Chapter 9

Representations for machine learning

At first, it might seem that the applicability of linear regression and classification to real-life problems is greatly limited. After all, it is not clear whether it is realistic (most of the time) to assume that the target variable is a linear combination of features. Fortunately, the applicability of linear regression is broader than originally thought. The main idea is to apply a non-linear transformation to the data matrix $\mathbf{x}$ prior to the fitting step, which then enables a non-linear fit. Obtaining such a useful feature representation is a central problem in machine learning.

We will first examine fixed representations for linear regression: polynomial curve fitting and radial basis function (RBF) networks. Then, we will discuss learning representations.

9.1 Radial basis function networks and kernel representations

The idea of radial basis function (RBF) networks is a natural generalization of the polynomial curve fitting and approaches from the previous Section. Given data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, we start by picking p points to serve as the "centers" in the input space $\mathcal{X}$. We denote those centers as $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p$. Usually, these can be selected from $\mathcal{D}$ or computed using some clustering technique (e.g. the EM algorithm, K-means).

When the clusters are determined using a Gaussian mixture model, the basis functions can be selected as

$$\phi_j(\mathbf{x}) \quad = \quad e^{-\frac{1}{2}(\mathbf{x}-\mathbf{c}_j)^T \Sigma_j^{-1} (\mathbf{x}-\mathbf{c}_j)},$$

where the cluster centers and the covariance matrix are found during clustering. When K-means or other clustering is used, we can use

$$\phi_j(\mathbf{x}) = e^{-\frac{\left\|\mathbf{x}-\mathbf{c}_j\right\|^2}{2\sigma_j^2}},$$

where σ_j's can be separately optimized; e.g. using a validation set. In the context of multidimensional transformations from $\mathbf{x}$ to $\mathbf{\Phi}$, the basis functions can also be referred to as *kernel functions*, i.e. $\phi_j(\mathbf{x}) = k_j(\mathbf{x}, \mathbf{c}_j)$. Matrix

$$\mathbf{\Phi} = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_p(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & & \\ \vdots & & \ddots & \\ \phi_0(\mathbf{x}_n) & & & \phi_p(\mathbf{x}_n) \end{bmatrix}$$

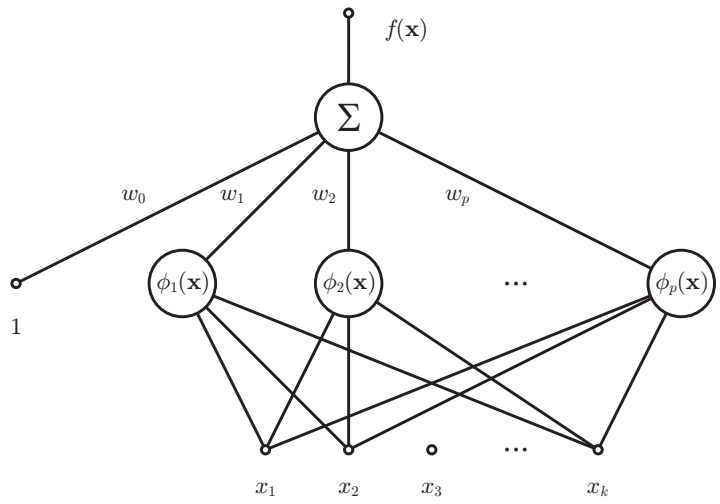

Figure 9.1: Radial basis function network.

is now used as a new data matrix. For a given input $\mathbf{x}$, the prediction of the target y will be calculated as

$$\begin{aligned} f(\mathbf{x}) &= w_0 + \sum_{j=1}^{p} w_j \phi_j(\mathbf{x}) \\ &= \sum_{j=0}^{p} w_j \phi_j(\mathbf{x}) \end{aligned}$$

where $\phi_0(\mathbf{x}) = 1$ and $\mathbf{w}$ is to be found. It can be proved that with a sufficiently large number of radial basis functions we can accurately approximate any function. As seen in Figure 9.1, we can think of RBFs as neural networks.

RBF networks and kernel representations are highly related. The main distinction is that kernel representations use any kernel function for the similarity measure $k(\mathbf{x}, \mathbf{c}_j) = \phi_j(\mathbf{x})$, where radial basis functions are one example of a kernel. In addition, if an RBF kernel is chosen, for kernel representations typical the centers are selected from the training dataset. For RBF networks, the selection of the centers is left generally as an important step, where they can be selected from the training set but can also be selected in other ways.

