Mathematical Engineering of Deep Learning

Book Draft

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

Contents

3 Simple Neural Networks - DRAFT

In the previous chapter we explored machine learning in general and also focused on linear models trained via gradient based optimization. In this chapter we move up a notch and consider logistic regression and multinomial (softmax) regression models used for regression and classification. These models often play a key role in statistical modeling and are also very important from a deep learning perspective. Logistic regression and multinomial regression models are shallow neural networks, but also involve non-linear activation functions and hence understanding their structure and means of training is a gateway to understanding general deep learning networks.

In exploring logistic and multinomial regression, we are introduced to some of the basic mathematical engineering elements of deep learning. These include the sigmoid activation function, the cross-entropy loss, the softmax function, and the convexity properties that these models enjoy. We also use the opportunity to consider simple but non trivial autoencoders that generalize PCA, introduced in the previous chapter.

In Section 3.1 we consider the statistical viewpoint of the logistic regression model. In the process we see how binary cross entropy loss results from maximum likelihood estimation. In Section 3.2 we consider the same model, only this time as a shallow neural network trained via gradient descent. In Section 3.3 we adapt the model to multi-class classification. Here we introduce the softmax function as well as the categorical cross entropy loss. In Section 3.4 we investigate feature engineering for such models and see how additional features extend the resulting classifiers to have non-linear decision boundaries. In Section 3.5 we move onto unsupervised learning and consider simple non-linear autoencoder models. In that section we also discuss general autoencoder concepts and describe a few applications of autoencoders. Note that a reader returning to Section 3.5 after reading Chapter 5, may also envision how the simple shallow autoencoders that we present in Section 3.5, can be generalized to deep autoencoders.

3.1 Logistic Regression in Statistics

Logistic regression can be viewed as the simplest non-linear neural network model. However, outside of the context of deep learning, logistic regression is a very popular statistical model. In this section we present logistic regression via a statistical viewpoint. We show how to estimate parameters via maximum likelihood estimation and in the process are introduced to the (binary) cross entropy loss function common in deep learning models including logistic regression, but also beyond.

Note that the statistical view of logistic regression also incorporates parameter uncertainty intervals, hypothesis tests, and other statistical inference aspects. Literature for such statistical inference aspects of logistic regression is provided to at the end of the chapter.

The Model

Logistic regression is a model for predicting the probability that a binary response is 1. It is suitable for classification tasks, a concept described in Section 2.2, as well as for prediction of proportions or probabilities. From a statistical perspective, it is defined by assuming that the distribution of the binary response variable, y, given the features, x, follows a Bernoulli distribution[1] with success probability $\phi(x)$ that is defined below. That is, if we represent the random variables of the feature vector and the (scalar) response via X and Y respectively, then

$$\mathbb{P}(Y = 1 \,|\, X = x) = \phi(x) \quad \text{and} \quad \mathbb{P}(Y = 0 \,|\, X = x) = 1 - \phi(x). \tag{3.1}$$

To capture the relationship between x and $\phi(x)$, the model uses the *odds* of $\phi(x)$, namely $\phi(x)/\big(1 - \phi(x)\big)$, via the *log odds* represented as the *logit* function,

$$\text{Logit}(u) = \log\Big(\frac{u}{1-u}\Big). \tag{3.2}$$

Using this notation, the logistic regression model assumes a linear (affine) relationship between the feature vector x and the log odds of $\phi(x)$. Namely,

$$\text{Logit}\big(\phi(x)\big) = b + w^\top x. \tag{3.3}$$

Here for feature vector $x \in \mathbb{R}^p$, the parameter space is $\Theta = \mathbb{R} \times \mathbb{R}^p$ and the parameters $\theta \in \Theta$ are denoted via $\theta = (b, w)$. In this case, the number of parameters is $d = p + 1$ similarly to the linear regression models of Section 2.3. Like the linear regression model introduced in Section 2.3, the scalar parameter b is called the *intercept* or *bias* and the vector parameter w is called the *regression parameter* or *weight vector*.

The fact that (3.3) presents $\text{Logit}(\cdot)$ on the left hand side in contrast to simply the expected response y as one would have in linear regression, sets logistic regression as a non-trivial form of a *generalized linear model* (GLM). We do not discuss general GLM further, nevertheless it is good to note that using GLM terminology, $\text{Logit}(\cdot)$ plays the role of a *link function*. See notes at the end of this chapter for details and references.

A closely related function to $\text{Logit}(\cdot)$, central to logistic regression and deep learning, is the *sigmoid function*, also called the *logistic function*. It is denoted $\sigma_{\text{Sig}}(\cdot)$ and is also discussed in Section 5.3 where it is plotted in Figure 5.6. Its expression is,

$$\sigma_{\text{Sig}}(u) = \frac{1}{1 + e^{-u}}. \tag{3.4}$$

It has domain $\mathbb{R}$, range $(0, 1)$, and is monotonically increasing. Note that the logit function and the sigmoid function are inverses. That is, $\sigma_{\text{Sig}}\big(\text{Logit}(u)\big) = \text{Logit}\big(\sigma_{\text{Sig}}(u)\big) = u$.

Applying $\sigma_{\text{Sig}}(\cdot)$ to the representation of the model in (3.3) we obtain the common representation of the logistic regression model,

$$\phi(x) = \mathbb{P}\Big(Y = 1 \,|\, X = x\,;\, \theta = (b, w)\Big) = \sigma_{\text{Sig}}(b + w^\top x) = \frac{1}{1 + e^{-(b + w^\top x)}}. \tag{3.5}$$

[1] This is a probability distribution of a random variable having only two outcomes, 0 or 1. It is a special case of the *binomial distribution* where the parameter for the "number of trials" is set at 1.

Side Note: The Logistic Distribution

When considering the statistical logistic regression model, it is also interesting to represent the relationship between X and Y via a continuous *latent variable*, denoted via Z. A latent variable is an unobserved quantity. In the case of logistic regression, Z allows us to use the linear model representation,

$$Z = b + w^\top X + \epsilon, \qquad \text{with} \qquad \begin{cases} Y = 1, & \text{if} \quad Z \geq 0, \\ Y = 0, & \text{if} \quad Z < 0. \end{cases} \tag{3.6}$$

Here the noise component ϵ follows a (standard) *logistic distribution* whose cumulative distribution function, $F_\epsilon(u) = \mathbb{P}(\epsilon \leq u)$, is given by $F_\epsilon(u) = \sigma_{\text{Sig}}(u)$. Such a representation agrees with (3.5) since,

$$\begin{aligned} \phi(x) = \mathbb{P}(Y = 1 \,|\, X = x\,; \theta) &= \mathbb{P}(Z \geq 0 \,|\, X = x\,; \theta) \\ &= \mathbb{P}(b + w^\top x + \epsilon \geq 0) \\ &= \mathbb{P}(b + w^\top x - \epsilon \geq 0) \\ &= \mathbb{P}(\epsilon \leq b + w^\top x) = \sigma_{\text{Sig}}(b + w^\top x). \end{aligned}$$

Note that in the step between the second line and the third line we use the fact that the (standard) logistic distribution of ϵ is symmetric about 0 and hence ϵ and $-\epsilon$ are equal in distribution.

We do not use the latent representation (3.6) any further. Yet it may be interesting to note that if one chooses to use a Gaussian distribution for ϵ in (3.6) in place of the logistic distribution, the model turns out to be a *probit regression model* instead of (3.3). Hence such a latent representation of the model is generally interesting.

Estimation Using the Maximum Likelihood Principle

When statistical assumptions are imposed, *maximum likelihood estimation* (MLE) is a very common statistical method for estimating parameters. While most deep learning models in this book do not use MLE or other statistical point estimation techniques directly, exploring how MLE works for logistic regression is insightful.

Central to MLE is the *likelihood function* $L : \Theta \to \mathbb{R}$. It is a function of the parameters $\theta \in \Theta$ which is obtained by the probability (or probability density) of the data for any given parameter value θ. The idea of MLE is to choose values for θ that maximize the likelihood (function). We see this in action for the logistic regression model now.

Consider the training observations $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ denoted in a similar way to the way $\mathcal{D}$ is defined for the linear model Section 2.3. Here each label $y^{(j)}$ is encoded via either 0 or 1, and the feature vector $x^{(j)}$ is a p-dimensional vector in $\mathbb{R}^p$.

A common assumption in statistics and machine learning is that all features-label pairs $(x^{(j)}, y^{(j)})$ are identically distributed and mutually independent; this is often denoted *i.i.d.*. With this assumption, we aim to estimate the parameters $\theta = (b, w)$. The likelihood function

is

$$L(\theta\,;\,\mathcal{D}) = \prod_{i=1}^{n} \mathbb{P}(Y = y^{(i)} \,|\, X = x^{(i)}\,;\,\theta), \tag{3.7}$$

where $L(\theta\,;\,\mathcal{D})$ is viewed as a function of θ given the observed sample $\mathcal{D}$. The product is due to the independence assumption arising as part of the i.i.d. assumption. In the context of the underlying logistic model, using (3.1) and (3.5), since the labels are either 0 or 1, the likelihood evaluates as

$$L(\theta\,;\,\mathcal{D}) = \prod_{i=1}^{n} \phi(x^{(i)})^{y^{(i)}} \left(1 - \phi(x^{(i)})\right)^{1-y^{(i)}} \text{where} \quad \phi(x) = \sigma_{\text{Sig}}(b + w^{\top} x). \tag{3.8}$$

The maximum likelihood estimate is defined as a value of the parameter θ which maximizes the likelihood function. That is, MLE is the value which maximizes the probability of the observations $\mathcal{D}$ assuming the logistic model.

With (3.8) available, one may proceed to optimize the likelihood directly. However, in this case of logistic regression, and in many other cases where MLE is applied to i.i.d. data, it is more convenient to maximize the logarithm of the likelihood called the *log-likelihood* denoted via $\ell(\theta\,;\,\mathcal{D}) = \log\left(L(\theta\,;\,\mathcal{D})\right)$. This is equivalent to maximizing the likelihood since the log function is a monotonic increasing function. For logistic regression, the log-likelihood expression is,

$$\ell(\theta; \mathcal{D}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi(x^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - \phi(x^{(i)})\right) \right], \tag{3.9}$$

and the maximum likelihood estimate (MLE) is represented via,

$$\hat{\theta}_{MLE} := \underset{\theta \in \mathbb{R}^1 \times \mathbb{R}^p}{\text{argmax}}\ \ell(\theta\,;\,\mathcal{D}) = \underset{\theta \in \mathbb{R}^1 \times \mathbb{R}^p}{\text{argmin}}\ -\frac{1}{n}\ell(\theta\,;\,\mathcal{D}), \tag{3.10}$$

since maximizing $\ell(\theta\,;\,\mathcal{D})$ is the same as minimizing $-\ell(\theta\,;\,\mathcal{D})$ and the positive factor $1/n$ does not change the optimization, yet is useful in the presentation that follows.

One can show that the function $-\frac{1}{n}\ell(\cdot\,;\,\mathcal{D})$ is convex and hence a global minimum exists. Aspects of optimization and convexity are also further overviewed in Section 4.1. However in contrast to the linear model where an explicit analytic solution as in (2.16) exists, in the case of logistic regression there is no analytic solution for the minimizer and hence optimization algorithms are needed to find the MLE (3.10).

The Binary Cross-Entropy Loss

We have already claimed throughout Chapter 2 that learning almost always involves optimization. The MLE based paradigm for logistic regression certainly reinforces this claim. We now reposition logistic regression MLE in terms of minimization of a loss function, following similar lines to the learning of the linear model of Section 2.3. This setup continues straight into more involved deep learning models that follow.

Recall the general loss function formulation which we first presented for linear models in (2.11) where the loss is $C(\theta\,;\,\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta)$. In that context of linear models the individual loss is $C_i(\theta) = C_i(\theta\,;\,y^{(i)}, \hat{y}^{(i)}) = (y^{(i)} - \hat{y}^{(i)})^2$. Logistic regression learning follows

similar lines except that $C_i(\theta)$ is not the quadratic loss. To see this, revisit the minimization form of (3.10) together with the log-likelihood expression (3.9). The scaled negative likelihood that needs to be minimized can then be represented as a loss function via

$$C(\theta\,;\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}\Big[-\Big(y^{(i)}\log\big(\hat{y}^{(i)}\big) + (1-y^{(i)})\log\big(1-\hat{y}^{(i)}\big)\Big)\Big], \tag{3.11}$$

where $\hat{y}^{(i)} = \phi(x^{(i)}) = \sigma_{\text{Sig}}(b + w^\top x^{(i)})$. This then implies that for logistic regression the loss for each data sample is $C_i(\theta) = \text{CE}_{\text{binary}}(y^{(i)}, \hat{y}^{(i)})$ where

$$\text{CE}_{\text{binary}}(y,\hat{y}) = -\Big(y\log\big(\hat{y}\big) + (1-y)\log\big(1-\hat{y}\big)\Big), \tag{3.12}$$

is called the *binary cross entropy* (estimate) applied to observation y and prediction $\hat{y}$.

In general the phrase "cross entropy" and specifically "entropy" is rooted in information theory. Appendix B outlines relationships between cross entropy and related quantities. However, these relationships are not critical for understanding of deep learning. In terms of relating (3.11) to the probabilistic meaning of cross entropy, see first the definition of the cross entropy for two probability distributions in (B.2) and then see the specialisation to distributions with binary outcomes in (B.5).

We also mention that while here, we arrived to the binary cross entropy as a bi-product of maximum likelihood estimation for logistic regression, in general, binary cross entropy has become the default loss function for more complex binary classification deep learning models, as surveyed in Chapter 5 and onwards.

Predicted Probabilities and Parameter Interpretability

For any observed or postulated feature vector $x^\star \in \mathbb{R}^p$, the output of logistic regression is $\hat{y} = \phi(x^\star)$. This is a probability which can be interpreted as in the left hand side of (3.1). Hence at its core, the logistic regression model (3.5) yields probabilities as outputs. Below we see how these probabilities can be used for classification, yet first let us consider prediction of probabilities or proportions.

Recall the breast cancer example presented in Section 2.2 where we overviewed concepts of binary classification. In that example, based on the Wisconsin breast cancer dataset, we used a training set to create two logistic regression models. The first model used the `smoothness_worst` feature as the single coordinate of x and the second model used all possible 30 features. We now continue to use this dataset, this time using all $n = 569$ observations for statistical inference and setting a single feature model based on the `area_mean` feature, and a two feature model based on `area_mean` and `texture_mean`.

