Chapter 1

Representation learning

In this book we have seen many ways to represent visual signals – in the spatial domain vs frequency domain, with pyramids and filter responses, and more. We have seen that the choice of representation is critical: each type of representation makes some operations easy and others hard. We also saw that deep neural nets can be thought of as transforming the data, layer by layer, from one representation into another and another, and that those representations are transferrable. If representations are so important, why not set our objective to be "come up with the best representation possible." In this chapter, we will try to do exactly that.

This chapter is primarily an investigation of *objectives*. We will identify different objective functions that capture properties of what it means to be a good representation. Then we will train deep nets to optimize these objectives, and investigate the resulting learned representations.

1.1 Technical setting

Before diving in, let us present the basic problem statement and the notation we will be using throughout this chapter. The goal of representation learning is to learn a mapping from datapoints, $\mathbf{x} \in \mathcal{X}$, to abstract representations, $\mathbf{z} \in \mathcal{Z}$, as schematized in Figure 1.1:

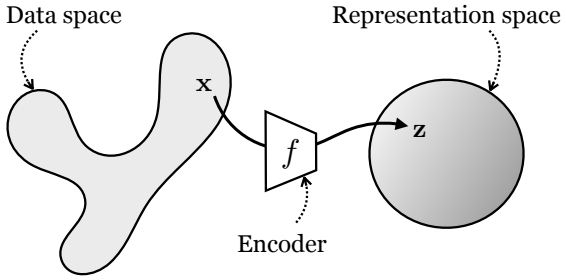


Figure 1.1: The goal in this chapter is to learn mappings from datapoints to abstract representations, which are typically simpler and more useful than the raw data.

We call this mapping an **encoding** and learn an **encoder** function $f : \mathcal{X} \to \mathcal{Z}$. Typically both $\mathbf{x}$ and $\mathbf{z}$ are high-dimensional vectors, and $\mathbf{z}$ is called a **vector embedding** of $\mathbf{x}$. The mapping f is trained so that $\mathbf{z}$ has certain desirable properties; common desirata include that $\mathbf{z}$ be lower dimensional than $\mathbf{x}$, that the distribution of $\mathbf{z}$'s, i.e. $p(\mathbf{z})$, has a simple structure (such as being the unit Normal distribution), and that the dimensions of $\mathbf{z}$ be independent factors, a.k.a. the representation should be *disentangled* [Bengio et al. 2013]. In this way $\mathcal{Z}$ is a simpler, more abstracted, or better organized representational space than $\mathcal{X}$, as reflected

in the shapes in Figure 1.1.

1.2 What makes for a good representation?

A good representation is one that makes subsequent problem solving easier. Below we discuss several desiderata of a good representation:

Compression A good representation is one that is parsimonious and captures just the essential characteristics of the data necessary for tasks of human interest. There are at least three senses in which compressed representations are desirable:

1. Compressed representations require less memory to store.

2. Compression is a way to achieve invariance to nuisance factors. Depending on what the representation will be used for, nuisances might include camera noise, lighting conditions, and viewing angle. If we can factor out those nuisances, and discard them from our representation, then the representation will be more useful for tasks like object recognition where camera and lighting may change without affecting the semantics of the observed object.

3. Compression is an embodiment of *Occam's razor*: among competing hypotheses that all explain the data equally well, the simplest is most likely to be true. As we will see below, and in the next chapter, many representation learning algorithms seek simple representations from which you can regenerate the raw data (e.g., autoencoders). Such representations explain the data in the formal sense that they assign high likelihood to the data (this is the same sense to which the idea of "Bayesian Occam's Razor" applies; see [MacKay et al. 2003] for a mathematical treatment of this idea). Therefore, we can state Occam's razor for representation learning: among two representations that fit the data equally well, prefer the more compressed.

This version of Occam's razor is also known as the "Minimum Description Length Principle" [Grünwald 2007]

Many representation learning algorithms capture these goals by penalizing or constraining the complexity of the learned representation. Such a penalty must be paired with some other objective or else it will lead to a degenerate solution: just make the representation contain nothing at all. Usually we try to compress the signal in certain ways while preserving other aspects of the information it carries. *Autoencoders* and *contrastive learning*, which we will describe below, are two representation learning algorithms based on the idea of compression.

Prediction The whole purpose of having a visual system is to be able to take actions that achieve desirable future outcomes. Predicting the future is therefore very important, and we often want representations that act as a good substrate for prediction.

The idea of prediction can be generalized to involve more than just predicting the *future*. Instead we may want to "predict" the past (given when I'm seeing today, what happened yesterday?), or we may want to predict gaps in our measurements, a problem known as **imputation**. Filling in missing pixels in a corrupted image is an example of imputation, as is superresolving a movie to play at a higher framerate. We may even want to perform more abstract kinds of predictions, like predicting what some other agent is thinking about (a problem called "Theory of Mind"). In general, "prediction" can refer to the inference of any arbitrary property of the world given observed data. Facilitating the prediction of important world properties – the future, the past, mental states, cause and effect, etc – is perhaps the defining property of a good representation.

Most representation learning algorithms in vision are about learning compressed encodings of the world around us that are also predictive of the future (and therefore a good substrate for decision making). Beyond just compression and prediction, some works have explored other goals, including that the representation be disentangled [Bengio et al. 2013], interpretable [Koh et al. 2020], and actionable [Soatto 2013] (you can use it as an effective

Learning Method	Learning Principle	Short Summary
Autoencoding	Compression	Remove redundant information
Contrastive	Compression	Achieve invariance to viewing transformations
Clustering	Compression	Quantize continuous data into discrete categories
"Prediction"	Prediction	Predict the future
Imputation	Prediction	Predict missing data
Pretext tasks	Prediction	Predict abstract properties of your data

Table 1.1: One way to categorize different representation learning methods.

substrate for control). All these goals – compression, prediction, interpretability, etc – are not in contrast to each other but rather overlap; for example, more compressed representations may be simpler and hence easier for a human to interpret.

Types of representation learners Representation learning algorithms are mostly differentiated by the objective function they optimize, as well as the constraints on their hypothesis space. We explore a variety of common objectives in the following sections, and summarize how these relate to the core principles of compression and prediction in Table 1.1.

1.3 Autoencoders

Autoencoders are one of the oldest and most common kind of representation learners [Rumelhart et al. 1985, Ballard 1987]. An autoencoder is a function that maps data back to itself (hence the "auto"):

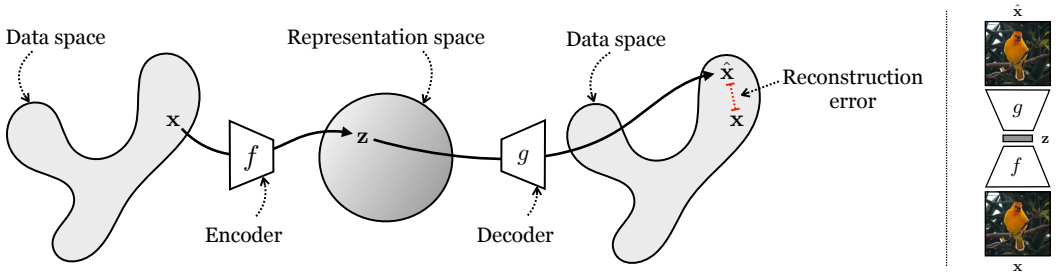


Figure 1.2: Autoencoder.

