19 Learning with fewer labeled examples

Many ML models, especially neural networks, often have many more parameters than we have labeled training examples. For example, a ResNet CNN (Sec. 14.3.2.4) with 50 layers has 23 million parameters. Transformer models (Sec. 15.5) can be even bigger. Of course these parameters are highly correlated, so they are not independent "degrees of freedom". Nevertheless, such big models are slow to train and, more importantly, they may easily overfit. This is particularly a problem when you do not have a large labeled training set. In this chapter, we discuss some ways to tackle this issue, beyond the generic regularization techniques we discussed in Sec. 13.5 such as early stopping, weight decay and dropout.

19.1 Data augmentation

Suppose we just have a single small labeled dataset. In some cases, we may be able to create artificially modified versions of the input vectors, which capture the kinds of variations we expect to see at test time, while keeping the original labels unchanged. This is called **data augmentation**.[1] We give some examples below, and then discuss why this approach works.

19.1.1 Examples

For image classification tasks, standard data augmentation methods include random crops, zooms, and mirror image flips, as illustrated in Fig. 19.1. [GVZ16] gives a more sophisticated example, where they render text characters onto an image in a realistic way, thereby creating a very large dataset of text "in the wild". They used this to train a state of the art visual text localization and reading system. Other examples of data augmentation include artifically adding background noise to clean speech signals, and artificially replacing characters or words at random in text documents.

If we afford to train and test the model many times using different versions of the data, we can learn which augmentations work best, using blackbox optimization methods such as RL (see e.g., [Cub+19]) or Bayesian optimization (see e.g., [Lim+19]); this is called **AutoAugment**. We can also learn to combine multiple augmentations together; this is called **AugMix** [Hen+20].

1. The term "data augmentation" is also used in statistics to mean the addition of auxiliary latent variables to a model in order to speed up convergence of posterior inference algorithms [DM01].

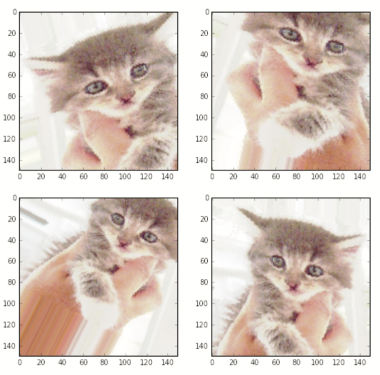


Figure 19.1: *Illustration of random crops, zooms and rotations of some cat images. From [Cho17]. Used with kind permission of Francois Chollet.*

19.1.2 Theoretical justification

Data augmentation often significantly improves performance (predictive accuracy, robustness, etc). At first this might seem like we are getting something for nothing, since we have not provided additional data. However, the data augmentation mechanism can be viewed as a way to algorithmically inject prior knowledge.

To see this, reall that in standard ERM training, we minimize the empirical risk

$$R(f) = \int \ell(f(\mathbf{x}), y) p^*(\mathbf{x}, \mathbf{y}) d\mathbf{x} d\mathbf{y} \tag{19.1}$$

where we approximate $p^*(\mathbf{x}, \mathbf{y})$ by the empirical distribution

$$p_D(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_{n=1}^{N} \delta(\mathbf{x} - \mathbf{x}_n) \delta(\mathbf{y} - \mathbf{y}_n) \tag{19.2}$$

We can think of data augmentation as replacing the empirical distribution with the following algorithmically smoothed distribution

$$p_D(\mathbf{x}, \mathbf{y}|A) = \frac{1}{N} \sum_{n=1}^{N} p(\mathbf{x}|\mathbf{x}_n, A) \delta(\mathbf{y} - \mathbf{y}_n) \tag{19.3}$$

where A is the data augmentation algorithm, which generates a sample $\mathbf{x}$ from a training point $\mathbf{x}_n$, such that the label ("semantics") is not changed. (A very simple example would be a Gaussian kernel, $p(\mathbf{x}|\mathbf{x}_n, A) = \mathcal{N}(\mathbf{x}|\mathbf{x}_n, \sigma^2 \mathbf{I})$.) This has been called **vicinal risk minimization** [Cha+01], since we are minimizing the risk in the vicinity of each training point $\mathbf{x}$. For more details on this perspective, see [Zha+17b; CDL19; Dao+19].

19.2 Transfer learning

This section is coauthored with Colin Raffel.

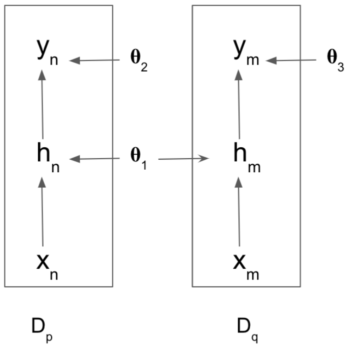


Figure 19.2: Illustration of transfer learning from dataset $\mathcal{D}_p$ to $\mathcal{D}_q$ using a neural network, in which the feature extractor is shared, but the final layer is domain specific. The parameters $\boldsymbol{\theta}_1$ are first trained on $\mathcal{D}_p$, and then optionally fine-tuned on $\mathcal{D}_q$. Thus the information in $\mathcal{D}_p$ is used to help the model work well on $\mathcal{D}_q$, but not vice versa.

Many data-poor tasks have some high-level structural similarity to other data-rich tasks. For example, consider the task of **fine-grained visual classification** of endangered bird species. Given that endangered birds are by definition rare, it is unlikely that a large quantity of diverse labeled images of these birds exist. However, birds bear many structural similarities across species - for example, most birds have wings, feathers, beaks, claws, etc. We therefore might expect that first training a model on a large dataset of non-endangered bird species and then continuing to train it on a small dataset of endangered species could produce better performance than training on the small dataset alone.

This is called **transfer learning**, since we are transferring information from one dataset to another, via a shared set of parameters. More precisely, we first perform a **pre-training phase**, in which we train a model with parameters $\boldsymbol{\theta}$ on a large dataset $\mathcal{D}_p$; this may be labeled or unlabeled. We then perform a second **fine-tuning phase** on the small labeled dataset $\mathcal{D}_q$ of interest. (The fact that the pre-training and fine-tuning tasks differ means that transfer learning is slightly different from semi-supervised learning, which we discuss in Sec. 19.6.) We discuss these two phases in more detail below, but for more information, see e.g., [Tan+18; Zhu+19] for recent surveys.

19.2.1 Fine-tuning

Suppose, for now, that we already have a pretrained classifier, $p(y|\mathbf{x}, \boldsymbol{\theta}_p)$, such as a CNN, that works well for inputs $\mathbf{x} \in \mathcal{X}_p$ (e.g. natural images) and outputs $y \in \mathcal{Y}_p$ (e.g., ImageNet labels), where the data comes from a distribution $p(\mathbf{x}, y)$ similar to the one used in training. Now we want to create a new model $q(y|\mathbf{x}, \boldsymbol{\theta}_q)$ that works well for inputs $\mathbf{x} \in \mathcal{X}_q$ (e.g. bird images) and outputs $y \in \mathcal{Y}_q$ (e.g., fine-grained bird labels). where the data comes from a distribution $q(\mathbf{x}, y)$ which may be different from p.

We will assume that the set of possible inputs is the same, so $\mathcal{X}_q \approx \mathcal{X}_p$ (e.g., both are RGB images), or that we can easily transform inputs from domain p to domain q (e.g., we can convert an RGB image to grayscale by dropping the chrominance channels and just keeping luminance). (If this is not the case, then we may need to use a method called domain adaptation, that modifies models to map between modalities, as discussed in Sec. 19.2.4.)

However, the output domains are usually different, i.e., $\mathcal{Y}_q \neq \mathcal{Y}_p$. For example, $\mathcal{Y}_p$ might be Imagenet labels and $\mathcal{Y}_q$ might be medical labels (e.g., types of diabetic retinopathy [Arc+19]). In this case, we need to "translate" the output of the pre-trained model to the new domain. This is easy to do with neural networks: we simply "chop off" the final layer of the original model, and add a new "head" to model the new class labels, as illustrated in Fig. 19.2. For example, suppose $p(y|\mathbf{x}, \boldsymbol{\theta}_p) = \mathcal{S}(y|\mathbf{W}_2\mathbf{h}(\mathbf{x}; \boldsymbol{\theta}_1) + \mathbf{b}_2)$, where $\boldsymbol{\theta}_p = (\mathbf{W}_2, \mathbf{b}_2, \boldsymbol{\theta}_1)$. Then we can construct $q(y|\boldsymbol{\theta}_q) = \mathcal{S}(y|\mathbf{W}_3^\top \mathbf{h}(\mathbf{x}; \boldsymbol{\theta}_1) + \mathbf{b}_3)$, where $\boldsymbol{\theta}_q = (\mathbf{W}_3, \mathbf{b}_3, \boldsymbol{\theta}_1)$ and $\mathbf{h}(\mathbf{x}; \boldsymbol{\theta}_1)$ is the shared nonlinear feature extractor.

After performing this "model surgery", we can fine-tune the new model with parameters $\boldsymbol{\theta}_q = (\boldsymbol{\theta}_1, \boldsymbol{\theta}_3)$, where $\boldsymbol{\theta}_1$ parameterizes the feature extractor, and $\boldsymbol{\theta}_3$ parameters the final linear layer that maps features to the new set of labels. If we treat $\boldsymbol{\theta}_1$ as "**frozen parameters**", then the resulting model $q(y|\mathbf{x}, \boldsymbol{\theta}_q)$ is linear in its parameters, so we have a convex optimization problem for which many simple and efficient fitting methods exist (see Part II). This is particularly helpful in the long-tail setting, where some classes are very rare [Kan+20]. However, a linear "decoder" may be too limiting, so we can also allow $\boldsymbol{\theta}_1$ to be fine-tuned as well, but using a lower learning rate, to prevent the values moving too far from the values estimated on $\mathcal{D}_p$.

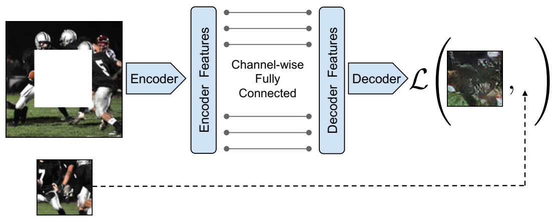
19.2.2 Supervised pre-training

The pre-training task may be supervised or unsupervised; the main requirements are that it can teach the model basic structure about the problem domain and that it is sufficiently similar to the downstream fine-tuning task. The notion of task similarity is not rigorously defined, but in practice the domain of the pre-training task is often more broad than that of the fine-tuning task (e.g., pre-train on all bird species and fine-tune on endangered ones).

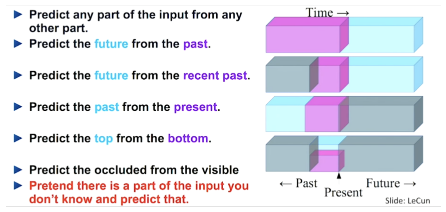
The most straightforward form of transfer learning is the case where a large labeled dataset is suitable for pre-training. For example, it is very common to use the ImageNet dataset (Sec. 14.3.1.2) to pretrain CNNs, which can then be used for an a variety of downstream tasks and datasets (see e.g., [Kol+19]). Imagenet has 1.28 million natural images, each associated with a label from one of 1,000 classes. The classes constitute a wide variety of different concepts, including animals, foods, buildings, musical instruments, clothing, and so on. The images themselves are diverse in the sense that they contain objects from many angles and in many sizes with a wide variety of backgrounds. This diversity and scale may partially explain why it has become a de-facto pre-training task for transfer learning in computer vision.

However, Imagenet pre-training has been shown to be less helpful when the domain of the fine-tuning task is quite different from natural images (e.g. medical images [Rag+19]). And in some cases where it is helpful (e.g., training object detection systems), it seems to be more of a speedup trick (by warm-starting optimization at a good point) rather than something that is essential, in the sense that one can achieve comparable performance on the downstream task when training from scratch, if done for long enough [HGD19].

Supervised pre-training is somewhat less common in non-vision applications. One notable exception is to pre-train on natural language inference data (i.e. whether a sentence implies or contradicts another) to learn vector representations of sentences [Con+17], though this approach has largely



(a)


- Predict any part of the input from any other part.
- Predict the future from the past.
- Predict the future from the recent past.
- Predict the past from the present.
- Predict the top from the bottom.
- Predict the occluded from the visible
- Pretend there is a part of the input you don't know and predict that.

(b)

Figure 19.3: (a) Context encoder for self-supervised learning. From [Pat+16]. Used with kind permission of Deepak Pathak. (b) Some other proxy tasks for self-supervised learning. From [LeC18]. Used with kind permission of Yann LeCun.

been supplanted by unsupervised methods (Sec. 19.2.3). Another non-vision application of transfer learning is to pre-train a speech recognition on a large English-labeled corpus before fine-tuning on low-resource languages [Ard+20].

19.2.3 Unsupervised pre-training (self-supervised learning)

It is increasingly common to use **unsupervised pre-training**, because unlabeled data is often easy to acquire, e.g., unlabeled images or text documents from the web.

For a short period of time it was common to pre-train deep neural networks using an unsupervised objective (e.g., reconstruction error, as discussed in Sec. 20.3) over the labeled dataset (i.e. ignoring the labels) before proceeding with standard supervised training [HOT06; Vin+10b; Erh+10]. While this technique is also called unsupervised pre-training, it differs from the form of pre-training for transfer learning we discuss in this section, which uses a (large) unlabeled dataset for pre-training before fine-tuning on a different (smaller) labeled dataset.

Pre-training tasks that use unlabeled data are often called **self-supervised** rather than unsupervised. This term is used because the labels are created by the algorithm, rather than being provided externally by a human, as in standard supervised learning. Both supervised and self-supervised learning are discriminative tasks, since they require predicting outputs given inputs. By contrast, other unsupervised approaches, such as some of those discussed in Chapter 20, are generative, since they predict outputs unconditionally.

There are many different self-supervised learning heuristics that have been tried (see e.g., [GR18; JT19; Ren19] for a review). We can identify at least three main broad groups, which we discuss below.

19.2.3.1 Imputation tasks

One approach to semi-supervised learning is to solve **imputation tasks**. In this approacg, we partition the input vector $\mathbf{x}$ into two parts, $\mathbf{x} = (\mathbf{x}_h, \mathbf{x}_v)$, and then try to predict the hidden part $\mathbf{x}_h$ given the remaining visible part, $\mathbf{x}_v$, using a model of the form $\hat{\mathbf{x}}_h = f(\mathbf{x}_v, \mathbf{x}_h = \mathbf{0})$. We can think of this as a "**fill-in-the-blank**" task; in the NLP community, this is called a **cloze task**. See Fig. 19.3 for some visual examples, and Sec. 19.5.5.3 for some NLP examples.

