Neural Networks - Notes

Matheus Rosso

Contents

Part I

The Elements of Statistical Learning

1 Neural networks (Chapter 11, pages 389-414)

- Neural network models construct linear combinations from original inputs X, named derived features Z, and then explain the response variable Y through nonlinear functions of those derived features $g(Z)$.

- **Projection pursuit regression (PPR):** this model is a special case of neural networks. Given an input vector $px1$ X, a response variable Y and parameters ω_m, consisting on a unit vector $(1xp)$ with $m \in \{1, 2, ..., M\}$, then the PPR model is defined as:

$$f(X) = \sum_{m=1}^{M} g_m(X\omega_m) \tag{1}$$

  Thus, the PPR model (1) is additive in derived features $V_m = X\omega_m$, and both functions $g_m(.)$ and parameters ω_m are estimated.

  - The function $g_m(X\omega_m)$ is called a *ridge function*. The derived features $V_m = X\omega_m$ are projections of X onto the space defined by parameters ω_m. The PPR model has the goal of finding the directions ω_m that best fits the data.
  - The PPR model, though simple in its form, is very general, since implies in a large set of different models. If M is taken to be arbitrarily large, then, for appropriate choices of functions $g_m(.)$, the PPR model (1) approximates well any continuous function in the space $\mathbb{R}^p$. Consequently, this class of models is named as a *universal approximator*.
  - As applies for neural networks in general, PPR models are not useful for interpretation, being most suited for prediction purposes. Actually, this does not hold for $M = 1$, whose PPR model is called a *single index model*.
  - Given training data $\{(x_i, y_i)\}_{i=1}^{N}$ and assuming a squared-error loss, the estimation of a PPR model should solve:

$$\underset{\{(g_m,\omega_m)\}_{m=1}^{M}}{\operatorname{argmin}} \sum_{i=1}^{N} \Big(y_i - \sum_{m=1}^{M} g_m(x_i\omega_m)\Big)^2 \tag{2}$$

– Estimation of (2) requires complexity constraints to avoid overfitting issues. An alternative to estimate $g_m(.)$ with appropriate complexity handling is by using *smoothing splines*. Firstly, assuming $M = 1$, smoothing splines choose the function $g(.)$ defined over derived features $v_i = x_i\omega$ through the following minimization problem:

$$\sum_{i=1}^{N}(y_i - g(v_i))^2 + \lambda \int g''(t)^2 dt \tag{3}$$

– Given an estimate of $g(.)$, as produced through smoothing splines, the *Gauss-Newton search* is applied for estimating ω. From a current estimate ω_{old}, the following approximation is defined:

$$g(x_i\omega) \approx g(x_i\omega_{old}) + g'(x_i\omega_{old})x_i(\omega - \omega_{old}) \tag{4}$$

Combining (2) and (4):

$$\sum_{i=1}^{N}(y_i - g(x_i\omega)^2 \approx \sum_{i=1}^{N} g'(x_i\omega_{old})^2\Big[\Big(x_i\omega_{old} + \frac{y_i - g(x_i\omega_{old})}{g'(x_i\omega_{old})}\Big) - x_i\omega\Big]^2 \tag{5}$$

Minimizing the right side of (5) is equivalent to fitting a weighted least squares regression of $x_i\omega_{old} + (y_i - g(x_i\omega_{old}))/g'(x_i\omega_{old})$ against the inputs x_i, where the weights are given by $g'(x_i\omega_{old})^2$ and there is no intercept term.

– From this estimation, a updated vector ω_{new} is derived. Since the first estimation of $g(.)$ has used an initial guess for ω, smoothing splines can be implemented again in order to generate a new estimate for $g(.)$. These two steps are iterated until some convergence criterion is satisfied.

– When $M > 1$, the PPR model is estimated in a forward stagewise manner, where each pair (g_m, ω_m) is added to the sum (1) at each stage.

– The definition of M can be based either on early stopping, when the next term m does not sufficiently improve the fit, or on cross-validation.

