CS-GY 9223, Fall 2020

Practical aspects of Deep Learning (I)

The course thus far has covered different categories of deep learning models, where the categorization is based on the underlying data distribution — whether the data distribution is:

- unstructured and static (MLPs),
- spatially coherent (convnets)
- temporally coherent (RNNs)
- unlabeled (generative models)
- changing in response to the output predictions (RL).

We will now briefly discuss some practical and real-world topics/challenges that are generally applicable to all deep learning problems. The first set of challenges deal with resource-limitedness while training the DL model, while the second set of challenges deal with limitations of any model when deployed in the real world.

First, we cover training challenges and how to overcome then. We will learn about:

- *meta-learning*, which addresses the limited samples issue
- *federated learning*, which addresses the limited memory issue

Meta-learning

Supervised (deep) learning has mainly gone after datasets such as MNIST or CIFAR-10/100, which have a small number of classes, and many samples per class.

But humans can generalize really well even with a very small number of examples per class! Think of the last time you saw the picture of an unknown animal. You clearly don't need hundreds of examples in order to learn a concept.

Even worse: in several applications, you *can't* get hundreds of examples anyway. Think of building an AI assistant to assist doctors in diagnosis: every test example may be new, critical cases are correlated with how rare they are, and large datasets are hard to find.

Therefore, it is of crucial importance moving forward to devise DL techniques that succeed with relatively few data points. An interesting early test bed is the *Omniglot* dataset, popularized in ML by Brendan Lake at NYU CDS, which can be thought of as "Transpose-MNIST" – lots of classes and very few samples per class. How do we effectively learn in this type of scenario?

Such problems fall into the realm of "few-shot" learning where "shots" here refer to the number of examples. For example, an *n-class k-shot* classification task requires learning to classify between n-classes using only k (potentially $\ll n$) examples per class.

If an ML agent were given a k-shot dataset, how should it solve such a challenging task? The rapidly growing field of *meta-learning* advocates the following principles:

- each learning agent trying to solve a new task is guided by a higher-level *meta-learner*
- the meta-learner possesses *meta-knowledge* (in the form of features, or pre-trained nets, or other quantities) which is imparted to the learning agents when they are being trained.

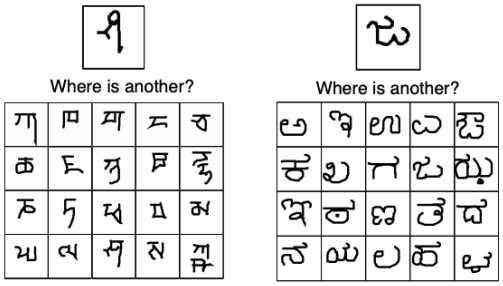


Figure 1: Omniglot dataset

- (here is the crux) the meta-learner *itself* can be trainable, and is able to learn from experience as it teaches different agents.

Approach 1: Transfer learning

Let us be concrete. A canonical example of the above approach iss *transfer learning*. We have actually already discussed transfer learning before (and implemented it for the case of R-CNN type object detection).

The high level idea is that given an ML task with a limited-sample dataset, one starts with a pre-trained *base network* that has already been trained on perhaps a bigger dataset (like ImageNet for images, or Wikipedia for NLP), and uses the given dataset to fine-tune to any new given task.

The problem, of course, is that one requires a good enough base model to start with. In the examples seen so far, the base model has been pre-trained using a massive dataset. The essence of pre-training is to get "good enough" features which generalize well for the given task, and it is not entirely clear if such "good enough" features could be learned in the few-shot setting. Below we will address more principled ways of performing transfer learning in the few-shot setting.

Approach 2: Model-agnostic meta learning (MAML)

Back to transfer learning. A different way of thinking about the few-shot learning problem is to visualize the tasks as different points in the parameter space. In this scenario, transfer learning/fine-tuning can be viewed as a souped-up initialization procedure where we initialize the weights at some known, excellent point in the parameter space, and use the available few-shot data to move the weights to some point better suited to the task.

Of course, this assumes that the new task we are trying to learn is somehow *close enough* to the base model that it can be trained via a few steps of gradient descent. As different tasks are solved, can the meta-learner update the base model itself? If trained over sufficiently many tasks, then perhaps the base model is no longer required to be trained using a *specific*, large dataset – it can be a general model whose only goal is to be "fine-tunable" to different tasks using a few number of gradient descent steps. In that sense, this approach would be *model-agnostic*.