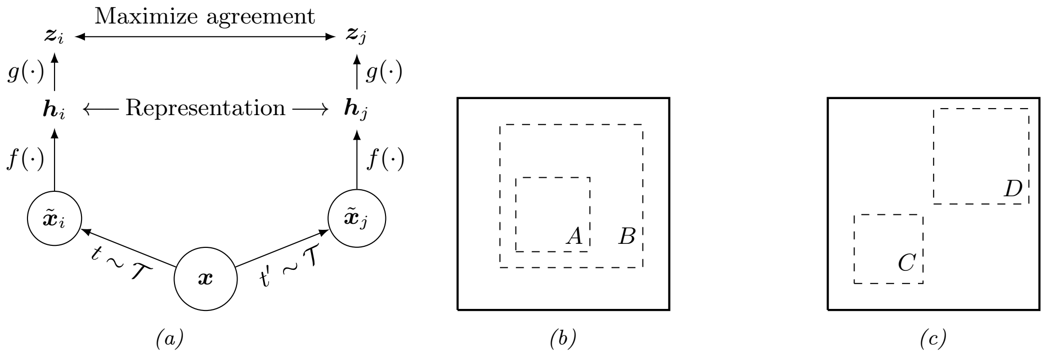


Figure 19.4: (a) Illustration of SimCLR training. $\mathcal{T}$ is a set of stochastic semantics-preserving transformations (data augmentations). (b-c) Illustration of the benefit of random crops. Solid rectangles represent the original image, dashed rectangles are random crops. On the left, the model is forced to predict the local view A from the global view B (and vice versa). On the right, the model is forced to predict the appearance of adjacent views (C,D). From Figures 2–3 of [Che+20b]. Used with kind permission of Ting Chen.

19.2.3.2 Proxy tasks

Another approach to SSL is to solve **proxy tasks**, also called **pretext tasks**. In this setup, we create pairs of inputs, $(\mathbf{x}_1, \mathbf{x}_2)$, and then train a Siamese network classifier (Fig. 16.5a) of the form $p(y|\mathbf{x}_1, \mathbf{x}_2) = p(y|f(r(\mathbf{x}_1), r(\mathbf{x}_2))$, where $r(\mathbf{x})$ is some function that performs "**representation learning**" [BCV13], and y is some label that captures the relationship between $\mathbf{x}_1$ and $\mathbf{x}_2$. For example, if $\mathbf{x}_1$ is an image patch, and $\mathbf{x}_2$ is a random rotation of $\mathbf{x}_1$ that we create, we can try to predict the rotation angle y [GSK18].

19.2.3.3 Contrastive tasks

The currently most popular approach to self-supervised learning is to use various kinds of **contrastive tasks**. The basic idea is to create pairs of examples that are semantically similar to each other, using data augmentation methods (Sec. 19.1), and then to ensure that the distance between their representations is closer (in embedding space) than the distance between two unrelated examples. This is exactly the same idea that is used in deep metric learning (Sec. 16.2.2) — the only difference is that the algorithm creates its own similar pairs, rather than relying on an externally provided measure of similarity, such as labels.

As an example of such an approach, consider the **SimCLR** (simple contrastive learning of visual representations) method of [Che+20b; Che+20c], which has shown state of the art performance on transfer learning and semi-supervised learning. The basic idea is as follows. Each input $\mathbf{x}$ is randomly modified to create two versions, $\mathbf{x}_i$ and $\mathbf{x}_j$. This is done for all N examples in the batch. We then compute a representation of each input, $\mathbf{h}_i = f(\mathbf{x}_i)$, as well as the final embedding, $\mathbf{z}_i = g(\mathbf{h}_i)$, which is then ℓ_2 normalized. See Fig. 19.4a.

Define the similarity between a pair of inputs as follows:

$$S_{ij} = \frac{\mathbf{z}_i^\top \mathbf{z}_j}{||\mathbf{z}_i|| \; ||\mathbf{z}_j||} \tag{19.4}$$

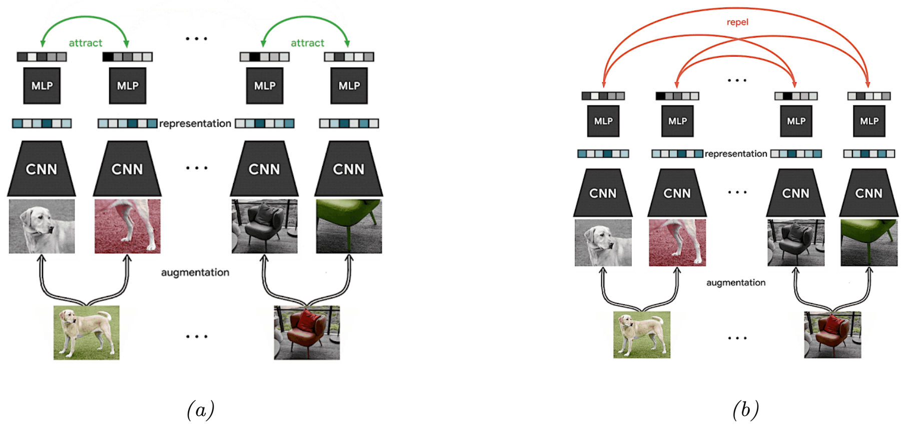


Figure 19.5: Visualization of SimCLR training. Each input image in the minibatch is randomly modified in two different ways (using cropping (followed by resize), flipping, and color distortion), and then fed into a Siamese network. The embeddings (final layer) for each pair derived from the same image is forced to be close, whereas the embeddings for all other pairs are forced to be far. From https://ai.googleblog.com/2020/04/advancing-self-supervised-and-semi.html. Used with kind permission of Ting Chen.

We now define the loss function for a positive pair (i, j) to be the normalized temperature cross-entropy (**NT-Xent**) loss, defined as follows:

$$\ell_{ij}(\boldsymbol{\theta}) = -\log \frac{\exp(S_{ij}/\tau)}{\sum_{k=1}^{2N} \mathbb{I}(k \neq i) \exp(S_{ik}/\tau)} \tag{19.5}$$

where $\tau > 0$ is a temperature parameter. (This is identical to the N-pairs loss discussed in Sec. 16.2.4.3, apart from the temperature term.) The final loss for a minibatch is given by

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \ell_{i,j=\text{pos}(i)}(\boldsymbol{\theta}) \tag{19.6}$$

where pos(i) is the (index of the) positive counterpart for example i. The net effect is to pull similar pairs close together, and to force dissimilar pairs far apart, as illustrated in Fig. 19.5.

A critical ingredient to the success of SimCLR is the choice of data augmentation methods. By using random cropping, they can force the model to predict local views from global views, as well as to predict adjacent views of the same image (see Fig. 19.4). After cropping, all images are resized back to the same size. In addition, they randomly flip the image some fraction of the time.

However, it turns out that distinguishing positive crops (from the same image) from negative crops (from different images) is often easy to do just based on color histograms. To prevent this kind of "cheating", they also apply a random color distortion, thus cutting off this "short circuit". The combination of random cropping and color distortion is found to work better than either method alone.

After training, the final g mapping (known as the **projection head**) is thrown away, and we use $\mathbf{h}_i = f(\mathbf{x}_i)$ as our representation in downstream tasks. One reason this may help, according to [Che+20b], is that the mapping $\mathbf{z}_i = g(\mathbf{h}_i)$ might throw away information that is not necessary for the contrastive task, such as the color or orientation of objects, but which may be useful for downstream tasks, so it is better to keep it. (In [Che+20c], they propose SimCLR2, which keeps "part of" the projection head g.)

SimCLR relies on large batch training, in order to ensure a sufficiently diverse set of negatives. When this is not possible, we can use a memory bank of past (negative) embeddings, which can be updated using exponential moving averaging (Sec. 4.4.2.2). This is known as **momentum contrastive learning** or **MoCo** [He+20].

19.2.4 Domain adaptation

Consider a problem in which we have inputs from different domains, such as a **source domain** $\mathcal{X}_s$ and **target domain** $\mathcal{X}_t$, but a common set of output labels, $\mathcal{Y}$. (This is the "dual" of transfer learning, since the input domains are different, but the output domains the same.) For example, the domains might be images from a computer graphics system and real images, or product reviews and movie reviews. We assume we do not have labeled examples from the target domain. Our goal is to fit the model on the source domain, and then modify its parameters so it works on the target domain. This is called (unsupervised) **domain adaptation** (see e.g., [KL19] for a review).

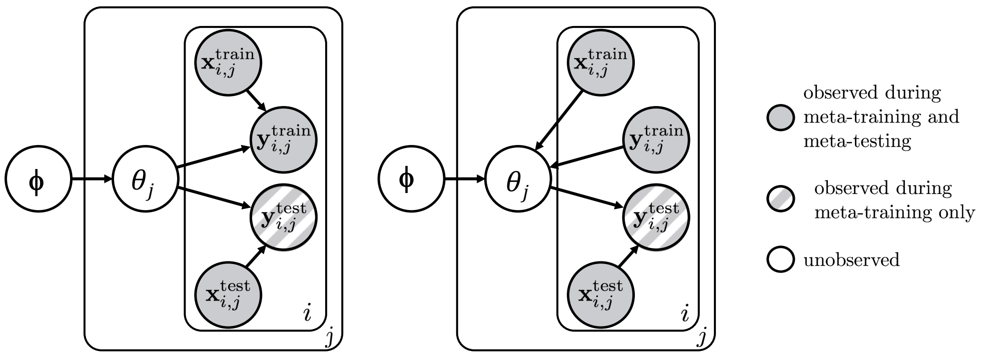
A common approach to this problem is to train the source classifier in such a way that it cannot distinguish whether the input is coming from the source or target distribution; in this case, it will only be able to use features that are common to both domains. This is called **domain adversarial learning** [Gan+16]. More formally, let $d_n \in \{s, t\}$ be a label that specifies if the data example n comes from domain s or t. We want to optimize

$$\min_{\boldsymbol{\phi}} \max_{\boldsymbol{\theta}} \frac{1}{N_s + N_t} \sum_{n \in \mathcal{D}_s, \mathcal{D}_t} \ell(d_n, f_{\boldsymbol{\theta}}(\mathbf{x}_n)) + \frac{1}{N_s} \sum_{m \in \mathcal{D}_s} \ell(y_m, g_{\boldsymbol{\phi}}(f_{\boldsymbol{\theta}}(\mathbf{x}_m))) \tag{19.7}$$

where $N_s = |\mathcal{D}_s|$, $N_t = |\mathcal{D}_t|$, f maps $\mathcal{X}_s \cup \mathcal{X}_t \to \mathcal{H}$, and g maps $\mathcal{H} \to \mathcal{Y}_t$. The objective in Eq. (19.7) minimizes the loss on the desired task of classifying y, but *maximizes* the loss on the auxiliary task of classifying the source domain d. This can be implemented by the **gradient sign reversal** trick, and is related to GANs (generative adversarial networks). See e.g., [Csu17; Wu+19] for some other approaches to domain adaptation.

19.3 Meta-learning *

The field of **meta learning**, also called **learning to learn** [TP97], is concerned with learning multiple related functions (often called "**tasks**"). For example, suppose we have a set of J related datasets, $\{\mathcal{D}^j : j = 1 : J\}$, where $\mathcal{D}^j = \{(\mathbf{x}_n^j, y_n^j) : n = 1 : N_j\}$, $\mathbf{x}_n^j \sim p(\mathbf{x})$, and $y_n^j = f^j(\mathbf{x}_n^j)$. (For example, $p(\mathbf{x})$ could be a distribution over images, f^1 could be a function mapping images to dog breeds, f^2 could be a function mapping images to car types, etc.) We can apply this inner algorithm to generate a labeled set $\mathcal{D}_{\text{meta}} = \{(\mathcal{D}^j, f^j) : j = 1 : J\}$, which we can use to learn an outer or "meta" algorithm or function $\hat{M}$ that *maps a dataset to a prediction function*, $\hat{f}^j = \hat{M}(\mathcal{D}^j)$ (c.f., [Min99]). By contrast, conventional supervised learning learns a function $\hat{f}$ that maps a single example to a label, $\hat{y}_n = \hat{f}(\mathbf{x}_n)$.



ϕ
θ_j
$\mathbf{x}_{i,j}^{\text{train}}$
$\mathbf{y}_{i,j}^{\text{train}}$
$\mathbf{y}_{i,j}^{\text{test}}$
$\mathbf{x}_{i,j}^{\text{test}}$
i
j
observed during meta-training and meta-testing
observed during meta-training only
unobserved

Figure 19.6: Graphical model corresponding to MAML. Left: generative model. Right: During meta-training, each of the task parameters $\boldsymbol{\theta}_j$'s are updated using their local datasets. The indices j are over tasks (meta datasets), and i are over instances within each task. Solid shaded nodes are always observed; semi-shaded (striped) nodes are only observed during meta training time (i.e., not at test time). From Figure 1 of [FXL18]. Used with kind permission of Chelsea Finn.

There are many other approaches to meta-learning (see e.g., [Van18] for a survey). In Sec. 19.3.1, we discuss one popular approach, known as **MAML**, which can be thought of as an approximate form of empirical Bayes in a hierarchical Bayesian model.

There are also many applications of meta-learning. We discuss one of the most popular examples in Sec. 19.4.

19.3.1 Model-agnostic meta-learning (MAML)

A natural approach to meta learning is to use a hierarchical Bayesian model, as illustrated in Fig. 19.6. The parameters for each task $\boldsymbol{\theta}_j$ are assumed to come from a common prior, $p(\boldsymbol{\theta}_j|\boldsymbol{\phi})$, which can be used to help pool statistical strength from multiple data-poor problems.

We could perform Bayesian inference for the task parameters $\boldsymbol{\theta}_j$ and the shared hyper-parameters $\boldsymbol{\phi}$, but a more efficient approach is to use the following empirical Bayes (Sec. 4.6.5.3) approximation:

$$\boldsymbol{\phi}^* = \operatorname*{argmax}_{\boldsymbol{\phi}} \frac{1}{J} \sum_{j=1}^{J} \log p(\mathcal{D}_{\text{valid}}^j | \hat{\boldsymbol{\theta}}_j(\boldsymbol{\phi}, \mathcal{D}_{\text{train}}^j)) \tag{19.8}$$

where $\hat{\boldsymbol{\theta}}_j = \hat{\boldsymbol{\theta}}(\boldsymbol{\phi}, \mathcal{D}_{\text{train}}^j)$ is a point estimate of the parameters for task j based on $\mathcal{D}_{\text{train}}^j$ and prior $\boldsymbol{\phi}$, and where we use a cross-validation approximation to the marginal likelihood (Sec. 5.4.4).