9.2 Learning representations

There are many approaches to learning representations. Two dominant approaches are (semi-supervised) matrix factorization techniques and neural networks. Neural networks build on the generalized linear models we have discussed, stacking multiple generalized linear models together. Matrix factorization techniques (e.g., dimensionality reduction, sparse coding) typically factorize the input data into a dictionary and a new representation (a basis). We will first discuss neural networks, and then discuss the many unsupervised and semisupervised learning techniques that are encompassed by matrix factorizations.

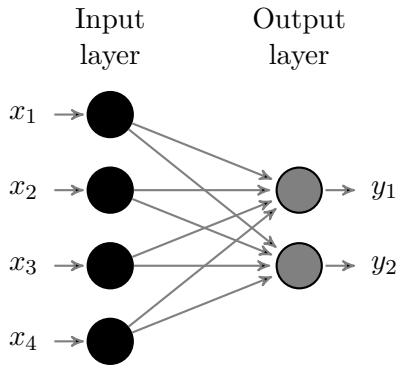

Figure 9.2: Generalized linear model, such as logistic regression.

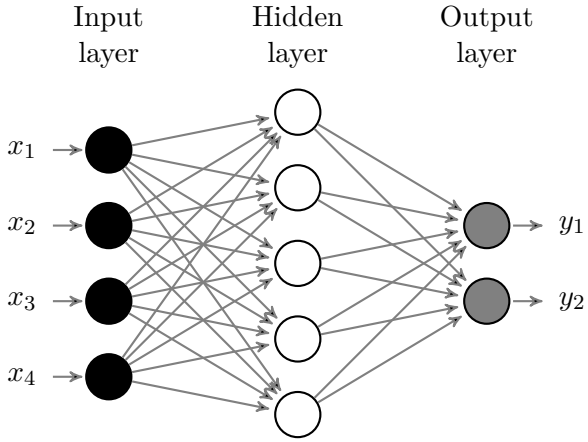

Figure 9.3: Standard two-layer neural network.

9.2.1 Neural networks

Neural networks are a form of supervised representation learning. As before, the goal is to learn a function of inputs, f, to produce a prediction of the target: $f(\mathbf{x})$. The addition of hidden layers, with non-linear activation functions, enables learning of nonlinear functions f. For some intuition, one can consider that all the first hidden layers constitute representation layer, with learning on the last layer corresponding to supervised prediction part. Figure 9.2 shows the graphical model for the generalized linear models we discussed in the previous chapters, where the weights and corresponding transfer can be thought of as being on the arrows (as they are not random variables). Figure 9.3 shows a neural network with one hidden-layer; this is called a two-layer neural network, as there are two layers of weights.

In the figure, the neural network inputs a 4-dimensional feature vector $\mathbf{x} = [x_1,\ x_2,\ x_3,\ x_4]$ (i.e., $d = 4$) and outputs a 2-dimensional prediction $\mathbf{y} = [y_1,\ y_2]$ (i.e., $m = 2$). The hidden layer consists of a mapping from $\mathbf{x}$ to a new representation that is 5-dimensional (i.e., $k_1 = 5$ as per the notation below). For the neural network, let each node in this hidden representation be indexed by $k \in \{1, \ldots, 5\}$. Each h_k consists of a transformation of a linear weighting of $\mathbf{x}$, such as a sigmoid transfer: $h_k = \sigma\left(\sum_{j=1}^{d} x_j w_{kj}\right) = \sigma\left(\mathbf{x}\mathbf{w}_k\right)$ where $\mathbf{w}_k \in \mathbb{R}^d$ is the weights on the first layer used to produce the kth node in the hidden representation.

Example 18: For a simple example, consider $d = 1$ (i.e., one input observation), $m = 1$ (i.e., one output), $k_1 = 2$ (i.e., 2-dimensional hidden layer) and a sigmoid transfer to get the first hidden layer. Assume we are given one instance (x, y). Then input observation x is transformed into

$$\mathbf{h} = [h_1,\ h_2], \qquad \text{with } h_1 = \sigma(x w_1^{(2)}) \text{ and } h_2 = \sigma(x w_2^{(2)}) \qquad \text{for } w_1^{(2)}, w_2^{(2)} \in \mathbb{R}.$$

To avoid transpose notation, we used $\mathbf{x} \in \mathbb{R}^{1\times d}$ to give one row of the data matrix $\mathbf{X} \in \mathbb{R}^{n\times d}$, and row vector $\mathbf{h} \in \mathbb{R}^{1\times k_1}$. We use the superscript notation to distinguish between the weights in the first and last layer. It may seem counter-intuitive why we label $\mathbf{w}^{(2)}$ for the input layer, and $\mathbf{w}^{(1)}$ for the output layer, but you will see below it makes notation simpler to start indexing from the output layer.