On the right is an example of an autoencoder applied to an image. At first glance this might not seem very useful: the output is the same as the input! The key trick is to impose constraints on the intermediate representation $\mathbf{z}$, so that it becomes useful. The most common constraint is compression: $\mathbf{z}$ is a low-dimensional, compressed representation of $\mathbf{x}$.

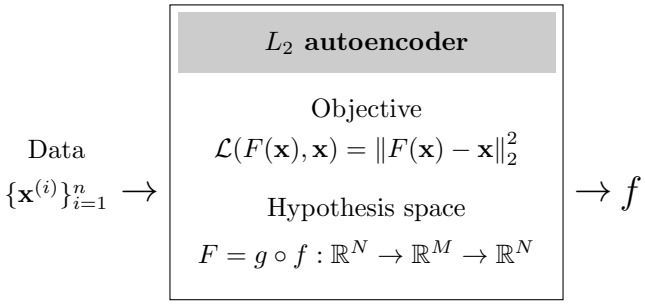
To be precise, an autoencoder F consists of two parts, an encoder f and a decoder g, with $F = f \circ g$. The encoder, $f : \mathbb{R}^N \to \mathbb{R}^M$, maps high-dimensional data $\mathbf{x} \in \mathbb{R}^N$ to a vector embedding $\mathbf{z} \in \mathbb{R}^M$. Typically, the key property is that $M < N$, that is we have performed **dimensionality reduction** (although autoencoders can also be constructed without this property, in which case they are not doing dimensionality reduction but instead may place other explicit or implicit constraints on $\mathbf{z}$). The decoder, $g : \mathbb{R}^M \to \mathbb{R}^N$ performs the inverse mapping to f, and ideally g is exactly the inverse function f^{-1}. Because it may be impossible to perfectly invert f, we use a loss function to penalize how far we are from perfect inversion, and a typical choice is the squared error "reconstruction loss", $\mathcal{L}_{\texttt{recon}}(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2$. Then the learning problem is:

$$f^*, g^* = \arg\min_{f,g} \mathbb{E}_{\mathbf{x}} \|\mathbf{x} - g(f(\mathbf{x}))\|_2^2 \tag{1.1}$$

The idea is to find a lower dimensional representation of the data from which we are able to reconstruct the data. An autoencoder is fine with throwing away any redundant features of the raw data but is not happy with actually losing information. The particular loss function determines what kind of information is preferred when the bottleneck cannot support preserving *all* the information and a hard choice has to be made. The L_2 loss decomposes as $\|\mathbf{x} - g(f(\mathbf{x}))\|_2^2 = \sqrt{\sum_i (x_i - g(f(\mathbf{x}))_i)^2}$, i.e. a sum over individual pixel errors. This means that it only cares about matching each individual pixel intensity $f(\mathbf{x})_i$ to the ground truth x_i and does not directly penalize patch-level statistics of $\mathbf{x}$ (i.e. statistics $\phi(\mathbf{x})$ that do not factorize as a product $\prod_i \psi_i(x_i)$ over per-pixel functions ψ_i for any possible set of functions $\{\psi_i\}$). Autoencoders can also be constructed using loss functions that penalize higher-order statistics of $\mathbf{x}$, and this allows, for example, penalizing errors in the reconstruction of edges, textures, and other perceptual structures beyond just pixels [Snell et al. 2017].

Autoencoders are one of our favorite algorithms. They may not seem like much at first glance, but they actually appear all over the place, and many methods in this book can be considered to be special kinds of autoencoders, if you squint. Two examples: the steerable pyramid from Chapter **??** is an autoencoder, and the CycleGAN algorithm from Chapter **??** is also an autoencoder. See if you can find more examples.

The learning diagram for the basic L_2 autoencoder looks like this:

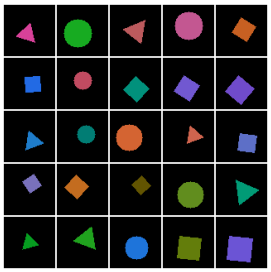


The optimizer is arbitrary but a typical choice would be gradient descent. The functional form of f and g are typically deep neural nets. The output is a learned data encoder f. We also get a learned decoder g as a byproduct, which has its own uses, but, for the purpose of representation learning, is usually discarded and only used as scaffolding for training what we really care about: f.

Closely related to autoencoders are other dimensionality reduction algorithms like Principle Components Ananlysis (PCA). In fact, an L_2 autoencoder, for which both f and g are linear functions, learns an M-dimensional embedding that spans the same subspace as a PCA projection to M-dimensions [Bourlard and Kamp 1988].

1.3.1 Experiment: Do autoencoders learn useful representations?

It is clear from the above that autoencoders will learn a *compressed* representation of the data, but do they learn a *useful* representation? Of course the answer to this question will depend on what we will use the representation for. Let's explore how autoencoders work on a simple data domain, consisting just of colored circles, triangles, and squares. The data consists of 64k images that look like this: Each shape has a randomized size, position,



rotation. Each shape's color is sampled from one of eight color classes (orange, green, purple, etc) plus a small random perturbation.

For the autoencoder architecture we use a convolutional encoder and decoder, each with 6 convolutional layers interspersed with relu nonlinearities and a 128-dimensional bottleneck

(i.e. $M = 128$). We train this autoencoder for 20k steps of stochastic gradient descent, using the Adam optimizer [Kingma and Ba 2014]) with a batch size of 128.

After training, does this autoencoder obtain a good representation of the data? To answer this question, we need ways of evaluating the quality of a representation. There are many ways and indeed how to evaluate representations is an open area of research. But here we will stick with a very simple approach: see if the nearest neighbors, in representational space, are meaningful.

We can test this in two ways: 1) for a given query, visualize the images in the dataset whose embeddings are nearest neighbors to the query's embedding, 2) measure the accuracy of a 1-nearest-neighbor classifier in embedding space. Below we show both these analyses: Recall that every layer of a neural net can be considered as an embedding (representation) of

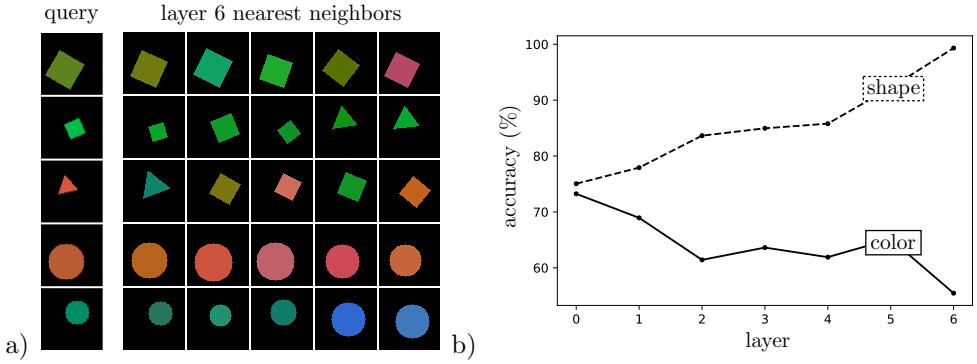


Figure 1.3: (a) Nearest neighbors, in autoencoder embedding space, to a set of query images. (b) Classification accurracy of a 1-nearest-neighbor classifier of color and shape using the embeddings at each layer of the autoencoder's encoder.

the data. On the left we show the nearest neighbors to a set of query images, using the layer 6 embeddings of the data as the feature space in which to measure distance ("nearness"). Since we have a 6-layer encoder, layer 6 is the bottleneck layer, the output of the full encoder f. Notice that the neighbors are indeed similar to the queries in terms of their colors, shapes, positions, and rotations – it seems the autoencoder produced a meaningful representation of the data!