Let us formalize this idea (which is called model-agnostic meta-learning, or MAML). There is a base model ϕ. There are J different tasks. We use the base model f_ϕ – ϕ are the weights – as initialization. For each new task dataset T_j, we form a loss function (based on few-shot samples) $L(f_\phi, T_j)$ – this

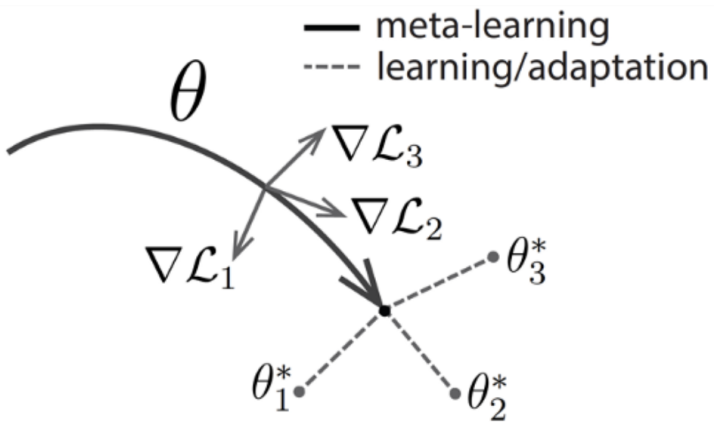


Figure 2: Model-agnostic meta learning (MAML)

stands for the model f with weights ϕ evaluated on training dataset T_j – and fine-tune these weights using gradient descent. In the simplest case, if we used *one* step of gradient descent, this could be written as:

$$\phi_j \leftarrow \phi - \alpha \nabla_\phi L(f_\phi, T_j).$$

If we used two steps of gradient descent, we would have to iterate the above equation twice. And so on. We use the final weights ϕ_j to solve task T_j.

The hope is that if the base model is good enough then the overall cumulative loss across different tasks at the adapted parameters is small as well. This is the *meta-loss function*:

$$M(\phi) = \sum_{j=1}^{J} L(f_{\phi_j}, T_j).$$

Notice the interesting nested structure:

- The meta-loss function $M(\phi)$ depends on the adapted weights ϕ_j
- which in turn depend on the base weights ϕ via one or more steps of gradient descent.

So we can update the base weights themselves by summing up the gradients computed during the adaptation:

$$\begin{aligned}\phi &\leftarrow \phi - \beta \nabla_\phi M(\phi) \\ &= \phi - \beta \sum_j \nabla L(f_{\phi_j}, T_j) \\ &= \phi - \beta \sum_j \nabla L(\phi - \alpha \nabla_\phi L(f_\phi, T_j)).\end{aligned}$$

Some further observations:

- the "samples" in the above update *correspond to different tasks*. One could use stochastic methods here to speed things up: the meta-learner *samples* a new learning agents, "teaches" them how to update their weights by giving them the base model, and "learns" a new set of base model weights.
- "generalization" here corresponds to the fact that after a while, MAML learns parameters that can be adapted to new, unseen tasks via fine-tuning.
- the above equation is specific to the learning agents in MAML using *one step of gradient descent*. But one could use any other optimization method here – k-steps of gradient descent, SGD, Adam, Hessian methods, whatever – call this method Alg. Then a general form of MAML is:

  $$\phi \leftarrow \phi - \beta \sum_j \nabla L(\text{Alg}_j)$$

  The only requirement is that there is some way to take the derivative of Alg in the chain rule – i.e., MAML *works by taking the gradient of gradient descent*!

One last point: The above gradient updates in MAML can be quite complicated. In particular, the meta-gradient update requires a gradient-of-a-gradient (due to the chain rule) and already needs tons of computations. If we want to increase this to k-steps of gradient descent, then we need higher-order gradients. A series of algorithmic improvements have improved this computational dependency on the complexity of the optimizer, but we won't cover it here.

Approach 2: Metric-based meta-learning

An alternative family of meta-learning approaches is learning *metric embeddings*. The high level idea is to learn embeddings (or latent representations) of all data points in a given dataset (similar to how we learned word embeddings in NLP). If the embeddings are meaningful, then the geometry of the embedding space should tell us class information (and we should be able to use simple geometric methods such as nearest neighbors or perceptrons to classify points).

An early approach (pioneered by LeCun and collaborators in the nineties and revived a few years ago) is *Siamese Networks*. The goal was to solve one-shot image classification tasks, where we are given a database of exactly one image in each class.

Imagine a training dataset $(x_1, x_2, \ldots, x_n)$. The label indices don't matter here since all the points are of distinct classes. Siamese nets work as follows.

- set up a Siamese network (pair of identical, weight-tied feedforward convnets, followed by a second network). The first part (pair of identical networks) f_θ consists of a standard convnet mapping data points to some latent set of features; we use this to evaluate every pair of data points and get outputs $f_\theta(x_i)$ and $/f_\theta(x_j)$.
- compute the coordinate-wise distances

  $$g(x_i, x_j) = |f_\theta(x_i) - f_\theta(x_j)|.$$

  This gives a vector which is a measure of similarity between the embeddings.
- Feed it through a second network that gives probabilities of matches, i.e., whether the two images are from the same class. A simple such network would :

  $$p(x_i, x_j) = \sigma(Wg(x_i, x_j))$$

- Apply standard data augmentation techniques (noise, distortion, etc) and train the network using SGD.

- Given a test image, match it with every point in the dataset. The final predicted class is the one with the max matching probability.

$$c(x) = \arg\max_{i \in S} P(x, x_i).$$

This idea was refined to the k-shot case via *Matching Networks* by Vinyals and coauthors in 2016. The steps are similar to the ones above, except that we don't compute distances in the middle, and use a trainable *attention* mechanism (instead of a standard MLP) to declare the final class:

$$c(x) = \arg\max_{i \in S} \sum_{i=1}^{n} \sum_{j=1}^{k} a(x_{nk}, x) y_{nk}.$$

Other attempts along this line of work include:

- Triplet networks (where we use three identical networks and train with triples of samples (x', x'', x^-) – the first two from the same class and the last from a different class.
- Prototypical Networks
- Relation Networks

among several others.