Once we have $\mathbf{h}$, we can pretend that $\mathbf{h}$ is the new input representation and go ahead and learn a (generalized) linear model on this last layer. Let's consider two cases: $y \in \mathbb{R}$

and $y \in \{0, 1\}$. If $y \in \mathbb{R}$, we use linear regression for this last layer and so learn weights $\mathbf{w}^{(2)} \in \mathbb{R}^2$ such that $\mathbf{h}\mathbf{w}^{(2)}$ approximates the true output y. If $y \in \{0, 1\}$, we use logistic regression for this last layer and so learn weights $\mathbf{w}^{(2)} \in \mathbb{R}^2$ such that $\sigma(\mathbf{h}\mathbf{w}^{(2)})$ approximates the true output y. □

Now we consider the more general case with any d, k_1, m. To provide some intuition for this more general setting, we will begin with one hidden layer, for the sigmoid transfer function and cross-entropy output loss. For logistic regression we estimated $\mathbf{W} \in \mathbb{R}^{d \times m}$, with $f(\mathbf{x}\mathbf{W}) = \sigma(\mathbf{x}\mathbf{W}) \approx \mathbf{y}$. We will predict an output vector $\mathbf{y} \in \mathbb{R}^m$, because it will make later generalizations more clear-cut and make notation for the weights in each layer more uniform. When we add a hidden layer, we have two parameter matrices $\mathbf{W}^{(2)} \in \mathbb{R}^{d \times k_1}$ and $\mathbf{W}^{(1)} \in \mathbb{R}^{k_1 \times m}$, where k_1 is the dimension of the hidden layer

$$\mathbf{h} = \sigma(\mathbf{x}\mathbf{W}^{(2)}) = \begin{bmatrix} \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:1}) \\ \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:2}) \\ \vdots \\ \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:k_1}) \end{bmatrix} \in \mathbb{R}^{k_1}$$

where the sigmoid function is applied to each entry in $\mathbf{x}\mathbf{W}^{(2)}$ and $\mathbf{h}\mathbf{W}^{(1)}$. This hidden layer is the new set of features and again you will do the regular logistic regression optimization to learn weights on $\mathbf{h}$:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{h}\mathbf{W}^{(1)}) = \sigma(\sigma(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}).$$

With the probabilistic model and parameter specified, we now need to derive an algorithm to obtain those parameters. As before, we take a maximum likelihood approach and derive gradient descent updates. This composition of transfers seems to complicate matters, but we can still take the gradient w.r.t. our parameters. We simply have more parameters now: $\mathbf{W}^{(2)} \in \mathbb{R}^{k_1 \times d}, \mathbf{W}^{(1)} \in \mathbb{R}^{1 \times k_1}$. Once we have the gradient w.r.t. each parameter matrix, we simply take a step in the direction of the negative of the gradient, as usual. The gradients for these parameters share information; for computational efficiency, the gradient is computed first for $\mathbf{W}^{(1)}$, and duplicate gradient information sent back to compute the gradient for $\mathbf{W}^{(2)}$. This algorithm is typically called *back propagation*, which we describe next.

In general, we can compute the gradient for any number of hidden layers. Denote each differentiable transfer function $f_1, \ldots, f_H$, ordered with f_1 as the output transfer, and $k_1, \ldots, k_{H-1}$ as the hidden dimensions with $H - 1$ hidden layers. Then the output from the neural network is

$$f_1\left(f_2\left(\ldots f_{H-1}\left(f_H\left(\mathbf{x}\mathbf{W}^{(H)}\right)\mathbf{W}^{(H-1)}\right)\ldots\right)\mathbf{W}^{(1)}\right)$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{k_1 \times m}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{k_2 \times k_1}$, $\ldots, \mathbf{W}^{(H)} \in \mathbb{R}^{d \times k_{H-1}}$.