With the estimated two feature model based on estimated parameters $\hat{\theta} = (\hat{b}, \hat{w}_1, \hat{w}_2)$, the predicted probability for an observation $x^\star \in \mathbb{R}^2$ is

$$\hat{y} = \sigma_{\text{Sig}}(\hat{b} + \hat{w}_1 x^\star_{\texttt{area\_mean}} + \hat{w}_2 x^\star_{\texttt{texture\_mean}}),$$

where for clarity we subscript coordinates of the features vector $x^\star$ with the feature name. Using (3.3) this can also be represented as,

$$\log\left(\frac{\hat{y}}{1-\hat{y}}\right) = \hat{b} + \hat{w}_1 x^\star_{\texttt{area\_mean}} + \hat{w}_2 x^\star_{\texttt{texture\_mean}}. \tag{3.13}$$

The representation in (3.13) is appealing since it endows the estimated parameters $\hat{b}$, $\hat{w}_1$, and $\hat{w}_2$ with a concrete *interpretation*. First observe that the log odds is linearly described by the features `area_mean` and `texture_mean`. This means that it is estimated that a unit increase in `area_mean` will see the log odds increase by $\hat{w}_1$, a unit increase in `texture_mean` will see the log odds increase by $\hat{w}_2$, and similarly when both features are at zero, the log odds is at the value $\hat{b}$. This interpretation of parameters is clearly not limited to a model with two features as it would work for any number of features.

In practice, it is more convenient to interpret the odds given by,

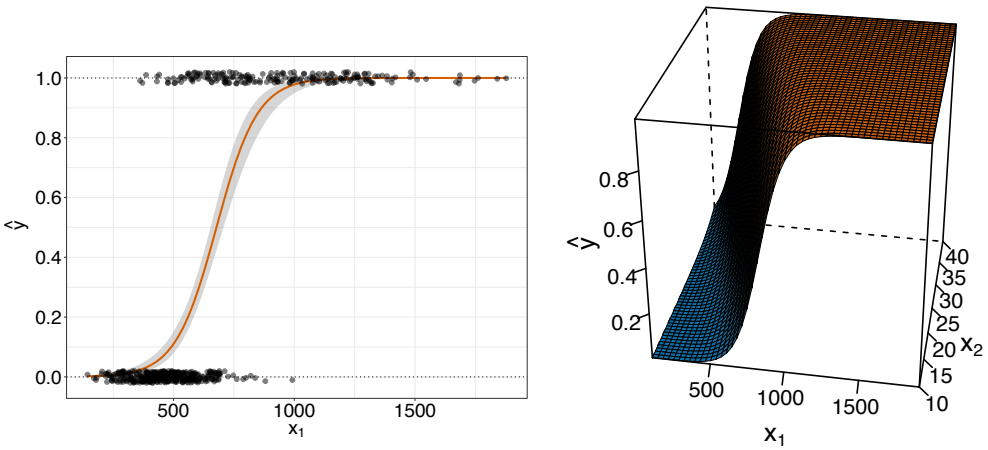
$$\frac{\hat{y}}{1-\hat{y}} = e^{\hat{b}+\hat{w}_1 x^\star_{\texttt{area\_mean}}+\hat{w}_2 x^\star_{\texttt{texture\_mean}}}.$$

Specifically, an increase of `area_mean` by one unit implies a multiplicative increase for the odds by a factor of $e^{\hat{w}_1}$, and similarly for `texture_mean` by a factor of $e^{\hat{w}_2}$. With this view it is typically common to consider the *odds ratio* where the meaning of the multiplicative factor $e^{\hat{w}_i}$ is the ratio between the odds of a model where the feature is at some level x_i between the odds at some level $x_i + 1$. This is especially useful where features are binary but also in general. Further note that when $e^{\hat{w}_i} > 1$ we say the feature has an increasing effect on the probability of the outcome being `positive`, and in the opposite direction a feature with $e^{\hat{w}_i} < 1$ yields a decrease in this probability when the feature is increased.

Such parameter interpretation transcends from linear models to logistic regression models as well as to other statistical models. This interpretability property often makes models extremely attractive in biostatistics and similar fields. It means that estimated parameters of the model are not only useful for their predictive ability, but also for reasoning about the relationships between variables.

As we progress beyond this chapter to more complicated deep neural networks, direct interpretability of parameter values is often lost. In such *non-interpretable* cases, the parameter estimate $\hat{\theta}$ is not a useful learning outcome on its own, but is rather only useful for the tasks of the model. Nevertheless we mention that an active area of machine learning and deep learning research is to seek *interpretable models*.

In Figure 3.1 we present the actual fit of the single feature model and the two features model for the Wisconsin breast cancer observations. The response $\hat{y}$ is the probability of malignant lumps whereas the data involves observations with $y = 1$ (`positive`, i.e., malignant) and $y = 0$ (`negative`, i.e., benign). In (a) we see the sigmoid function applied to $\hat{b} + \hat{w}_1 x_1$ for the estimated single feature model directly and also plot all the observations with a slightly random jitter on the y axis so that they can be visualized easily, where we cut off high outliers. In (b) the sigmoid function is applied to $\hat{b} + \hat{w}_1 x_1 + \hat{w}_2 x_2$ yielding a surface where we omit plotting the actual observations. In both the univariate and multivariate cases it is evident that the models present a monotonic relationship between each of the variables and the predicted probability. This property holds for any number of features.



(a) (b)

Figure 3.1: Logistic regression models for probability prediction fit to the Wisconsin breast cancer dataset. (a) A $p = 1$ model with the feature `area_mean` (x_1)and a confidence band. (b) A $p = 2$ model based on the features `area_mean` (x_1) and `texture_mean` (x_2).

Logistic Regression for Classification is a Linear Classifier

In addition to the application of probability prediction as described above, logistic regression models can also be naturally used for binary classification. We have already seen how to convert such models into binary classifiers. This was first seen in Section 2.2 where we introduced the threshold based classifier in (2.5) with the label prediction $\widehat{\mathcal{Y}}$ set to 1 (`positive`) if $\hat{y} > \tau$ and 0 (`negative`) otherwise.

As an illustration, consider the two features breast cancer prediction model with a response surface as in Figure 3.1 (b). By setting $\tau = 0.5$ for this model we get a classifier as illustrated in Figure 3.2. The red region corresponds to potential feature vectors that would be classified as `negative` (benign) and the blue region corresponds to `positive` (malignant). It is seen that many, but not all, of the training samples are correctly classified.

Interestingly the classification boundary appears like a straight line, or more precisely a hyperplane in the feature space. One may then wonder if this is a property of logistic regression. We now show that in fact, logistic regression based binary classifiers are *linear classifiers*. Such linear classifiers separate the feature space via a single hyperplane, $\mathcal{H}(x) = \breve{b} + \breve{w}^\top x$, where $\breve{b} \in \mathbb{R}$ and $\breve{w} \in \mathbb{R}^p$ are the parameters of the hyperplane. Since every hyperplane cuts Euclidean space into two half-spaces, a natural way to use a hyperplane for classification of a feature vector $x^\star \in \mathbb{R}^p$ is,

$$\widehat{\mathcal{Y}} = \begin{cases} 0 \text{ (`negative`)}, & \text{if} \quad \mathcal{H}(x^\star) \leq 0, \\ 1 \text{ (`positive`)}, & \text{if} \quad \mathcal{H}(x^\star) > 0. \end{cases} \tag{3.14}$$

This means that if $x^\star$ falls in one of the half spaces the classification is `negative` and in the other it is `positive`. The distinction on the boundry is arbitrary.

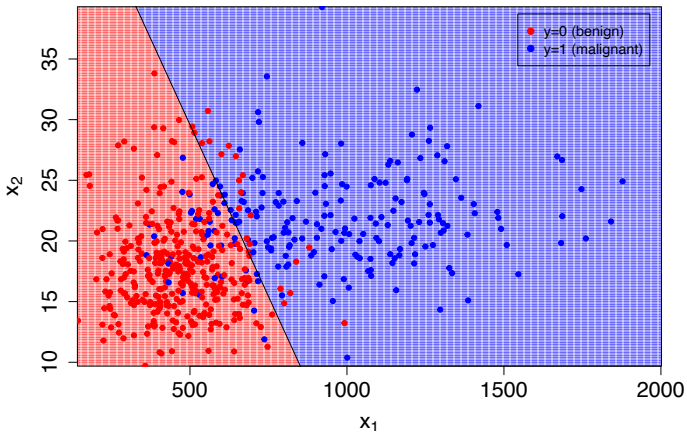


Figure 3.2: Binary classification of malignant lumps (blue dots) and benign lumps (red dots) using a logistic model with two features, `area_mean` (x_1) and `texture_mean` (x_2).

To see that logistic regression is a linear classifier consider estimated model parameters $\hat{\theta} = (\hat{b}, \hat{w})$ and probability prediction $\hat{y}$. The `positive` classification then occurs if $\hat{y} > \tau$, i.e., $\sigma_{\text{Sig}}(\hat{b} + \hat{w}^\top x^\star) > \tau$. We can then apply the Logit$(\cdot)$ function (3.2) to this inequality. Since Logit$(\cdot)$ is a monotonic increasing function and is the inverse of the sigmoid function, we obtain,

$$\hat{b} + \hat{w}^\top x^\star > \log(\frac{\tau}{1-\tau}), \qquad \text{or} \qquad \hat{b} + \log(\tau^{-1} - 1) + \hat{w}^\top x^\star > 0.$$

We thus see that the hyperplane parameters associated with logistic regression classification are $\breve{b} = \hat{b} + \log(\tau^{-1} - 1)$ and $\breve{w} = \hat{w}$. This shows that logistic regression with a threshold rule yields a linear classifier. Note in fact that with the same argument, any model of the form $\hat{y} = \sigma(\hat{b} + \hat{w}^\top x^\star)$ where $\sigma(\cdot)$ is some bijection (invertible function) from $\mathbb{R}$ to $\mathbb{R}$ yields a linear classifier. This naturally includes the linear model based binary classification outlined in Section 2.3. It also includes the probit model mentioned earlier in this section, as well as any shallow neural network for binary classification that has a strictly monotonic activation function, a concept introduced in the next section. Other very common linear classifiers, including support vector machine models are not surveyed in this book.

Note also that one often transforms classifiers with linear decision boundaries into more expressive classifiers via feature engineering. In Section 3.4 we explore how feature engineering based transformations of the features may yield non-linear decision boundaries for the models studied in this chapter.

3.2 Logistic Regression as a Shallow Neural Network

We now position the logistic regression model as a deep learning model. However, as we shortly explain, it is not really "deep" but is rather "shallow" since it does not have hidden layers. This is the first instance in this book where we explicitly consider deep learning models in some mathematical detail. The general (fully connected) deep learning model is

outlined in Chapter 5 and our presentation here serves as a shallow introduction. Note that the linear model of Section 2.3 can also be positioned as a deep learning model; we shed light on this too. Further, the multinomial regression model of the next section is a close relative of logistic regression, and as we show in that section, it is also a shallow neural network.

Logistic Regression is an Artificial Neuron

Let us first represent the logistic regression model (3.5) as

$$\hat{y} = \underbrace{\sigma\left(\overbrace{b + w^\top x}^{z}\right)}_{a}. \tag{3.15}$$

Observe that in (3.15), we omit the subscript from $\sigma_{\text{Sig}}(\cdot)$ used in (3.5). We call $\sigma(\cdot)$ a scalar *activation function* which in the case of logistic regression needs to be $\sigma_{\text{Sig}}(\cdot)$, but in other cases can be a different function. Section 5.3 is devoted to specific forms of such activation functions beyond the sigmoid function. At this point, let us just mention a trivial alternative, the *identity activation function* $\sigma(z) = z$. With this identity activation function, the model in (3.15) is clearly just the linear model $\hat{y} = b + w^\top x$.

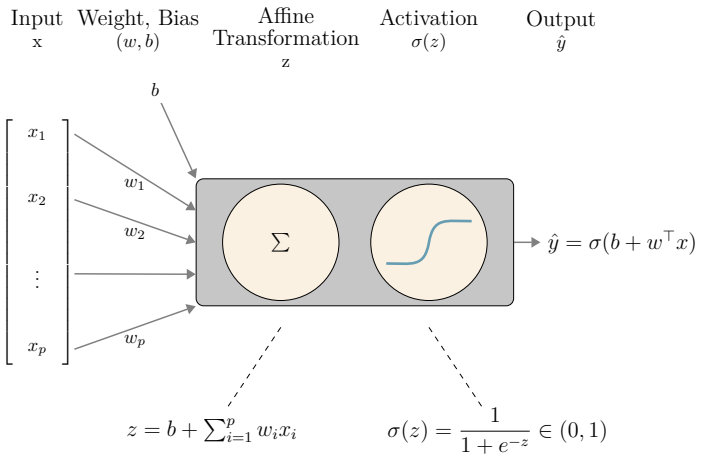


Figure 3.3: Logistic regression represented with neural network terminology as a shallow neural network. The gray box represents an artificial neuron composed of an affine transformation to create z and an activation $\sigma(z)$.

The form of (3.15) represents what we may call a *single layer* of a deep learning model, a *shallow neural network*, or simply an *artificial neuron*. In this case the vector inputs $x \in \mathbb{R}^p$ are transformed to a scalar output $\hat{y} \in \mathbb{R}$. However, in general, deep learning models (as well as shallow neural networks) allow for vector outputs. The next section presents such a case.

Observe that the artificial neuron is composed of an affine transformation $z = b + w^\top x$ followed by a (generally) non-linear transformation $a = \sigma(z)$. This notation of using z for the result of the affine transformation and a for the result of the non-linear transformation is common in deep learning models and heavily used in Chapter 5. Note that the actual definition of an artificial neuron is sometimes taken as the combination of z and a, sometimes

just as the non-linear result a, and sometimes as the computation mechanism defined by the right hand side of (3.15). Figure 3.3 summarizes the components of the artificial neuron focusing on the specific $\sigma(\cdot) = \sigma_{\text{Sig}}(\cdot)$ activation function.

A "deeper" deep learning model would have "hidden layers" based on composition of constructs similar to (3.15). This would involve multiple z and a values computed along the way. Thus when there is a single layer as in (3.15), the neural network is called *shallow* and otherwise it is called *deep*. Logistic regression is the simplest non-linear scalar output shallow neural network that one can consider.