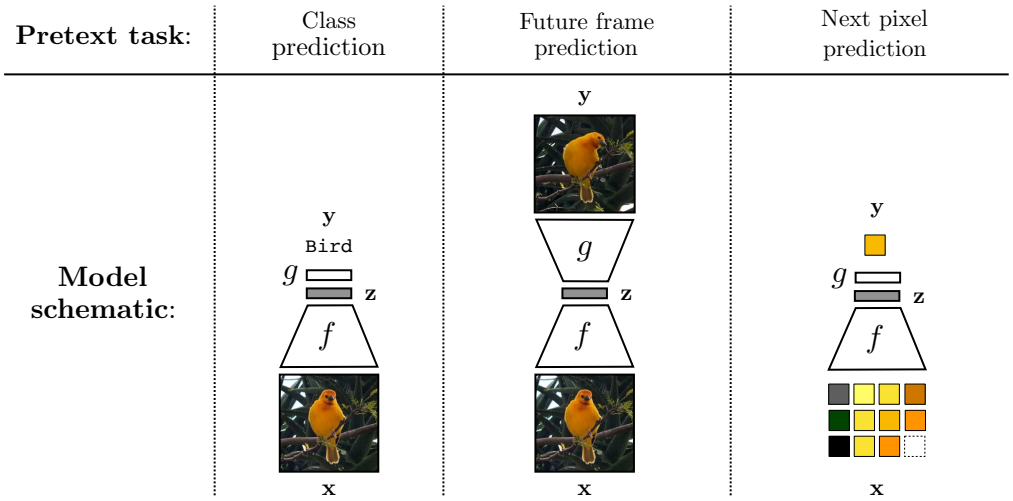
Next we will probe a bit deeper, and ask "how effective are these embeddings at classifying key properties of the data?" On the right we show the accuracy of a 1-nearest-neighbor color classifier (between the 8 color classes) and a shape classifier (circle vs triangle vs square) applied on embedding vectors at each layer of the autoencoder's encoder. The 0-th layer corresponds to measuring distance in the raw pixel space – interestingly, the color classifier does best on this raw represention of the data! That's because the pixels are a more or less direct representation of color. Color classification performance gets worse and worse as we go deeper in the encoder. On the other hand, shape is not explicit in raw pixels, so measuring distance in pixel-space does not yield good shape nearest neighbors. Deeper layers of the encoder give representations that are increasingly sensitive to shape similarity, and shape classification performance gets better. In general, there is no one representation that is universally the best. Each is good at capturing some properties of the data and bad at capturing others. As we go deeper into an autoencoder, the embeddings tend to become more abstracted and therefore better at capturing abstract properties like shape and worse at capturing superficial properties like color. For many tasks – object recognition, geometry understanding, future prediction – the salient information is rather abstracted from the raw data, and therefore for these tasks deeper embeddings tend to work better.

1.4 Predictive encodings

Neuroscientists think the brain also uses prediction to better encode sensory signals, but focus on a different part of the problem. The idea of *predictive coding* states that the sensory cortex only transmits the difference between its predictions and the actual observed signal [Huang and Rao 2011]. This chapter presents how to learn a representation that can make good predictions in the first place; "predictive coding" focuses on one thing you can do with such a representation: use it to compress future signals, just transmitting the surprises.

We have already seen many kinds of predictive learning, indeed almost any function can be thought of as making a "prediction". In predictive representation learning, the goal is not to make predictions per se, but to use prediction as a way to train a useful representation. The way to do this is first encode the data into an embedding $\mathbf{z}$, then map from the embedding to your target prediction. This gives a composition of functions, just like with the autoencoder, which can be trained with a prediction task yet yield a good data encoder. The prediction task is a **pretext task** for learning good representations.

Different kinds of prediction tasks have been proposed for learning good representations, and depending on the properties you want in your representation different prediction tasks will be best. Examples include predicting future frames in a video [Recasens et al. 2021] and predicting the next pixel in an image given a sequence of preceding pixels [Chen et al. 2020]. It is also possible to use an image's semantic class as the prediction target. In that case, the prediction problem is identical to training an image classifier, but the goal is very different. Rather than obtaining a good classifier at the end, our goal is instead to obtain a good image encoding (which is predictive of semantics) [Donahue et al. 2014]. These three examples are visualized below:



These tasks look a lot like supervised learning. We avoid calling it "supervised" because that connotes that we have examples of input-output pairs on the target task. Here that would be input data and exemplar output representations. But we don't have that! The "supervision" in this setup is a pretext task that we hope induces good representations.

Let us now describe the predictive learning problem more formally. Let $\mathbf{y}$ be the prediction target. Then predictive representation learning looks like this:

Predictive rep learning

Data: $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{n} \rightarrow$

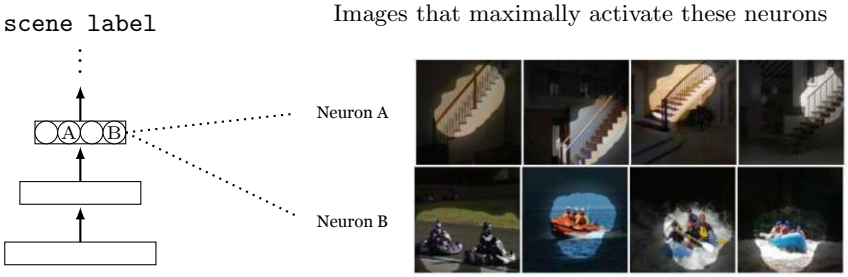
Objective: $\mathcal{L}(F(\mathbf{x}), \mathbf{y}) = D(F(\mathbf{x}), \mathbf{y})$

Hypothesis space: $F = f \circ g : \mathbb{R}^N \rightarrow \mathbb{R}^M \rightarrow \mathbb{R}^K$

$\rightarrow f$

D is some distance function, e.g., L_2. Just like with the autoencoder, f and g are usually neural nets but may be any family of functions – often g is just a single linear layer. Unlike with autoencoders, there is no standard setting for the relative dimensionalities of N, M, and K; instead it depends on the prediction task. If the task is image classification, then N will be the (large) dimensionality of the input pixels, M will usually be much lower dimensional, and K will be the number of image classes (to output K-dimensional class probability vectors).

Object detectors emerge from scene-level supervision The real power of these pretexts tasks lies not in solving the tasks themselves but in acquiring useful image embeddings **z** as a byproduct of solving the pretext task. The amazing thing that ends up happening is that z-space may have *emergent structure* that was not explicit in either the raw training data nor the pretext task. As a case study, consider the work of [Zhou et al. 2015]. They trained a CNN to perform scene classification – is this image a living room, bedroom, kitchen, etc. The net did wonderfully at that task, but that wasn't the point. The researchers instead peeled apart the net and looked at what input images were causing different neurons within the net to fire. What they found is that there were neurons, on hidden layers in the net, that fired selectively whenever the input image was of a specific object class. For example, one particular neuron would fire when the input was a staircase, and another neuron would fire predominantly for inputs that were rafts. The images below show four of the top images that activate these two particular neurons. These particular neurons are on convolutional layers, so really each is a filter response; the highlighted regions indicate where the feature map for that filter exceeds a threshold:



What this shows is that *object* detectors – i.e. neurons that selectively fire when they see a particular object class – emerge in the hidden layers of a CNN trained only to perform *scene* classification. This makes sense in retrospect – how else would the net recognize scenes if not first by identifying their constituent objects? – but it was quite a shock for the community to see it for the first time. It gave some evidence that the way we humans parse and recognize scenes may match the way CNNs also internally parse and recognize scenes.