Backpropagation algorithm

We will start by deriving back propagation for two layers; the extension to multiple layers will be more clear given this derivation. Due to the size of the network, we will often learn

with stochastic gradient descent. Therefore, we will first compute this gradient assuming we only have one sample $(\mathbf{x}, \mathbf{y})$.

The back-propagation algorithm is simply gradient descent on a non-convex objective, with a careful ordering of computation to avoid repeating computation. In particular, one first propagates forward and computes variable $\mathbf{h} = f_2(\mathbf{x}\mathbf{W}^{(2)}) \in \mathbb{R}^{1\times k}$ and then $\hat{\mathbf{y}} = f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}) = f_1(\mathbf{h}\mathbf{W}^{(1)})$. We then compute the error between our prediction $\hat{\mathbf{y}}$ and the true label. We take the gradient of this error (loss) w.r.t. to our parameters; in this case, for efficient computation, the best ordering is to compute the gradient w.r.t. to the last parameter $\mathbf{W}^{(1)}$ first, and then $\mathbf{W}^{(2)}$. This is the reason for the term back-propagation, since the error is propagated backward from the last layer first.

The choices then involve picking the transfers at each layer, the number of hidden nodes and the loss for the last layer. The matching convex loss $L(\cdot, y)$ depends on the chosen $p(y|\mathbf{x})$ and corresponding transfer function for the last layer of the neural network, just as with generalized linear models. For ease of notation, we define this error function as

$$\text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}) = \sum_{k=1}^{m} L(f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}_{:k}), \mathbf{y}_k)$$

for one sample $(\mathbf{x}, \mathbf{y})$. For example, for $p(y = 1|\mathbf{x})$ Gaussian and identity transfer f_2, we get $\text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}) = (f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)} - y)^2$. If $p(y = 1|\mathbf{x})$ is a Bernoulli distribution, then we would chose the logistic regression loss (the cross entropy).

As before, we will compute gradients of the loss w.r.t. our parameters. First, we take the partial derivative w.r.t. the parameters $\mathbf{W}^{(1)}$ (assuming $\mathbf{W}^{(2)}$ is fixed).

$$\begin{aligned}
\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(1)}_{jk}} &= \frac{\partial L(f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}), \mathbf{y})}{\partial \mathbf{W}^{(1)}_{jk}} \\
&= \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{W}^{(1)}_{jk}} \qquad \triangleright\ \hat{\mathbf{y}}_k = f_1(\mathbf{h}\mathbf{W}^{(1)}_{:k})
\end{aligned}$$

where only $\hat{\mathbf{y}}_k$ is affected by $\mathbf{W}^{(1)}_{jk}$ in the loss, and so the gradient for the others is zero. Continuing,

$$\begin{aligned}
\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(1)}_{jk}} &= \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \frac{\partial \boldsymbol{\theta}^{(1)}_k}{\partial \mathbf{W}^{(1)}_{jk}} \qquad \triangleright\ \boldsymbol{\theta}^{(1)}_k = \mathbf{h}\mathbf{W}^{(1)}_{:k} \\
&= \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \mathbf{h}_j
\end{aligned}$$

At this point these equations are abstract; but they are simple to compute for the losses and transfers we have examined. For example, for $L(\hat{\mathbf{y}}_k, \mathbf{y}_k) = \frac{1}{2}(\hat{\mathbf{y}}_k - \mathbf{y}_k)^2$, and f_2 the identity, we get

$$\begin{aligned}
\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} &= (\hat{\mathbf{y}}_k - \mathbf{y}_k) \\
\frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} &= 1
\end{aligned}$$

giving

$$\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(1)}_{jk}} = \left(\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k}\right) \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \mathbf{h}_j = (\hat{\mathbf{y}}_k - \mathbf{y}_k)\mathbf{h}_j.$$

The gradient update is as usual with $\mathbf{W}^{(1)} = \mathbf{W}^{(1)} - \alpha(\hat{\mathbf{y}} - \mathbf{y})\mathbf{h}^\top$ for some step-size α.