Training Logistic Regression

We have already seen in (3.10) how maximum likelihood estimation positions parameter fitting of logistic regression as an optimization problem. We then saw that maximization of the likelihood is identical to minimization of the loss $C(\theta\,;\,\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta)$, with $C_i(\theta)$ defined via the cross entropy cost (3.12). Importantly, when considered as a deep learning model or a machine learning model, one sometimes ignores the maximum likelihood interpretation of logistic regression and starts off with the cross entropy loss as an engineered loss function which requires minimization.

When statistical software packages are used to estimate parameters of logistic regression, this optimization is typically tackled using second-order methods; see Section 4.6 for an overview of such techniques. In general, such methods make use of the *Hessian matrix* of the loss function or approximations of it; see Appendix A for a review.

In contrast to statistical practices, in deep learning and machine learning culture, one often considers the number of features p to be very large in which case standard second-order optimization methods tend to struggle computationally. Thus when treated as a deep learning model, the default technique used for logistic regression training is gradient descent. Gradient descent was already introduced in Section 2.4 in the context of the linear model, and variants of gradient based learning are studied in detail in the next chapter as well; see sections 4.2 and 4.3.

General deep learning models do not have explicit expressions for the gradient of $C(\theta\,;\,\mathcal{D})$ and certainly not for the Hessian matrix. Hence learning such models requires computational techniques for gradient evaluation. The most common technique is the backpropagation algorithm, described in Section 5.4, which is a form of automatic differentiation, overviewed in Section 4.4. However, in the case of logistic regression, like the linear model, there are explicit expressions both for the gradient and the Hessian. We see these now.

In the case of logistic regression, the gradient of $C(\theta\,;\,\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta)$ with respect to $\theta = (b, w) \in \mathbb{R} \times \mathbb{R}^p$ is a vector in $\mathbb{R}^d$ with $d = p + 1$. It is denoted as $\nabla C(\theta)$ and can be represented as the average of the gradients of each $C_i(\theta)$. Namely,

$$\nabla C(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i(\theta). \tag{3.16}$$

This general relationship between the gradient of the total loss function $\nabla C(\theta)$ and the gradients of the loss for each observation $\nabla C_i(\theta)$ is common throughout deep learning. In the case of logistic regression we have that $C_i(\theta) = \text{CE}_{\text{binary}}(y^{(i)}, \hat{y}^{(i)})$ with $\text{CE}_{\text{binary}}(\cdot, \cdot)$

from (3.12). We thus require expressions for the gradient of this binary cross entropy loss with observation label $y^{(i)}$ and predicted value $\hat{y}^{(i)} = \sigma_{\text{Sig}}(b + w^\top x^{(i)})$. That is,

$$\nabla C_i(\theta) = -\nabla \left(y^{(i)} \log \sigma_{\text{Sig}}(b + w^\top x^{(i)}) + (1 - y^{(i)}) \log \left(1 - \sigma_{\text{Sig}}(b + w^\top x^{(i)})\right)\right),$$

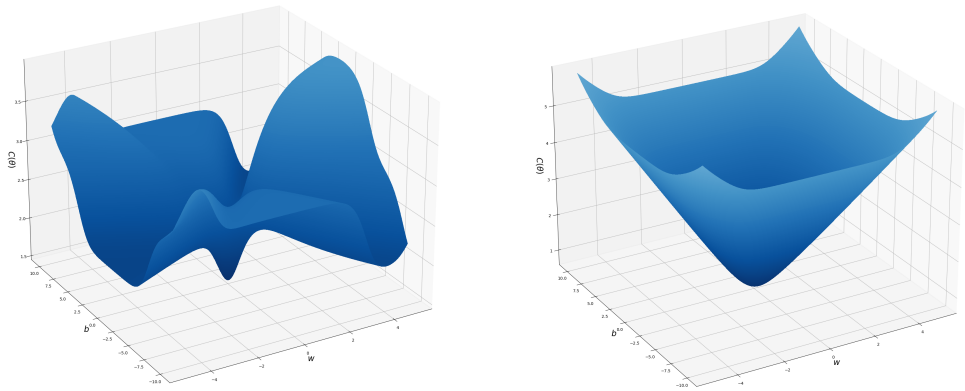
where the gradient is with respect to the vector $\theta = (b, w)$. Now using basic differentiation rules and the structure of $\sigma_{\text{Sig}}(\cdot)$, we obtain,

$$\begin{aligned} \frac{\partial C_i}{\partial b} &= \sigma_{\text{Sig}}(b + w^\top x^{(i)}) - y^{(i)}, \\ \frac{\partial C_i}{\partial w_j} &= \left(\sigma_{\text{Sig}}(b + w^\top x^{(i)}) - y^{(i)}\right) x_j^{(i)} \qquad \text{for} \qquad j = 1, \ldots, p. \end{aligned}$$

Thus in combining these components we can represent the $d = p + 1$ dimensional gradient vector as,

$$\nabla C_i\,(\theta) = \left(\sigma_{\text{Sig}}(w^\top x^{(i)} + b) - y^{(i)}\right) \begin{bmatrix} 1 \\ x^{(i)} \end{bmatrix}, \tag{3.17}$$

where the first scalar expression on the right hand side of (3.17) is the prediction difference for observation i and the second expression is the vector of features for observation i including the constant 1 feature.



(a) **(b)**

Figure 3.4: The loss landscape of logistic regression for a synthetic dataset. (a) Using the squared loss $C_i(\theta) = (y^{(i)} - \hat{y}^{(i)})^2$. (b) Using the binary cross entropy loss $C_i(\theta) = \text{CE}_{\text{binary}}(y^{(i)}, \hat{y}^{(i)})$.

Some Benefits of Cross Entropy Loss

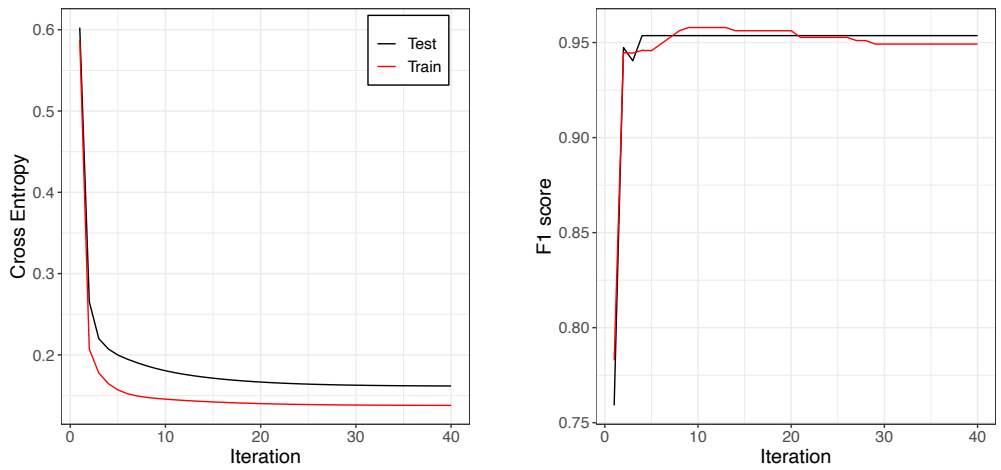
If one considers the problem of logistic regression training purely based on an optimization approach and not based on a maximum likelihood approach, then the cross entropy loss can potentially be replaced by other loss functions such as for example quadratic cost. However, it turns out that using the cross entropy loss, generally yields desirable loss landscapes.

As an illustration consider Figure 3.4 based on logistic regression with synthetic data of a single feature ($p = 1$ and $d = 2$). The parameters to be optimized are the scalars b and w. In (a) we use the squared error loss and in (b) we use the cross entropy loss. It is clear from

this image, that at least in this case, the cross entropy loss landscape is more manageable to navigate for an optimization algorithm like gradient descent. In fact, in the case of logistic regression, cross entropy always yields a convex loss landscape (further details are in the next chapter), while other losses such as the squared error loss generally yield non-convex loss landscapes often presenting multiple local minima as well as saddle points.

Importantly, when considering classification (or probability prediction problems), deep learning models that are more complex than logistic regression are still often easier to optimize using the cross entropy loss than the squared error loss or other losses. In such more complex models, the neat mathematical convexity property that logistic regression enjoys with cross entropy is lost, and multiple local minima can exit. Nevertheless, computational and research experience has shown that in general using cross entropy is preferable.

As an illustration of gradient descent applied to a concrete example we return to the Wisconsin breast cancer dataset, now splitting the observations as we did in Chapter 2 into $n_{\text{train}} = 456$ and $n_{\text{test}} = 113$. We use a model with $p = 10$ features ($d = 11$) and learn the parameters via gradient descent with some arbitrary initilization. In doing so, we obtain trajectories as in Figure 3.5.



(a) **(b)**

Figure 3.5: Training logistic regression via gradient descent for the Wisconsin breast cancer dataset with an 80-20 train-test split (we can treat the test set as a validation set). (a) Loss over iterations. (b) Performance over iterations using the F_1 score when $\tau = 0.5$.

3.3 Multi-class Problems with Softmax

The *multinomial regression model* as it is known in statistics, or *softmax regression* as it is known in machine learning[2] is the generalization of logistic regression from the binary response case to the case of $K > 2$ classes. Now the response random variable Y takes on values $1, 2, \ldots, K$. The feature vector remains X just like in logistic regression.

[2]In some machine learning fields, this is also called *softmax logistic regression*, *multinomial logistic regression*, or *multi-class logistic regression*.

We denote the training observations via $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, where each label $y^{(j)}$ is one of K class values $\{1, \ldots, K\}$. The purpose of the model is to predict class probability vectors, or if used for classification, to predict a class in a multi-class setting.

The Model

Just like logistic regression predicts two classes and uses $\phi(x)$ for the probability of the positive response, in multinomial regression the predicted response is the probability vector

$$\boldsymbol{\phi}(x) = \big(\phi_1(x), \ldots, \phi_K(x)\big), \tag{3.18}$$

where,

$$\phi_k(x) = \mathbb{P}(Y = k \mid X = x) \quad \text{for} \quad k = 1, \ldots, K. \tag{3.19}$$

To compare (3.19) with (3.1), observe that (3.1) is like (3.19) with $K = 2$ where $\phi(x) = \phi_1(x)$ and $1 - \phi(x) = \phi_2(x)$.

The name "multinomial regression" stems from the *multinomial distribution* which is a probability distribution over vectors of length K consisting of non-negative integers that sum up to some specified integer N. Specifically a random vector $(U_1, \ldots, U_K)$ follows a multinomial distribution with parameters N (positive integer) and $\boldsymbol{\phi} = (\phi_1, \ldots, \phi_k)$ (probability vector) if,

$$\mathbb{P}\left(U_1 = u_1, \ldots, U_K = u_K\right) = \frac{N!}{u_1! \cdot \ldots \cdot u_K!} \prod_{k=1}^{K} \phi_k^{u_k} \qquad \text{for} \qquad \sum_{k=1}^{K} u_k = N,$$

with the probability at 0 when $\sum_{i=k}^{K} u_k \neq N$. A specific case of the multinomial distribution with $N = 1$ is called a *categorical distribution*. In this case the random vector $(U_1, \ldots, U_K)$ is like a one-hot encoded vector since exactly one coordinate is 1 and the rest are 0s.

A statistical assumption in multinomial regression is that the one-hot encoded response, given the features vector $X = x$, follows a categorical distribution. The random vector of the one-hot encoded response of Y is denoted $(Y_1, \ldots, Y_k)$ where $Y_k = \mathbf{1}\{Y = k\}$, i.e., the kth coordinate equals 1 if $Y = k$ and equals 0 otherwise. Now the categorical distribution assumption is,

$$\mathbb{P}\left(Y_1 = u_1, \ldots, Y_K = u_K \mid X = x\right) = \prod_{k=1}^{K} \phi_k(x)^{u_k}. \tag{3.20}$$

The key model assumption in multinomial regression[3] is in the way in which the probability vector $\boldsymbol{\phi}(x)$ depends on the features vector x. There are two possible parameterizations, which we refer to as the statistical parameterization and the machine learning parameterization. The former presents a unique (*identifiable*) model while the latter is slightly simpler but has some redundant parameters.

Starting with the statistical parameterization, it assumed that,

$$\phi_k(x) = \frac{e^{b_k + w_{(k)}^\top x}}{1 + \sum_{j=1}^{K-1} e^{b_j + w_{(j)}^\top x}} \qquad \text{for} \qquad k = 1, \ldots, K-1, \tag{3.21}$$

[3]Like logistic regression, this assumption is rooted in the theory of generalized linear models (GLM), a topic we defer to the notes and references at the end of the chapter.

and further,

$$\phi_K(x) = 1 - \sum_{j=1}^{K-1} \phi_j(x). \tag{3.22}$$

This directly generalizes the sigmoidal relationship of logistic regression (3.5) from $K = 2$ to $K \geq 2$. With this statistical parameterization, the parameters of multinomial regression are

$$\theta = (b_1, \ldots, b_{K-1}, w_{(1)}, \ldots, w_{(K-1)}) \in \underbrace{\mathbb{R} \times \ldots \times \mathbb{R}}_{K-1 \text{ times}} \times \underbrace{\mathbb{R}^p \times \ldots \times \mathbb{R}^p}_{K-1 \text{ times}} := \Theta, \tag{3.23}$$

and hence the number of parameters is $d = (K-1)(p+1)$. The scalar parameters $b_1, \ldots, b_{K-1}$ are bias (intercept) parameters and each of the vector parameters $w_{(1)}, \ldots, w_{(K-1)}$ is a weight vector (regression parameter). With this parameterization, we may also view the final bias b_K and final weight vector $w_{(K)}$ as the scalar 0 and vector 0 respectively. Such a restriction on b_K and $w_{(K)}$ allows us to combine (3.21) and (3.22) into,

$$\phi_k(x) = \frac{e^{b_k + w_{(k)}^\top x}}{\sum_{j=1}^K e^{b_j + w_{(j)}^\top x}} \qquad \text{for} \qquad k = 1, \ldots, K. \tag{3.24}$$

Moving onto the machine learning parameterization the last class also has free parameters b_K and $w_{(K)}$ like the other classes. In this case the parameter space is $\Theta = \mathbb{R}^K \times (\mathbb{R}^p)^K$ and thus $d = K(p+1)$. Now again the representation (3.24) is valid, yet unlike the statistical parameterization, the last term in the summation in the denominator is not restricted to be 1. In this sense the machine learning parameterization is simpler, however when estimating θ with maximum likelihood estimation there is never a unique θ that maximizes the likelihood (or minimizes the loss).