1.5 Self-supervised learning

Predictive learning is great when we have good prediction targets that induce good representations. What if we don't have labeled targets provided to us? Instead we could try to cook up targets out of the raw data itself – for example, we could decide that the topright pixel's color will be the "label" of the image. This idea is called "self-supervision". It looks like this:

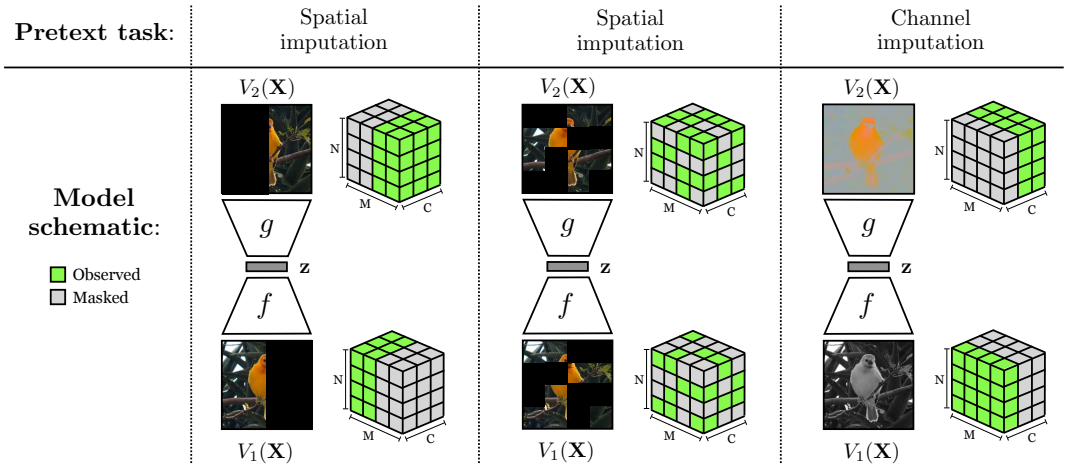
$$\text{Data} \quad \{\mathbf{X}^{(i)}\}_{i=1}^{n} \rightarrow \boxed{\begin{array}{c}\textbf{Self-sup rep learning} \\ \text{Objective} \\ \mathcal{L}(F(\mathbf{x}), \mathbf{y}) = D(F(V_1(\mathbf{X})), V_2(\mathbf{X})) \\ \text{Hypothesis space} \\ F = f \circ g : \mathbb{R}^N \rightarrow \mathbb{R}^M \rightarrow \mathbb{R}^K\end{array}} \rightarrow f$$

V_1 and V_2 are two different functions of the *full data tensor* $\mathbf{X}$. For example, V_1 might be the left side of the image $\mathbf{X}$ and V_2 could be the right side, so the pretext task is to predict the right side of an image from its left side. In fact, several of the examples we gave for predictive learning are of the self-supervised variety: predicting a future frame, or a next

pixel, can be cooked up just by splitting to a video into past and future frames, or splitting an image into previous and next pixels in a raster order sequence.

1.6 Imputation

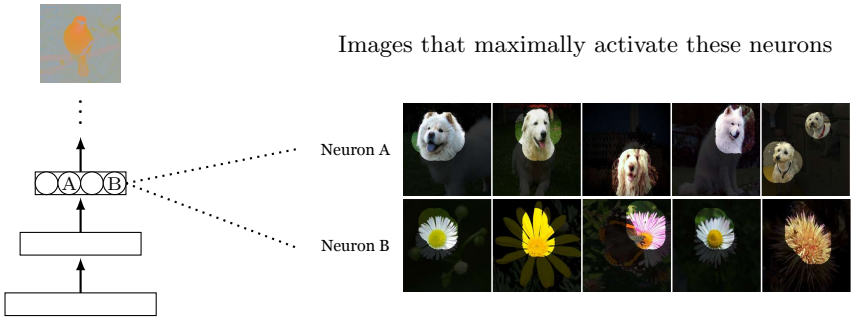
Imputation is a special case of self-supervised learning, where the prediction targets are missing elements of the input data. For example, predicting missing pixels is an imputation problem, as is colorizing a black and white photo (i.e. predicting missing color channels):



Imputation – whether over spatial masks or missing channels – can result in effective visual representations [Vincent et al. 2008, Pathak et al. 2016, He et al. 2022, Zhang et al. 2016, Larsson et al. 2016, Zhang et al. 2017]. Notice that predicting future frames and next pixels – our examples from above – are also imputation problems.

Above we described how object detectors emerge as a byproduct of training a net to perform scene classification. What do you think emerges as a byproduct of training a net to perform colorization?

It may surprise you to find that the answer is: object detectors once again! This certainly surprised us when we saw the following results in [Zhang et al. 2016]:



In fact, object detectors emerge in CNNs for just about any reasonable pretext task – scene recognition, colorization, inpainting missing pixels, and more. What may be going on is that the notion of "objects" is not just a human invention but rather maps onto some fundamentally useful structure out there in the world, and any roughly similar intelligence – an AI system, an alien on Mars – will learn to see the world in a similar way as we do, i.e. as composed of the modular and compositional parts we call objects.

1.7 Abstract pretext tasks

Other varieties of self-supervised learning set up more abstract prediction problems, rather than just aiming to predict missing data. For example, we may try to predict if an image has been rotated 90 degrees [Komodakis and Gidaris 2018], or we may aim to predict the relative position of two image patches given their appearance [Doersch et al. 2015]. These pretext tasks can induce effective visual representations because solving them requires learning about semantic and geometric regularities in the world, such as that clouds tend to appear near the top of an image.

1.8 Clustering

One way to compress a signal is dimensionality reduction, of which we saw an example of above with the autoencoder. Another way is to quantize the signal into discrete categories, an operation known also as **clustering**. Mathematically, clustering is a function $f : \{\mathbf{x}\}_{i=1}^{N} \rightarrow \{1, \ldots, k\}$, i.e. a mapping from the members of a dataset $\{\mathbf{x}\}_{i=1}^{N}$ to k integer classes (k can potentially be unbounded). Representing integers with one-hot codes, clustering looks like this:

Another name for clustering, which appears here and there in the representation learning literature, is **vector quantization**.

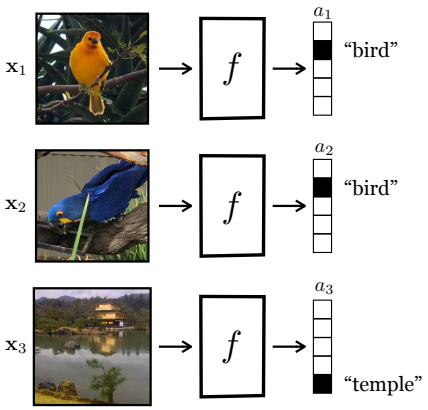


Figure 1.4: You can think of clustering as being just like image labeling, except that that labels are self-discovered rather than being predefined.

