CHAPTER 12

Unsupervised Learning

In previous chapters, we have largely focused on classification and regression problems, where we use supervised learning with training samples that have both features/inputs and corresponding outputs or labels, to learn hypotheses or models that can then be used to predict labels for new data.

In contrast to supervised learning paradigm, we can also have an unsupervised learning setting, where we only have features but no corresponding outputs or labels for our dataset. On natural question aries then: if there are no labels, what are we learning?

One canonical example of unsupervised learning is *clustering*, which we will learn about in Section 12.1. In clustering, the goal is to develop algorithms that can reason about "similarity" among data points's features, and group the data points into clusters.

Autoencoders are another family of unsupervised learning algorithms, which we will look at in Section 12.2, and in this case, we will be seeking to obtain insights about our data by learning compressed versions of the original data. Or, in other words, by finding a good lower-dimensional feature representations of the same data set. Such insights might help us to discover and characterize underlying factors of variation in data, which can aid in scientific discovery; to compress data for efficient storage or communication; or to preprocess our data prior to supervised learning, perhaps to reduce the amount of data that is needed to learn a good classifier or regressor.

12.1 Clustering

Oftentimes a dataset can be partitioned into different categories. A doctor may notice that their patients come in cohorts and different cohorts respond to different treatments. A biologist may gain insight by identifying that bats and whales, despite outward appearances, have some underlying similarity, and both should be considered members of the same category, i.e., "mammal". The problem of automatically identifying meaningful groupings in datasets is called clustering. Once these groupings are found, they can be leveraged toward interpreting the data and making optimal decisions for each group.

12.1.1 Clustering formalisms

Mathematically, clustering looks a bit like classification: we wish to find a mapping from datapoints, x, to categories, y. However, rather than the categories being predefined labels, the categories in clustering are automatically discovered *partitions* of an unlabeled dataset.

Because clustering does not learn from labeled examples, it is an example of an *unsupervised* learning algorithm. Instead of mimicking the mapping implicit in supervised training pairs $\{x^{(i)}, y^{(i)}\}_{i=1}^{n}$, clustering assigns datapoints to categories based on how the unlabeled data $\{x^{(i)}\}_{i=1}^{n}$ is *distributed* in data space.

Intuitively, a "cluster" is a group of datapoints that are all nearby to each other and far away from other clusters. Let's consider the following scatter plot. How many clusters do you think there are?

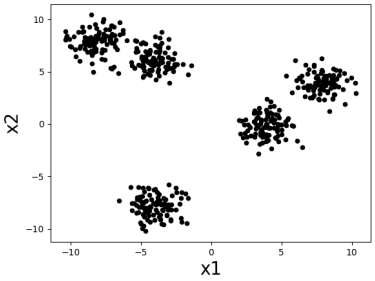


Figure 12.1: A dataset we would like to cluster. How many clusters do you think there are?

There seem to be about five clumps of datapoints and those clumps are what we would like to call clusters. If we assign all datapoints in each clump to a cluster corresponding to that clump, then we might desire that nearby datapoints are assigned to the same cluster, while far apart datapoints are assigned to different clusters.

In designing clustering algorithms, three critical things we need to decide are:

- How do we measure *distance* between datapoints? What counts as "nearby" and "far apart"?
- How many clusters should we look for?
- How do we evaluate how good a clustering is?

We will see how to begin making these decisions as we work through a concrete clustering algorithm in the next section.

12.1.2 The k-means formulation

One of the simplest and most commonly used clustering algorithms is called k-means. The goal of the k-means algorithm is to assign datapoints to k clusters in such a way that the variance within clusters is as small as possible. Notice that this matches our intuitive idea that a cluster should be a tightly packed set of datapoints.

Similar to the way we showed that supervised learning could be formalized mathematically as the minimization of an objective function (loss function + regularization), we will show how unsupervised learning can also be formalized as minimizing an objective function. Let us denote the cluster assignment for a datapoint $x^{(i)}$ as $y^{(i)} \in \{1, 2, \ldots, k\}$, i.e., $y^{(i)} = 1$ means we are assigning datapoint $x^{(i)}$ to cluster number 1. Then the k-means

We will be careful to distinguish between the k-means *algorithm* and the k-means *objective*. As we will see, the k-means algorithm can be understood to be just one optimization algorithm which finds a local optimum of the k-means objective.