Next, we compute the partial gradient with respect to $\mathbf{W}^{(2)}$. Now, however, the entire output variable $\mathbf{y} \in \mathbb{R}^{1\times m}$ is affected by the choice of $\mathbf{W}^{(2)}_{ij}$ for all $i \in \{1, \ldots, k_2\}$, $j \in \{1, \ldots, k_1\}$. Therefore, we need to take the partial derivative w.r.t. all of $\mathbf{y}$.

$$\begin{aligned}
\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(2)}_{ij}} &= \frac{\partial \sum_{k=1}^m L(f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)}_{:k}), \mathbf{y}_k)}{\partial \mathbf{W}^{(2)}_{ij}} \\
&= \sum_{k=1}^m \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{W}^{(2)}_{ij}} && \triangleright\ \hat{\mathbf{y}}_k = f_1(\mathbf{h}\mathbf{W}^{(1)}_{:k}) = f_1(\boldsymbol{\theta}^{(1)}_k) \\
&= \sum_{k=1}^m \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \frac{\partial \boldsymbol{\theta}^{(1)}_k}{\partial \mathbf{W}^{(2)}_{ij}}.
\end{aligned}$$

Continuing,

$$\begin{aligned}
\frac{\partial \boldsymbol{\theta}^{(1)}_k}{\partial \mathbf{W}^{(2)}_{ij}} = \frac{\partial \mathbf{h}\mathbf{W}^{(1)}_{:k}}{\partial \mathbf{W}^{(2)}_{ij}} &= \frac{\partial \sum_{l=1}^k \mathbf{h}_l \mathbf{W}^{(1)}_{lk}}{\partial \mathbf{W}^{(2)}_{ij}} \\
&= \frac{\partial \sum_{l=1}^k f_2(\mathbf{x}\mathbf{W}^{(2)}_{:l})\mathbf{W}^{(1)}_{lk}}{\partial \mathbf{W}^{(2)}_{ij}} \\
&= \sum_{l=1}^k \mathbf{W}^{(1)}_{lk} \frac{\partial f_2(\mathbf{x}\mathbf{W}^{(2)}_{:l})}{\partial \mathbf{W}^{(2)}_{ij}} \\
&= \mathbf{W}^{(1)}_{jk} \frac{\partial f_2(\mathbf{x}\mathbf{W}^{(2)}_{:j})}{\partial \mathbf{W}^{(2)}_{ij}}
\end{aligned}$$

because $\frac{\partial f_2(\mathbf{x}\mathbf{W}^{(2)}_{:l})}{\partial \mathbf{W}^{(2)}_{ij}} = 0$ for $l \neq j$. Now continuing the chain rule

$$\begin{aligned}
\frac{\partial f_2(\mathbf{x}\mathbf{W}^{(2)}_{:j})}{\partial \mathbf{W}^{(2)}_{ij}} &= \frac{\partial f_2(\boldsymbol{\theta}^{(2)}_j)}{\partial \boldsymbol{\theta}^{(2)}_j} \frac{\partial \boldsymbol{\theta}^{(2)}_j}{\partial \mathbf{W}^{(2)}_{ij}} && \triangleright\ \boldsymbol{\theta}^{(2)}_j = \mathbf{x}\mathbf{W}^{(2)}_{:j} \\
&= \frac{\partial f_2(\boldsymbol{\theta}^{(2)}_j)}{\partial \boldsymbol{\theta}^{(2)}_j} \mathbf{x}_i.
\end{aligned}$$

Putting this back together, we get

$$\begin{aligned}
\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}^{(2)}_{ij}} &= \sum_{k=1}^m \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \frac{\partial \boldsymbol{\theta}^{(1)}_k}{\partial \mathbf{W}^{(2)}_{ij}} \\
&= \sum_{k=1}^m \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}^{(1)}_k)}{\partial \boldsymbol{\theta}^{(1)}_k} \mathbf{W}^{(1)}_{jk} \frac{\partial f_2(\boldsymbol{\theta}^{(2)}_j)}{\partial \boldsymbol{\theta}^{(2)}_j} \mathbf{x}_i.
\end{aligned}$$

Notice that some of gradient is the same as for $\mathbf{W}^{(1)}$, i.e.

$$\boldsymbol{\delta}_k^{(1)} = \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}}$$