To compute the point estimate of the task parameters, $\hat{\boldsymbol{\theta}}_j$, we use K steps of a gradient ascent procedure starting at $\boldsymbol{\phi}$ with a learning rate of η. This can be shown to be equivalent to an approximate MAP estimate using a Gaussian prior centered at $\boldsymbol{\phi}$, where the strength of the prior is controlled by the number of gradient steps [San96; Gra+18]. (This is an example of **fast adapation** of the task specific weights starting from the shared prior $\boldsymbol{\phi}$.)

Thus we see that learning the prior $\boldsymbol{\phi}$ is equivalent to learning a good initializer for task-specific learning. This is known as **model-agnostic meta-learning** or **MAML** [FAL17].

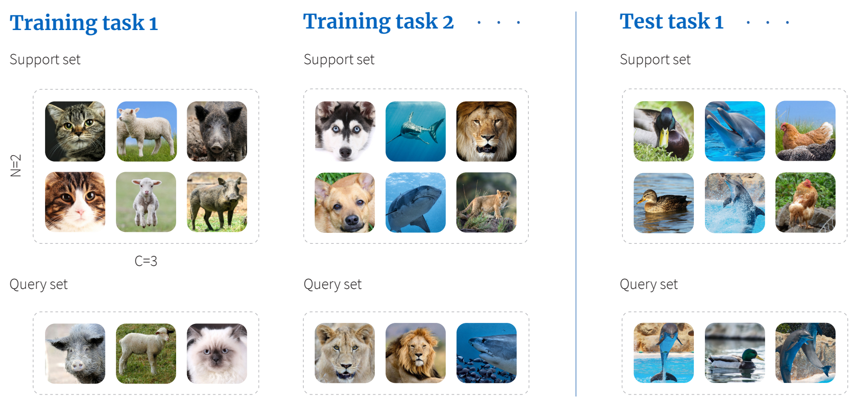


Figure 19.7: Illustration of meta-learning for few-shot learning. Here, each task is a 3-way-2-shot classification problem because each training task contains a support set with three classes, each with two examples. From https://www.borealisai.com/en/blog/tutorial-2-few-shot-learning-and-meta-learning-i. Copyright (2019) Borealis AI. Used with kind permission of Simon Prince and April Cooper.

19.4 Few-shot learning *

People can learn to predict from very few labeled examples. This is called **few-shot learning**. In the extreme in which the person or system learns from a single example of each class, this is called **one-shot learning**, and if no labeled examples are given, it is called **zero-shot learning**.

A common way to evaluate methods for FSL is to use **C-way N-shot classification**, in which the system is expected to learn to classify C classes using just N training examples of each class. Typically N and C are very small, e.g., Fig. 19.7 illustrates the case where we have $C = 3$ classes, each with $N = 2$ examples. Since the amount of data from the new domain (here, ducks, dolphins and hens) is so small, we cannot expect to learn from scratch. Therefore we turn to meta-learning.

During training, the meta-algorithm M trains on a labeled support set from group j, returns a predictor f^j, which is then evaluated on a disjoint query set also from group j. We optimize M over all J groups. Finally we can apply M to our new labeled support set to get f^{test}, which is applied to the query set from the test domain. This is illustrated in Fig. 19.7. We see that there is no overlap between the classes in the two training tasks ({cat, lamb, pig} and {dog, shark, lion}) and those in the test task ({duck, dolphin, hen}). Thus the algorithm M must learn to predict image classes in general rather than any particular set of labels.

There are many approaches to few-shot learning. We discuss one such method in Sec. 19.4.1. For more methods, see e.g., [Wan+20b].

19.4.1 Matching networks

One approach is to few shot learning is learn a distance metric on some other dataset, and then to use $d_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{x}')$ inside of a nearest neighbor classifier. Essentially this defines a semi-parametric model of the form $p_{\boldsymbol{\theta}}(y|\mathbf{x}, \mathcal{S})$, where $\mathcal{S}$ is the small labeled dataset (known as the support set), and $\boldsymbol{\theta}$ are the parameters of the distance function. This approach is widely used for **fine-grained classification**

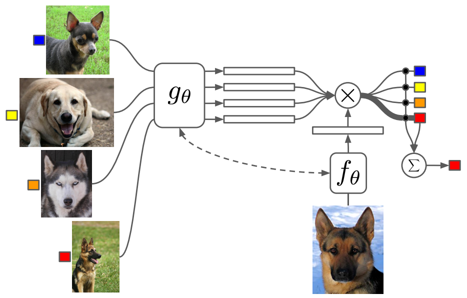


Figure 19.8: Illustration of a matching network for one-shot learning. From Figure 1 of [Vin+16]. Used with kind permission of Oriol Vinyals.

tasks, where there are many different visually similar categories, such as face images from a gallery, or product images from a catalog.

An extension of this approach is to learn a function of the form

$$p_{\boldsymbol{\theta}}(y|\mathbf{x}, \mathcal{S}) = \mathbb{I}\left(y = \sum_{n \in \mathcal{S}} a_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{x}_n; \mathcal{S}) y_n\right) \tag{19.9}$$

where $a_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{x}_n; \mathcal{S}) \in \mathbb{R}^+$ is some kind of adaptive similarity kernel. For example, we can use an **attention kernel** of the form

$$a(\mathbf{x}, \mathbf{x}_n; \mathcal{S}) = \frac{\exp(c(f(\mathbf{x}), g(\mathbf{x}_n)))}{\sum_{n'=1}^{N} \exp(c(f(\mathbf{x}), g(\mathbf{x}_{n'})))} \tag{19.10}$$

where $c(\mathbf{u}, \mathbf{v})$ is the cosine distance. (We can make f and g be the same function if we want.) Intuitively, the attention kernel will compare $\mathbf{x}$ to $\mathbf{x}_n$ in the context of all the labeled examples, which provides an implicit signal about which feature dimensions are relevant. (We discuss attention mechanisms in more detail in Sec. 15.4.) This is called a **matching network** [Vin+16]. See Fig. 19.8 for an illustration.

We can train the f and g functions using multiple small datasets, as in meta-learning (Sec. 19.3). More precisely, let $\mathcal{D}$ be a large labeled dataset (e.g., ImageNet), and let $p(\mathcal{L})$ be a distribution over its labels. We create a task by sampling a small set of labels (say 25), $\mathcal{L} \sim p(\mathcal{L})$, and then sampling a small support set of examples from $\mathcal{D}$ with those labels, $\mathcal{S} \sim \mathcal{L}$, and finally sampling a small test set with those same labels, $\mathcal{T} \sim \mathcal{L}$. We then train the model to predict the test labels given the support set, i.e., we optimize the following objective:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \mathbb{E}_{\mathcal{L} \sim p(\mathcal{L})}\left[\mathbb{E}_{\mathcal{S} \sim \mathcal{L}, \mathcal{T} \sim \mathcal{L}}\left[\sum_{(\mathbf{x}, y) \in \mathcal{T}} \log p_{\boldsymbol{\theta}}(y|\mathbf{x}, \mathcal{S})\right]\right] \tag{19.11}$$

After training, we freeze $\boldsymbol{\theta}$, and apply Eq. (19.9) to a test support set $\mathcal{S}$.

19.5 Word embeddings

Words are categorical random variables, so their corresponding one-hot vector representations are sparse. The problem with this binary representation is that semantically similar words may have very different vector representations. For example, the pair of related words "man" and "woman" will be Hamming distance 1 apart, as will the pair of unrelated words "man" and "banana".

The standard way to solve this problem is to use **word embeddings**, in which we map each sparse one-hot vector, $\mathbf{s}_{n,t} \in \{0,1\}^M$, representing the t'th word in document n, to a lower-dimensional dense vector, $\mathbf{z}_{n,t} \in \mathbb{R}^D$, such that semantically similar words are placed close by. This can significantly help with data sparsity. There are many ways to learn such embeddings, as we discuss below.

Before discussing methods, we have to define what we mean by "semantically similar" words. We will assume that two words are semantically similar if they occur in similar contexts. This is known as the **distributional hypothesis** [Har54], which is often summarized by the phase (originally from [Fir57]) "a word is characterized by the company it keeps". Thus the methods we discuss will all learn a mapping from a word's context to an embedding vector for that word.

19.5.1 Methods based on SVD

In this section, we discuss a simple way to learn word embeddings based on singular value decomposition (Sec. 7.5) of a term-frequency count matrix.

19.5.1.1 Latent semantic indexing (LSI)

Let C_{ij} be the number of times "term" i occurs in "context" j. The definition of what we mean by "term" is application-specific. In English, we often take it to be the set of unique tokens that are separated by punctuation or whitespace; for simplicity, we will call these "words". However, we may preprocess the text data to remove very frequent or infrequent words, or perform other kinds of preprocessing. as we discuss in Sec. 10.4.3.1.

The definition of what we mean by "context" is also application-specific. In this section, we count how many times word i occurs in each document $j \in \{1,\ldots,N\}$ from a set or **corpus** of documents; the resulting matrx $\mathbf{C}$ is called a **term-document frequency matrix**, as in Fig. 10.10. (Sometimes we apply the TF-IDF transformation to the counts, as discussed in Sec. 10.4.3.2.)

Let $\mathbf{C} \in \mathbb{R}^{M \times N}$ be the count matrix, and let $\hat{\mathbf{C}}$ be the rank K approximation that minimizes the following loss:

$$\mathcal{L}(\hat{\mathbf{C}}) = ||\mathbf{C} - \hat{\mathbf{C}}||_F = \sum_{ij} (C_{ij} - \hat{C}_{ij})^2 \tag{19.12}$$

One can show that the minimizer of this is given by the rank K truncated SVD approximation, $\hat{\mathbf{C}} = \mathbf{USV}$. This means we can represent each c_{ij} as a bilinear product:

$$c_{ij} \approx \sum_{k=1}^{K} u_{ik} s_k v_{jk} \tag{19.13}$$

We define $\mathbf{u}_i$ to be the embedding for word i, and $\mathbf{s} \odot \mathbf{v}_j$ to be the embedding for context j.

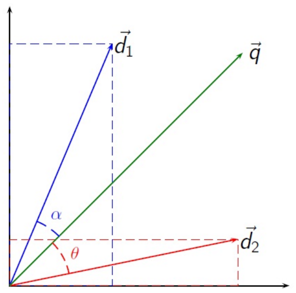


Figure 19.9: Illustration of the cosine similarity between a query vector $\mathbf{q}$ *and two document vectors* $\mathbf{d}_1$ *and* $\mathbf{d}_2$*. Since angle* α *is less than angle* θ*, we see that the query is more similar to document 1. From* *https://en.wikipedia.org/wiki/Vector_space_model*. *Used with kind permission of Wikipedia author Riclas.*

We can use these embeddings for **document retrieval**. The idea is to compute an embedding for the query words using $\mathbf{u}_i$, and to compare this to the embedding of all the documents or contexts $\mathbf{v}_j$. This is known as **latent semantic indexing** or **LSI** [Dee+90].

In more detail, suppose the query is a bag of words $w_1, \ldots, w_B$; we represent this by the vector $\mathbf{q} = \frac{1}{B}\sum_{b=1}^{B} \mathbf{u}_{w_b}$, where $\mathbf{u}_{w_b}$ is the embedding for word w_b. Let document j be represented by $\mathbf{v}_j$. We then rank documents by the **cosine similarity** between the query vector and document, defined by

$$\text{sim}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q}^\top \mathbf{d}}{||\mathbf{q}|| \, ||\mathbf{d}||} \tag{19.14}$$

where $||\mathbf{q}|| = \sqrt{\sum_i q_i^2}$ is the ℓ_2-norm of $\mathbf{q}$. This measures the angles between the two vectors, as shown in Fig. 19.9. Note that if the vectors are unit norm, cosine similarity is the same as inner product; it is also equal to the squared Euclidean distance, up to a change of sign and an irrelevant additive constant:

$$||\mathbf{q} - \mathbf{d}||^2 = (\mathbf{q} - \mathbf{d})^\top(\mathbf{q} - \mathbf{d}) = \mathbf{q}^\top\mathbf{q} + \mathbf{d}^\top\mathbf{d} - 2\mathbf{q}^\top\mathbf{d} = 2(1 - \text{sim}(\mathbf{q}, \mathbf{d})) \tag{19.15}$$

19.5.1.2 Latent semantic analysis (LSA)

Now suppose we define context more generally to be some local neighborhood of words $j \in \{1, \ldots, M^h\}$, where h is the window size. Thus C_{ij} is how many times word i occurs in a neighborhood of type j. We can compute the SVD of this matrix as before, to get $c_{ij} \approx \sum_{k=1}^{K} u_{ik} s_k v_{jk}$. We define $\mathbf{u}_i$ to be the embedding for word i, and $\mathbf{s} \odot \mathbf{v}_j$ to be the embedding for context j. This is known as **latent semantic analysis** or **LSA** [Dee+90].

For example, suppose we compute $\mathbf{C}$ on the British National Corpus.[2] For each word, let us retrieve the K nearest neighbors in embedding space ranked by cosine similarity (i.e., normalized inner product). If the query word is "dog", and we use $h = 2$ or $h = 30$, the nearest neighbors are as follows:

2. This example is taken from [Eis19, p312].

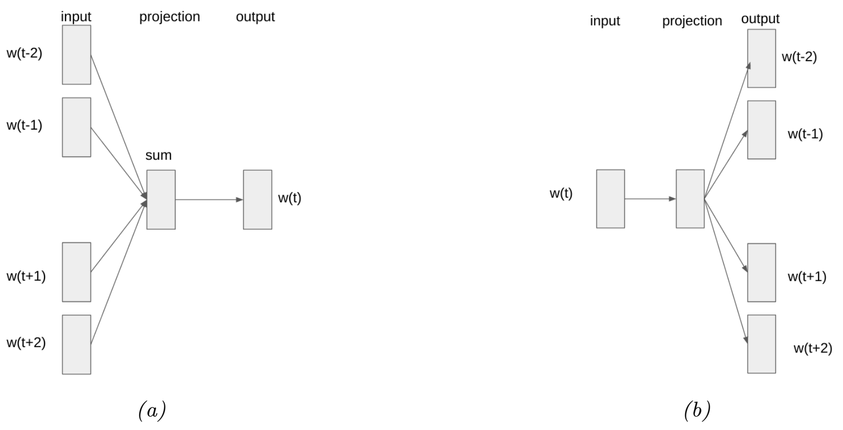


Figure 19.10: Illustration of word2vec model with window size $H = 2$. (a) CBOW version. (b) Skip-gram version. .

```
h=2: cat, horse, fox, pet, rabbit, pig, animal, mongrel, sheep, pigeon
h=30: kennel, puppy, pet, bitch, terrier, rottwiler, canine, cat, to bark
```

The 2-word context window is more sensitive to syntax, while the 30-word window is more sensitive to semantics. The "optimal" value of context size h depends on the application.