The Softmax Function and Multinomial Regression as a Shallow Neural Network

A common function in deep learning, especially when considering classification problems, is the $\mathbb{R}^K \to \mathbb{R}^K$ *softmax activation function*. For $z = (z_1, \ldots, z_K) \in \mathbb{R}^K$, it is defined as,

$$S_{\text{Softmax}}(z) = \frac{1}{\sum_{i=1}^K e^{z_i}} \begin{bmatrix} e^{z_1} \\ \vdots \\ e^{z_K} \end{bmatrix}. \tag{3.25}$$

Softmax and its derivative expressions are further discussed in Section 5.3. At this point let us just point out that for any vector $z \in \mathbb{R}^K$, the result of $S_{\text{Softmax}}(z)$ is a probability vector.

Now with $S_{\text{Softmax}}(\cdot)$ defined we can revisit the multinomial regression model (3.24) and represent it as the softmax of a vector of length K that has $b_k + w_{(k)}^\top x$ at the kth coordinate. More succinctly we have,

$$\hat{y} = \underbrace{S_{\text{Softmax}}(\overbrace{b + Wx}^{z \in \mathbb{R}^K})}_{a \in \mathbb{R}^K}, \tag{3.26}$$

where $\hat{y}$ is the prediction of $\boldsymbol{\phi}(x)$ as in (3.18), and the K dimensional vector b and the $K \times p$ dimensional matrix W are

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_K \end{bmatrix}, \qquad \text{and} \qquad W = \begin{bmatrix} — & w_{(1)}^\top & — \\ — & w_{(2)}^\top & — \\ & \vdots & \\ — & w_{(K)}^\top & — \end{bmatrix}. \tag{3.27}$$

Compare (3.26) with (3.15) of logistic regression. In the logistic regression case, z is a scalar and so is a. In contrast, in the multinomial regression case the affine transformation converts a vector $x \in \mathbb{R}^p$ into a vector $z \in \mathbb{R}^K$ and then the softmax activation retains the same dimension to arrive at a.

Note also that (3.26) is valid both in the statistical parameterization and the machine learning parameterization. In the former, the last bias b_K and the last row $w_{(K)}^\top$ of (3.27) are zeros, where in the latter these are free variables.

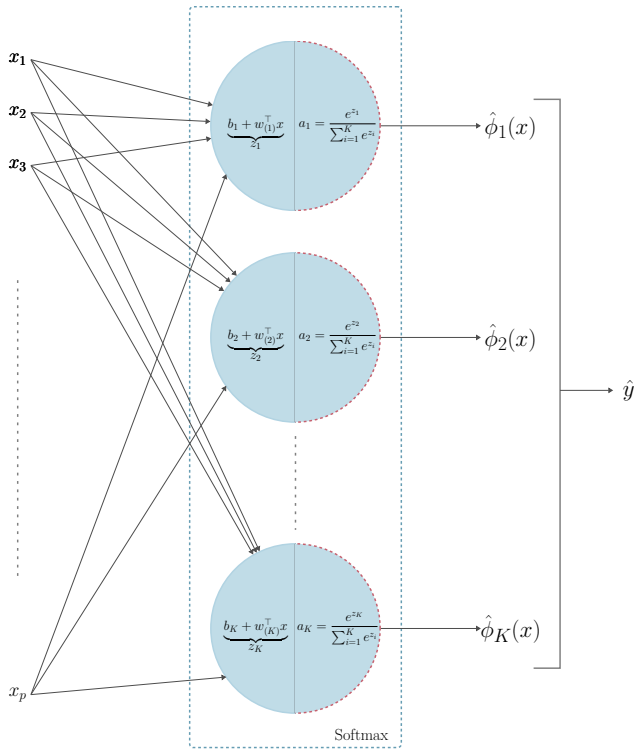


Figure 3.6: Multinomial regression as a neural network model with output $\hat{y}$ which is a probability vector. Note that each a_i is a function of all of $z_1, \ldots, z_K$ due to the softmax operation.

This representation positions multinomial regression as a shallow neural network similarly to the way logistic regression is a shallow neural network. The difference is that multinomial regression has vector outputs while logistic regression has a scalar output. Figure 3.6 illustrates the multinomial regression model as a neural network. Here each circle can again be viewed as an "artificial neuron" however note that the softmax activation affects all

neurons together via the normalization in the denominator of (3.25). Hence the activation value of each neuron is not independent of the activation values of the other neurons.

Likelihood and Cross Entropy

Now that we understand the multinomial regression model, both from a statistical perspective using the probabilistic interpretation (3.20) and as a deep learning model using (3.26), let us consider maximum likelihood estimation, and equivalently loss function minimization.

To obtain an estimate $\hat{\theta}$ using maximum likelihood estimation, we follow a procedure analogous to the one used for logistic regression. Specifically, the likelihood of the model for the data $\mathcal{D}$ is defined as in (3.7). Now using the probability law of the categorical distribution, (3.20), we obtain the likelihood,

$$L(\theta\,;\,\mathcal{D}) = \prod_{i=1}^{n}\prod_{k=1}^{K} \phi_k(x^{(i)})^{\mathbf{1}\{y^{(i)}=k\}} \qquad \text{for} \qquad \theta \in \Theta,$$

where we use the parameter space Θ as in the statistical parameterization[4] (3.23). Now considering the log-likelihood by applying $\log(\cdot)$ to $L(\theta\,;\,\mathcal{D})$ we obtain,

$$\begin{aligned}
\ell(\theta\,;\,\mathcal{D}) &= \sum_{i=1}^{n}\sum_{k=1}^{K} \mathbf{1}\{y^{(i)}=k\}\log\phi_k(x^{(i)}) \\
&= \sum_{i=1}^{n} \log \phi_{y^{(i)}}(x^{(i)}) \\
&= \sum_{i=1}^{n} \left(b_{y^{(i)}} + w_{(y^{(i)})}^{\top} x^{(i)} - \log \sum_{j=1}^{K} e^{b_j + w_{(j)}^{\top} x^{(i)}} \right).
\end{aligned} \tag{3.28}$$

In the second step, we use the subscript $y^{(i)}$ because except for the index k where $y^{(i)} = k$, all other summands of the internal sum of the first row are zero. The third step follows from (3.24).

Now similarly to our development of logistic regression MLE in (3.10), we can represent the problem as a minimization problem and define the estimator via,

$$\widehat{\theta}_{MLE} = \underset{\theta\in\Theta}{\operatorname{argmin}} \; -\frac{1}{n}\ell(\theta\,;\,\mathcal{D}).$$

Further, we can also view this problem as a loss minimization problem, where as before the loss is of the form $C(\theta\,;\,\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta)$, and $C_i(\theta)$ is the loss for an individual observation. Based on the expressions of the log-likelihood (3.28) and using $\hat{y}^{(i)}$ for the vector $\boldsymbol{\phi}(x^{(i)})$, the loss for an individual observation is,

$$C_i(\theta) = -\sum_{k=1}^{K} \mathbf{1}\{y^{(i)}=k\}\log\hat{y}_k^{(i)} = -\log\hat{y}_{y^{(i)}}^{(i)}. \tag{3.29}$$

[4] Alternatively one may opt for the machine learning parameterization with the bigger parameter space. In this case the MLE is never unique.

Here, the first and second equalities arise from the first and the second lines of (3.28) respectively. The expressions on the right hand side of (3.29) are in fact called the *categorical cross entropy*. More precisely, for a label $y \in \{1, \ldots, K\}$ and a probability vector $\hat{y}$ of length K, we define the categorical cross entropy as,

$$\mathrm{CE}_{\mathrm{categorical}}(y, \hat{y}) = -\sum_{k=1}^{K} \mathbf{1}\{y = k\} \log \hat{y}_k = -\log \hat{y}_y, \tag{3.30}$$

where in the final expression, $\hat{y}_y$ means taking the element at index y from the vector $\hat{y}$. We thus see that

$$C_i(\theta) = \mathrm{CE}_{\mathrm{categorical}}(y^{(i)}, \hat{y}^{(i)}). \tag{3.31}$$

We have already seen the binary cross entropy in (3.12). Similar to this, the categorical cross entropy (3.30) is the empirical estimate of the cross entropy of two probability distributions appearing in (B.2) of Appendix B. We note here that in many contexts of deep learning, one just uses the phrase "cross entropy" without the prefix "binary" or "categorical".[5] Then the distinction between $\mathrm{CE}_{\mathrm{binary}}(\cdot, \cdot)$ and $\mathrm{CE}_{\mathrm{categorical}}(\cdot, \cdot)$ is based on context and the type of arguments y and $\hat{y}$. In the former y is a binary outcome in $\{0, 1\}$ and $\hat{y}$ is a scalar probability in $[0, 1]$. In the latter, y is a multi-class choice in $\{1, \ldots, K\}$ and $\hat{y}$ is a probability vector of length K in $[0, 1]^K$. The binary cross entropy is a special case of the categorical cross entropy and $\mathrm{CE}_{\mathrm{binary}}(y, \hat{y})$ with $y \in \{0, 1\}$ and $\hat{y} \in [0, 1]$ can be represented as $\mathrm{CE}_{\mathrm{categorical}}(2 - y, [\hat{y}, 1 - \hat{y}]^\top)$.

It may be useful to also see other notational forms for the loss of an individual observation $C_i(\theta)$. Making use of (3.26) we obtain

$$C_i\big(\theta\big) = \mathrm{CE}_{\mathrm{categorical}}\big(y^{(i)}, S_{\mathrm{Softmax}}(b + W x^{(i)})\big), \tag{3.32}$$

where we parameterize θ to be composed of the vector b and the matrix W. Namely, $\Theta = \mathbb{R}^K \times \mathbb{R}^{K \times p}$. Alternatively,

$$C_i(\theta) = -(b_{y^{(i)}} + w_{(y^{(i)})}^\top x^{(i)}) + \log \sum_{j=1}^{K} e^{b_j + w_{(j)}^\top x^{(i)}}. \tag{3.33}$$

Compare (3.33) to the last line of (3.28).

Derivatives and Learning

Like logistic regression, inference for multinomial regression can be efficiently carried out using second-order methods. Nevertheless, when multinomial regression is viewed as a deep learning model with very large p, one often uses gradient descent (first-order optimization) in place of second-order methods. We now present the derivative expressions of $C_i(\theta)$. These expressions allow one to use gradient descent just as in the case of logistic regression.

Using either the statistical parameterziation with $d = (K - 1)(p + 1)$ parameters or the machine learning parameterziation with $d = K(p + 1)$ parameters, we have that $\theta = (b, W)$ consists of both a vector b and a matrix W. Thus in dealing with the gradient of $C_i(\theta)$ with respect to θ, denoted as $\nabla C_i(\theta)$, one needs to either vectorize θ into a vector of length d, or

[5] In practice, when deep learning systems implement the binary and categorical cross entropy, $\log(u)$ in (3.12) or (3.30) is replaced with $\log(u + \varepsilon)$ where ε is a small fixed parameter. This allows to seamlessly include the probability $u = 0$ as an input.

alternatively, to make use of the derivative of a real valued function with respect to a matrix, as defined in Appendix A, (A.7). Our presentation uses both variants, and it is a prelude for further derivative expressions involving matrix valued parameters, used in Chapter 5 and the chapters that follow.

Let us first consider the derivative of (3.33) with respect to the individual scalar elements of $\theta = (b, W)$. Specifically, for $k = 1, \ldots, K$ we obtain,

$$\frac{\partial C_i}{\partial b_k} = \frac{e^{b_k + w_{(k)}^\top x^{(i)}}}{\sum_{j=1}^K e^{b_j + w_{(j)}^\top x^{(i)}}} - \mathbf{1}\{y^{(i)} = k\},$$
$$\frac{\partial C_i}{\partial w_{k,\ell}} = \left(\frac{e^{b_k + w_{(k)}^\top x^{(i)}}}{\sum_{j=1}^K e^{b_j + w_{(j)}^\top x^{(i)}}} - \mathbf{1}\{y^{(i)} = k\} \right) x_\ell^{(i)}, \qquad \text{for} \qquad \ell = 1, \ldots, p.$$

These derivatives can then be placed in a d dimensional vector[6] to form $\nabla C_i(\theta)$.

Now observe that the expressions above involve the softmax function where the ratio of the exponent with the sum of exponents in each expression is the kth coordinate of $S_{\text{Softmax}}(b + Wx^{(i)})$. This hints at a simpler expressions and indeed we may use unit vector notation (e_j) in place of the indicator functions $\mathbf{1}\{\cdot\}$ to obtain,

$$\frac{\partial C_i}{\partial b} = S_{\text{Softmax}}(b + Wx^{(i)}) - e_{y^{(i)}},$$
$$\frac{\partial C_i}{\partial W} = \left(S_{\text{Softmax}}(b + Wx^{(i)}) - e_{y^{(i)}}\right) {x^{(i)}}^\top.$$

With such a representation the first expression which is a derivative with respect to a vector is in fact composed of the coordinates associated with the bias vector b of the gradient $\nabla C_i(\theta)$. Further, the second expression is a derivative of a real valued function with respect to a matrix, which we represent using the notation of (A.7) in Appendix A to represent the matrix with (k, ℓ)th element denoting $\frac{\partial C_i}{\partial w_{k,\ell}}$.

In general, when one implements gradient based optimization as in Step 3 of Algorithm 2.1 of Chapter 2, or using one of the multiple variants presented in Chapter 4, these derivatives need to be used with the right shape dimensions taken into account.

We also mention that it is sometimes common to subsume the bias term expressions within the weight parameter expressions by augmenting the feature vector to be of length $p+1$ where the first feature is the constant 1. This practice was already used in the linear regression treatment of Section 2.3, and could have also been employed in the logistic regression treatment of Section 3.1. However we mention that from a deep learning perspective[7] the weight parameters in W often receive a different treatment than the bias parameters in b and thus keeping b separate from W notationally is instructive.

[6] In fact, when one seeks a Hessian expression for $C_i(\theta)$, using such a d dimensional vector layout is the standard approach. We skip presentation of such a Hessian expression here, but note that (in contrast to more complicated deep learning models) it is tractable, and is often used in second-order methods for multinomial regression.

[7] This is made clearer in chapters 5 and 6, for example, in the contexts of dropout and convolutional neural networks.

Note that like logistic regression, the loss minimization problem of multinomial regression is convex and thus with proper hyper-parameter choices (e.g., learning rate), a global minimum can in principle always be reached. We establish the convexity of this optimization problem in the next chapter.