Computing these components only needs to be done once for $\mathbf{W}^{(1)}$, and this information propagated back to get the gradient for $\mathbf{W}^{(2)}$. The difference is in the gradient $\frac{\partial \boldsymbol{\theta}^{(1)}}{\partial \mathbf{W}^{(2)}}$, because $\mathbf{h}$ relies on $\mathbf{W}^{(2)}$. For $\mathbf{W}^{(1)}$, $\mathbf{h} = f_2(\mathbf{x}_i \mathbf{W}^{(2)})$ is a constant, and so does not affect the gradient for $\mathbf{W}^{(1)}$. The final gradient is

$$\begin{aligned}
\frac{\partial \text{Err}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)})}{\partial \mathbf{W}_{ij}^{(2)}} &= \left(\sum_{k=1}^{m} \boldsymbol{\delta}_k^{(1)} \mathbf{W}_{jk}^{(1)} \right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \mathbf{x}_i \\
&= \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)} \right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}} \mathbf{x}_i
\end{aligned}$$

If another layer is added before $\mathbf{W}^{(2)}$, then the information propagated backward is

$$\boldsymbol{\delta}_j^{(2)} = \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)} \right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}}$$

and $\mathbf{x}_i$ is replaced with $\mathbf{h}_i^{(2)}$. The gradient for $\mathbf{W}_{ij}^{(3)}$ is

$$\left(\mathbf{W}_{j:}^{(2)} \boldsymbol{\delta}^{(2)} \right) \frac{\partial f_3(\boldsymbol{\theta}_j^{(3)})}{\partial \boldsymbol{\theta}_j^{(3)}} \mathbf{x}_i$$

Example 19: Let $p(y = 1|\mathbf{x})$ be a Bernoulli distribution, with f_1 and f_2 both sigmoid functions. The loss is the cross-entropy. We can derive the two-layer update rule with these settings, by plugging-in above.

$$\begin{aligned}
L(\hat{y}, y) &= -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}) && \triangleright \text{ cross-entropy} \\
\frac{\partial L(\hat{y}, y)}{\partial \hat{y}} &= -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} \\
f_2(\mathbf{x}\mathbf{W}_{:j}^{(2)}) &= \sigma(\mathbf{x}\mathbf{W}_{:j}^{(2)}) = \frac{1}{1 + \exp(-\mathbf{x}\mathbf{W}_{:j}^{(2)})} \\
f_1(\mathbf{h}\mathbf{W}_{:k}^{(1)}) &= \sigma(\mathbf{h}\mathbf{W}_{:k}^{(1)}) = \frac{1}{1 + \exp(-\mathbf{h}\mathbf{W}_{:k}^{(1)})} \\
\partial \sigma(\theta) &= \sigma(\theta)(1 - \sigma(\theta))
\end{aligned}$$

Now we can compute the backpropagation update by first propagating forward

$$\begin{aligned}
\mathbf{h} &= \sigma(\mathbf{x}\mathbf{W}^{(2)}) \\
\hat{\mathbf{y}} &= \sigma(\mathbf{h}\mathbf{W}^{(1)})
\end{aligned}$$

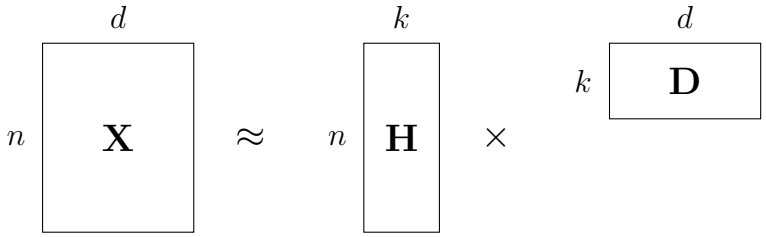

Figure 9.4: Matrix factorization of data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$.

and then propagating the gradient back

$$\begin{aligned}
\boldsymbol{\delta}_k^{(1)} &= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_1(\boldsymbol{\theta}_k^{(1)})}{\partial \boldsymbol{\theta}_k^{(1)}} \\
&= \left(-\frac{\mathbf{y}_k}{\hat{\mathbf{y}}_k} + \frac{1 - \mathbf{y}_k}{1 - \hat{\mathbf{y}}_k} \right) \hat{\mathbf{y}}_k (1 - \hat{\mathbf{y}}_k) \\
&= -\mathbf{y}_k (1 - \hat{\mathbf{y}}_k) + (1 - \mathbf{y}_k) \hat{\mathbf{y}}_k \\
&= \hat{\mathbf{y}}_k - \mathbf{y}_k \\
\frac{\partial}{\partial \mathbf{W}_{jk}^{(1)}} &= \boldsymbol{\delta}_k^{(1)} \mathbf{h}_j \\
\boldsymbol{\delta}_j^{(2)} &= \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)} \right) \mathbf{h}_j (1 - \mathbf{h}_j) \\
\frac{\partial}{\partial \mathbf{W}_{ij}^{(2)}} &= \boldsymbol{\delta}_j^{(2)} \mathbf{x}_i
\end{aligned}$$