19.5.1.3 PMI

In practice LSA (and other similar methods) give much better results if we replace the raw counts C_{ij} with **pointwise mutual information (PMI)** [CH90], defined as

$$\mathbb{PMI}(i, j) = \log \frac{p(i, j)}{p(i)p(j)} \tag{19.16}$$

If word i is strongly associated with context j, we will have $\mathbb{PMI}(i, j) > 0$. If the PMI is negative, it means i and j co-occur less often that if they were independent; however, such negative correlations can be unreliable, so it is common to use the **positive PMI**: $\mathbb{PPMI}(i, j) = \max(\mathbb{PMI}(i, j), 0)$. In [BL07], they show that SVD applied to the PPMI matrix results in word embeddings that perform well on a many tasks related to word meaning. See Sec. 19.5.3 for a theoretical model that explains this empirical performance.

19.5.2 Word2vec

In this section, we discuss the popular **word2vec** model from [Mik+13a; Mik+13b], which are "shallow" neural nets for predicting a word given its context. In Sec. 19.5.3, we will discuss the connections with SVD of the PMI matrix.

There are two versions of the word2vec model. The first is called CBOW, which stands for "continuous bag of words". The second is called skipgram. We discuss both of these below.

19.5.2.1 Word2vec CBOW model

In the continuous bag of words (**CBOW**) model (see Fig. 19.10(a)), the log likelihood of a sequence of words is computed using the following model:

$$\log p(\mathbf{w}) = \sum_{t=1}^{T} \log p(w_t|\mathbf{w}_{t-H:t+H}) = \sum_{t=1}^{T} \log \frac{\exp(\mathbf{v}_{w_t}^\top \overline{\mathbf{v}}_t)}{\sum_{w'} \exp(\mathbf{v}_{w'}^\top \overline{\mathbf{v}}_t)} \tag{19.17}$$

$$= \sum_{t=1}^{T} \mathbf{v}_{w_t}^\top \overline{\mathbf{v}}_t - \log \sum_{w'} \exp(\mathbf{v}_{w'}^\top \overline{\mathbf{v}}_t) \tag{19.18}$$

where $\mathbf{v}_{w_t}$ is the vector for the word at location w_t, and

$$\overline{\mathbf{v}}_t = \frac{1}{2H} \sum_{h=1}^{H} (\mathbf{v}_{w_{t+h}} + \mathbf{v}_{w_{t-h}}) \tag{19.19}$$

is the average of the word vectors in the window of size H around word w_t. Thus we try to predict each word given its context. The model is called CBOW because it uses a bag of words assumption for the context, and represents each word by a continuous embedding.

19.5.2.2 Word2vec Skip-gram model

In CBOW, each word is predicted from its context. A variant of this is to predict the context (surrounding words) given each word. This yields the following objective:

$$\log p(\mathbf{w}) \approx \sum_{t=1}^{T} \left[\sum_{h=1}^{H_t} \log p(w_{t-h}|w_t) + \log p(w_{t+h}|w_t) \right] \tag{19.20}$$

$$= \sum_{t=1}^{T} \left[\sum_{h=1}^{H_t} \mathbf{u}_{w_{t-h}}^\top \mathbf{v}_{w_t} + \mathbf{u}_{w_{t+h}}^\top \mathbf{v}_{w_t} - 2 \log \sum_{w'} \exp(\mathbf{v}_{w_t}^\top \mathbf{u}_{w'}) \right] \tag{19.21}$$

where H_t is a randomly sampled window length for location t, and $\mathbf{u}_w$ is the embedding vector of word w when used as a context word as opposed to a predicted word. See Fig. 19.10(b) for an illustration.

We can approximate this objective by sampling a single context word c_t for each w_t, instead of summing over all words $w_{t-h}, \ldots, w_{t+h}$ in the window:

$$\log p(\mathbf{w}) \approx \sum_{t} \log p(c_t|w_t) = \sum_{t} \left[\mathbf{u}_{c_t}^\top \mathbf{v}_{w_t} - \log \sum_{w'} \exp(\mathbf{v}_{w_t}^\top \mathbf{u}_{w'}) \right] \tag{19.22}$$

This model is known as the **skipgram model**, since we are working with H-gram models, but skipping most of the terms.

To avoid the costly sum over all words w', we can sum over a smaller set of negative words for w_t:

$$\log p(\mathbf{w}) \approx \sum_{t} \left[\mathbf{u}_{c_t}^\top \mathbf{v}_{w_t} - \sum_{w' \in \text{neg}_t} \log(1 - \sigma(\mathbf{v}_{w_t}^\top \mathbf{u}_{w'})) \right] \tag{19.23}$$

where neg_t are a set of randomly chosen words that do not occur in the context window of w_t. This approach is called **skip-gram with negative sampling** (**SGNS**), and is much faster than standard skip-gram training. SGNS training is slower than CBOW training, but tends to learn better word embeddings than CBOW.

In [LG14], they show that SGNS is equivalent to SVD applied to a shifted version of the PMI matrix, defined as

$$\text{shiftedPMI}(i,j) \triangleq \mathbb{PMI}(i,j) - \log(N^-) \tag{19.24}$$

where N^- is the number of negative contexts sampled for each word i.

19.5.3 RAND-WALK model of word embeddings

Word embeddings significantly improve the performance of various kinds of NLP models compared to using one-hot encodings for words. It is natural to wonder why the above word embeddings work so well. In this section, we give a simple generative model for text documents that explains this phenomenon, based on [Aro+16].

Consider a sequence of words $w_1, \ldots, w_T$. We assume each word is generated by a latent context or discourse vector $\mathbf{z}_t \in \mathbb{R}^D$ using the following **log bilinear language model**, similar to [MH07]:

$$p(w_t = w|\mathbf{z}_t) = \frac{\exp(\mathbf{z}_t^\top \mathbf{v}_w)}{\sum_{w'} \exp(\mathbf{z}_t^\top \mathbf{v}_{w'})} = \frac{\exp(\mathbf{z}_t^\top \mathbf{v}_w)}{Z(\mathbf{z}_t)} \tag{19.25}$$

where $\mathbf{v}_w \in \mathbb{R}^D$ is the embedding for word w, and $Z(\mathbf{z}_t)$ is the partition function. We assume $D < M$, the number of words in the vocabulary.

Let us further assume the prior for the word embeddings $\mathbf{v}_w$ is an isotropic Gaussian, and that the latent topic $\mathbf{z}_t$ undergoes a slow Gaussian random walk. (This is therefore called the **RAND-WALK** model.) Under this model, one can show that $Z(\mathbf{z}_t)$ is approximately equal to a fixed constant, Z, independent of the context. This is known as the **self-normalization property** of log-linear models [AK15]. Furthermore, one can show that the pointwise mutual information of predictions from the model is given by

$$\mathbb{PMI}(w, w') \approx \frac{\mathbf{v}_w^\top \mathbf{v}_{w'}}{D} \tag{19.26}$$

We can therefore fit the RAND-WALK model by matching the model's predicted values for PMI with the empirical values, i.e., we minimize

$$\mathcal{L}(\mathbf{V}) = \sum_{w,w'} X_{w,w'} (\mathbb{PMI}(w, w') - \mathbf{v}_w^\top \mathbf{v}_{w'})^2 \tag{19.27}$$

where $X_{w,w'}$ is the number of times w and w' occur next to each other. This objective can be seen as a frequency-weighted version of the SVD loss in Eq. (19.12).

Furthermore, some additional approximations can be used to show that the NLL for the RAND-WALK model is equivalent to the CBOW and SGNS word2vec objectives. We can also derive the objective for the popular **GloVe** model of [PSM14a]. (GloVe stands for "global Vectors for word representation".)

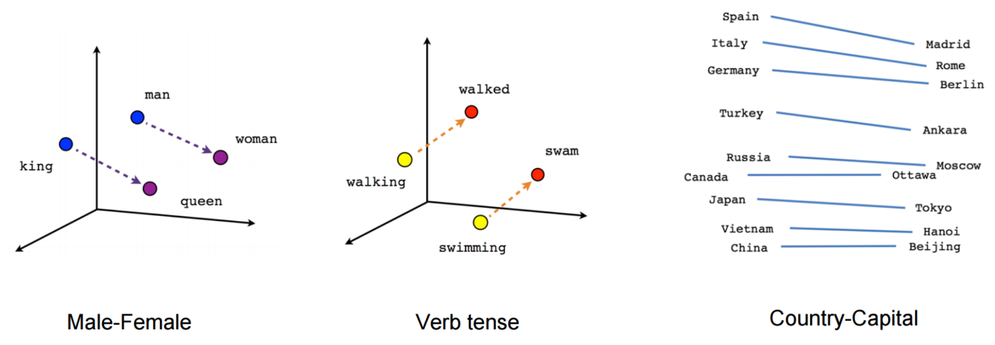


Figure 19.11: Visualization of arithmetic operations in word2vec embedding space. From https://www.tensorflow.org/tutorials/representation/word2vec.

19.5.4 Word analogies

One of the most remarkable properties of word embeddings produced by word2vec, GloVe, and other similar methods is that the learned vector space seems to capture relational semantics in terms of simple vector addition. In particular, we can solve **word analogy** problems of the form "a is to b as c is to ?", written $a : b :: c :?$, by computing

$$d^* = \operatorname*{argmin}_d ||(\mathbf{v}_a - \mathbf{v}_b) - (\mathbf{v}_c - \mathbf{v}_d)|| \tag{19.28}$$

For example, we have man:woman::king:queen, and indeed we find that

$$\mathbf{v}_{\text{man}} - \mathbf{v}_{\text{woman}} \approx \mathbf{v}_{\text{king}} - \mathbf{v}_{\text{queen}} \tag{19.29}$$

We can verify this empirically using the script at word_embedding_spacy.py, which uses a pretrained word embedding model. See Fig. 19.11.

In [PSM14a], they conjecture that $a : b :: c : d$ holds iff for every word w in the vocabulary, we have

$$\frac{p(w|a)}{p(w|b)} \approx \frac{p(w|c)}{p(w|d)} \tag{19.30}$$

In [Aro+16], they show that this follows from the RAND-WALK modeling assumptions in Sec. 19.5.3. See also [AH19; EDH19] for other explanations of why word analogies work, based on different modeling assumptions.

19.5.5 Contextual word embeddings

Consider the sentences "I was eating an apple" and "I bought a new phone from Apple". The meaning of the word "apple" is different in both cases, but a fixed word embedding (of the type discussed above) would not be able to capture this. In this section, we consider **contextual word embeddings**, where the embedding of a word is a function of all the words in its context (usually a sentence). There are many methods for creating such contextual word embeddings. [Wen19] provides a good summary, which we draw on below.

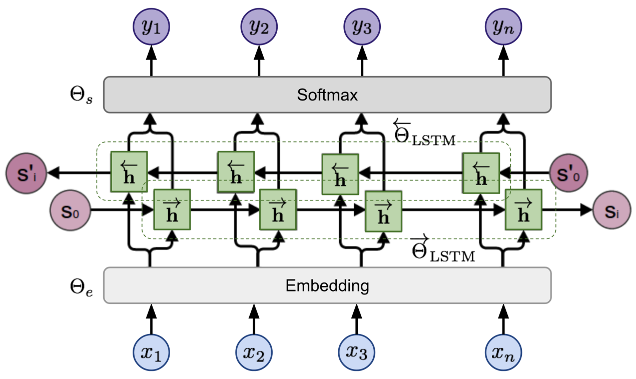


Figure 19.12: Illustration of ELMo bidirectional language model. Here $y_t = x_{t+1}$ when acting as the target for the forwards LSTM, and $y_t = x_{t-1}$ for the backwards LSTM. (We add bos *and* eos *sentinels to handle the edge cases.) From Weng2019LM. Used with kind permission of Lilian Weng.*

19.5.5.1 ELMo

In [Pet+18], they present a method called **ELMo**, which is short for "Embeddings from Language Model". The basic idea is to fit two RNN language models, one left-to-right, and one right-to-left, and then to combine their hidden state representations to come up with a contextual embedding for each word. Unlike a biRNN (Sec. 15.2.2), which needs an input-output pair, ELMo is trained in an unsupervised way, to maximize the log likelihood of the input sentence $\mathbf{x}_{1:T}$:

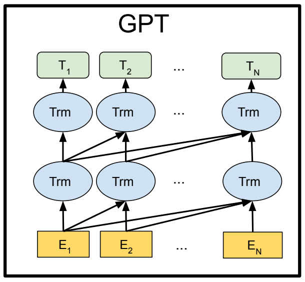
$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{t=1}^{T} [\log p(x_t|\mathbf{x}_{1:t-1}; \boldsymbol{\theta}_e, \boldsymbol{\theta}^{\rightarrow}, \boldsymbol{\theta}_s) + \log p(x_t|\mathbf{x}_{t+1:T}; \boldsymbol{\theta}_e, \boldsymbol{\theta}^{\leftarrow}, \boldsymbol{\theta}_s)] \tag{19.31}$$

where $\boldsymbol{\theta}_e$ are the shared parameters of the embedding layer, $\boldsymbol{\theta}_s$ are the shared parameters of the softmax output layer, and $\boldsymbol{\theta}^{\rightarrow}$ and $\boldsymbol{\theta}^{\leftarrow}$ are the parameters of the two RNN models. (They use LSTM RNNs, described in Sec. 15.2.6.2.) See Fig. 19.12 for an illustration.

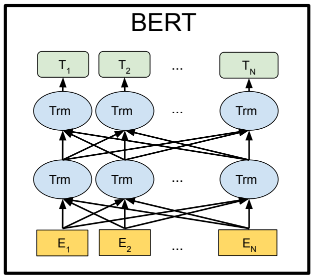
After training, we define the contextual representation $\mathbf{r}_t = [\mathbf{e}_t, \mathbf{h}^{\rightarrow}_{t,1:L}, \mathbf{h}^{\leftarrow}_{t,1:L}]$, where L is the number of layers in the LSTM. We then learn a task-specific set of linear weights to map this to the final context-specific embedding of each token: $\mathbf{r}_t^j = \mathbf{r}_t^{\top}\mathbf{w}^j$, where j is the task id. If we are performing a syntactic task like **part-of-speech** (**POS**) tagging (i.e., labeling each word as a noun, verb, adjective, etc), then the task will learn to put more weight on lower layers. If we are performing a semantic task like **word sense disambiguation** (**WSD**), then the task will learn to put more weight on higher layers. In both cases, we only need a small amount of task-specific labeled data, since we are just learning a single weight vector, to map from $\mathbf{r}_{1:T}$ to the target labels $\mathbf{y}_{1:T}$.