Classification with Multinomial Regression Yields Convex Polytope Decision Regions

Recall that with multi-class classification, our goal is to provide a prediction $\widehat{\mathcal{Y}}$ for the label $\{1, \ldots, K\}$ associated with the input feature vector. In Section 2.2 we introduced concepts of binary classification and then via the example of the linear model in Section 2.3 we saw strategies for adapting binary classifiers to a multi-class classifier. This was via the one vs. rest and one vs. one approaches. In contrast to these approaches, the multinomial model provides a direct solution for multi-class classification since the output $\hat{y}$ is already a probability vector over $\{1, \ldots, K\}$. This approach is also common in more complicated deep learning models presented in the chapters that follow.

In general, an output vector $\hat{y}$, which is a probability vector, can be used to create a classifier by choosing the class that has the highest probability (and breaking ties arbitrarily). Namely,

$$\widehat{\mathcal{Y}} = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} \ \hat{y}_k. \tag{3.34}$$

This approach is called a *maximum a posteriori probability* (MAP) decision rule since it simply chooses $\widehat{\mathcal{Y}}$ as the class that is most probable. It is the most common decision rule when using deep learning models for classification. As a side note, for binary classification, a MAP decision agrees with binary classification threshold prediction (2.5) with a threshold of $\tau = 0.5$, and similarly to the fine-tuning of the parameter τ, in the multi-class case we may also deviate from MAP decisions and adjust (3.34) if needed.

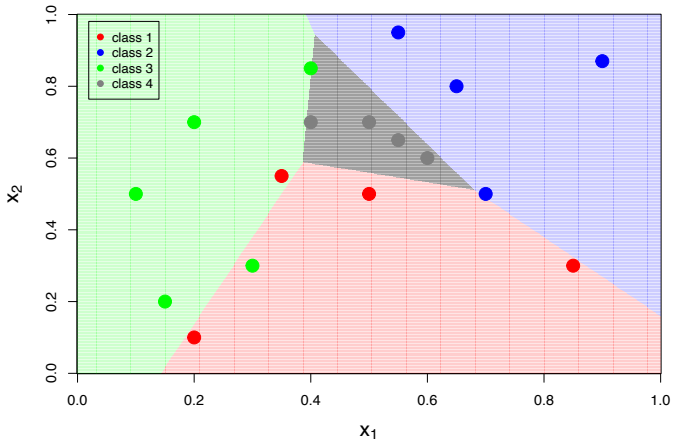


Figure 3.7: Multinomial regression for multi-class classification applied to synthetic data. In this example the training accuracy is 15/17.

As an example, consider Figure 3.7 that represents a MAP based rule applied to the output of a multinomial model trained on $n = 17$ synthetic data observations with $p = 2$ features and $K = 4$ classes. Like the binary classification example of Figure 3.2, this multi-class classification example illustrates the output of $\widehat{\mathcal{Y}}(x^\star)$ for any $x^\star \in \mathbb{R}^p$. Similarly to the logistic regression case, we observe that the decision boundaries are straight lines. This is not a coincidence but rather a property of multinomial regression.

Denote by $\mathcal{C}_k$ the set of input features vectors $x \in \mathbb{R}^p$ such that $\widehat{\mathcal{Y}}(x) = k$. We now see that $\mathcal{C}_k$ is an intersection of half spaces, i.e., it is a convex polytope. To see this, consider some arbitrary point $x^\star \in \mathbb{R}^p$ and fix some class label $k \in \{1, \ldots, K\}$. We can now compare $\hat{y}_k(x^\star)$ and $\hat{y}_j(x^\star)$ for all other labels j. Specifically, since the denominator of the softmax expression (3.24) is independent of the index, $\hat{y}_k(x^\star) \geq \hat{y}_j(x^\star)$ if and only if

$$e^{b_k + w_{(k)}^\top x^\star} \geq e^{b_j + w_{(j)}^\top x^\star}, \qquad \text{or} \qquad (b_k - b_j) + (w_{(k)} - w_{(j)})^\top x^\star \geq 0.$$

Thus by defining a hyperplane $\mathcal{H}_{kj}$ with $\mathcal{H}_{kj}(x) = (b_k - b_j) + (w_{(k)} - w_{(j)})^\top x$, we see that $\hat{y}_k(x^\star) \geq \hat{y}_j(x^\star)$ holds if and only if $\mathcal{H}_{kj}(x^\star) \geq 0$ for all other classes j. Now $\widehat{\mathcal{Y}}(x^\star) = k$ only if $\hat{y}_k(x^\star) \geq \hat{y}_j(x^\star)$ for all other j. Hence $\mathcal{C}_k$ is an intersection of $K - 1$ hyperplanes.

As a concrete example we return to MNIST digit classification explored in Section 2.3. In that section we used the one vs. rest and one vs. one strategies to build classifiers based on linear models. See Table 2.1 where we presented confusion matrices for these classifiers when trained on the 60,000 MNIST train images and tested on the 10,000 test images. We now report the performance of multinomial regression on this dataset as well as the application of one vs. rest and one vs. one with logistic regression.[8] A summary is in Table 3.1. Our purpose with this comparison is not to claim which method is best since in practice all these methods are significantly beat by convolutional neural networks as presented in Chapter 6. We rather aim to highlight that a direct multi-class strategy such as multinomial regression requires training and application of a single model while the other approaches require multiple models, and are not significantly superior.

Table 3.1: Different approaches for creating an MNIST digit classifier. It is evident that in general one vs. one classifiers outperform one vs. rest classifiers, yet have many more parameters. Further, on the same type of classification scheme, logistic regression generally outperforms linear regression. Finally observe that multinomial regression with only a single model and a low number of parameters, generally performs almost as good as the top scheme (logistic regression with one vs. one).

Strategy and model type	Number of models to train	Total number of parameters	Test accuracy
Linear regression one vs. rest	10	7,850	0.8603
Linear regression one vs. one	45	35,325	0.9297
Logistic regression one vs. rest	10	7,850	0.9174
Logistic regression one vs. one	45	35,325	0.9321
Multinomial regression	1	7,850 or 7,065	0.9221

[8] The linear regression based classifiers use the pseudo-inverse and hence the accuracy on the test set of size 10,000 is exact (up to numerical error). The logistic and multinomial classifiers were trained with gradient based learning with an ADAM algorithm with a learning rate of 0.02, mini-batches of size 2000, and 200 epochs; see Chapter 4. With this gradient based learning there is room for error with the accuracy as it depends on the optimization run. Hence the best achievable accuracy with the non-linear classifiers can potentially be slightly better.

3.4 Beyond Linear Decision Boundaries

Recall figures 3.1, 3.2, and 3.7. Logistic regression naturally yields a sigmoidal relationship between the input features and the response probability. In a classification setting this translates to linear (affine) decision boundaries. Similarly, multinomial regression used for classification also yields straight line boundaries via convex polytope decision boundaries. One may then ask if these shallow neural network models can create other forms of response curves or decision boundaries? We now show that this is indeed possible via feature engineering following a similar spirit to the way feature engineering was introduced in Section 2.2.

Enhancing the Sigmoidal Response

Recall first the linear regression example illustrated in Figure 2.3 of Chapter 2. In display (b) of that figure we saw how linear regression with an engineered quadratic feature yields a curve that better fits the data. A similar type of idea may also be applied in the case of logistic regression.

As an example consider a dataset $\mathcal{D}$ where each observation is for a different geographic location and where the feature vector has a single coordinate which is the level of precipitation at that location (in millimeters/month). For each observation, the associated $y^{(i)} \in \{0, 1\}$, determines the absence (0) or presence (1) of a certain species. Observations from such a dataset[9] are presented in Figure 3.8. At locations i that are not too dry or not too wet, the species tends to be present, whereas when the precipitation is very low or very high the species tends to be absent.

Such a relationship between the precipitation and the probability of presence of the species cannot be captured by a sigmoidal curve since $\sigma_{\text{Sig}}(b + w^\top x)$ is a monotonic function of x. In fact, when fitting a sigmoidal curve to this data, the response, plotted via the red curve of Figure 3.8, tends to be almost flat in the region of the observations because in this case $\hat{w}_1$ is close to zero.

Nevertheless, if we wish to use logistic regression for this problem we may do so via feature engineering. Similarly to the housing price example of Section 2.2 we introduce a new feature $x_2 = x_1^2$. With the addition of this new feature, the model (as a function of the single feature $x = x_1$) becomes,

$$\phi(x) = \frac{1}{1 + e^{-(b + w_1 x + w_2 x^2)}}.$$

Following from basic properties of the parabola $b + w_1 x + w_2 x^2$, if $w_2 < 0$ then we have that $\phi(x) \to 0$ as $x \to -\infty$ or $x \to \infty$ and further $\phi(x)$ is maximized at $x = -w_1/w_2$ which is the maximal point of the parabola. Similarly, if $w_2 > 0$ the shape of $\phi(x)$ is reversed and $\phi(x)$ has a minimum point at the minimum point of the parabola. In both cases, $\phi(x)$ is symmetric about $x = -w_1/w_2$.

Fitting logistic regression to this feature engineered model yields a negative value for $\hat{w}_2$ and this results in the blue curve of Figure 3.8. We thus see that feature engineering in the context of logistic regression allows us to extend the form of the response beyond the sigmoid function.

[9] This is a synthetic dataset motivated by ecological applications.

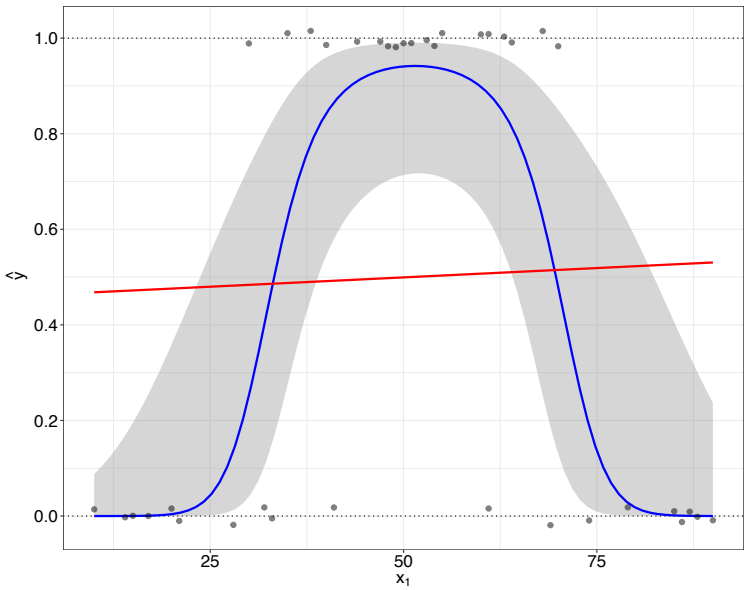


Figure 3.8: Logistic regression fit with prediction $\hat{y}$ for a feature engineered model with the feature $x_2 = x_1^2$. The response curve is plotted in blue together with confidence bounds. The red curve is a sigmoidal function fit to the single feature $x = x_1$.

The General Setup of Polynomial Feature Engineering

As in the example above, in general it is quite common to use powers for feature engineering and this makes the linear combination of the engineered features a polynomial. In examples with a small number of features such as the $p = 1$ example above, we can tweak feature engineering visually. However, when p is large, other performance based methods are needed.

When there are initially p input features we can automate the creation of more features by choosing each new feature as a *power product* or *monomial* form $x_1^{k_1} x_2^{k_2} \cdot \ldots \cdot x_p^{k_p}$ for some non-negative integers $k_1, k_2, \ldots, k_p$. With such a process it is common to limit the *degree* by a constant r via

$$k_1 + k_2 + \ldots + k_p \leq r.$$

For example when $r = 2$ the set of engineered features is,

$$\tilde{x} = (x_1, x_2 \ldots, x_p, x_1^2, x_2^2, \ldots, x_p^2, x_1x_2, x_1x_3, \ldots, x_{p-1}x_p).$$

In this case $\tilde{x} \in \mathbb{R}^{p(p+3)/2}$ and thus $d = 1 + p(p+3)/2$ for logistic regression.[10] So if for example there are initially $p = 1,000$ features then there are about half a million engineered features with $d = 501,501$. One may of course cull the number of engineered features to reduce the number of learned parameters. This can sometimes be done via regularization introduced in Section 2.5.

[10] This also holds for any model where the number of parameters is one plus the number of features such as for example linear regression.

Let us also go beyond $r = 2$ to higher degrees of the monomial features. From basic combinatorics, we have that the number of non-negative integer solutions to $k_1 + \ldots + k_p = \ell$ is $(\ell + p - 1)!/\big((p-1)!\,\ell!\big)$ and thus when using a polynomial feature scheme with degree up to r we have for logistic regression

$$d = \sum_{\ell=0}^{r} \frac{(\ell + p - 1)!}{(p-1)!\,\ell!},$$

and for multinomial regression, d is K or $K-1$ times this number with the machine learning or statistical parameterizations respectively.

Using *Stirling's approximation* of factorials we have that when ℓ is significantly smaller than p then $d \approx p^r/r!$ for logistic regression. As an example with $p = 1,000$ input features and setting $r = 4$ we have approximately 4.16×10^{10} parameters or and exact number of $42,084,793,751$. This is over 40 trillion parameters to learn!

Having 40 trillion parameters in a model is indeed borderline astronomical and as of today still infeasible. Thus for non-small p, going beyond $r = 2$ is rarely used in practice. In fact, in Section 5.2 of Chapter 5 we argue that with deeper neural networks one may sometimes get more expressivity without creating such a large number of parameters.

Versatile Classification Boundaries

To further explore feature engineering let us now consider classification problems. We have seen above that both logistic regression and multinomial regression yield decision boundaries defined by hyperplanes which we may generally denote via $\mathcal{H}(x)$. Each such hyperplane has parameters $\breve{b} \in \mathbb{R}$ and $\breve{w} \in \mathbb{R}^p$ which depend on the estimated model parameters in a simple manner. Decision rules for classification with given input $x^\star \in \mathbb{R}^p$ are then made based on if $\mathcal{H}(x^\star) \geq 0$ or not (where the case of multinomial regression involves multiple such hyperplanes and comparisons).