Recall that *variance* is a measure of how "spread out" data is, defined as the mean squared distance from the average value of the data.

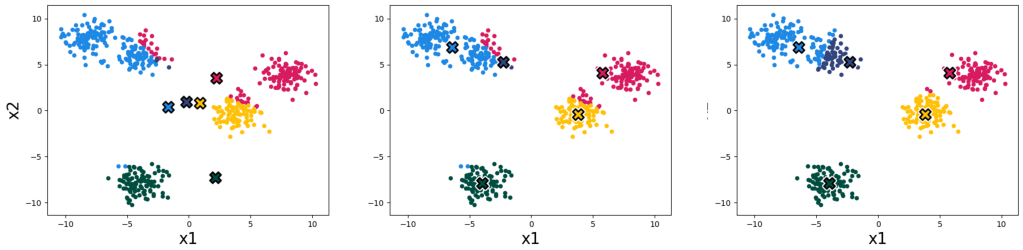
objective can be quantified with the following objective function (which we also call the "k-means objective" or "k-means loss"):

$$\sum_{j=1}^{k} \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j) \left\| x^{(i)} - \mu^{(j)} \right\|^2 , \tag{12.1}$$

where $\mu^{(j)} = \frac{1}{N_j} \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j) x^{(i)}$ and $N_j = \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j)$, so that $\mu^{(j)}$ is the mean of all datapoints in cluster j, and using $\mathbb{1}(\cdot)$ to denote the indicator function (which takes on value of 1 if its argument is true and 0 otherwise). The inner sum (over data points) of the loss is the variance of datapoints within cluster j. We sum up the variance of all k clusters to get our overall loss.

12.1.3 K-means algorithm

The k-means algorithm minimizes this loss by alternating between two steps: given some initial cluster assignments: 1) compute the mean of all data in each cluster and assign this as the "cluster mean", and 2) reassign each datapoint to the cluster with nearest cluster mean. Fig. 12.2 shows what happens when we repeat these steps on the dataset from above.



1) Initial assignments 2) Update means 3) Update assignments

Figure 12.2: The first three steps of running the k-means algorithm on this data. Datapoints are colored according to the cluster to which they are assigned. Cluster means are the larger X's with black outlines.

Each time we reassign the data to the nearest cluster mean, the k-means loss decreases (the datapoints end up closer to their assigned cluster mean), or stays the same. And each time we recompute the cluster means the loss *also* decreases (the means end up closer to their assigned datapoints) or stays the same. Overall then, the clustering gets better and better, according to our objective – until it stops improving. After four iterations of cluster assignment + update means in our example, the k-means algorithm stops improving. Its final solution is shown in Fig. 12.3. It seems to arrive at something reasonable!

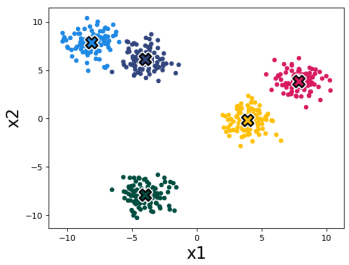


Figure 12.3: K-means loss stops improving; final result.

Now let's write out the algorithm in complete detail:

K-MEANS$(k, \tau, \{x^{(i)}\}_{i=1}^{n})$

```
1  μ, y = random initialization
2  for t = 1 to τ
3      y_old = y
4      for i = 1 to n
5          y^(i) = arg min_j ||x^(i) − μ^(j)||^2
6      for j = 1 to k
7          μ^(j) = (1/N_j) Σ_{i=1}^{n} 𝟙(y^(i) = j) x^(i)
8      if 𝟙(y = y_old)
9          break
10 return μ, y
```

Study Question: Why do we have the "break" statement on line 9? Could the clustering improve if we ran it for more iterations after this point? Has it converged?

The for-loop over the n datapoints assigns each datapoint to the nearest cluster center. The for-loop over the k clusters updates the cluster center to be the mean of all datapoints currently assigned to that cluster. As suggested above, it can be shown that this algorithm reduces the loss in Eq. 12.1 on each iteration, until it converges to a local minimum of the loss.

It's like classification except the algorithm *picked* what the classes are rather than being given examples of what the classes are.