19.5.5.2 GPT

In [Rad+18], they propose a model called **GPT**, which is short for "Generative Pre-training Transformer". It is very similar to ELMo, except it replaces the two LSTMs with a single (unidirectional) transformer decoder. See Fig. 19.13a for an illustration. In addition, the training objective is slightly



(a)



(b)

Figure 19.13: Illustration of (a) GPT and (b) BERT. E_t is the embedding vector for the input token at location t, and T_t is the output target to be predicted. From Figure 3 of [Dev+19]. Used with kind permission of Ming-Wei Chang.

different. Rather than first training a language model, and then training a linear decoder for each task, they jointly optimize on a large unlabeled dataset, and a small labeled dataset. In the classification setting, the loss is given by $\mathcal{L} = \mathcal{L}_{\text{cls}} + \lambda\mathcal{L}_{\text{LM}}$, where $\mathcal{L}_{\text{cls}} = \sum_{(\mathbf{x},y)\in\mathcal{D}} \log p(y|\mathbf{x})$ is the classification loss and $\mathcal{L}_{\text{LM}} = -\sum_t p(x_t|\mathbf{x}_{1:t-1})$ is the language modeling loss.

In [Rad+19], they propose **GPT-2**, which is a larger version of GPT, trained on a large web corpus called **WebText**. They also eliminate any task-specific training, and instead just train it as a language model. At test time, they tell the system what task they want it to perform by feeding in a specially structured sentence, called the **prompt**, and then generating from the model conditioned on this input. This is called **zero-shot task transfer**.

For example, to perform **abstractive summarization** of some input text $\mathbf{x}_{1:T}$ (as opposed to **extractive summarization**, which just selects a subset of the input words), we sample from $p(\mathbf{x}_{T+1:T+100}|[\mathbf{x}_{1:T}; \text{TL;DR}])$, where **TL;DR** is a special token added to the end of the input text, which tells the system the user wants a summary. TL;DR stands for "too long; didn't read" and frequently occurs in webtext followed by a human-created summary. By adding this token to the input, the user hopes to "trigger" the transformer decoder into a state in which it enters summarization mode. This works because the model has been exposed to (document, summary) pairs, where the summary was prefixed by the TL;DR token. (It is also possible to explicitly train a model to perform certain tasks, by telling it what task to perform as part of the input, and giving it the desired output, as discussed in Sec. 19.5.5.4.)

More recently, OpenAI released **GPT-3** [Bro+20], which is an even larger version of GPT-2, but based on the same principles.

19.5.5.3 BERT

In this section, we describe the **BERT** model (Bidirectional Encoder Representations from Transformers) of [Dev+19]. This can be thought of as a symmetric version of BERT that is trained to maximize the following pseudo-likelihood:

$$p(\mathbf{x}|\theta) \propto \prod_{t=1}^{T} p(x_t|\mathbf{x}_{-t}, \boldsymbol{\theta}) \tag{19.32}$$

We compute $p(x_t|\mathbf{x}_{-t})$ using a transformer model, in which we mask out the t'th token from the input, and then try to predict it given all the others. This is called the **fill-in-the-blank** or **cloze** task. In practice, we mask out multiple input tokens (up to 15%), and predict them all in parallel, for speed. Thus a typical input might be

```
Let's make [MASK] chicken! [SEP] It [MASK] great with orange sauce.
```

where [SEP] is a separator token inserted between two sentences. The desired target labels are "some" and "tastes". The loss is only computed at the masked locations; this is therefore called a **masked language model**. (This is similar to a denoising autoencoder, Sec. 20.3.2). More precisely, the objective is as follows:

$$\log p_{\boldsymbol{\theta}}(\overline{\mathbf{x}}|\hat{\mathbf{x}}) \approx \sum_{t=1}^{T} m_t \log p_{\boldsymbol{\theta}}(x_t|\hat{\mathbf{x}}) = \sum_{t=1}^{T} m_t \log \frac{\exp(\mathbf{h}_{\boldsymbol{\theta}}(\hat{\mathbf{x}})_t^{\top} \mathbf{e}(x_t))}{\sum_{x'} \exp(\mathbf{h}_{\boldsymbol{\theta}}(\hat{\mathbf{x}})_t^{\top} \mathbf{e}(x'))} \tag{19.33}$$

where $\hat{\mathbf{x}}$ is the masked input sentence, $\overline{\mathbf{x}}$ are the masked tokens, $m_t = 1$ iff location t is masked, $\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{x})$ is the hidden representation of the transformer, and $\mathbf{e}(x)$ is the embedding for token x.

After training BERT in an unsupervised way, we can use $\mathbf{r}_t = \mathbf{h}(\mathbf{x})_t$ as the contextual embedding for word x_t. For dense prediction problems, we can learn a linear decoder to compute $p(y_t|\mathbf{r}_t)$. To tackle whole sentence classification tasks, we need to aggregate these word embeddings, $\{\mathbf{r}_t\}$. A simple approach would be to average the embeddings by computing $\overline{\mathbf{r}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{r}_t$, and then train a model of the form $p(y|\overline{\mathbf{r}})$. However, the more common approach is to prepend a special [CLS] token to the beginning of each sentence, and to use its embedding as a representation of the entire input.

Fig. 19.14 illustrates how BERT can be used to tackle a variety of NLP tasks, such as single sentence classifiction (e.g., for **sentiment analysis**), sentence pair classification (e.g., for **natural language entailment**), single sentence tagging (e.g., for **named entity recognition**), or sentence pair tagging (e.g., for **question answering**. Interestingly, [TDP19] shows that BERT rediscovers the standard NLP pipeline, in which different layers perform tasks such as part of speech (POS) tagging, parsing, named entity relationship (NER) detection, semantic role labeling (SRL), coreference resolution, etc. However, the extent to which BERT and other giant language models "understand" text in a robust way (beyond incidental co-occurrence statistics) has recently been called into question (see e.g., [NK19; BK20]).

19.5.5.4 T5

Many models are trained in an unsupervised way, and then fine-tuned on specific tasks. It is also possible to train a single model to perform multiple tasks, by telling the system what task to perform as part of the input sentence, and then training it as a seq2seq model, as illustrated in Fig. 19.16. This is the approach used in **T5** [Raf+19], which stands for "Text-to-text Transfer Transformer". The model is a standard seq2seq transformer, that is trained on supervised $(\mathbf{x}, \mathbf{y})$ pairs, as well as unsupervised $(\mathbf{x}', \mathbf{x}'')$ pairs, where $\mathbf{x}'$ is a masked version of $\mathbf{x}$, and $\mathbf{x}''$ are the missing tokens that need to be predicted.

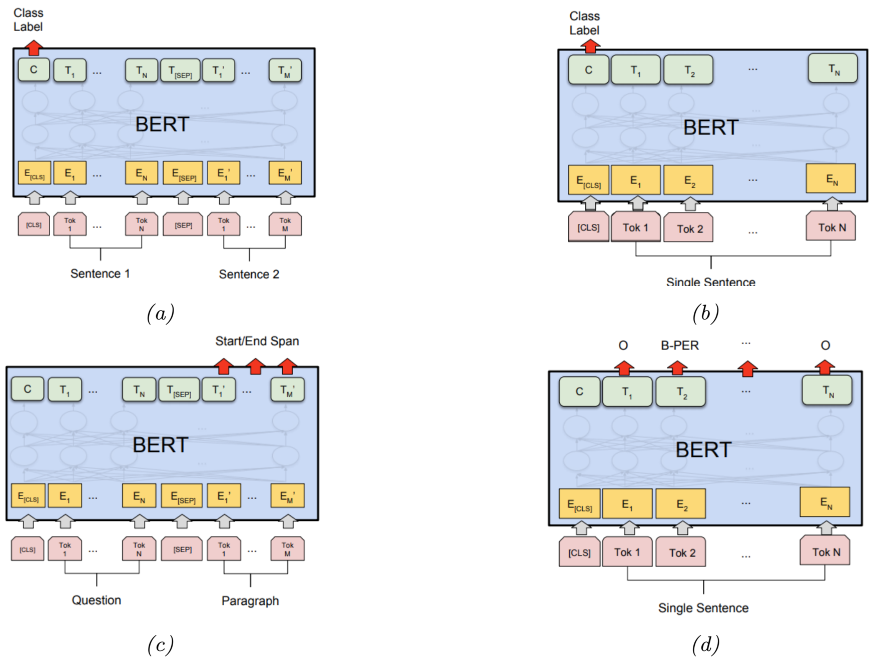


Figure 19.14: Illustration of how BERT can be used for different kinds of supervised NLP tasks. (a) Sentence-pair classification (e.g., entailment). (b) Single sentence classification (e.g., sentiment). (c) Sentence pair tagging (e.g., question answering). (d) Single sentence tagging (e.g., named entity recognition, where the tags are "outside", "begin-person", "inside-person", "begin-place", "inside-place", etc). From Figure 4 of [Dev+19]. Used with kind permission of Ming-Wei Chang.

T: In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail… Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

Q1: What causes precipitation to fall? A1: **gravity**
Q2: What is another main form of precipitation besides drizzle, rain, snow, sleet and hail? A2: **graupel**
Q3: Where do water droplets collide with ice crystals to form precipitation? A3: **within a cloud**

Figure 19.15: Question-answer pairs for a sample passage in the SQuAD dataset. Each of the answers is a segment of text from the passage. This can be solved using sentence pair tagging. The input is the paragraph text T and the question Q. The output is a tagging of the relevant words in T that answer the question in Q. From Figure 1 of [Raj+16]. Used with kind permission of Percy Liang.

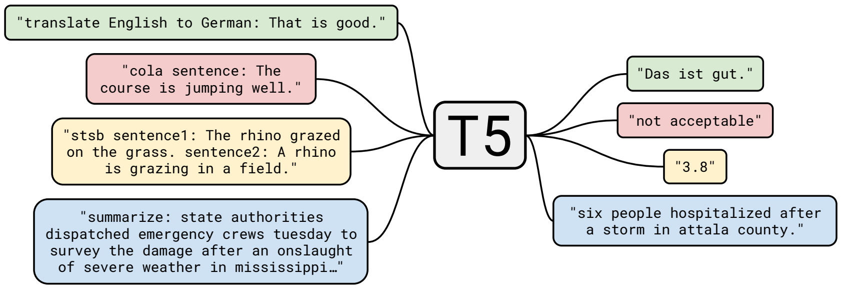

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

*Figure 19.16: Illustration of how the T5 model ("Text-to-text Transfer Transformer") can be used to perform multiple NLP tasks, such as translating English to German; determining if a sentence is linguistic valid or not (**CoLA** stands for "Corpus of Linguistic Acceptability"); determining the degree of semantic similarity (**STSB** stands for "Semantic Textual Similarity Benchmark"); and abstractive summarization. From Figure 1 of [Raf+19]. Used with kind permission of Colin Raffel.*

19.6 Semi-supervised learning

This section is co-authored with Colin Raffel.

Many recent successful applications of machine learning are in the supervised learning setting, where a large dataset of labeled examples are available for training a model. However, in many practical applications it is expensive to obtain this labeled data. Consider the case of automatic speech recognition: Modern datasets contain thousands of hours of audio recordings [Pan+15; Ard+20]. The process of annotating the words spoken in a recording is many times slower than realtime, potentially resulting in a long (and costly) annotation process. To make matters worse, in some applications data must be labeled by an expert (such as a doctor in medical applications) which can further increase costs.

Semi-supervised learning can alleviate the need for labeled data by taking advantage of unlabeled data. The general goal of semi-supervised learning is to allow the model to learn the high-level structure of the data distribution from unlabeled data and only rely on the labeled data for learning the fine-grained details of a given task. Whereas in standard supervised learning we assume that we have access to samples from the joint distribution of data and labels $\mathbf{x}, y \sim p(\mathbf{x}, y)$, semi-supervised learning assumes that we additionally have access to samples from the marginal distribution of $\mathbf{x}$, namely $\mathbf{x} \sim p(\mathbf{x})$. Further, it is generally assumed that we have many more of these unlabeled samples since they are typically cheaper to obtain. Continuing the example of automatic speech recognition, it is often much cheaper to simply record people talking (which would produce unlabeled data) than it is to transcribe recorded speech. Semi-supervised learning is a good fit for the scenario where a large amount of unlabeled data has been collected and the practitioner would like to avoid having to label all of it.

19.6.1 Self-training and pseudo-labeling

An early and straightforward approach to semi-supervised learning is **self-training** [Scu65; Agr70; McL75]. The basic idea behind self-training is to use the model itself to infer predictions on unlabeled

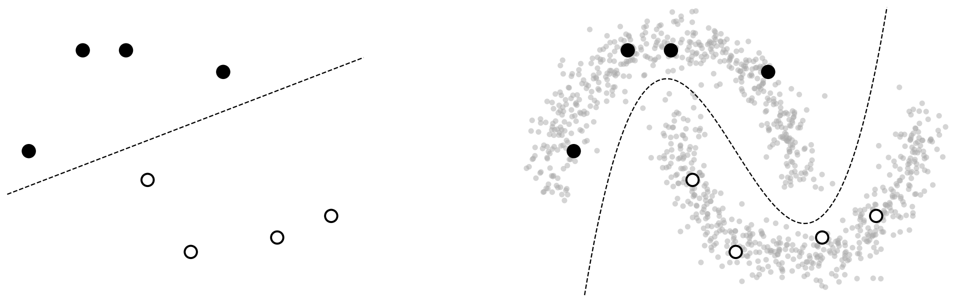


Figure 19.17: Illustration of the benefits of semi-supervised learning for a binary classification problem. Labeled points from each class are shown as black and white circles respectively. (a) Decision boundary we might learn given only unlabeled data. (b) Decision boundary we might learn if we also had a lot of unlabeled data points, shown as smaller grey circles.

data, and then treat these predictions as labels for subsequent training. Self-training has endured as a semi-supervised learning method because of its simplicity and general applicability; i.e. it is applicable to any model that can generate predictions for the unlabeled data. Recently, it has become common to refer to this approach as "**pseudo-labeling**" [Lee13] because the inferred labels for unlabeled data are only "pseudo-correct" in comparison with the true, ground-truth targets used in supervised learning.