Clustering follows from the principle of compression: if we can well summarize a signal with just a discrete category label, then this summary can serve as a lighter weight and more abstracted substrate for further reasoning. If this idea seems abstract just consider the words "antelope", "giraffe", and "zebra". Those words are discrete category labels (i.e. integers) which summarize huge conceptual sets (just think of all the individual lives and richly diverse personalities you are lumping together with the simple word "antelope"). Words, then, are clusters! – they are mappings from data to integers – clustering is the problem of making up new words for things.

Words, of course, are given additional structure when used in a language – grammar, connotations, etc – beyond just being a set of clusters. The same kind of structure can be added on top of visual clusters.

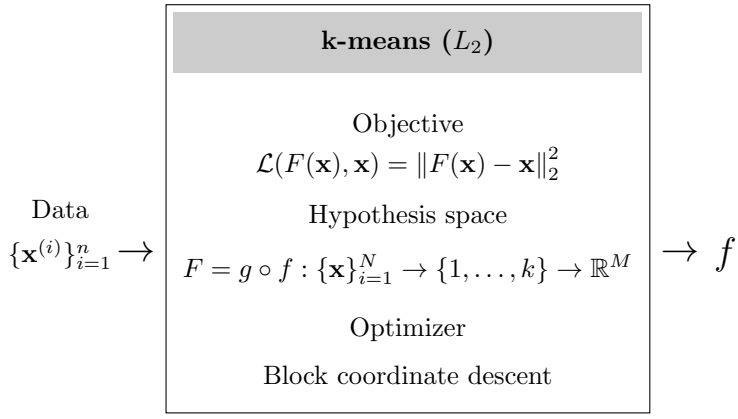
Many clustering algorithms not only partition the data but also compute a representation of the data within each cluster; this representation is sometimes called a **code vector**. The most common code vector is the **cluster center**, μ, i.e. the mean value of all datapoints assigned to the cluster. Clusters can also be represented by other statistics of the data assigned to them, such as the variance of this data or some arbitrary embedding vector, but this is less common. The set of cluster centers, $\{\mu_i\}_{i=1}^{k}$, is a representation of a whole data*set*. They summarize the main modes of behavior in the dataset.

1.8.1 K-means

There are many types of clustering algorithm but we will illustrate the basic principles just with one example, perhaps the most popular clustering algorithm of them all, **k-means**. K-means is a clustering algorithm that partitions the data into k clusters. Each cluster is also represented with a code vector $\mathbf{z} \in \mathbb{R}^M$. The k-means objective is to minimize the distance between each datapoint and the code vector of the cluster it is assigned to. This way, the code vector assigned to each datapoint will be a good approximation to the value of that datapoint, and we will have a faithful, but compressed, representation of the dataset.

There are two free parameter sets to optimize over: the code vectors and the cluster assignments. The cluster assignments can be represented with our clustering function f, and we will represent the code vectors for each cluster with a function $g : \{1, \dots, k\} \to \mathbb{R}^M$, where the data dimensionality is M. Both f and g can be implemented as lookup tables, since for both the input is a countable set. The k-means algorithm amounts to just filling in the ranges of these two lookup tables.

Now we are ready to present the full k-means algorithm, viewed as a learning algorithm. As you will see below, the learning diagram looks almost the same as for an autoencoder! The key differences are 1) the bottleneck is discrete integers rather than continuous vectors, and 2) the optimizer is slightly different (we will delve into it below).



There are two functions we are optimizing over: the encoder f and the decoder g. f can be parameterized by a set of integers $\{a_i\}_{i=1}^N$, $a_i \in \{1, \dots, k\}$ which specify the cluster assignment for each datapoint, that is, $f(\mathbf{x}^{(i)}) = a_i$. g can be parameterized by a set of k code vectors $\{\mathbf{z}_j\}_{j=1}^k$, $\mathbf{z}_j \in \mathbb{R}^M$, one for each cluster, so we have $F(\mathbf{x}^{(i)}) = \mathbf{z}_{a_i}$. g is differentiable w.r.t. to its parameters but f is not, because the parameters of f are discrete variables. This means that gradient descent will not be a suitable optimization algorithm (the gradient $\frac{\partial f}{\partial a_i}$ is undefined). Instead we will use the optimization strategy described next.

Optimizing k-means k-means uses an optimization algorithm called **block coordinate descent**. This algorithm splits up the optimization parameters into multiple "blocks". Then it alternates between *fully* minimizing the objective w.r.t. each block of parameters. In k-means there are two parameter blocks: $\{a_i\}_{i=1}^N$ and $\{\mathbf{z}_j\}_{j=1}^k$. So, we need to derive update rules that find the minimizer of the k-means objective w.r.t. each block. It turns out there are simple solutions for both:

$$a_i \leftarrow \arg\min_{j \in \{1,\dots,k\}} \left\| \mathbf{z}_j - \mathbf{x}^{(i)} \right\|_2^2 \quad \triangleleft \quad \text{Assign datapoint to nearest cluster} \tag{1.2}$$

$$\mathbf{z}_j \leftarrow \frac{1}{N} \sum_{i=1}^{N} 1(a_i = j) \mathbf{x}^{(i)} \quad \triangleleft \quad \text{Set code vector to cluster center} \tag{1.3}$$

These steps are repeated until a fixed point is reached, which occurs when all datapoints assigned to each cluster are closest to that cluster's code vector.

Why are these the correct updates? Equation 1.2 is straightforward: it's just a brute force enumeration of all k possible assignments, from which we select the one that minimizese the k-means objective for that datapoint ($\mathcal{L}(F(\mathbf{x}^{(i)}, \mathbf{x}^{(i)}))$. Equations 1.3 is the minimizer w.r.t. each $\mathbf{z}_j$ since our task is to minimize the objective over all datapoints, i.e. $\sum_{i=1}^{N} \left\|\mathbf{z}_{a_i} - \mathbf{x}^{(i)}\right\|_2^2$, which can be rewritten as $\sum_{j=1}^{k}\sum_{i=1}^{N} 1(a_i = j) \left\|\mathbf{z}_j - \mathbf{x}^{(i)}\right\|_2^2$. Each term in the outer sum is affected by only one of the k $\mathbf{z}_j$'s. Looking at one in isolation we are seeking $\arg\min_{\mathbf{z}_j} \sum_{i'=1}^{N} \left\|\mathbf{z}_j - \mathbf{x}^{(i')}\right\|_2^2$, where i' enumerates all the datapoints for which $a_i = j$. Recall that the point that minimizes the sum of squared distances from a dataset is the mean of the dataset. This yields our update in Equation 1.3: just set the code to be the mean of all datapoints assigned to that cluster.

Now we can see where the name k-*means* comes from: the optimal code vectors are the cluster means! The algorithm is very simple: compute the cluster means given current data assignments; re-assign each datapoint to nearest cluster mean; re-compute the means; and so on until convergence. An example of applying this algorithm to a simple dataset of 2D points is given in Figure 1.5.