Now in a feature engineered scenario the hyperplane parameters are adapted to be of larger dimension and in place of $\mathcal{H}(x^\star)$ we evaluate $\tilde{\mathcal{H}}(\tilde{x}^\star)$ with parameters $\breve{\breve{b}}$ and $\breve{\breve{w}}$. As an example consider a scenario with $p = 2$ features where we carry out polynomial feature engineering as above with $r = 2$. Then the number of parameters is now $d = 6$ and the hyperplane comparison is then,

$$\breve{\breve{b}} + \breve{\breve{w}}_1 x_1 + \breve{\breve{w}}_2 x_2 + \breve{\breve{w}}_3 x_1^2 + \breve{\breve{w}}_4 x_2^2 + \breve{\breve{w}}_5 x_1 x_2 \geq 0. \tag{3.35}$$

Now here the set of input feature vectors $(x_1, x_2) \in \mathbb{R}^2$ that satisfies the above inequality is no longer a half space but rather represents a curved subset of $\mathbb{R}^2$. We thus see that such feature engineering enables non linear decision boundaries.

As an example, consider Figure 3.9 based on synthetic data that requires binary classification with two input features. In (a) we see that for this type of data, logistic regression without feature engineering is not suitable. The linear decision boundary is not able to capture the pattern in the data. In (b) we expand the set of features and get a decision boundary of the form (3.35). It appears that going from $d = 3$ learned parameters to $d = 6$ learned parameters is worthwhile since the pattern in the data is well captured in (b).

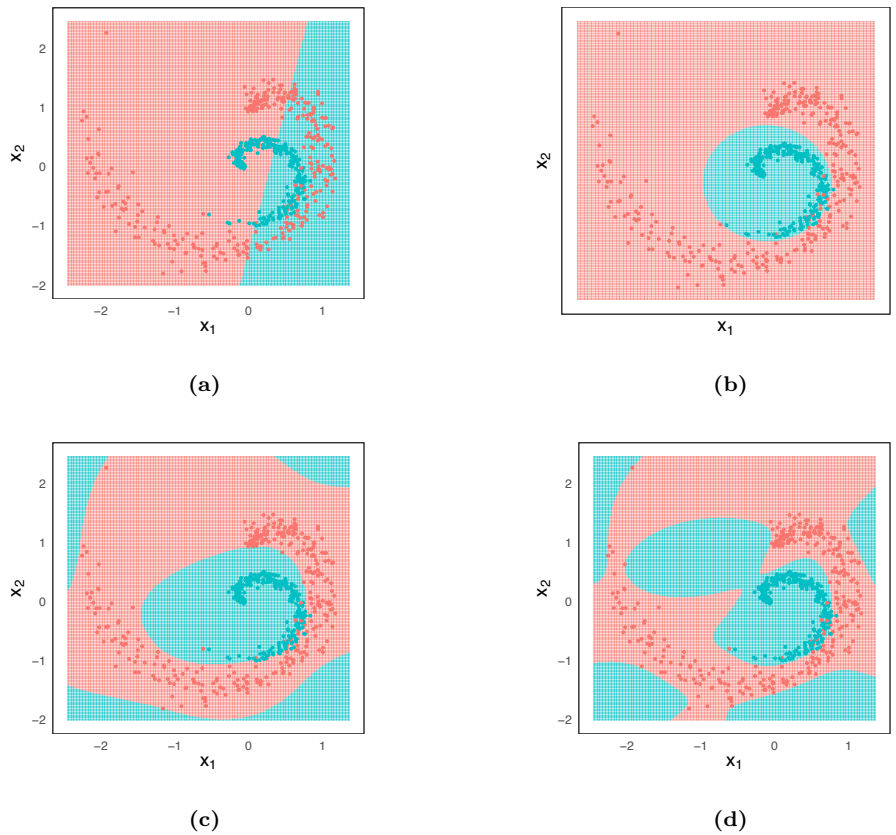
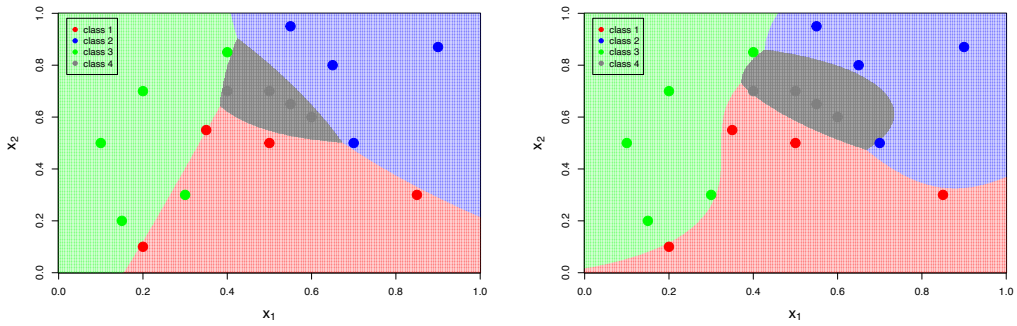


Figure 3.9: Decision boundaries for binary classification with an expanded set of features for synthetic data with $p = 2$ input features. (a) No feature engineering ($r = 1$, $d = 3$). (b) Quadratic $r = 2$, $d = 6$. (c) Quartic $r = 4$, $d = 15$. (d) $r = 8$, $d = 45$.

We may continue with higher orders in an attempt to gain more expressivity. However, as we see in (c) and (d), for this data, the higher order models yield obscure classification decision boundaries. In this example these higher orders certainly appear like an overfit. Such overfitting would probably lead to high generalization error (recall the discussion in Section 2.5). Moreover, the new set of features could become highly correlated and that can cause difficulty in inference of parameters when taking a statistical approach.

As another visual example, return to the synthetic multi-class classification example illustrated in Figure 3.7. We expand the set of features for this example and plot the decision regions in Figure 3.10. As this is just a synthetic example, our purpose is to show that by increasing the number of engineered features we can get more curvature in the decision boundaries. In (a) we consider quadratic features and in (b) we consider an extreme case of $r = 8$ which has 180 parameters with the machine learning parameterization.



(a) (b)

Figure 3.10: Decision boundaries with an expanded set of features for the multinomial regression model ($K = 4$). (a) $r = 2$, $d = 24$ (b) $r = 8$, $d = 180$.

3.5 Shallow Autoencoders

So far in this chapter we explored logistic and multinomial regression. Both of these models are shallow neural networks, which do not involve hidden layers, for supervised learning where the data is labelled. They are special cases of "deep learning models". The same goes for the linear regression model of Section 2.3 (it was made evident that linear regression is also a deep learning model in Section 3.2). Thus, all the key supervised learning models that we discussed up to this point are simple neural networks that fall under the umbrella of deep learning.

We now devote this last section of this chapter to simple neural networks that are used for unsupervised learning where the unlabelled data is of the form $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$. Namely we introduce *autoencoder* architectures and focus primarily on shallow versions of these architectures. Recall that in Section 2.1 we presented an overview of the concept of unsupervised learning in the context of machine learning activities, and in Section 2.6 we surveyed basic unsupervised learning techniques including principle components analysis (PCA). As we see in this current section, PCA can be cast as a special case of an autoencoder and this positions PCA as a form of (simple) neural network based learning as well.

Autoencoder Principles

Before we explore several varied applications of *autoencoders*, let us focus on their basic architecture. Consider Figure 3.11 which presents a schematic of an autoencdoer with a single *hidden layer*. The input $x \in \mathbb{R}^p$ is transformed into a *bottleneck*, also called the *code*, which is some $\tilde{x} \in \mathbb{R}^m$ and is the hidden layer of the model. Then the bottleneck is further transformed into the output $\hat{x} \in \mathbb{R}^p$. The part of the autoencoder that transforms the input into the bottleneck is called the *encoder* and the part of the autoencoder that transforms the bottleneck to the output is called the *decoder*. Both the encoder and the decoder have parameters that are to be learned.

Note that many other deep learning models in this book will also have hidden layers. In fact, representing and computing gradients for the parameters of such layers is the focus of

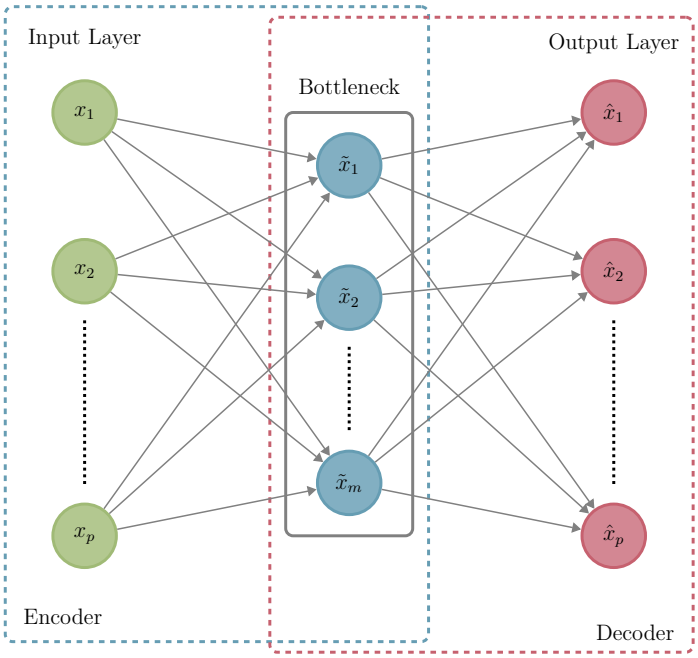


Figure 3.11: An autoencoder architecture with an encoder, decoder, and a bottleneck in between which is the single hidden layer of this neural network.

Chapter 5 and stands at the heart of deep learning. In this autoencoder example, the single hidden layer is also the bottleneck of the autoencoder and thus we informally consider this shallow autoencoder as "simple". Other "deeper" autoencoders may have multiple hidden layers of which one should be treated as the bottleneck or code.

Interestingly for input x, once parameters are trained, we generally expect the autoencoder to generate output $\hat{x}$ that is as similar to the input x as possible. This property of autoencoders, after which they are named, may at first seem obscure since it means that the autoencoder is a form of an identity function. However, this architecture is actually very useful for a variety of applications, some of which are detailed in this section and others appearing as parts of more complicated models such as for example sequence models which we discuss in Chapter 7.

For now, as the most basic application, consider the activity of data reduction already surveyed in Section 2.6 in the context of PCA and SVD (singular value decomposition). For this, assume that the dimension of the bottleneck m is significantly smaller than the input and output dimension p (in other non-data reduction applications we may also have $m \geq p$). For example, return to the case of MNIST digits (initially introduced in Section 1.4) where $p = 784$. For our example here, assume we have an autoencoder with $m = 30$.

If indeed a trained autoencoder yields $x \approx \hat{x}$ then it means that we have an immediate data reduction method. With the trained encoder we are able to convert digit images, each of size $28 \times 28 = 784$, into much smaller vectors, each of size 30. With the trained decoder we

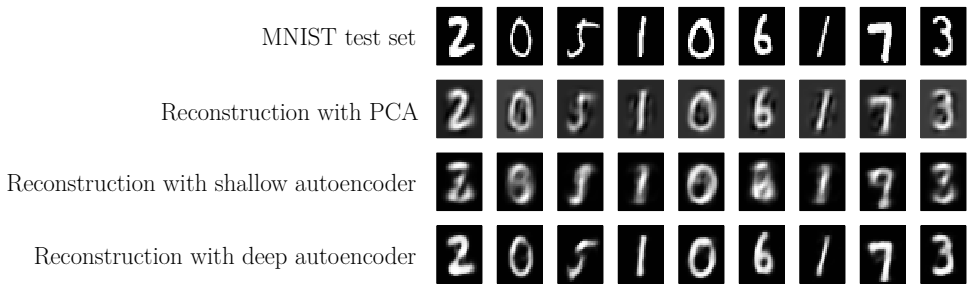


Figure 3.12: Reconstruction of the test set of MNIST (first row) using various types of autoencoders.

are able to convert back and get an approximation of the original image. This choice of m implies a rather remarkable compression factor of about 26.

Figure 3.12 presents the compression/decompression effect of several types of autoencoders with $m = 30$. The first row presents untouched MNIST images. The second row presents the effect of reducing the images via PCA (a shallow linear autoencoder) and the reverting back to the image. The third row presents the effect of a shallow non-linear autoencoder. Finally the last row presents the effect with a richer autoencoder that has several hidden layers (a deep autoencoder).

There are multiple other applications for autoencoders which we soon survey. However, let us first formulate these types of models more precisely.

Single Layer Autoencoders

As already mentioned above, we may view an autoencoder as a function $f_\theta : \mathbb{R}^p \to \mathbb{R}^p$, where θ are the trainable parameters of this function. These parameters θ ideally influence the function's operation such that $f_\theta(x^\star) \approx x^\star$ where $x^\star$ is an arbitrary observation from either the seen or unseen data. The approximate equality, "$\approx$", can be considered informally as closeness of two vectors.

In practice when faced with training data $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$, we train the autoencoder (learn the parameters θ) such that $C(\theta\,; \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta)$ is minimized. This is a similar loss function setup to that used in supervised contexts such as linear regression, logistic regression, multinomial regression, or the deep learning models that are in the chapters that follow. For autoencoders, we construct the loss for an individual observation, $C_i(\theta)$, as some distance penalty measure between the input observation $x^{(i)}$ and the output $\hat{x}^{(i)} = f_\theta(x^{(i)})$. Contrast this with supervised learning where we compare the observed label and the predicted label. That is, with autoencoders the target of the model is the input in contrast to a label $y^{(i)}$ in supervised learning.

The most straightforward choice for the distance penalty in $C_i(\theta)$ is the square of the Euclidean distance, namely,

$$C_i(\theta) = \|x^{(i)} - f_\theta(x^{(i)})\|^2 \quad \text{and thus} \quad C(\theta\,;\,\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - f_\theta(x^{(i)})\|^2. \tag{3.36}$$

With this cost structure, learning the parameters, θ, of an autoencoder based on data $\mathcal{D}$ is the process of minimizing $C(\theta\,;\,\mathcal{D})$.

In general, autoencoders may have architectures similar to the fully connected deep neural networks that we study in Chapter 5 as well as to extensions in the chapters that follow. This may include multiple hidden layers, convolutional layers, and other constructs. At this point, to understand the key concepts, let us revert to Figure 3.11 and consider autoencoders composed of the same components of that figure.

Specifically, we decompose $f_\theta(\cdot)$ to be a composition of the encoder function denoted via $f^{[1]}_{\theta^{[1]}}(\cdot)$ and the decoder function denoted via $f^{[2]}_{\theta^{[2]}}(\cdot)$, where $\theta^{[1]}$ are the parameters of the encoder and $\theta^{[2]}$ are the parameters of the decoder.[11] That is,

$$\hat{x} = f_\theta(x) = \big(f^{[2]}_{\theta^{[2]}} \circ f^{[1]}_{\theta^{[1]}}\big)(x) = f^{[2]}_{\theta^{[2]}}\big(f^{[1]}_{\theta^{[1]}}(x)\big),$$

where the notation using the square bracketed superscripts for the functions and parameters is in agreement with the notation we use for layers of deep neural networks in Chapter 5 and onwards.