12.1.4 Using gradient descent to minimize k-means objective

We can also use gradient descent to optimize the k-means objective. To show how to apply gradient descent, we first rewrite the objective as a differentiable function *only of* μ:

$$L(\mu) = \sum_{i=1}^{n} \min_j \left\| x^{(i)} - \mu^{(j)} \right\|^2 \quad . \tag{12.2}$$

The k-means algorithm presented above is a form of *block coordinate descent*, rather than gradient descent. For certain problems, and in particular k-means, this method can converge faster than gradient descent.

$L(\mu)$ is the value of the k-means loss given that we pick the *optimal* assignments of the datapoints to cluster means (that's what the $\min_j$ does). Now we can use the gradient $\frac{\partial L(\mu)}{\partial \mu}$ to find the values for μ that achieve minimum loss when cluster assignments are optimal. Finally, we read off the optimal cluster assignments, given the optimized μ, just by assigning datapoints to their nearest cluster mean:

$$y^{(i)} = \arg\min_j \left\| x^{(i)} - \mu^{(j)} \right\|^2 \quad . \tag{12.3}$$

$L(\mu)$ is a smooth function except with kinks where the nearest cluster changes; that means it's differentiable almost everywhere, which in practice is sufficient for us to apply gradient descent.

This procedure yields a local minimum of Eq. 12.1, as does the standard k-means algorithm we presented (though they might arrive at different solutions). It might not be a global optimum since the objective is not convex (due to $\min_j$, as the minimum of multiple convex functions is not necessarily convex).

12.1.5 Importance of initialization

The standard k-means algorithm, as well as the variant that uses gradient descent, both are only guaranteed to converge to a local minimum, not necessarily a global minimum of the loss. Thus the answer we get out depends on how we initialize the cluster means.

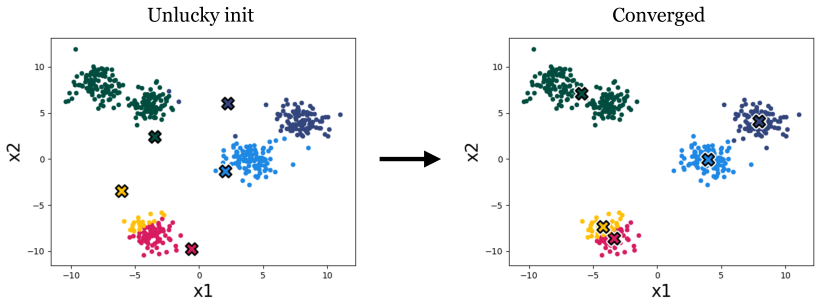


Figure 12.4: With the initialization of the means to the left, the yellow and red means end up splitting what perhaps should be one cluster in half.

Figure 12.4 is an example of a different initialization on our toy data, which results in a worse converged clustering:

A variety of methods have been developed to pick good initializations (for example, check out the *k-means++* algorithm). One simple option is to run the standard k-means algorithm multiple times, with different random initial conditions, and then pick from these the clustering that achieves the lowest k-means loss.

12.1.6 Importance of k

A very important choice in cluster algorithms is the number of clusters we are looking for. Some advanced algorithms can automatically infer a suitable number of clusters, but most of the time, like with k-means, we will have to pick k – it's a hyperparameter of the algorithm.

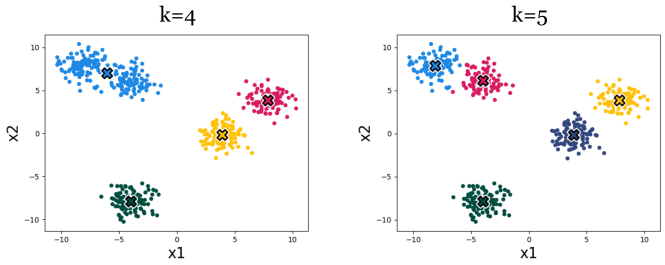


Figure 12.5: Example of k-means run on our toy data, with two different values of k. Setting k=4, on the left, results in one cluster being merged, compared to setting k=5, on the right. Which clustering do you think is better? How could you decide?