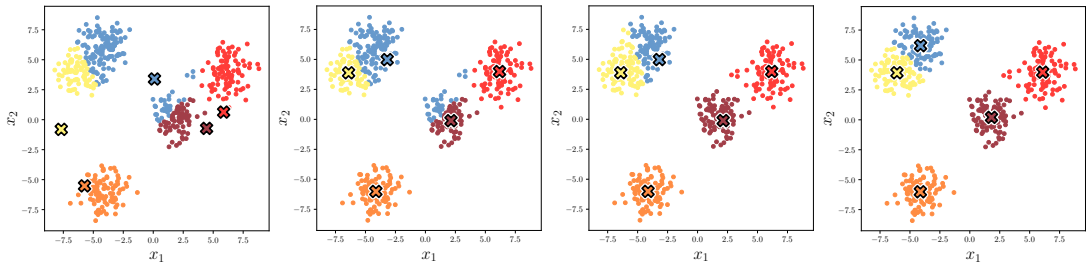


Figure 1.5: Iterations of k-means applied to a simple 2D dataset, with $k = 5$. The code vectors assigned to each cluster are marked with an x. a) Initialization. b) Update code vector to be cluster means. c) Update assignments. d) Converged solution (here occurs after 4 updates of both codes and assignments).

k-means from multiple perspectives: This section has presented k-means as a representation learning algorithm. In other texts you may encounter other views on k-means, e.g., as a way of interpreting your data or as simple generative model. These views are all complementary and all simultaneously true. Here we showed that k-means is like an autoencoder with a discrete bottleneck. In the next chapters we will encounter generative models, including one called a Gaussian Mixture Model (GMM). k-means can also be viewed as a vanilla form of a GMM. Later we will show that another kind of autoencoder, called a variational autoencoder (VAE), is a continuous version of a GMM. So we have a rich tapestry of relationships: autoencoders and VAEs are continous versions of k-means and GMMs, respectively. GMMs and VAEs are formal probabilistic models that, respectively, extend k-means and autoencoders (which do not come with probabilistic semantics). This is just one set of connections that can be made. In this book, be on the look out for more connections like this. It can be confusing at first to see multiple different perspectives on the same method: which one is correct? But rarely is there a single correct perspective. It is useful to identify the delta between each new model you encounter and all the models you already know. Usually there is a small delta w.r.t. *many* things you already know, and rarely is there no meaningful connection between any two arbitrary models. As an exercise, try picking a random concept on a random page in this book (or, for a challenge, any book on your bookshelf). What is the delta between that concept and k-means? In our experience, it will almost always be very small (but sometimes it takes a lot of effort to see the connection).

1.8.2 Clustering in vision

In vision, the problem of clustering is related to the idea of **perceptual organization**, which we cover in detail in Chapter **??**. We humans see the world as organized into different levels of perceptual structure: contours and surfaces, objects and events. These structures are groupings, or clusters, of related visual elements: a contour is a group of points that form a line, an object is a group of parts that form a cohesive, nameable whole, etc. Algorithms like k-means, in a suitable feature space, can discover them.

1.9 Contrastive learning

Dimensionality reduction and clustering algorithms learn compressed representations by creating an **information bottleneck**, that is, by constraining the number of bits available in the representation. An alternative compression strategy is to *supervise* what information should be thrown away. **Contrastive learning** is one such approach where a representation is supervised to be *invariant* to certain viewing transformations, resulting in a compressed representation that only captures the properties that are common between the different views.

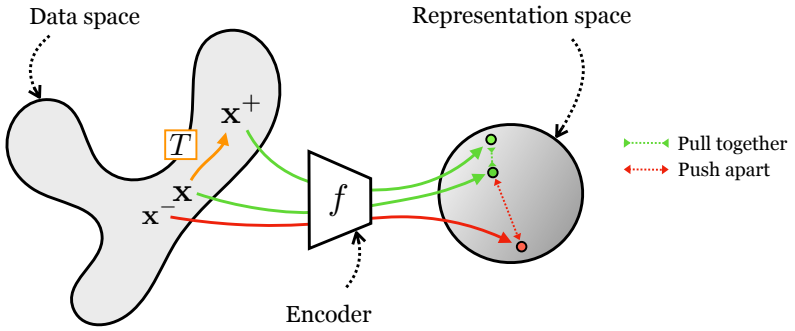
Contrastive learning is actually more closely related to clustering than it may at first seem. Contrastive learning maps similar datapoints to similar embeddings. Clustering is just the extreme version of this where there are only k distinct embeddings and similar datapoints get mapped to the *exact same* embedding.

Learning invariant representations is classic goal of computer vision. Recall that this was one of the reasons we use convolutional image filters: convolution is equivariant with camera translation, and invariance can then be achieved simply by pooling over filter responses. CNNs, through their convolutional architecture, bake translation invariance into the *hypothesis space*. In this section we will see how to incentivize invariances instead through the *objective* function.

The idea is to simply penalize deviations from the invariance we want. Suppose T is a transformation we wish our representation to be invariant to. Then we may use a loss of the form $\|f(T(\mathbf{x})) - f(\mathbf{x})\|_2^2$ to learn an encoder f that is invariant to T. We call such a loss an **alignment** loss [Wang and Isola 2020].

That seems easy enough, but you may have noticed a flaw: what if f just learns to output the zero vector all the time? Trivial alignment can be achieved when there is representational collapse, and all datapoints get mapped to the same arbitrary vector.

Contrastive learning fixes this issue by coupling an alignment loss with a second loss that pushes apart embeddings of datapoints for which we do not want an invariant representation. The supervision for contrastive learning comes in the form of *positive pairs* and *negative pairs*. Positive pairs are two datapoints we wish to align in z-space; if we wish for invariance to T then a positive pair should be constructed as $\mathbf{x}$ and $\mathbf{x}^+ = T(\mathbf{x})$. Negative pairs, $\{\mathbf{x}, \mathbf{x}^-\}$, are two datapoints that should be represented differently in z-space. Commonly, negative pairs are random independently sampled datapoints $\{\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\} \sim p_{\texttt{data}}(\mathbf{x})$. Given such data pairings, the objective is to pull together the positive pairs and push apart the negative pairs:



This kind of contrastive learning results in an embedding that is invariant to a transformation T. Extending this to achieve invariance to a *set* of transformations $\{T_1, \ldots, T_n\}$ is

straightforward – just apply the same loss for each of $T_1, \ldots, T_n$.

A second kind of contrastive learning is based on cooccurrence, where the goal is to learn a common representation of all cooccurring signals. This form of contrastive learning is useful for learning, e.g., an audiovisual representation where the embedding of an image matches the embedding of the sound for that same scene. Or, returning to our colorization example, an image representation where the embedding of the grayscale channels matches the embedding of the color channels. In both these cases we are learning to align cooccurring sensory signals. This kind of contrastive learning is schematized in Figure 1.6.

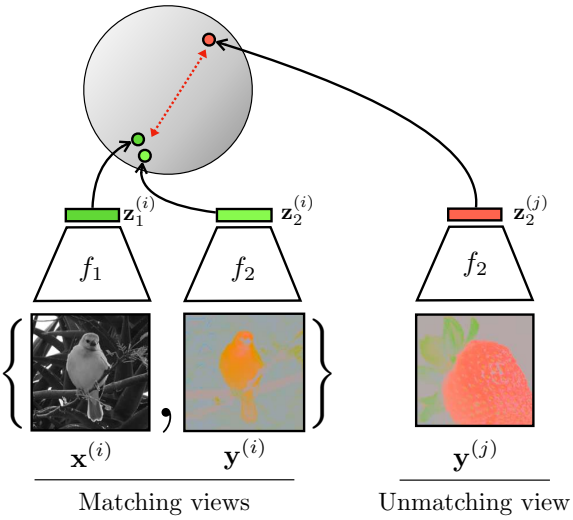


Figure 1.6: Contrastive learning from multiple views of the data. Figure inspired by [Tian et al. 2020].