Algorithmically, self-training typically follows one of the following two procedures. In the first approach, pseudo-labels are first predicted for the entire collection of unlabeled data and the model is re-trained (possibly from scratch) to convergence on the combination of the labeled and (pseudo-labeled) unlabeled data. Then, the unlabeled data is re-labeled by the model and the process repeats itself until a suitable solution is found. The second approach instead continually generates predictions on randomly-chosen batches of unlabeled data and immediately trains the model against these pseudo-labels. Both approaches are currently common in practice; the first "offline" variant has been shown to be particularly successful when leveraging giant collections of unlabeled data [Yal+19; Xie+20] whereas the "online" approach is often used as one component of more sophisticated semi-supervised learning methods [Soh+20]. Neither variant is fundamentally better than the other. Offline self-training can result in training the model on "stale" pseudo-labels, since they are only updated each time the model converges. However, online pseudo-labeling can incur larger computational costs since it involves constantly "re-labeling" unlabeled data.

Self-training can suffer from an obvious problem: If the model generates incorrect predictions for unlabeled data and then is re-trained on these incorrect predictions, it can become progressively worse and worse at the intended classification task until it eventually learns a totally invalid solution. This issue has been dubbed **confirmation bias** [TV17] because the model is continually confirming its own (incorrect) bias about the decision rule.

A common way to mitigate confirmation bias is to use a "selection metric" [RHS05] which heuristically tries to only retain pseudo-labels that are correct. For example, assuming that a model outputs probabilities for each possible class, a frequently-used selection metric is to only retain pseudo-labels whose largest class probability is above a threshold [Yar95; RHS05]. If the model's

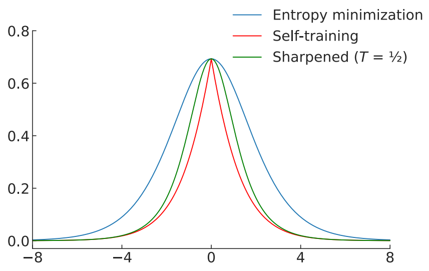


Figure 19.18: Comparison of the entropy minimization, self-training, and "sharpened" entropy minimization loss functions for a binary classification problem.

class probability estimates are well-calibrated, then this selection metric will only retain labels that are highly likely to be correct (according to the model, at least). More sophisticated selection metrics can be designed according to the problem domain.

19.6.2 Entropy minimization

Self-training has the implicit effect of encouraging the model to output low-entropy (i.e. high-confidence) predictions. This effect is most apparent in the online setting with a cross-entropy loss, where the model minimizes the following loss function $\mathcal{L}$ on unlabeled data:

$$\mathcal{L} = -\max_c \log p_\theta(y = c|\mathbf{x}) \tag{19.34}$$

where $p_\theta(y|\mathbf{x})$ is the model's class probability distribution given input $\mathbf{x}$. This function is minimized when the model assigns all of its class probability to a single class c^*, i.e. $p(y = c^*|\mathbf{x}) = 1$ and $p(y \neq c^*)|\mathbf{x}) = 0$.

A closely-related semi-supervised learning method is **entropy minimization** [GB05], which minimizes the following loss function:

$$\mathcal{L} = -\sum_{c=1}^{C} p_\theta(y = c|\mathbf{x}) \log p(y = c|\mathbf{x}) \tag{19.35}$$

Note that this function is also minimized when the model assigns all of its class probability to a single class. We can make the entropy-minimization loss in Eq. (19.35) equivalent to the online self-training loss in Eq. (19.34) by replacing the first $p_\theta(y = c|\mathbf{x})$ term with a "one-hot" vector that assigns a probability of 1 for the class that was assigned the highest probability. In other words, online self-training minimizes the cross-entropy between the model's output and the "hard" target $\arg\max p_\theta(y|\mathbf{x})$, whereas entropy minimization uses the the "soft" target $p_\theta(y|\mathbf{x})$. One way to trade off between these two extremes is to adjust the "temperature" of the target distribution by raising each probability to the power of $1/T$ and renormalizing; this is the basis of the **mixmatch** method of [Ber+19a; Ber+19b; Xie+19b]. At $T = 1$, this is equivalent to entropy minimization; as $T \to 0$, it becomes hard online self-training. A comparison of these loss functions is shown in Fig. 19.18.

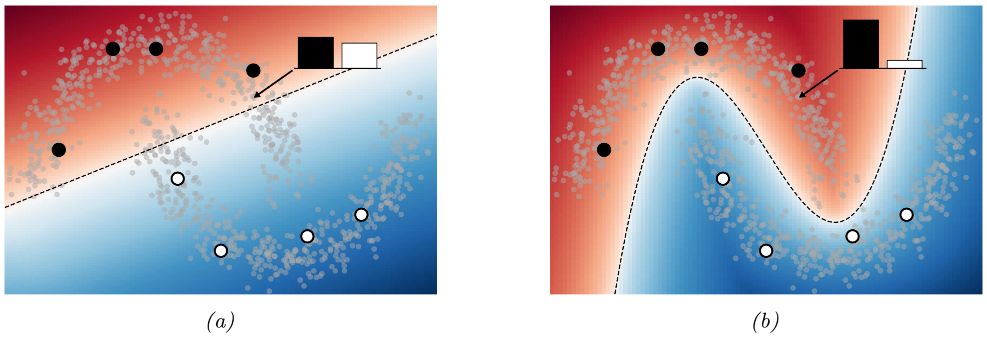


Figure 19.19: Visualization demonstrating how entropy minimization enforces the cluster assumption. The classifier assigns a higher probability to class 1 (black dots) or 2 (white dots) in red or blue regions respectively. The predicted class probabilities for one particular unlabeled datapoint is shown in the bar plot. In (a), the decision boundary passes through high-density regions of data, so the classifier is forced to output high-entropy predictions. In (b), the classifier avoids high-density regions and is able to assign low-entropy predictions to most of the unlabeled data.

19.6.2.1 The cluster assumption

Why is entropy minimization a good idea? A basic assumption of many semi-supervised learning methods is that the decision boundary between classes should fall in a low-density region of the data manifold. This effectively assumes that the data corresponding to different classes are clustered together. A good decision boundary, therefore, should not pass through clusters; it should simply separate them. Semi-supervised learning methods that make the "**cluster assumption**" can be thought of as using unlabeled data to estimate the shape of the data manifold and moving the decision boundary away from it.

Entropy minimization is one such method. To see why, first assume that the decision boundary between two classes is "smooth", i.e. the model does not abruptly change its class prediction anywhere in its domain. This is true in practice for simple and/or regularized models. In this case, if the decision boundary passes through a high-density region of data, it will by necessity produce high-entropy predictions for some samples from the data distribution. Entropy minimization will therefore encourage the model to place its decision boundary in low-density regions of the input space to avoid transitioning from one class to another in a region of space where data may be sampled. A visualization of this behavior is shown in Fig. 19.19.

19.6.2.2 Input-output mutual information

An alternative justification for the entropy minimization objective was proposed by Bridle, Heading, and MacKay [BHM92], where it was shown that it naturally arises from maximizing the mutual information (Sec. 6.3) between the data and the label (i.e. the input and output of a model). Denoting

$\mathbf{x}$ as the input and y as the target, the input-output mutual information can be written as

$$\mathcal{I}(y;\mathbf{x}) = \iint p(y,\mathbf{x}) \log \frac{p(y,\mathbf{x})}{p(y)p(\mathbf{x})} dy d\mathbf{x} \tag{19.36}$$

$$= \iint p(y|\mathbf{x})p(\mathbf{x}) \log \frac{p(y,\mathbf{x})}{p(y)p(\mathbf{x})} dy d\mathbf{x} \tag{19.37}$$

$$= \int p(\mathbf{x}) d\mathbf{x} \int p(y|\mathbf{x}) \log \frac{p(y|\mathbf{x})}{p(y)} dy \tag{19.38}$$

$$= \int p(\mathbf{x}) d\mathbf{x} \int p(y|\mathbf{x}) \log \frac{p(y|\mathbf{x})}{\int p(\mathbf{x})p(y|\mathbf{x}) d\mathbf{x}} dy \tag{19.39}$$

Note that the first integral is equivalent to taking an expectation over $\mathbf{x}$, and the second integral is equivalent to summing over all possible values of the class y. Using these relations, we obtain

$$\mathcal{I}(y;\mathbf{x}) = \mathbb{E}_{\mathbf{x}} \left[\sum_{i=1}^{L} p(y_i|\mathbf{x}) \log \frac{p(y_i|\mathbf{x})}{\mathbb{E}_{\mathbf{x}}[p(y_i|\mathbf{x})]} \right] \tag{19.40}$$

$$= \mathbb{E}_{\mathbf{x}} \left[\sum_{i=1}^{L} p(y_i|\mathbf{x}) \log p(y_i|\mathbf{x}) \right] - \mathbb{E}_{\mathbf{x}} \left[\sum_{i=1}^{L} p(y_i|\mathbf{x}) \log \mathbb{E}_{\mathbf{x}}[p(y_i|\mathbf{x})] \right] \tag{19.41}$$

$$= \mathbb{E}_{\mathbf{x}} \left[\sum_{i=1}^{L} p(y_i|\mathbf{x}) \log p(y_i|\mathbf{x}) \right] - \sum_{i=1}^{L} \mathbb{E}_{\mathbf{x}}[p(y_i|\mathbf{x})] \log \mathbb{E}_{\mathbf{x}}[p(y_i|\mathbf{x})]] \tag{19.42}$$

Since we had initially sought to *maximize* the mutual information, and we typically *minimize* loss functions, we can convert this to a suitable loss function by negating it:

$$\mathcal{I}(y;\mathbf{x}) = -\mathbb{E}_{\mathbf{x}} \left[\sum_{i=1}^{L} p(y_i|\mathbf{x}) \log p(y_i|\mathbf{x}) \right] + \sum_{i=1}^{L} \mathbb{E}_{\mathbf{x}}[p(y_i|\mathbf{x})] \log \mathbb{E}_{\mathbf{x}}[p(y_i|\mathbf{x})]] \tag{19.43}$$

The first term is exactly the entropy minimization objective in expectation. The second term specifies that we should maximize the entropy of the expected class prediction, i.e. the average class prediction over our training set. This encourages the model to predict each possible class with equal probability, which is only appropriate when we know a priori that all classes are equally likely.

19.6.3 Co-training

Co-training [BM98] is also similar to self-training, but makes an additional assumption that there are two complementary "views" (i.e. independent sets of features) of the data, both of which can be used separately to train a reasonable model. After training two models separately on each view, unlabeled data is classified by each model to obtain candidate pseudo-labels. If a particular pseudo-label receives a low-entropy prediction (indicating high confidence) from one model and a high-entropy prediction (indicating low confidence) from the other, then that pseudo-labeled datapoint is added to the training set for the low-confidence model. Then, the process is repeated with the new, larger training datasets. The procedure of only retaining pseudo-labels when one of the models is confident ideally builds up the training sets with correctly-labeled data.

Co-training makes the strong assumption that there are two informative-but-independent views of the data, which may not be true for many problems. The **Tri-Training** algorithm [ZL05] circumvents this issue by instead using *three* models that are first trained on independently-sampled (with replacement) subsets of the labeled data. Ideally, initially training on different collections of labeled data results in models that do not always agree on their predictions. Then, pseudo-labels are generated for the unlabeled data independently by each of the three models. For a given unlabeled datapoint, if two of the models agree on the pseudo-label, it is added to the training set for the third model. This can be seen as a selection metric, because it only retains pseudo-labels where two (differently initialized) models agree on the correct label. The models are then re-trained on the combination of the labeled data and the new pseudo-labels, and the whole process is repeated iteratively.

19.6.4 Label propagation on graphs

If two datapoints are "similar" in some meaningful way, we might expect that they share a label. This idea has been referred to as the **manifold assumption**. **Label propagation** is a semi-supervised learning technique that leverages the manifold assumption to assign labels to unlabeled data. Label propagation first constructs a graph where the nodes are the data examples and the edge weights represent the degree of similarity. The node labels are known for nodes corresponding to labeled data but are unknown for unlabeled data. Label propagation then propagates the known labels across edges of the graph in such a way that there is minimal disagreement in the labels of a given node's neighbors. This provides label guesses for the unlabeled data, which can then be used in the usual way for supervised training of a model.

More specifically, the basic label propagation algorithm [ZG02] proceeds as follows: First, let $w_{i,j}$ denote a non-negative edge weight between $\mathbf{x}_i$ and $\mathbf{x}_j$ that provides a measure of similarity for the two (labeled or unlabeled) datapoints. Assuming that we have M labeled datapoints and N unlabeled datapoints, define the $(M+N) \times (M+N)$ transition matrix $\mathbf{T}$ as having entries

$$\mathbf{T}_{i,j} = \frac{w_{i,j}}{\sum_k w_{k,j}} \tag{19.44}$$

$\mathbf{T}_{i,j}$ represents the probability of propagating the label for node j to node i. Further, define the $(M+N) \times C$ label matrix $\mathbf{Y}$, where C is the number of possible classes. The ith row of $\mathbf{Y}$ represents the class probability distribution of datapoint i. Then, repeat the following steps until the values in $\mathbf{Y}$ do not change significantly: First, use the transition matrix $\mathbf{T}$ to propagate labels in $\mathbf{Y}$ by setting $\mathbf{Y} \leftarrow \mathbf{T}\mathbf{Y}$. Then, re-normalize the rows of Y by setting $\mathbf{Y}_{i,c} \leftarrow \mathbf{Y}_{i,c} / \sum_k \mathbf{Y}_{k,c}$. Finally, replace the rows of $\mathbf{Y}$ corresponding to labeled datapoints with their one-hot representation (i.e. $\mathbf{Y}_{i,c} = 1$ if datapoint i has ground-truth label c and 0 otherwise). After convergence, guessed labels are chosen based on the highest class probability for each datapoint in $\mathbf{Y}$.

This algorithm iteratively uses the similarity of datapoints (encoded in the weights used to construct the transition matrix) to propagate information from the (fixed) labels onto the unlabeled data. At each iteration, the label distribution for a given datapoint is computed as the weighted average of the label distributions for all of its connected datapoints, where the weighting corresponds to the edge weights in $\mathbf{T}$. It can be shown that this procedure converges to a single fixed point, whose computational cost mainly involves the inversion of the matrix of unlabled-to-unlabled transition probabilities [ZG02].

The overall approach can be seen as a form of **transductive learning**, since it is learning to predict labels for a fixed unlabeled dataset, rather than learning a model that generalizes. However, given the induced labeling. we can perform **inductive learning** in the usual way.