Figure 12.5 shows an example of the effect. Which result looks more correct? It can be hard to say! Using higher k we get more clusters, and with more clusters we can achieve lower within-cluster variance – the k-means objective will never increase, and will typically strictly decrease as we increase k. Eventually, we can increase k to equal the total number of datapoints, so that each datapoint is assigned to its own cluster. Then the k-means objective is zero, but the clustering reveals nothing. Clearly, then, we cannot use the k-means objective itself to choose the best value for k. In subsection 12.1.7.1, we will discuss some ways of evaluating the success of clustering beyond its ability to minimize the k-means objective, and it's with these sorts of methods that we might decide on a proper value of k.

Alternatively, you may be wondering: why bother picking a single k? Wouldn't it be nice to reveal a *hierarchy* of clusterings of our data, showing both coarse and fine groupings? Indeed *hierarchical clustering* is another important class of clustering algorithms, beyond k-means. These methods can be useful for discovering tree-like structure in data, and they work a bit like this: initially a coarse split/clustering of the data is applied at the root of the tree, and then as we descend the tree we split and cluster the data in ever more fine-grained ways. A prototypical example of hierarchical clustering is to discover a taxonomy of life, where creatures may be grouped at multiple granularities, from species to families to kingdoms.

12.1.7 k-means in feature space

Clustering algorithms group data based on a notion of *similarity*, and thus we need to define a *distance metric* between datapoints. This notion will also be useful in other machine learning approaches, such as nearest-neighbor methods that we see in Chapter 9. In k-means and other methods, our choice of distance metric can have a big impact on the results we will find.

Our k-means algorithm uses the Euclidean distance, i.e., $\left\|x^{(i)} - \mu^{(j)}\right\|$, with a loss function that is the square of this distance. We can modify k-means to use different distance metrics, but a more common trick is to stick with Euclidean distance but measured in a *feature space*. Just like we did for regression and classification problems, we can define a feature map from the data to a nicer feature representation, $\phi(x)$, and then apply k-means to cluster the data in the feature space.

In fact, using a simple distance metric in feature space can be equivalent to using a more sophisticated distance metric in the data space, and this trick forms the basis of *kernel methods*, which you can learn about in more advanced machine learning classes.

As a simple example, suppose we have two-dimensional data that is very stretched out in the first dimension and has less dynamic range in the second dimension. Then we may want to scale the dimensions so that each has similar dynamic range, prior to clustering. We could use standardization, like we did in Chapter 5.

If we want to cluster more complex data, like images, music, chemical compounds, etc., then we will usually need more sophisticated feature representations. One common practice these days is to use feature representations learned with a neural network. For example, we can use an autoencoder to compress images into feature vectors, then cluster those feature vectors.

12.1.7.1 How to evaluate clustering algorithms

One of the hardest aspects of clustering is knowing how to evaluate it. This is actually a big issue for all unsupervised learning methods, since we are just looking for patterns in the data, rather than explicitly trying to predict target values (which was the case with supervised learning).

Remember, evaluation metrics are *not* the same as loss functions, so we can't just measure success by looking at the k-means loss. In prediction problems, it is critical that the evaluation is on a held-out test set, while the loss is computed over training data. If we evaluate on training data we cannot detect overfitting. Something similar is going on with the example in Section 12.1.6, where setting k to be too large can precisely "fit" the data (minimize the loss), but yields no general insight.

One way to evaluate our clusters is to look at the **consistency** with which they are found when we run on different subsamples of our training data, or with different hyperparameters of our clustering algorithm (e.g., initializations). For example, if running on several bootstrapped samples (random subsets of our data) results in very different clusters, it should call into question the validity of any of the individual results.

If we have some notion of what **ground truth** clusters should be, e.g., a few data points that we know should be in the same cluster, then we can measure whether or not our

discovered clusters group these examples correctly.

Clustering is often used for **visualization** and **interpretability**, to make it easier for humans to understand the data. Here, human judgment may guide the choice of clustering algorithm. More quantitatively, discovered clusters may be used as input to **downstream tasks**. For example, we may fit a different regression function on the data within each cluster. Figure 12.6 gives an example where this might be useful. In cases like this, the success of a clustering algorithm can be indirectly measured based on the success of the downstream application (e.g., does it make the downstream predictions more accurate).