In Figure 1.6, we refer to the two cooccurring signals – color and grayscale – as two different "views" of the total data tensor $\mathbf{X}$, just like we did in the sections above: $\mathbf{x} = V_1(\mathbf{X})$, $\mathbf{y} = V_1(\mathbf{X})$. You can think of these views either as resulting from sensory cooccurrences or as two transformations of $\mathbf{X}$, where the transformation in the color example is channel dropping. Thus, the two kinds of contrastive learning we have presented are really one and the same: any two signals can be considered transformations of a combined "total-signal", and any signal and its transformation can be considered two cooccurring ways of measuring the underlying world.

Nonetheless, it is often easiest to conceptualize these two approaches separately, and below we give learning diagrams for each:

Data
$\{\mathbf{x}^{(i)}\}_{i=1}^N, T \longrightarrow$

Constrastive learning (transformations)

Objective

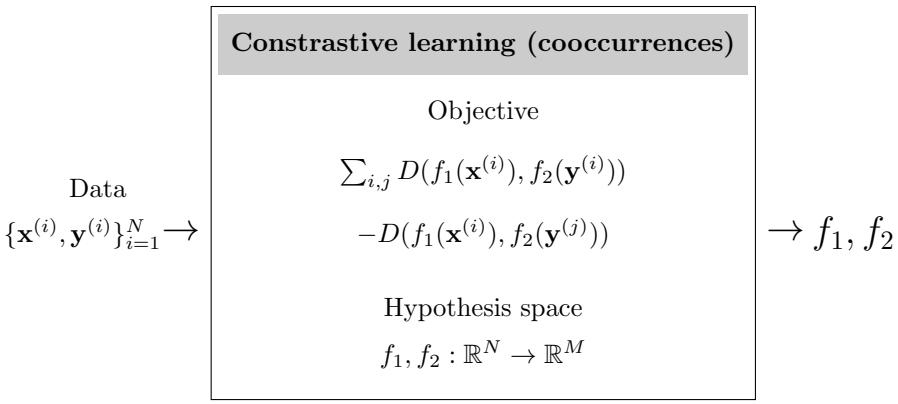
$\sum_{i,j} D(f(T(\mathbf{x}^{(i)})), f(\mathbf{x}^{(i)}))$

$-D(f(\mathbf{x}^{(i)}), f(\mathbf{x}^{(j)}))$

Hypothesis space

$f : \mathbb{R}^N \to \mathbb{R}^M$

$\longrightarrow f$



In these diagrams, D is a distance function. Above we give just one simple form for the contrastive objective above; many variations have been proposed. Three of the most popular are the "constrastive loss" [Hadsell et al. 2006] (an older definition of the term, now overloaded with our more general notion of contrastive losses being any that pull together positive samples and push apart negative samples), the **triplet loss** [Chechik et al. 2010], and the **InfoNCE loss** [Oord et al. 2018]. The "contrastive loss" and triplet loss add the concept of a **margin** to the vanilla formulation: they only push/pull when distance is less than a specified margin – otherwise points are considered far enough apart (or close enough together). The InfoNCE loss is a variation that treats establishing a contrast as a classification problem: it tries to move points apart until you can classify the positive sample, for a given anchor, separately from all the negatives. The general formulation of these losses takes as input an anchor $\mathbf{x}$, a positive example $\mathbf{x}^+$, and one or more negative examples $\mathbf{x}^-$. The positive and negative may be defined based on transformations, coocurrences, or something else. The full learning objective is to sum over many samples of anchors, positives, and negatives, sampled set evaluated according to the losses as follows:

$$\mathcal{L}(\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-) = \max(D(f(\mathbf{x}), f(\mathbf{x}^+) - m_{\texttt{pos}}, 0) - \max(m_{\texttt{neg}} - D(f(\mathbf{x}), f(\mathbf{x}^-), 0) \quad \triangleleft \text{ "contrastive"} \quad (1.4)$$

$$\mathcal{L}(\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-) = \max(D(f(\mathbf{x}), f(\mathbf{x}^+) - D(f(\mathbf{x}), f(\mathbf{x}^-) - m, 0) \quad \triangleleft \text{ triplet} \quad (1.5)$$

$$\mathcal{L}(\mathbf{x}, \mathbf{x}^+, \{\mathbf{x}_i^-\}_{i=1}^N) = -\log \frac{e^{f(\mathbf{x})^T f(\mathbf{x}^+)/\tau}}{e^{f(\mathbf{x})^T f(\mathbf{x}^+)/\tau} + \sum_i e^{f(\mathbf{x})^T f(\mathbf{x}_i^-)/\tau}} \quad \triangleleft \text{ InfoNCE} \quad (1.6)$$

Notice that the InfoNCE loss is a log softmax over a vector of scores $f_1(\mathbf{x})^T f_2(c)/\tau$ with $c = \{x^+, x_1^-, \ldots, x_N^-\}$; you can therefore think of this loss as corresponding to a classification problem where the ground truth class is x^+ and the other possible classes are $\bar{}_1, \ldots, x_N^-\}$ (refer to Chapter **??** to revisit softmax classification).

1.9.1 Alignment and uniformity

Wang and Isola (2020) showed that the contrastive loss (specifically the InfoNCE form) encourages two simple properties of the embeddings: alignment and uniformity. We have already seen that alignment is the property that two views in a positive pair will map to the same point in embedding point: i.e. the mapping is invariant to the difference between the views. **Uniformity** comes from the negative term, which encourages embeddings to spread out and tend toward an evenly spread, uniform distribution. Importantly, for this to work out mathematically, the embeddings must be *normalized*, that is, each embedding vector must be a unit vector. Otherwise, the negative term can push embeddings toward being infinitely far apart from each other. Fortunately, it is standard practice in contrastive learning (and many other forms of representation learning) to apply L_2 normalization to the embeddings. The result is that the embeddings will tend toward a uniform distribution

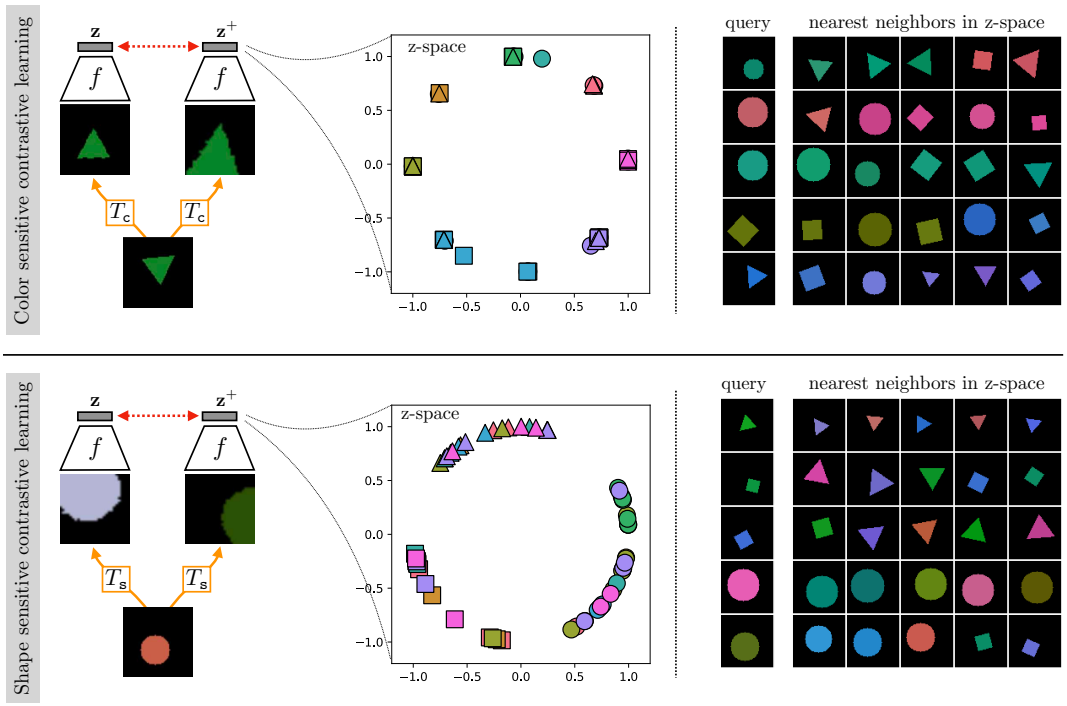


Figure 1.7: Contrastive learning on colored shapes using two different transformations for creating positive pairs. The choice of transformation controls which features the embedding becomes sensitive to and which it becomes invariant to.

over the surface of the M-dimensional hypersphere, where M is the dimensionality of the embeddings. See [Wang and Isola 2020] Theorem 1 for a formal statement of this fact.