In general, one may construct all kinds of structures for the encoder, decoder, and their parameters. In our case, we consider a specific single layer neural network structure. We define,

$$\begin{array}{rcl} f^{[1]}_{\theta^{[1]}}(u) & = & S^{[1]}(b^{[1]} + W^{[1]}u) \quad \text{for} \quad u \in \mathbb{R}^p \qquad \text{(Encoder)} \\ f^{[2]}_{\theta^{[2]}}(u) & = & S^{[2]}(b^{[2]} + W^{[2]}u) \quad \text{for} \quad u \in \mathbb{R}^m \qquad \text{(Decoder)}, \end{array} \tag{3.37}$$

where the notation is somewhat similar to (3.26). Specifically for $\ell = 1, 2$, $b^{[\ell]}$ are vectors, $W^{[\ell]}$ are matrices, and $S^{[\ell]}(\cdot)$ are vector activation functions with $S^{[1]} : \mathbb{R}^m \to \mathbb{R}^m$ and $S^{[2]} : \mathbb{R}^p \to \mathbb{R}^p$. Before describing the exact details of these functions, let us focus on the autoencoder parameters.

The encoder parameters $\theta^{[1]}$ are composed of the bias $b^{[1]} \in \mathbb{R}^m$ and weight matrix $W^{[1]} \in \mathbb{R}^{m\times p}$, and the decoder parameters $\theta^{[2]}$ are composed of the bias $b^{[2]} \in \mathbb{R}^p$ and weight matrix $W^{[2]} \in \mathbb{R}^{p\times m}$. Thus the complete list of parameters for the autoencoder is,

$$\theta = (b^{[1]}, W^{[1]}, b^{[2]}, W^{[2]}). \tag{3.38}$$

Returning to the vector activation functions $S^{[1]}(\cdot)$ and $S^{[2]}(\cdot)$, we construct these based on scalar activation functions $\sigma^{[\ell]} : \mathbb{R} \to \mathbb{R}$ for $\ell = 1, 2$ such as the sigmoid function (3.4), the identity function $\sigma(u) = u$, or one of many other variants (see also Section 5.3). Specifically, we set $S^{[\ell]}(z)$ to be the element wise application of $\sigma^{[\ell]}(\cdot)$ on each of the coordinates of z.

[11] An alternative way to denote these parameters would have been using ϕ (in place of $\theta^{[1]}$) for encoder and θ (in place of $\theta^{[2]}$) for the decoder. This is the notation used in variational autoencoders in Chapter 8.

Namely,

$$S^{[\ell]}(z) = \begin{bmatrix} \sigma^{[\ell]}(z_1) \\ \vdots \\ \sigma^{[\ell]}(z_r) \end{bmatrix}. \tag{3.39}$$

The choice of the type of scalar activation function may depend on the domain of the input x since the output of the model aims to reconstruct the input. For example, use of the sigmoid function for $\sigma^{[2]}(\cdot)$ restricts the output to be in the range $[0, 1]$ and this will clearly be unsuitable in cases where the input is not limited to this range.

With this notation in place, it may also be useful to see the individual representation of the bottleneck units $\tilde{x}_1, \ldots, \tilde{x}_m$, and the outputs $\hat{x}_1, \ldots, \hat{x}_p$. Specifically, with $a_k = \tilde{x}_k$,

$$\tilde{x}_i = \sigma^{[1]}\Big(b_i^{[1]} + \sum_{k=1}^{p} w_{i,k}^{[1]} x_k\Big), \quad \text{for} \quad i = 1, \ldots, m,$$
$$\hat{x}_j = \sigma^{[2]}\Big(b_j^{[2]} + \sum_{k=1}^{m} w_{j,k}^{[2]} a_k\Big). \quad \text{for} \quad j = 1, \ldots, p.$$

This set of equations is the first non-shallow (single hidden layer) neural network which we see in the book. Note also that we often use the notation $a_k = \tilde{x}_k$ as it is an 'activation' of a *unit* or neuron within the encoder.

Also, it may be useful to see the loss function representation as,

$$C(\theta\,;\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - \underbrace{S^{[2]}(W^{[2]}S^{[1]}(W^{[1]}x^{(i)} + b^{[1]}) + b^{[2]})}_{f_\theta(x^{(i)})}\|^2. \tag{3.40}$$

With this loss function, for given data $\mathcal{D}$, the learned autoencoder parameters $\hat{\theta}$ are given by a solution to the optimization problem $\min_\theta C(\theta\,;\mathcal{D})$.

PCA is an Autoencoder

It turns out that autoencoders generalize principal component analysis (PCA), introduced in Section 2.6. We overview the details by seeing that PCA is essentially a shallow autoencoder with identity activation functions $\sigma^{[\ell]}(u) = u$ for $\ell = 1, 2$, also known as a *linear autoencoder*. Specifically, we now summarize how PCA yields one possible solution to the learning optimization problem for linear autoencoders. Note that some of the mathematical details below may be skipped on a first reading without loss of continuity. The key outcome of this subsection is that PCA and linear autoencoders are essentially the same.

Consider the loss (3.40) with identity activation functions. In this case the vector activation functions $S^{[\ell]}(\cdot)$ of (5.3) are each vector identity functions and (3.40) reduces to,

$$C(\theta;\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - W^{[2]}W^{[1]}x^{(i)} - W^{[2]}b^{[1]} - b^{[2]}\|^2. \tag{3.41}$$

Now it can be shown[12] that by considering the de-meaned data, we can formulate the objective without the bias vectors $b^{[1]}$ and $b^{[2]}$ in (3.41) and focus on optimizing the loss function,

$$C(W^{[1]}, W^{[2]}\,; \mathcal{D}_{\text{de-meaned}}) = \frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - W^{[2]}W^{[1]}x^{(i)}\|^2. \tag{3.42}$$

Here when the data is denoted $\mathcal{D}_{\text{de-meaned}}$, we reuse the notation of the data samples $x^{(i)}$, taking them now as de-meaned feature vectors (see also (2.42) in Chapter 2). It can be shown that optimization of this new loss function (3.42) is equivalent to optimization of (3.41). Specifically, if minimizing (3.42) and obtaining minimizers $\hat{W}^{[1]}$ and $\hat{W}^{[2]}$, then $\hat{b}^{[1]}$ can be set to any value and $\hat{b}^{[2]}$ has a specific expression. With these, together with $\hat{W}^{[1]}$ and $\hat{W}^{[2]}$ minimize (3.41).

Now it is possible to go one step further in reducing the parameter space. In fact, it can be shown that for the optimum of (3.42) we have $\hat{W}^{[1]} = \hat{W}^{[2]\dagger}$. This is the pseudoinverse as introduced in Section 2.3. Specifically when $\hat{W}^{[2]}$ is full column rank the pseudoinverse can be represented as,

$$\hat{W}^{[2]\dagger} = (\hat{W}^{[2]\top}\hat{W}^{[2]})^{-1}\hat{W}^{[2]\top},$$

and thus the matrix in (3.42) is,

$$\hat{W}^{[2]}\hat{W}^{[1]} = \hat{W}^{[2]}\hat{W}^{[2]\dagger} = \hat{W}^{[2]}(\hat{W}^{[2]\top}\hat{W}^{[2]})^{-1}\hat{W}^{[2]\top}.$$

This is the $p \times p$ projection matrix which projects vectors of length p onto the m dimensional column space of $W^{[2]}$. Now using the QR decomposition[13] this projection matrix may be represented as $VV^\top$ where the $p \times m$ matrix V has orthonormal columns. This means that an equivalent optimization problem to the problem of minimizing (3.42) is

$$\min_{V \in \mathbb{R}^{p\times m}\ V^\top V = I_m,} \quad \frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - VV^\top x^{(i)}\|^2, \quad \text{where} \quad x^{(i)} \in \mathcal{D}_{\text{de-meaned}}, \tag{3.43}$$

with I_m denoting the $m \times m$ identity matrix. This *constrained optimization* problem limits the search space to the space of $p \times m$ matrices that have orthonormal columns. Any minimizer V^* of (3.43) can then be used as a minimizer of the losses (3.41) or (3.42) by setting,

$$\hat{W}^{[1]} = V^{*\top}, \quad \text{and} \quad \hat{W}^{[2]} = V^*.$$

Now let us see the connection to PCA. Recall (2.51) representing the encoded lower dimensional PCA data as $\widetilde{X} = XV$ where the $n \times p$ matrix X is a (de-meaned) data matrix as in Section 2.6, the $n \times m$ matrix $\widetilde{X}$ is the reduced data matrix, and the matrix V has columns that are singular vectors from the SVD decomposition of X. Hence the projection of each data point $x^{(i)}$ into this lower dimensional space is given by

$$\tilde{x}^{(i)} = V^\top x^{(i)} \tag{3.44}$$

with V the $p \times m$ matrix of columns $v_1, \ldots, v_m$ that are an orthonormal basis of a reduced subspace of $\mathbb{R}^p$. Further, with PCA, the matrix V can also be used to reconstruct the data

[12] This is shown by considering the derivative with respect to $b^{[2]}$ as a first step and then reorganizing the objective (3.41) with the expression of $b^{[2]}$ that sets the objective to 0. See also the notes and references at the end of the chapter

[13] See the notes and references of Chapter 1 for suggested linear algebra background reading.

as a decoder. Specifically, the decoded data points in the original space can be obtained via

$$\hat{x}^{(i)} = V\tilde{x}^{(i)}. \tag{3.45}$$

Now piecing together the encoder in (3.44) and the decoder in (3.45), the reconstruction error is,

$$\frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - \hat{x}^{(i)}\|^2 = \frac{1}{n}\sum_{i=1}^{n} \|x^{(i)} - VV^\top x^{(i)}\|^2. \tag{3.46}$$

We see that (3.46) agrees with the objective in (3.43) and further the matrix V of PCA agrees with the constraint $V^\top V = I_m$. Now from the *Eckart-Young-Mirsky theorem* appearing in (2.53) of Section 2.6 as well as the relationship between SVD and PCA captured in (2.51) we have that V of PCA is indeed one of the optimal solutions[14] of (3.43). Hence in summary, PCA is a (shallow) linear autoencoder.

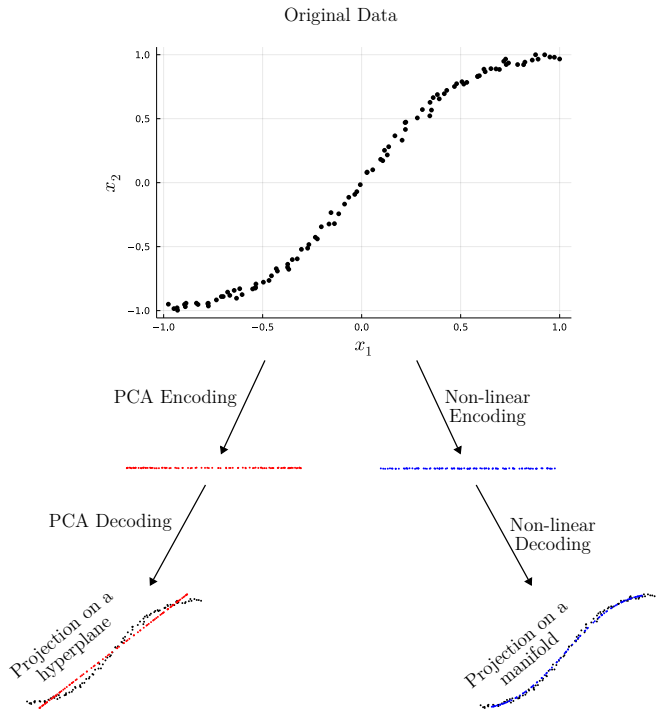


Figure 3.13: Encoding and decoding of synthetic $p = 2$ data with a linear autoencoder (PCA) in red and a non-linear shallow autoencoder (blue). The reconstruction of PCA falls on a hyperplane while the non-linear autoencoder projects onto a manifold that is not an hyperplane.

In practice, one would not use gradient based optimization to learn the parameters of linear autoencoders, but rather employ algorithms from numerical linear algebra. Further, the specific basis vectors obtained via PCA are insightfull since they order vectors according to their variance contributions. In contrast, optimizing (3.43) without considering PCA, yields arbitrary V (with orthonormal columns). Nevertheless, as we see now, our positioning of PCA as a special case of the (non-linear) autoencoder is insightful.

[14] There is an infinite number of solutions.

Autoencoders as a Form of Non-linear PCA

As we discussed above, encoding and decoding with PCA projects the p dimensional feature vector x onto an m dimensional subspace. When $p = 2$ and $m = 1$ this can be viewed as a projection of points from the plane onto a line, when $p = 3$ and $m = 2$ this is a projection of points from three dimensional space onto some plane, and similarly in more realistic higher dimensions of p, we project onto a linear subspace of dimension m. As such, the bottleneck of the linear encoder (PCA) encodes the location of the points on this projected space.

Linear subspace projection is sometimes a sufficient data reduction technique and at other times is not. In such cases there are other multiple forms of *non-linear PCA* where points are projected onto manifolds that are generally curved. Since autoencoders generalize PCA, they present us with one such rich class of non-linear PCA models.

As an illustration consider Figure 3.13 where we consider synthetic data with $p = 2$ which we wish to encode with $m = 1$. This means that the bottleneck layer, or the code, represents a value on the real line for each data point. If we use PCA (red) then this encoding translates to a location on a linear subspace of $\mathbb{R}^2$. However, if we modify the identity activation functions in the autoencoder to be non-linear (blue), then the projection is on a manifold which is generally curved. In this example the non-linear activation functions are taken as tanh functions; see also Section 5.3.

As a further example, consider using an autoencoder on MNIST where $p = 28 \times 28 = 784$ and we use $m = 2$. We encode this via PCA, a shallow non-linear autoencoder, and a deep autoencoder that has hidden layers. In Figure 3.14 we present scatter plots of the codes for various types of autoencoders for both the training and test sets. That is, the autoencoders are trained on the training set and the codes presented are both for the training set, and for the test set data. While the training and testing does not directly involve the labels (the digits 0–9), in our visualization we color the code points based on the labels. This allows us to see how different labels are generally encoded onto different regions of the code space. Recall also Figure 2.14 (b) which is of a similar nature.