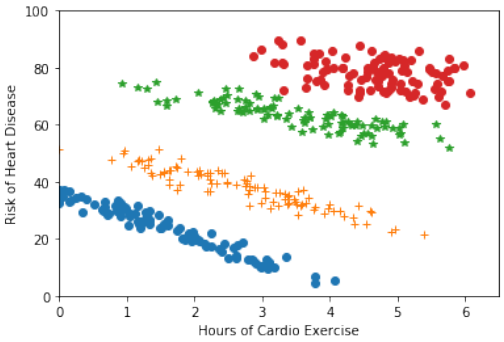


Figure 12.6: Averaged across the whole population, risk of heart disease positively correlates with hours of exercise. However, if we cluster the data, we can observe that there are four subgroups of the population which correspond to different age groups, and within each subgroup the correlation is negative. We can make better predictions, and better capture the presumed true effect, if we cluster this data and then model the trend in each cluster separately.

12.2 Autoencoder structure

Assume that we have input data $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$, where $x^{(i)} \in \mathbb{R}^d$. We seek to learn an autoencoder that will output a new dataset $\mathcal{D}_{out} = \{a^{(1)}, \ldots, a^{(n)}\}$, where $a^{(i)} \in \mathbb{R}^k$ with $k < d$. We can think about $a^{(i)}$ as the new *representation* of data point $x^{(i)}$. For example, in Fig. 12.7 we show the learned representations of a dataset of MNIST digits with $k = 2$. We see, after inspecting the individual data points, that unsupervised learning has found a compressed (or *latent*) representation where images of the same digit are close to each other, potentially greatly aiding subsequent clustering or classification tasks.

Formally, an autoencoder consists of two functions, a vector-valued *encoder* $g : \mathbb{R}^d \to \mathbb{R}^k$ that deterministically maps the data to the representation space $a \in \mathbb{R}^k$, and a *decoder* $h : \mathbb{R}^k \to \mathbb{R}^d$ that maps the representation space back into the original data space.

In general, the encoder and decoder functions might be any functions appropriate to the domain. Here, we are particularly interested in neural network embodiments of encoders and decoders. The basic architecture of one such autoencoder, consisting of only a single layer neural network in each of the encoder and decoder, is shown in Figure 12.8; note that bias terms W_0^1 and W_0^2 into the summation nodes exist, but are omitted for clarity in the figure. In this example, the original d-dimensional input is compressed into $k = 3$ dimensions via the encoder $g(x; W^1, W_0^1) = f_1({W^1}^T x + W_0^1)$ with $W^1 \in \mathbb{R}^{d \times k}$ and $W_0^1 \in \mathbb{R}^k$, and where the non-linearity f_1 is applied to each dimension of the vector. To recover (an approximation to) the original instance, we then apply the decoder $h(a; W^2, W_0^2) = f_2({W^2}^T a + W_0^2)$, where f_2 denotes a different non-linearity (activation function). In general, both the de-

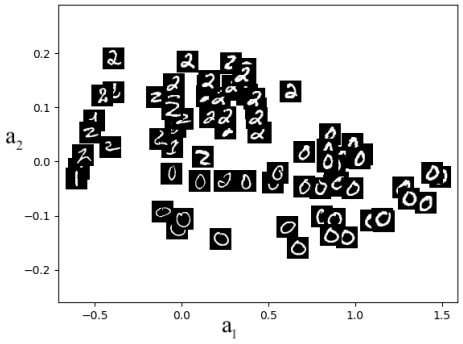


Figure 12.7: Compression of digits dataset into two dimensions. The input $x^{(i)}$, an image of a handwritten digit, is shown at the new low-dimensional representation (a_1, a_2).

coder and the encoder could involve multiple layers, as opposed to the single layer shown here. Learning seeks parameters W^1, W_0^1 and W^2, W_0^2 such that the reconstructed instances, $h(g(x^{(i)}; W^1, W_0^1); W^2, W_0^2)$, are close to the original input $x^{(i)}$.