The update simply consists of stepping in the direction of these gradients, as is usual for gradient descent. We start with some initial $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ (say filled with random values), and then apply the gradient descent rules with these gradients. □

9.2.2 Unsupervised learning and matrix factorization

Another strategy to obtaining a new representation is through matrix factorization. The data matrix $\mathbf{X}$ is factorized into a dictionary $\mathbf{D}$ and a basis or new representation $\mathbf{H}$ (see Figure 9.4). In fact, many unsupervised learning algorithms (e.g., dimensionality reduction, sparse coding) and semi-supervised learning algorithms (e.g., supervised dictionary learning) can actually be formulated as matrix factorizations. We will look at k-means clustering and principal components analysis as an example. The remaining algorithms are simply summarized in the below table. This general approach to obtaining a new representation using factorization is called **dictionary learning**.

K-means clustering is an unsupervised learning problem to group data points into k clusters by minimizing distances to the mean of each cluster. This problem is not usually thought of as a representation learning approach, because the cluster number is not typically used as a representation. However, we nonetheless start with k-means because it is an intuitive example of how these unsupervised learning algorithms can be thought of as matrix

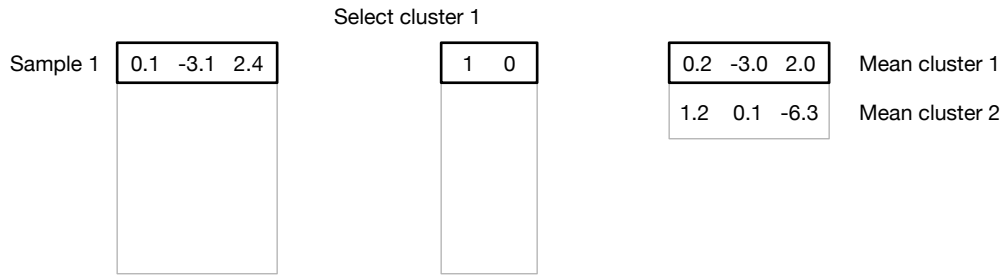

Figure 9.5: K-means clustering as a matrix factorization for data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$.

factorization. Further, the clustering approach can be seen as a representation learning approach, because it is a learned discretization of the space. We will discuss this view of k-means after discussing it as a matrix factorization.

Imagine that you have two clusters ($k = 2$), with data dimension $d = 3$. Let $\mathbf{d}_1$ be the mean for cluster 1 and $\mathbf{d}_2$ the mean for cluster 2. The goal is to minimize the squared ℓ_2 distance of each data point $\mathbf{x}$ to its cluster center

$$\|\mathbf{x} - \sum_{i=1}^{2} 1\left(\mathbf{x} \text{ in cluster } i\right) \mathbf{d}_i\|_2^2 = \|\mathbf{x} - \mathbf{h}\mathbf{D}\|_2^2$$

where $\mathbf{h} = [1 \quad 0]$ or $\mathbf{h} = [0 \quad 1]$ and $\mathbf{D} = [\mathbf{d}_1 \; ; \; \mathbf{d}_2]$. An example is depicted in Figure 9.5. For a point $\mathbf{x} = [0.1 \quad -3.1 \quad 2.4]$, $\mathbf{h} = [1 \quad 0]$, meaning it is placed in cluster 1 with mean $\mathbf{d}_1 = [0.2 \quad -3.0 \quad 2.0]$. It would incur more error to place $\mathbf{x}$ in cluster 2 which has a mean that is more dissimilar: $\mathbf{d}_2 = [1.2 \quad 0.1 \quad -6.3]$.

The overall minimization is defined across all the samples, giving loss

$$\min_{\substack{\mathbf{H} \in \{0,1\}^{n \times k}, \mathbf{1}\mathbf{H} = \mathbf{1} \\ \mathbf{D} \in \mathbb{R}^{k \times d}}} \|\mathbf{X} - \mathbf{H}\mathbf{D}\|_F^2.$$

Different clusters vectors $\mathbf{h}$ are learned for each $\mathbf{x}$, but the dictionary of means is shared amongst all the data points. The specified optimization should pick dictionary $\mathbf{D}$ of means that provides the smallest distances to points in the training dataset.