Keeping in mind that one application of such data reduction is to help separate the data, it is evident that as model complexity increases (moving right along the displays of the figure), somewhat better separation occurs in the data. In particular, compare (d) based on the test set using PCA, and (f) based on the test set using the deep autoencoder. Refer also to Figure 3.12 which illustrates the reconstruction effect for various types of autoencoders (here $m = 30$). In terms of reconstruction, it is also evident in this case that more complex models exhibit better reconstruction ability.

Applications and Architectures

We have already seen the archetypical autoencoder application, namely data reduction. Yet there are many more applications and associated architectures of autoencoders. We now discuss some of these. We first consider various ways in which data reduction can be employed to help with additional machine learning activities. We then discuss *de-noising* which is a different application to data reduction. We close the chapter with ways in which interpolation in the *latent space* of the bottleneck are useful. The discussion here is just a brief summary and more information is suggested in the notes and references at the end of the chapter.

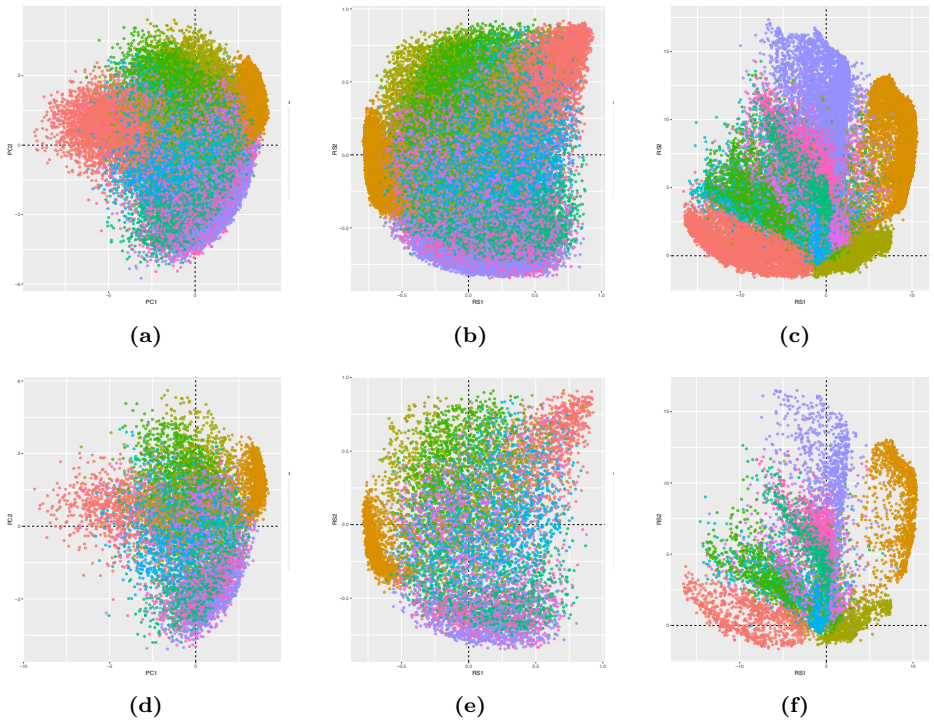


Figure 3.14: Autoencoders applied to MNIST with each of the 10 digits color coded. (a)–(c) are for the training set and (d)–(f) are for the test set. In the first column with (a) and (d) we use PCA. In the middle column with (b) and (e) we use a shallow non-linear autoencoder. In the last column with (c) and (f) we use a deep autoencoder.

In terms of uses of data reduction, let us consider both supervised and unsupervised learning. In the supervised setting, whenever a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ is used where the dimension of $x^{(i)}$ is p, one may attempt to reduce the dimension of the input features to $m < p$ and obtain a dataset $\widetilde{\mathcal{D}} = \{(\tilde{x}^{(1)}, y^{(1)}), \ldots, (\tilde{x}^{(n)}, y^{(n)})\}$. With such a dataset, training a supervised learning model to predict y in terms of $\tilde{x} \in \mathbb{R}^m$ may sometimes yield much better performance than using the original $x \in \mathbb{R}^p$. This process requires training an autoencoder as well as training of the model. Then in production with unseen data, for any $x^\star$ we first use the trained encoder to obtain to obtain $\tilde{x}^\star$. We then use the trained model on $\tilde{x}^\star$ to obtain $\hat{y}$. Note that with such an activity we do not use the decoder per-se.

To take this application of data reduction even further, consider a situation with $y^{(i)} \in \mathbb{R}^q$ where q is not small (high dimensional labels such as segmentation maps on images for example). In this case one may encode both the feature vector and the label, each with their own autoencoder, to obtain encoded data of the form $\widetilde{\mathcal{D}} = \{(\tilde{x}^{(1)}, \tilde{y}^{(1)}), \ldots, (\tilde{x}^{(n)}, \tilde{y}^{(n)})\}$, say with $\tilde{x}^{(i)} \in \mathbb{R}^{m_p}$, $\tilde{y}^{(i)} \in \mathbb{R}^{m_q}$, $m_p < p$, and $m_q < q$. One may then train a supervised model using this $\widetilde{\mathcal{D}}$. Such a model predicts $\hat{\tilde{y}}^\star \in \mathbb{R}^{m_q}$ for each encoded feature vector $\tilde{x}^\star \in \mathbb{R}^{m_p}$. With this setup, when the model is used in production, one first observes a feature vector $x^\star$, then uses the feature vector encoder to create $\tilde{x}^\star$, then uses the model to predict $\hat{\tilde{y}}^\star$, and finally uses the label decoder to obtain $\hat{y}^\star$. Hence with such an activity, in production the encoder of the feature vector and the decoder of the label are used.

In terms of unsupervised learning, there are many secondary ways to use applications of autoencoder based data reduction as well. As one example, assume that we wish to cluster images. A naive approach may be to treat each image as a vector and then use an algorithm such as K-means (see Section 2.6) to cluster the vectors. This approach has many drawbacks since the distance between images is based on the exact locations (indices) of pixels within an image. For example two images of the same object with slightly different camera locations would be generally "far" when comparing the Euclidean distance between the associated vectors. A more suitable approach is to use an autoencoder for data reduction of the image to a low dimension and then to perform clustering on the reduced dimensional codes. See for example Figure 3.14 (f). Here $m = 2$ and with such encoding, clustering the vectors on the two dimensional code space will generally work well.

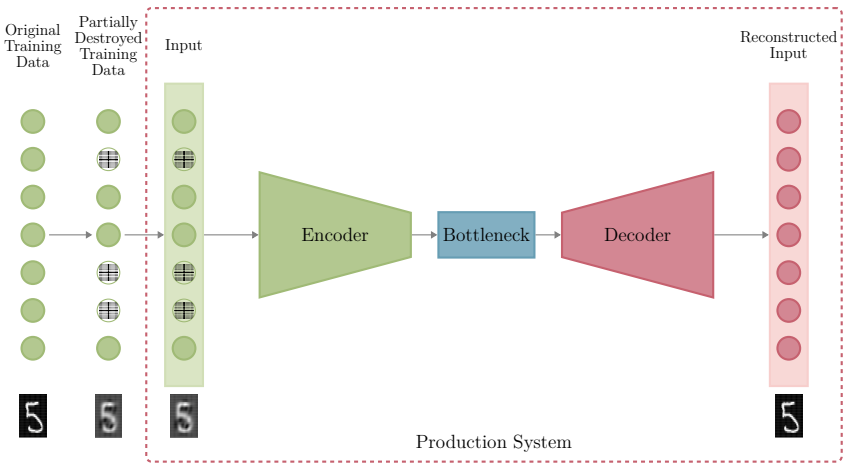


Figure 3.15: A denoising autoencoder where during training partially destroyed (noised) data is fed into the input. The production system is the trained autoencoder.

We now move away from data reduction and consider an architecture called the *denoising autoencoder* which is illustrated in Figure 3.15. This model learns to remove noise during the reconstruction step for noisy input data. It takes in partially corrupted input and learns to recover the original denoised input. It relies on the hypothesis that high-level representations are relatively stable and robust to entry corruption and that the model is able to extract characteristics that are useful for the representation of the input distribution.

In terms of architectures, denoising autoencoders exhibits a similar architecture to the usual autoencoder as they involve an encoder $f^{[1]}_{\theta^{[1]}}(\cdot)$ and decoder $f^{[2]}_{\theta^{[2]}}(\cdot)$. However a key difference is that during the learning process it is trained on noisy samples and the loss function is modified. First the initial input x is corrupted into $\breve{x}$ via some predefined stochastic mapping performing the partial destruction. In practice the main corruption mappings include adding Gaussian noise, masking noise (a fraction of the input chosen at random for each example is forced to 0), or salt-and-pepper noise (a fraction of the input chosen at random for each example is set to its minimum or maximum value with uniform probability).

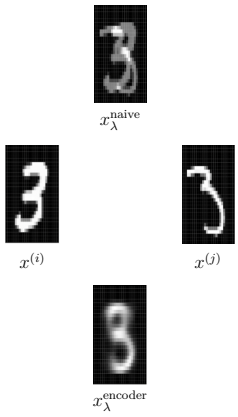


Figure 3.16: Interpolation of images with $\lambda = 1/2$. The left and right images of $x^{(i)}$ and $x^{(j)}$ are raw images. The top image is the naive interpolation and the bottom image is obtained via interpolation using an autoencoder.

Now with the corrupted input $\breve{x}$, the autoencoder encodes and decodes $\breve{x}$ in the usual way yielding a reconstructed $\hat{\breve{x}}$ via,

$$f_\theta(\breve{x}) = f^{[2]}_{\theta^{[2]}} \circ f^{[1]}_{\theta^{[1]}}(\breve{x}) \longrightarrow \hat{\breve{x}}.$$

However, instead of using a loss like (3.36) which would compare $\breve{x}$ and $\hat{\breve{x}}$ we modify the loss function for each observation i to be $C_i(\theta) = \|x^{(i)} - \hat{\breve{x}}^{(i)}\|^2$. Hence during training, we seek parameters θ that try to remove the noise as much as possible. When used in production on an unseen data sample $x^\star$, the data corruption step is clearly not employed. Instead, at this point the autoencoder output $f_\theta(x^\star)$ is conditioned to generate outputs $\hat{x}^\star$ that are denoised.

As a third general application of autoencoders, let us consider *interpolation on the latent space*. Take $x^{(i)}$ and $x^{(j)}$ from $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$ and consider the convex combination

$$x_\lambda^{\text{naive}} = \lambda x^{(i)} + (1-\lambda)x^{(j)},$$

for some $\lambda \in [0, 1]$. Ideally such an x_λ^{naive} may serve as a weighted average between the two observations where λ clearly captures which of the observations has more weight. However with most data, such arithmetic on the associated feature vectors is too naive and often meaningless for similar reasons to those discussed above in the context of K-means clustering of images. For example, in the top image of Figure 3.16 we see such interpolation with $\lambda = 1/2$.

When considering the latent space representation of the images it is often possible to create a much more meaningful interpolation between the images. An example is in the bottom image of Figure 3.16. To carry out this interpolation we train an autoencoder and then encode $x^{(i)}$ and $x^{(j)}$ to obtain $\tilde{x}^{(i)}$ and $\tilde{x}^{(j)}$. We then interpolate on the codes, and finally decode $\tilde{x}_\lambda$ to obtain an interpolated image. That is, using the notation of (3.37) and omitting parameter subscripts we have,

$$x_\lambda^{\text{encoder}} = f^{[2]}\Big(\lambda f^{[1]}(x^{(i)}) + (1-\lambda) f^{[1]}(x^{(j)})\Big).$$

This property of latent space representations and the ability to interpolate with these representations, also plays a role in the context of generative models discussed in Chapter 8. At this point let us mention that one potential application of such interpolation is for design purposes, say in art or architecture, where one chooses two samples as a starting point and then uses interpolation to see other samples lying "in between".

Notes and References

A comprehensive book on applied logistic and multinomial regression in the context of statistics is [187] where one can find out about *confidence intervals*, *hypothesis tests*, and other aspects of inference. The statistical origins of logistic regression are most probably due to Chester Ittner Bliss whom developed the related *probit regression model* in the 1930's; see [42]. Probit was initially used for bioassay studies. See also [95] for an historical account where the development of the logistic function (the sigmoid function) as a solution to logistic differential equations is presented. In the context of machine learning, logistic regression may be viewed as one example of a probabilistic generative model, see for example Section 4.2 of [39].

These days, in the context of deep learning, logistic regression is treated as the simplest general non-linear (shallow) neural network. However, the original simplest (non-linear) neural network is not actually logistic regression but rather the *perceptron*; see [353]. That model, created by Rosenblatt in 1958, differs from logistic regression in that the activation function is the ; see (5.16) in Chapter 5. With this activation function, gradient based optimization is not possible. Yet Rosenblatt introduced the which finds a classifier in finite time when the data is *linearly separable*; see also [314].

The phrase "softmax" for (3.25) started to be used in the machine learning community at around 1990; see [63]. The presentation of both logistic regression and multinomial regression as specific cases of generalized linear models is described in [106]. We also mention several variants of these models. These include *Dirichlet regression* which is used to analyse continuous proportions and compositional response data; see [108], [149], and [178]. Also *ordinal regression* is relevant for predicting an ordinal response variable. Examples of ordinal models are the ordered logit and ordered probit models. A comprehensive reference for analysing categorical variables is [5] and for ordinal variables see [6]. See also chapters 10 and 12 of [179]. Related terms from the world of machine learning are *ranking learning*, also known as *learning to rank*. This field builds on *ordinal regression*; see for example [374]. See [91] as a general reference for inference using maximum likelihood estimation in the context of biostatistics as well as the classic theoretical statistics book [94]. Further, see [44] for additional computational aspects including optimization of multinomial regression using second-order methods.

Our focus on autoencoders in this chapter is mostly on the most popular architecture where the hidden layer presents a lower number of units than the inputs. This specific architecture is called *undercomplete* as opposed to the *overcomplete* case presenting more hidden units than the input. We mention that autoencoders in general, and specifically overcomplete architectures, go hand in hand with regularisation, achieving sparse representations of the code; see for example chapter 14 of [142]. A general overview of autoencoder applications and architectures is in [74]. Specifically, different variants of autoencoders have recently emerged including *sparse autoencoders*, *contractive autoencoders*, *robust autoencoders*, and *adversarial autoencoders*. One particular class explored in Chapter 8 is *variational autoencoders*. For relationships between PCA and autoencoders see [332] as well as the more classic works [21] and [53].