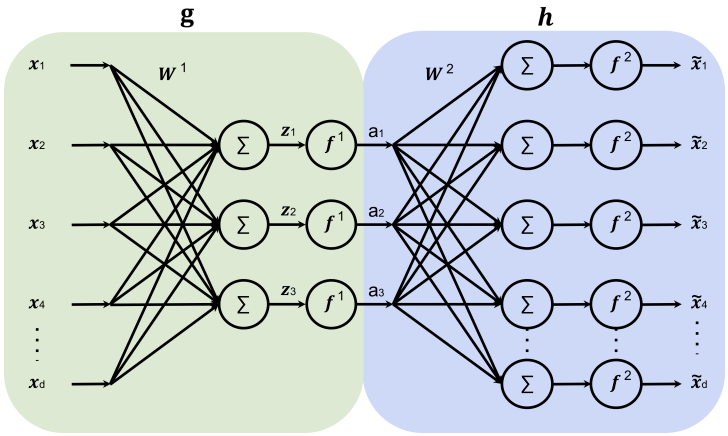


Figure 12.8: Autoencoder structure, showing the encoder (left half, light green), and the decoder (right half, light blue), encoding inputs x to the representation a, and decoding the representation to produce $\tilde{x}$, the reconstruction. In this specific example, the representation (a_1, a_2, a_3) only has three dimensions.

12.2.1 Autoencoder Learning

We learn the weights in an autoencoder using the same tools that we previously used for supervised learning, namely (stochastic) gradient descent of a multi-layer neural network to minimize a loss function. All that remains is to specify the loss function $\mathcal{L}(\tilde{x}, x)$, which tells us how to measure the discrepancy between the reconstruction $\tilde{x} = h(g(x; W^1, W_0^1); W^2, W_0^2)$ and the original input x. For example, for continuous-valued x it might make sense to use squared loss, i.e., $\mathcal{L}_{SE}(\tilde{x}, x) = \sum_{j=1}^{d} (x_j - \tilde{x}_j)^2$. Learning then seeks to optimize the parameters of h and g so as to minimize the reconstruction error, measured according to this loss function:

$$\min_{W^1, W_0^1, W^2, W_0^2} \sum_{i=1}^{n} \mathcal{L}_{SE}\left(h(g(x^{(i)}; W^1, W_0^1); W^2, W_0^2), x^{(i)}\right)$$

Alternatively, you could think of this as *multi-task learning*, where the goal is to predict each dimension of x. One can mix-and-match loss functions as appropriate for each dimension's data type.

12.2.2 Evaluating an autoencoder

What makes a good learned representation in an autoencoder? Notice that, without further constraints, it is always possible to perfectly reconstruct the input. For example, we could let $k = d$ and h and g be the identity functions. In this case, we would not obtain any compression of the data.

To learn something useful, we must create a *bottleneck* by making k to be smaller (often much smaller) than d. This forces the learning algorithm to seek transformations that describe the original data using as simple a description as possible. Thinking back to the digits dataset, for example, an example of a compressed representation might be the digit label (i.e., 0–9), rotation, and stroke thickness. Of course, there is no guarantee that the learning algorithm will discover precisely this representation. After learning, we can inspect the learned representations, such as by artificially increasing or decreasing one of the dimensions (e.g., a_1) and seeing how it affects the output $h(a)$, to try to better understand what it has learned.

As with clustering, autoencoders can be a preliminary step toward building other models, such as a regressor or classifier. For example, once a good encoder has been learned, the decoder might be replaced with another neural network that is then trained with supervised learning (perhaps using a smaller dataset that does include labels).

12.2.3 Linear encoders and decoders

We close by mentioning that even linear encoders and decoders can be very powerful. In this case, rather than minimizing the above objective with gradient descent, a technique called *principal components analysis* (PCA) can be used to obtain a closed-form solution to the optimization problem using a singular value decomposition (SVD). Just as a multilayer neural network with nonlinear activations for regression (learned by gradient descent) can be thought of as a nonlinear generalization of a linear regressor (fit by matrix algebraic operations), the neural network based autoencoders discussed above (and learned with gradient descent) can be thought of as a generalization of linear PCA (as solved with matrix algebra by SVD).

12.2.4 Advanced encoders and decoders

Advanced neural networks that build on the encoder-decoder conceptual decomposition have become increasingly powerful in recent years. One family of applications are *generative* networks, where new outputs that are "similar to" but different from any existing training sample are desired. In *variational autoencoders* the compressed representation encompasses information about the probability distribution of training samples, e.g., learning both mean and standard deviation variables in the bottleneck layer or latent representation. Then, new outputs can be generated by random sampling based on the latent representation variables and feeding those samples into the decoder. For instance, *Transformers* use *multiple* encoder and decoder blocks, together with a self-attention mechanism to make predictions about potential next outputs resulting from sequences of inputs. Such transformer networks have many applications in natural language processing and elsewhere.