The success of label propagation depends heavily on the notion of similarity used to construct the weights between different nodes (datapoints). For simple data, measuring the Euclidean distance between datapoints can be sufficient. However, for complex and high-dimensional data the Euclidean distance might not meaningfully reflect the likelihood that two datapoints share the same class. The similarity weights can also be set arbitrarily according to problem-specific knowledge. For a few examples of different ways of constructing the similarity graph, see Zhu [Zhu05, chapter 3]. For some recent papers that use this approach in conjunction with deep learning, see e.g., [BRR18; Isc+19].

19.6.5 Consistency regularization

Consistency regularization leverages the simple idea that perturbing a given datapoint (or the model itself) should not cause the model's output to change dramatically. Since measuring consistency in this way only makes use of the model's outputs (and not ground-truth labels), it is readily applicable to unlabeled data and therefore can be used to create appropriate loss functions for semi-supervised learning. This idea was first proposed under the framework of "learning with pseudo-ensembles" [BAP14], with similar variants following soon thereafter [LA16; SJT16].

In its most general form, both the model $p_\theta(y|\mathbf{x})$ and the transformations applied to the input can be stochastic. For example, in computer vision problems we may transform the input by using data augmentation like randomly rotating or adding noise the input image, and the network may include stochastic components like dropout (Sec. 13.5.4) or weight noise [Gra11]. A common and simple form of consistency regularization first samples $\mathbf{x}' \sim q(\mathbf{x}'|\mathbf{x})$ (where $q(\mathbf{x}'|\mathbf{x})$ is the distribution induced by the stochastic input transformations) and then minimizes the loss $\|p_\theta(y|\mathbf{x}) - p_\theta(y|\mathbf{x}')\|^2$. In practice, the first term $p_\theta(y|\mathbf{x})$ is typically treated as fixed (i.e. gradients are not propagated through it). In the semi-supervised setting, the combined loss function over a batch of labeled data $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_M, y_M)$ and unlabeled data $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ is

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{M} \log p_\theta(y = y_i|\mathbf{x}_i) + \lambda \sum_{j=1}^{N} \|p_\theta(y|\mathbf{x}_j) - p_\theta(y|\mathbf{x}'_j)\|^2 \tag{19.45}$$

where λ is a scalar hyperparameter that balances the importance of the loss on unlabeled data and, for simplicity, we write $\mathbf{x}'_j$ to denote a sample drawn from $q(\mathbf{x}'|\mathbf{x}_j)$.

The basic form of consistency regularization in Eq. (19.45) reveals many design choices that impact the success of this semi-supervised learning approach. First, the value chosen for the λ hyperparameter is important. If it is too large, then the model may not give enough weight to learning the supervised task and will instead start to reinforce its own bad predictions (as with confirmation bias in self-training). Since the model is often poor at the start of training before it has been trained on much labeled data, it is common in practice to initialize set λ to zero and increase its value over the course of training.

A second important consideration are the random transformations applied to the input, i.e., $q(\mathbf{x}'|\mathbf{x})$. Generally speaking, these transformations should be designed so that they do not change the label of $\mathbf{x}$. As mentioned above, a common choice is to use domain-specific data augmentations. It has recently been shown that using strong data augmentations that heavily corrupt the input (but, arguably, still do not change the label) can produce particularly strong results [Xie+19b; Ber+19b; Soh+20].

The use of data augmentation requires expert knowledge to determine what kinds of transformations are label-preserving and appropriate for a given problem. An alternative technique, called **virtual adversarial training** (VAT), instead transforms the input using an analytically-found perturbation designed to maximally change the model's output (similar to adversarial examples, see Sec. 14.6). Specifically, VAT computes a perturbation $\boldsymbol{\delta}$ that approximates $\boldsymbol{\delta} = \text{argmax}_{\boldsymbol{\delta}} \mathbb{KL}\left(p_\theta(y|\mathbf{x}) \| p_\theta(y|\mathbf{x}+\boldsymbol{\delta})\right)$. The approximation is done by sampling $\mathbf{d}$ from a multivariate Gaussian distribution, initializing $\boldsymbol{\delta} = \mathbf{d}$, and then setting

$$\boldsymbol{\delta} \leftarrow \nabla_{\boldsymbol{\delta}} \mathbb{KL}\left(p_\theta(y|\mathbf{x}) \| p_\theta(y|\mathbf{x}+\boldsymbol{\delta})\right)|_{\boldsymbol{\delta}=\xi\mathbf{d}} \tag{19.46}$$

where ξ is a small constant, typically 10^{-6}. VAT then sets

$$\mathbf{x}' = \mathbf{x} + \epsilon \frac{\boldsymbol{\delta}}{\|\boldsymbol{\delta}\|_2} \tag{19.47}$$

and proceeds as usual with consistency regularization (as in Eq. (19.45)), where ϵ is a scalar hyperparameter that sets the L2-norm of the perturbation applied to $\mathbf{x}$.

Consistency regularization can also profoundly affect the geometry properties of the training objective, and the trajectory of SGD, such that performance can particularly benefit from non-standard training procedures. For example, the Euclidean distances between weights at different training epochs is significantly larger for objectives that use consistency regularization. Athiwaratkun, Izmailov, and Wilson [AIW19] show that a variant of **stochastic weight averaging** (SWA) [Izm+18] can achieve state-of-the-art performance on semi-supervised learning tasks by exploiting the geometric properties of consistency regularization.

A final consideration when using consistency regularization is the function used to measure the difference between the network's output with and without perturbations. Equation (19.45) uses the squared L2 distance (also referred to as the Brier score), which is a common choice [SJT16; TV17; LA16; Ber+19a]. It is also common to use the KL divergence $\mathbb{KL}\left(p_\theta(y|\mathbf{x}) \| p_\theta(y|\mathbf{x}')\right)$ in analogy with the cross-entropy loss (i.e. KL divergence between ground-truth label and prediction) used for labeled examples [Miy+18; Ber+19b; Xie+19b]. The gradient of the squared-error loss approaches zero as the model's predictions on the perturbed and unperturbed input differ more and more, assuming the model uses a softmax nonlinearity on its output. Using the squared-error loss therefore has a possible advantage that the model is not updated when its predictions are very unstable. However, the KL divergence has the same scale as the cross-entropy loss used for labeled data, which makes for more intuitive tuning of the unlabeled loss hyperparameter λ. A comparison of the two loss functions is shown in Fig. 19.20.

19.6.6 Deep generative models *

Generative models provide a natural way of making use of unlabeled data through learning a model of the marginal distribution by minimizing $\mathcal{L}_U = -\sum_n \log p_{\boldsymbol{\theta}}(\mathbf{x}_n)$. Various approaches have leveraged generative models for semi-supervised by developing ways to use the model of $p_{\boldsymbol{\theta}}(\mathbf{x}_n)$ to help produce a better supervised model.

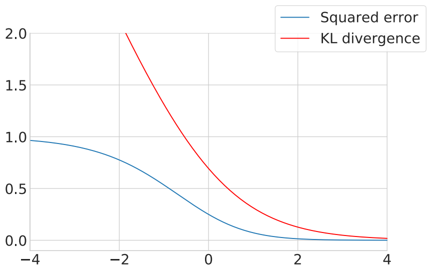


Figure 19.20: Comparison of the squared error and KL divergence lossses for a consistency regularization. This visualization is for a binary classification problem where it is assumed that the model's output for the unperturbed input is 1. The figure plots the loss incurred for a particular value of the logit (i.e. the pre-activation fed into the output sigmoid nonlinearity) for the perturbed input. As the logit grows towards infinity, the model predicts a class label of 1 (in agreement with the prediction for the unperturbed input); as it grows towards negative infinity, the model predictions class 0. The squared error loss saturates (and has zero gradients) when the model predicts one class or the other with high probability, but the KL divergence grows without bound as the model predicts class 0 with more and more confidence.

19.6.6.1 Variational autoencoders

In Sec. 20.3.5, we describe the variational autoencoder (VAE), which defines a probabilistic model of the joint distribution of data $\mathbf{x}$ and latent variables $\mathbf{z}$. Data is assumed to be generated by first sampling $\mathbf{z} \sim p(\mathbf{z})$ and then sampling $\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$. For learning, the VAE uses an encoder $\mathbf{q}_\lambda(\mathbf{z}|\mathbf{x})$ to approximate the posterior and a decoder $p_\theta(\mathbf{x}|\mathbf{z})$ to approximate the likelihood. The encoder and decoder are typically deep neural networks. The parameters of the encoder and decoder can be jointly trained by maximizing the evidence lower bound (ELBO) of data.

The marginal distribution of latent variables $p(\mathbf{z})$ is often chosen to be a simple distribution like a diagonal-covariance Gaussian. In practice, this can make the latent variables $\mathbf{z}$ more amenable to downstream classification thanks to the facts that $\mathbf{z}$ is typically lower-dimensional than $\mathbf{x}$, that $\mathbf{z}$ is constructed via cascaded nonlinear transformations, and that the dimensions of the latent variables are designed to be independent. In other words, the latent variables can provide a (learned) representation where data may be more easily separable. In [Kin+14], this approach is called **M1** and it is indeed shown that the latent variables can be used to train stronger models when labels are scarce. (The general idea of unsupervised learning of representations to help with downstream classification tasks is described further in Sec. 19.2.3.)

An alternative approach to leveraging VAEs, also proposed in [Kin+14] and called **M2**, has the form

$$p_{\boldsymbol{\theta}}(\mathbf{x}, y) = p_{\boldsymbol{\theta}}(y) p_{\boldsymbol{\theta}}(\mathbf{x}|y) = p_{\boldsymbol{\theta}}(y) \int p_{\boldsymbol{\theta}}(\mathbf{x}|y, \mathbf{z}) p_{\boldsymbol{\theta}}(\mathbf{z}) d\mathbf{z} \tag{19.48}$$

where $\mathbf{z}$ is a latent variable, $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I})$ is the latent prior, $p_{\boldsymbol{\theta}}(y) = \text{Cat}(y|\boldsymbol{\pi})$ the label prior, and $p_{\boldsymbol{\theta}}(\mathbf{x}|y, \mathbf{z}) = p(\mathbf{x}|f_{\boldsymbol{\theta}}(y, \mathbf{z}))$ is the likelihood, such as a Gaussian, with parameters computed by f (a deep neural network). The main innovation of this approach is to assume that data is generated according to both a latent class variable y as well as the continuous latent variable $\mathbf{z}$. The class variable y is observed for labeled data and unobserved for unlabled data.

To compute the likelihood for labeled data, $p_{\boldsymbol{\theta}}(\mathbf{x}, y)$, we need to marginalize over $\mathbf{z}$, which can do approximately by using an inference network of the form $q_{\boldsymbol{\omega}}(\mathbf{z}|y, \mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\boldsymbol{\omega}}(y, \mathbf{x}), \text{diag}(\sigma^2_{\boldsymbol{\omega}}(\mathbf{x}))$. We then use the following variational lower bound

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}, y) \geq \mathbb{E}_{q_{\boldsymbol{\omega}}(\mathbf{z}|\mathbf{x},y)} \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|y, \mathbf{z}) + \log p_{\boldsymbol{\theta}}(y) + \log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\omega}}(\mathbf{z}|\mathbf{x}, y)\right] = -\mathcal{L}(\mathbf{x}, y) \tag{19.49}$$

as is standard for VAEs (see Sec. 20.3.5). The only difference is that we observe two kinds of data: $\mathbf{x}$ and y.

To compute the likelihood for unlabeled data, $p_{\boldsymbol{\theta}}(\mathbf{x})$, we need to marginalize over $\mathbf{z}$ and y, which can do approximately by using an inference network of the form

$$q_{\boldsymbol{\omega}}(\mathbf{z}, y|\mathbf{x}) = q_{\boldsymbol{\omega}}(\mathbf{z}|\mathbf{x})q_{\boldsymbol{\omega}}(y|\mathbf{x}) \tag{19.50}$$

$$q_{\boldsymbol{\omega}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\boldsymbol{\omega}}(\mathbf{x}), \text{diag}(\sigma^2_{\boldsymbol{\omega}}(\mathbf{x})) \tag{19.51}$$

$$q_{\boldsymbol{\omega}}(y|\mathbf{x}) = \text{Cat}(y|\boldsymbol{\pi}_{\boldsymbol{\omega}}(\mathbf{x})) \tag{19.52}$$

Note that $q_{\boldsymbol{\omega}}(y|\mathbf{x})$ acts like a discriminative classifier, that imputes the missing labels. We then use the following variational lower bound:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) \geq \mathbb{E}_{q_{\boldsymbol{\omega}}(\mathbf{z},y|\mathbf{x})} \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|y, \mathbf{z}) + \log p_{\boldsymbol{\theta}}(y) + \log p_{\boldsymbol{\theta}}(\mathbf{z}) - \log q_{\boldsymbol{\omega}}(\mathbf{z}, y|\mathbf{x})\right] \tag{19.53}$$

$$= -\sum_y q_{\boldsymbol{\omega}}(y|\mathbf{x})\mathcal{L}(\mathbf{x}, y) + \mathbb{H}\left(q_{\boldsymbol{\omega}}(y|\mathbf{x})\right) = -\mathcal{U}(\mathbf{x}) \tag{19.54}$$

Note that the discriminative classifier $q_{\boldsymbol{\omega}}(y|\mathbf{x})$ is only used to compute the loglikelihood of the unlabeled data, which is undesirable. We can therefore add an extra classification loss on the supervised data, to get the following overall objective function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}_L}\left[\mathcal{L}(\mathbf{x}, y)\right] + \mathbb{E}_{\mathbf{x}\sim\mathcal{D}_U}\left[\mathcal{U}(\mathbf{x})\right] + \alpha\mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}_L}\left[-\log q_{\boldsymbol{\omega}}(y|\mathbf{x})\right] \tag{19.55}$$

where α is a hyperparameter that controls the relative weight of generative and discriminative learning.

Of course, the probablistic model used in M2 is just one of many ways to decompose the dependencies between the observed data, the class labels, and the continuous latent variables. There are also many ways other than variational inference to perform approximate inference. The best technique will be problem dependent, but overall the main advantage of the generative approach is that we can incorporate domain knowledge. For example, we can model the missing data mechanism, since the absence of a label may be informative about the underlying data (e.g., people may be reluctant to answer a survey question about their health if they are unwell).

