can go the other way around, adapting the training data to look like the test data, *before* training f on the adapted training data. In either case, the trick is to make the distribution of test and train data identical, so that a method that works on one will work just as well on the other.

Commonly, we only have access to labels at train time, but may have plentiful unlabeled inputs $\{x'_i\}_{i=1}^N$ at test time. This setting is sometimes referred to as **unsupervised domain adaptation**. Here we cannot make use of test labels but we can still align the test inputs to look like the training inputs (or vice versa). One way to do so is to use a generative adversarial network (Chapter **??**) that "translates" the data in one domain to look identically distributed as the data in the other domain [Zhu et al. 2017, Hoffman et al. 2018]. A simple version of this algorithm is given below:

Algorithm 2: Unpaired domain adaptation via translation

---

1 **Input:** training data $\{x^{(i)}_{\texttt{train}}, y^{(i)}_{\texttt{train}}\}_{i=1}^N$, test data, $\{x'^{(i)}_{\texttt{test}}\}_{i=1}^M$
2 **Output:** trained model F
3 **Train predictor on train:** $f = \arg\min_f \mathbb{E}_{x_{\texttt{train}}, y_{\texttt{train}}}[\mathcal{L}(f(x_{\texttt{train}}), y_{\texttt{train}})]$
4 **Train translator $\mathcal{X}_{\texttt{test}}$ to $\mathcal{X}_{\texttt{train}}$:**
$G = \arg\min_G \max_D \mathbb{E}_{x_{\texttt{train}}}[\log D(x_{\texttt{train}}))] + \mathbb{E}_{x_{\texttt{test}}}[1 - D(G(x_{\texttt{test}}))]$
5 return $F = f \circ G$

1.7 RNNs from the lens of adaptation

RNNs update their hidden states based as they process a sequence of data. These changes in hidden state change the behavior of the mapping from input to output. Therefore, RNNs *adapt* as they process data. Now consider that learning is the problem of adapting future behavior as a function of past data. RNN's can be run on sequences of frames in a movie or on sequences of pixels in an image. They can also be run on sequences of training points in a training dataset! If you do that, then the RNN is doing "learning". Learning such an RNN can be considered meta-learning. For example, let's see how we can make an RNN do supervised learning. Supervised learning is the problem of taking a set of examples $\{x^{(i)}, y^{(i)}\}_{i=1}^N$ and inferring a function $f : x \to y$. Without loss of generality, we can define an ordering over the input set, so the learning problem is to take the sequence $\{x^{(1)}, y^{(1)}, \dots, x^{(n)}, y^{(n)}\}$ as input and produce f as output.

The goal of supervised learning is that after training on the training data, we should do well on any random test query. After processing the training sequence $\{x^{(1)}, y^{(1)}, \dots, x^{(n)}, y^{(n)}\}$, an RNN will have arrived at some setting of its hidden state. Now if we feed the RNN a new query $x^{(n+1)}$, the mapping it applies, which produces an output $y^{(n+1)}$, can be considered the learned function f. This function is defined by the parameters of the RNN along with its hidden state that was "learned" from the training sequence. Since we can apply this "f" to any arbitrary query $x^{(n+1)}$, what we have is a learned function that operates just like a function learned by any other learning algorithm; it's just that in this case the learning algorithm was an RNN.

Bibliography

J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, P. Isola, K. Saenko, A. Efros, and T. Darrell. Cycada: Cycle-consistent adversarial domain adaptation. In *International conference on machine learning*, pages 1989–1998. PMLR, 2018.

J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.