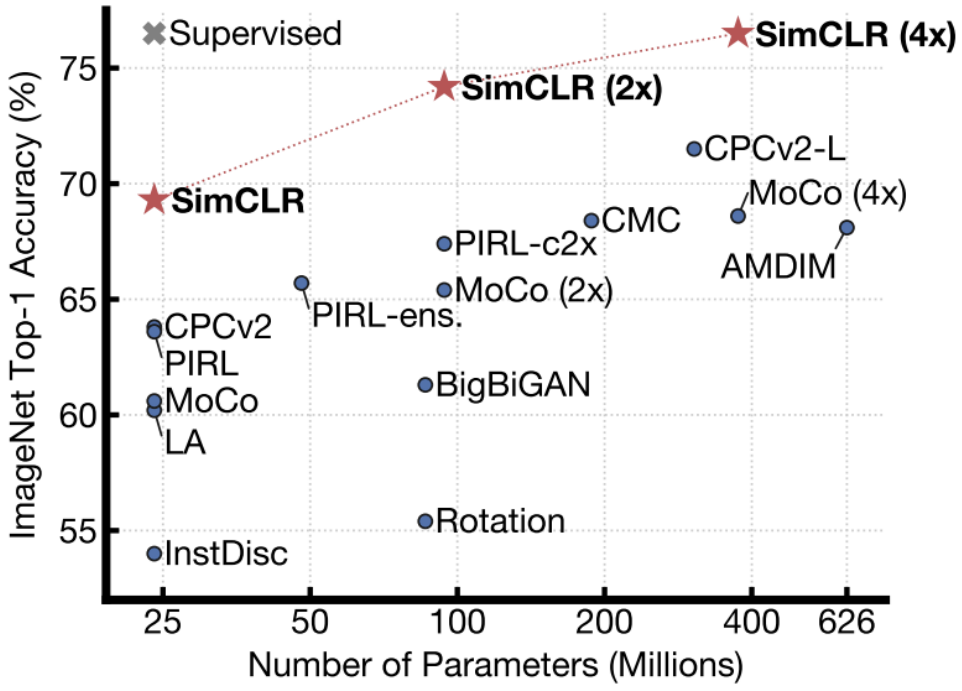
Self-supervised learning

One last note: this idea of using Siamese networks to learn useful embedding features for unlabeled/few-shot datasets is rather similar to the *next-sentence-prediction* task that we used to learn BERT-style representations in NLP.

We can use similar techniques for other data types too! For example, imagine that we were trying to learn embeddings for image- or video- data. The Siamese network idea works here too – as long as we develop a *contrastive pretext* task that enables us to devise embeddings and compare pairs (or triples) of inputs. The above example of Siamese networks corresponded to a "same-class-classification" pretext task. But we could think of others:

- for images, one candidate pretext task could be to predict relative transformations: given two images, predict whether one is a rotation/crop/color transformation of the second.
- for video, one candidate pretext task could be *shuffle-and-learn* where given three frames, the goal is to shuffle the order back ti a temporally coherent manner.
- For audio-video, a candidate pretext task could be match whether the given audio corresponds to the video or not.
- Jigsaw puzzles: the input is a bunch of shuffled tiles, and the goal is to predict a permutation.

All these methods have been applied to varying degrees of success, culminating in SIMCLR (published this year by Hinton and co-authors) which reaches AlexNet-level performance on image classification using 100X fewer labeled samples.



{ width = 50% }

Federated Learning

Let us consider a new setting where the data is distributed over different agents, and the goal is to learn a common model that works for all. Reasons for this setup may include:

- Memory bottlenecks: datasets are so large that a single computing cannot access directly.
- Privacy concerns: datasets may be private to individual agents.

The field of *federated learning* aims to address this. Recent success stories were reported by NVIDIA.

Let us consider the simple "centralized" setting where there is a main node and several worker nodes, each with their own datasets. Multiple settings can be considered here:

- *Federated gradients*: the main node transmits the model weights to all workers, who then use it to compute gradients *with respect to their private dataset*. Each worker node relays the gradient back, and the main node accumulates all of them to get the next set of weights. If the datasets are imbalanced then the main node weights the gradients accordingly. Android phones used this to train next-word prediction on Google keyboard.
- *Federated averaging*: similar to above, except that each worker updates its own weights via gradient descent and relays the new weights back; the main node then averages the models themselves. This pushes some of the computation onto the workers themselves, and has certain benefits in terms of latency.

One can extend this in the distributed case where there is no main node, and everyone is a worker.

This is the “decentralized” setting, and each worker stores a copy of the model, and their exchanges model updates with their neighbor before local averaging. Training finishes when consensus is reached – and this algorithm is called consensus-based distributed SGD; see our paper for details.