Principal components analysis (PCA) is a standard dimensionality reduction technique, where the input data $\mathbf{x} \in \mathbb{R}^{1 \times d}$ is projected into a lower dimensional $\mathbf{h} \in \mathbb{R}^{1 \times k}$ spanned by the space of principal components. These principal components are the directions of maximal variance in the data. To obtain these k principal components $\mathbf{D} \in \mathbb{R}^{k \times d}$, the common solution technique is to obtain the singular value decomposition of the data matrix $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \in \mathbb{R}^{n \times d}$, giving

$$\mathbf{D} = \mathbf{V}_k^\top \in \mathbb{R}^{k \times d}$$
$$\mathbf{H} = \mathbf{U}_k \mathbf{\Sigma}_k \in \mathbb{R}^{n \times k}$$

where $\mathbf{\Sigma}_k \in \mathbb{R}^{k \times k}$ consists of the top largest k singular values (in descending order) and $\mathbf{U}_k \in \mathbb{R}^{n \times k}$ and $\mathbf{V}_k \in \mathbb{R}^{k \times d}$ are the corresponding singular vectors, i.e., $\mathbf{U}_k = \mathbf{U}_{:,1:k}$ and $\mathbf{V}_k = \mathbf{V}_{:,1:k}$. The new representation for $\mathbf{X}$ (using PCA) is this $\mathbf{H}$. Note that PCA does

not subselect features, but rather creates new features: The generated $\mathbf{h}$ is not a subset of the original $\mathbf{x}$.

This dimensionality reduction technique can also be formulated as a matrix factorization. The corresponding optimization has been shown to be

$$\min_{\mathbf{D}\in\mathbb{R}^{k\times d},\mathbf{H}\in\mathbb{R}^{n\times k}} \|\mathbf{X}-\mathbf{H}\mathbf{D}\|_F^2$$

One simple way to see why is to recall the well-known Eckart-Young-Mirsky theorem that the rank k matrix $\hat{\mathbf{X}}$ that best approximates $\mathbf{X}$, in terms of minimal Frobenius norm, is $\hat{\mathbf{X}} = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^\top$.

As with k-means clustering, it may be hard to immediately see why $\mathbf{h}$ generated by PCA could be useful as a representation. In fact, PCA is often used for visualization, and so is not always used for representation learning. For visualization, the projection is often aggressive to two or three dimensions. In general, however, the projection to lower dimensions has the property that it removes noise and maintains only the most meaningful directions. This projection, therefore, helps speed learning by reducing the number of features and promoting generalization, by preventing overfitting to the noise.

Sparse coding takes a different approach, where the input data is expanded into a sparse representation. Sparse coding is biologically motivated [14], based on sparse activations for memory in the mammalian brain. Another interpretation is that sparse coding effectively discretizes the space, like k-means clustering, but with overlapping clusters and an associated magnitude of how much a point belongs to that cluster.

A common strategy to obtain sparse representations is to use a sparse regularizer on the learned representation $\mathbf{h}$. This corresponds to the optimization

$$\min_{\mathbf{D}\in\mathbb{R}^{k\times d},\mathbf{H}\in\mathbb{R}^{n\times k}} \|\mathbf{X}-\mathbf{H}\mathbf{D}\|_F^2 + \lambda\sum_{i=1}^{k}\|\mathbf{H}_{:i}\|_1 + \lambda\sum_{i=1}^{k}\|\mathbf{D}_{i:}\|_2^2$$

As discussed in Section 5.5.2, the ℓ_1 regularizer promotes zeroed entries, and so prefers $\mathbf{H}$ with as many zeros as possible. A regularizer is also added to $\mathbf{D}$, to ensure that $\mathbf{D}$ does not become too large; otherwise, all the weight in $\mathbf{D}\mathbf{H}$ would be shifted to $\mathbf{D}$. As an exercise, see if you can explain why, and what this means for identifiability.

In general, there are many variants of unsupervised learning algorithms that actually correspond to factorizing the data matrix; additional details are given in Appendix B. We additionally give details on how to learn these factorizations in the appendix. As with previous algorithms, they are simply gradient descent on the (matrix) variables. The only distinction here is that it is common to use block coordinate descent, instead of the more standard gradient descent algorithm. This distinction is minor, and it would be perfectly valid to use standard gradient descent.