A result of this analysis is we may explicit decompose contrastive learning into one loss for alignment and another for uniformity, with the following forms:

$$\mathcal{L}_{\texttt{align}}(f;\alpha) = \mathbb{E}_{(\mathbf{x},\mathbf{x}^+)\sim p_{\texttt{pos}}}[\left\|f(\mathbf{x}) - f(\mathbf{x}^+)\right\|_2^\alpha] \tag{1.7}$$

$$\mathcal{L}_{\texttt{unif}}(f;t) = \log \mathbb{E}_{\mathbf{x}\sim p_{\texttt{data}}}[e^{-t\left\|f(\mathbf{x})-f(\mathbf{x}^+)_2^2\right\|}] \tag{1.8}$$

$$\mathcal{L}(f;\alpha,t,\lambda) = \mathcal{L}_{\texttt{align}}(f;\alpha) + \lambda\mathcal{L}_{\texttt{unif}} \tag{1.9}$$

where $p_{\texttt{pos}}$ is the distribution of positive pairs and α, t, and λ are hyperparameters of the losses.

1.9.2 Experiment: designing embeddings with contrastive learning

Using the alignment and uniformity objective defined above, we will now illustrate how one can design embeddings with desired invariances. We will use the shape dataset described in Section 1.3, and will use the same encoder architecture and optimizer as from that section (a convnet with 6 layers, Adam optimizer, 20k iterations of SGD, batch size of 128). In contrast to the autoencoder experiment, however, we will set the dimensionality of the embedding to $M = 2$, so that we can visualize it in a 2D plot.

Suppose we wish to obtain an embedding that is sensitive to color and invariant to shape. Then we should choose a view transformation $T_{\texttt{c}}$ that preserves color while changing shape. A simple choice, that turns out to work, is for $T_{\texttt{c}}(\mathbf{x})$ to simply output a crop from the image $\mathbf{x}$ (this transformation does not really change the object's shape but still ends up resulting in shape-invariant embeddings since color is a much more obvious cue for the convnet to pick up on and use for solving the contrastive problem). The result of contrastive learning, using $L_{\texttt{align}} + L_{\texttt{unif}}$ on data generated from $T_{\texttt{c}}$ is shown on the top row of Figure 1.7. Notice

that the trained f maps colors to be clustered into our color classes while the clusters are spread out more or less uniformly across the surface of a circle in embedding space (a 1D hypersphere, since the embeddings are 2D and are normalized).

We can repeat the same experiment but with a view transformation $T_{\mathtt{s}}$ designed to be invariant to color. To do so we simply use the same transformation as for $T_{\mathtt{c}}$ (i.e. cropping) plus add a random shift in hue, brightness, saturation of the shape's color. Training on views generated from $T_{\mathtt{s}}$ results in the embeddings on the bottom row of Figure 1.7. Now the embeddings become invariant to color but cluster shapes and spread out the three clusters to be roughly uniformly spaced around the hypersphere.

1.10 Concluding remarks

This chapter, like much of this book, is about representations. Different representations make different problems easy – or hard. It is important to remember that every representation involves a set of tradeoffs: a good representation for one task may be a bad representation for another task. One of the current goals of computer vision research is to find *general-purpose representations*, and what this means is not that the representation will be good for all tasks but that it will be good just for the tasks humans care about, which is a very tiny subset of all possible tasks.

Bibliography

D. H. Ballard. Modular learning in neural networks. In *AAAI*, volume 647, pages 279–284, 1987.

Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

H. Bourlard and Y. Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4):291–294, 1988.

G. Chechik, V. Sharma, U. Shalit, and S. Bengio. Large scale online learning of image similarity through ranking. *Journal of Machine Learning Research*, 11(3), 2010.

M. Chen, A. Radford, R. Child, J. Wu, H. Jun, D. Luan, and I. Sutskever. Generative pretraining from pixels. In *International Conference on Machine Learning*, pages 1691–1703. PMLR, 2020.

C. Doersch, A. Gupta, and A. A. Efros. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE international conference on computer vision*, pages 1422–1430, 2015.

J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655. PMLR, 2014.

P. D. Grünwald. *The minimum description length principle*. MIT press, 2007.

R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16000–16009, 2022.

Y. Huang and R. P. Rao. Predictive coding. *Wiley Interdisciplinary Reviews: Cognitive Science*, 2(5):580–593, 2011.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

P. W. Koh, T. Nguyen, Y. S. Tang, S. Mussmann, E. Pierson, B. Kim, and P. Liang. Concept bottleneck models. In *International Conference on Machine Learning*, pages 5338–5348. PMLR, 2020.

N. Komodakis and S. Gidaris. Unsupervised representation learning by predicting image rotations. In *International Conference on Learning Representations (ICLR)*, 2018.

G. Larsson, M. Maire, and G. Shakhnarovich. Learning representations for automatic colorization. In *European conference on computer vision*, pages 577–593. Springer, 2016.

D. J. MacKay, D. J. Mac Kay, et al. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

A. v. d. Oord, Y. Li, and O. Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2536–2544, 2016.

A. Recasens, P. Luc, J.-B. Alayrac, L. Wang, F. Strub, C. Tallec, M. Malinowski, V. Patraucean, F. Altché, M. Valko, et al. Broaden your views for self-supervised video learning. *International Conference on Computer Vision*, 2021.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

J. Snell, K. Ridgeway, R. Liao, B. D. Roads, M. C. Mozer, and R. S. Zemel. Learning to generate images with perceptual similarity metrics. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 4277–4281. IEEE, 2017.

S. Soatto. Actionable information in vision. In *Machine learning for computer vision*, pages 17–48. Springer, 2013.

Y. Tian, D. Krishnan, and P. Isola. Contrastive multiview coding. In *European conference on computer vision*, pages 776–794. Springer, 2020.

P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

T. Wang and P. Isola. Understanding contrastive representation learning through alignment and uniformity on the hypersphere. In *International Conference on Machine Learning*, pages 9929–9939. PMLR, 2020.

R. Zhang, P. Isola, and A. A. Efros. Colorful image colorization. In *European conference on computer vision*, pages 649–666. Springer, 2016.

R. Zhang, P. Isola, and A. A. Efros. Split-brain autoencoders: Unsupervised learning by cross-channel prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1058–1067, 2017.

B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Object detectors emerge in deep scene cnns. *ICLR*, 2015.