19.6.6.2 Generative adversarial networks

Generative adversarial networks (GANs) (described in more detail in the sequel to this book, [Mur22]) are a popular class of generative models that learn an implicit model of the data distribution. They consist of a generator network, which maps samples from a simple latent distribution to the data space, and a critic network, which attempts to distinguish between the outputs of the generator and samples from the true data distribution. The generator is trained to generate samples that the critic classifies as "real".

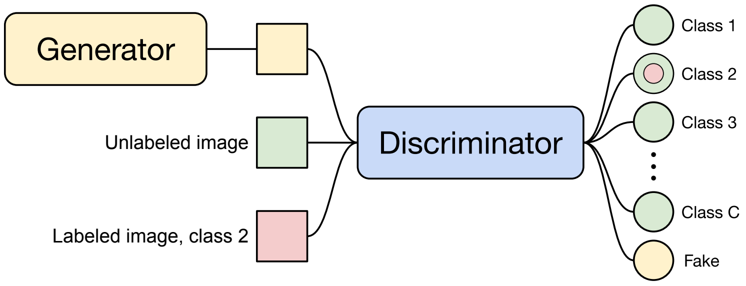


Figure 19.21: Diagram of the semi-supervised GAN framework. The discriminator is trained to output the class of labeled datapoints (red), a "fake" label for outputs from the generator (yellow), and any label for unlabeled data (green).

Since standard GANs do not produce a learned latent representation of a given datapoint and do not learn an explicit model of the data distribution, we cannot use the same approaches as were used for VAEs. Instead, semi-supervised learning with GANs is typically done by modifying the critic so that it outputs either a class label or "fake" instead of simply classifying real vs. fake [Sal+16; Ode16]. For labeled real data, the critic is trained to output the appropriate class label, and for unlabeled real data, it is trained to raise the probability of any of the class labels. As with standard GAN training, the critic is trained to classify outputs from the generator as fake and the generator is trained to fool the critic.

In more detail, let $p_\theta(y|\mathbf{x})$ denote the critic with $C+1$ outputs corresponding to C classes plus a "fake" class, and let $G(\mathbf{z})$ denote the generator which takes as input samples from the prior distribution $p(\mathbf{z})$. Let us assume that we are using the standard cross-entropy GAN loss as originally proposed in [Goo+14]. Then the critic's loss is

$$-\mathbb{E}_{\mathbf{x},y\sim p(\mathbf{x},y)} \log p_\theta(y|\mathbf{x}) - \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})} \log[1-p_\theta(y=C+1|\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})} \log p_\theta(y=C+1|G(\mathbf{z})) \quad (19.56)$$

This tries to maximize the probability of the correct class for the labeled examples, to minimize the probability of the fake class for real unlabeled examples, and to maximize the probability of the take class for generated examples. The generator's loss is simpler, namely

$$\mathbb{E}_{z\sim p(\mathbf{z})} \log p_\theta(y=C+1|G(\mathbf{z})) \quad (19.57)$$

A diagram visualizing the semi-supervised GAN framework is shown in Fig. 19.21.

19.6.6.3 Normalizing flows

Normalizing flows (described in more detail in the sequel to this book, [Mur22]) are a tractable way to define deep generative models. More precisely, they define an invertible mapping $f_\theta : \mathcal{X} \to \mathcal{Z}$, with parameters θ, from the data space $\mathcal{X}$ to the latent space $\mathcal{Z}$. The density in data space can be written starting from the density in the latent space using the change of variables formula:

$$p(x) = p(f(x)) \cdot \left|\det\left(\frac{\partial f}{\partial x}\right)\right|. \quad (19.58)$$

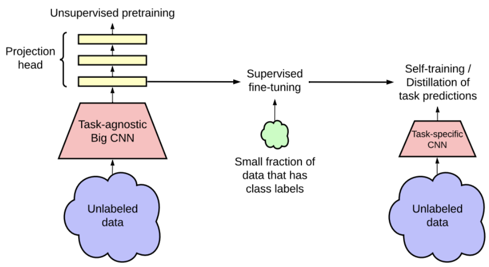


Figure 19.22: Combinng self-supervised learning on unlabeled data (left), supervised fine-tuning (middle), and self-training on pseudo-labeled data (right). From Figure 3 of [Che+20c]. Used with kind permission of Ting Chen

We can extend this to semi-supervised learning, as proposed in [Izm+20]. For class labels $y \in \{1 \ldots \mathcal{C}\}$, we can specify the latent distribution, conditioned on a label k, as Gaussian with mean μ_k and covariance Σ_k: $p(z|y=k) = \mathcal{N}(z|\mu_k, \Sigma_k)$. The marginal distribution of z is then a Gaussian mixture. The likelihood for labeled data is then

$$p_{\mathcal{X}}(x|y=k) = \mathcal{N}\left(f(x)|\mu_k, \Sigma_k\right) \cdot \left|\det\left(\frac{\partial f}{\partial x}\right)\right|, \tag{19.59}$$

and the likelihood for data with unknown label is $p(x) = \sum_k p(x|y=k)p(y=k)$.

For semi-supervised learning we can then maximize the joint likelihood of the labeled $\mathcal{D}_\ell$ and unlabeled data $\mathcal{D}_u$:

$$p(\mathcal{D}_\ell, \mathcal{D}_u|\theta) = \prod_{(x_i,y_i)\in\mathcal{D}_\ell} p(x_i, y_i) \prod_{x_j\in\mathcal{D}_u} p(x_j), \tag{19.60}$$

over the parameters θ of the bijective function f, which learns a density model for a Bayes classifier.

Given a test point x, the model predictive distribution is given by

$$p_{\mathcal{X}}(y|x) = \frac{p(x|y)p(y)}{p(x)} = \frac{\mathcal{N}\left(f(x)|\mu_y, \Sigma_y\right)}{\sum_{k=1}^{\mathcal{C}} \mathcal{N}(f(x)|\mu_k, \Sigma_k)}. \tag{19.61}$$

We can make predictions for a test point x with the Bayes decision rule $y = \arg\max_{c\in\{1,\ldots,\mathcal{C}\}} p(y=c|x)$.

19.6.7 Combining self-supervised and semi-supervised learning

It is possible to combine self-supervised and semi-supervised learning. For example, [Che+20c] using SimCLR (Sec. 19.2.3.3) to perform self-supervised representation learning on the unlabeled data, they then fine-tune this representation on a small labeled dataset (as in transfer learning, Sec. 19.2), and finally, they apply the trained model back to the original unlabeled dataset, and distill the predictions from this teacher model T into a student model S. (**Knowledge distillation** is the name given to the approach of training one model on the predictions of another, as originally proposed in [HVD14].)

That is, after fine-tuning T, they train S by minimizing

$$\mathcal{L}(T) = -\sum_{\mathbf{x}_i \in \mathcal{D}} \left[\sum_y p^T(y|\mathbf{x}_i;\tau) \log p^S(y|\mathbf{x}_i;\tau) \right] \tag{19.62}$$

where $\tau > 0$ is a temperature parameter applied to the softmax output, which is used to perform **label smoothing**. If S has the same form as T, this is known as **self-training**, as discussed in Sec. 19.6.1. However, normally the student S is smaller than the teacher T. (For example, T might be a high capacity model, and S is a lightweight version that runs on a phone.) See Fig. 19.22 for an illustration of the overall approach.

19.7 Active learning

In **active learning**, the goal is to identify the true predictive mapping $y = f(\mathbf{x})$ by querying as few $(\mathbf{x}, y)$ points as possible. There are three main variants. In **query synthesis**, the algorithm gets to choose any input $\mathbf{x}$, and can ask for its corresponding output $y = f(\mathbf{x})$. In **pool-based active learning**, there is a large, but fixed, set of unlabeled data points, and the algorithm gets to ask for a label for one or more of these points. Finally, in **stream-based active learning**, the incoming data is arriving continuously, and the algorithm must choose whether it wants to request a label for the current input or not.

There are various closely related problems. In **Bayesian optimization** (Sec. 8.8.3), the goal is to estimate the location of the global optimum $\mathbf{x}^* = \text{argmin}_{\mathbf{x}} f(\mathbf{x})$ in as few queries as possible; typically we fit a surrogate (response surface) model to the intermediate $(\mathbf{x}, y)$ queries, to decide which question to ask next. In **experiment design**, the goal is to infer a parameter vector of some model, using carefully chosen data samples $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, i.e. we want to estimate $p(\boldsymbol{\theta}|\mathcal{D})$ using as little data as possible. (This can be thought of as an unsupervised, or generalized, form of active learning.)

In this section, we give a brief review of the pool based approach to active learning. For more details, see e.g., [Set12] for a review.

19.7.1 Decision-theoretic approach

In the decision theoretic approach to active learning, proposed in [KHB07; RM01], we define the utility of querying $\mathbf{x}$ in terms of the **value of information**. In particular, we define the utility of issuing query $\mathbf{x}$ as

$$U(\mathbf{x}) \triangleq \mathbb{E}_{p(y|\mathbf{x},\mathcal{D})} \left[\min_a R(a|\mathcal{D}) - R(a|\mathcal{D}, (\mathbf{x}, y)) \right] \tag{19.63}$$

where $R(a|\mathcal{D}) = \mathbb{E}_{p(\theta|\mathcal{D})} [\ell(\theta, a)]$ is the posterior expected loss of taking some future action a given the data $\mathcal{D}$ observed so far. Unfortunately, evaluating $U(\mathbf{x})$ for each $\mathbf{x}$ is quite expensive, since for each possible response y we might observe, we have to update our beliefs given $(\mathbf{x}, y)$ to see what affect it might have on our future decisions (similar to look ahead search technique applied to belief states).

19.7.2 Information-theoretic approach

In the information theoretic approach to active supervised learning, we avoid using task-specific loss functions, and instead focus on learning our model as well as we can. In particular, [Lin56] proposed to define the utility of querying $\mathbf{x}$ in terms of **information gain** about the parameters $\boldsymbol{\theta}$, i.e., the reduction in entropy:

$$U(\mathbf{x}) \triangleq \mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D})\right) - \mathbb{E}_{p(y|\mathbf{x},\mathcal{D})}\left[\mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D},\mathbf{x},y)\right)\right] \tag{19.64}$$

(Note that the first term is a constant wrt $\mathbf{x}$, but we include it for later convenience.) Exercise 19.1 asks you to show that this objective is identical to the expected change in the posterior over the parameters which is given by

$$U'(\mathbf{x}) \triangleq \mathbb{E}_{p(y|\mathbf{x},\mathcal{D})}\left[\mathbb{KL}\left(p(\boldsymbol{\theta}|\mathcal{D},\mathbf{x},y)\|p(\boldsymbol{\theta}|\mathcal{D})\right)\right] \tag{19.65}$$

Using symmetry of the mutual information, we can rewrite Eq. (19.64) as follows:

$$\begin{aligned} U(\mathbf{x}) &= \mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D})\right) - \mathbb{E}_{p(y|\mathbf{x},\mathcal{D})}\left[\mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D},\mathbf{x},y)\right)\right] & (19.66)\\ &= \mathbb{I}(\boldsymbol{\theta},y|\mathcal{D},\mathbf{x}) & (19.67)\\ &= \mathbb{H}\left(p(y|\mathbf{x},\mathcal{D})\right) - \mathbb{E}_{p(\boldsymbol{\theta}|\mathcal{D})}\left[\mathbb{H}\left(p(y|\mathbf{x},\boldsymbol{\theta})\right)\right] & (19.68)\end{aligned}$$

The advantage of this approach is that we now only have to reason about the uncertainty of the predictive distribution over outputs y, not over the parameters $\boldsymbol{\theta}$.

Eq. (19.68) has an interesting interpretation. The first term prefers examples $\mathbf{x}$ for which there is uncertainty in the predicted label. Just using this as a selection criterion is called **maximum entropy sampling** [SW87]. However, this can have problems with examples which are inherently ambiguous or mislabeled. The second term in Eq. (19.68) will discourage such behavior, since it prefers examples $\mathbf{x}$ for which the predicted label is fairly certain once we know $\boldsymbol{\theta}$; this will avoid picking inherently hard-to-predict examples. In other words, Eq. (19.68) will select examples $\mathbf{x}$ for which the model makes confident predictions which are highly diverse. This approach has therefore been called **Bayesian active learning by disagreement** or **BALD** [Hou+12].

This method can be used to train classifiers for other domains where expert labels are hard to acquire, such as medical images or astronomical images [Wal+20].

19.7.3 Batch active learning

So far, we have assumed a greedy or **myopic** strategy, in which we select a single example $\mathbf{x}$, as if it were the last datapoint to be selected. But sometimes we have a budget to collect a set of B samples, call them $(\mathbf{X},\mathbf{Y})$. In this case, the information gain criterion becomes $U(\mathbf{X}) = \mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D})\right) - \mathbb{E}_{p(\mathbf{Y}|\mathbf{X},\mathcal{D})}\left[\mathbb{H}\left(p(\boldsymbol{\theta}|\mathbf{Y},\mathbf{X},\mathcal{D})\right)\right]$. Unfortunately, optimizing this is NP-hard in the horizon length B [KLQ95; KG05].

Fortunately, under certain conditions, the greedy strategy is near-optimal, as we now explain. First note that, for any given $\mathbf{X}$, the information gain function $f(\mathbf{Y}) \triangleq \mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D})\right) - \mathbb{H}\left(p(\boldsymbol{\theta}|\mathbf{Y},\mathbf{X},\mathcal{D})\right)$ maps a set of labels $\mathbf{Y}$ to a scalar. It is clear that $f(\emptyset) = 0$, and that f is non-decreasing, meaning $f(Y^{\text{large}}) \geq f(Y^{\text{small}})$, due to the "more information never hurts" principle. Furthermore, [KG05] proved that f is **submodular**. As a consequence is that a sequential greedy approach is within a constant factor of optimal. If we combine this greedy technique with the BALD objective, we get a method called **BatchBALD** [KAG19].

19.8 Exercises

Exercise 19.1 [Information gain equations]

Consider the following two objectives for evaluating the utility of querying a datapoint $\mathbf{x}$ in an active learning setting:

$$U(\mathbf{x}) \triangleq \mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D})\right) - \mathbb{E}_{p(y|\mathbf{x},\mathcal{D})}\left[\mathbb{H}\left(p(\boldsymbol{\theta}|\mathcal{D},\mathbf{x},y)\right)\right] \tag{19.69}$$

$$U'(\mathbf{x}) \triangleq \mathbb{E}_{p(y|\mathbf{x},\mathcal{D})}\left[\mathbb{KL}\left(p(\boldsymbol{\theta}|\mathcal{D},\mathbf{x},y)\|p(\boldsymbol{\theta}|\mathcal{D})\right)\right] \tag{19.70}$$

Prove that these are equal.