- **Neural networks:** these models are structured similarly to PPR model, but in a more general manner. Given an input vector X and K numerical response variables Y_k, or K classes for a single categorical variable, the *single hidden layer back-propagation network*, or *single layer perceptron* is given by:

$$Z_m = \sigma(\alpha_{m0} + X\alpha_m) \text{ for } m \in \{1, 2, ..., M\} \tag{6}$$

$$T_k = \beta_{k0} + Z\beta_k \text{ for } k \in \{1, 2, ..., K\} \tag{7}$$

$$f_k(X) = g_k(T) \text{ for } k \in \{1, 2, ..., K\} \tag{8}$$

Where $Z = (Z_1, Z_2, ..., Z_M)$ and $T = (T_1, T_2, ..., T_K)$.

- The *activation function* $\sigma(.)$ may be the sigmoid function or the Gaussian radial basis function, which gives rise to the *radial basis function network*.
  - Neural network estimation using the Gaussian radial basis function as $\sigma(.)$ increases the difficulty of weights estimation.
- Note that equation (6) provides the linear combinations from the original inputs, so each Z_m is a derived feature. From this equation, weights α_{ml} emerge, being specific to each pair of derived feature Z_m and input X_l.
- Equation (7) combines all derived features with weights β_{km} that are specific to each pair of Y_k and derived feature Z_m. Consequently, if $\sigma(.)$ is nonlinear, then the linear combinations of original inputs are transformed non-linearly already when the derived features are constructed.
- Equation (8) uses functions $g_k(.)$ for each Y_k that process all linear combinations T of derived features Z to produce an output $f_k(X)$, which may be either the prediction for numerical response variable Y_k or the probability of Y to belong to class k.
  - For regression problems, generally $g_k(T) = T_k$.
  - For classification problems, each function $g_k(.)$ may be given by the *soft-max function*:

$$g_k(T) = \frac{\exp(T_k)}{\sum_{l=1}^{K} \exp(T_l)} \tag{9}$$

- If all original inputs X constitute a visible layer, equation (6) represents a **hidden layer**, while equation (8) points to an **output layer**. With only one hidden layer, as model based on (6)-(8), figure 1 illustrates this single layer neural network:

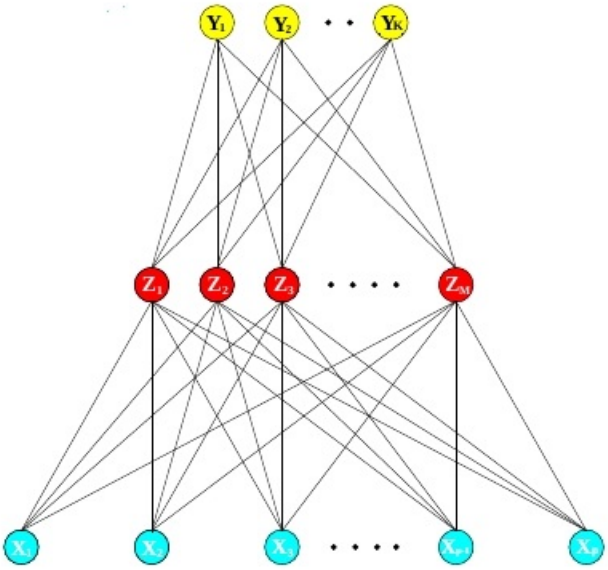

FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Figure 1.1: Single layer neural network

– The intercepts α_{m0} and β_{k0} are understand as *bias* introduced into the neural network.

– Considering $\sigma(.)$ to be the identity function, then the single layer neural network model consists on a linear model in the inputs. For example, given $g_k(T)$ as in (9), then the model is equivalent to a multinomial logistic regression model.

– Considering $\sigma_(.)$ to be the sigmoid function, then there is a nonlinear generalization of the linear model, with the activation rate (parameter s in $\sigma(sv)$) controlling for how linear $\sigma(.)$ is around zero (the larger s, the steeper the activation function).

  * If $\sigma(.)$ is the sigmoid and the loss function is the deviance, then each hidden unit Z_m consists on a linear logistic regression model.

– The single layer neural network can be seen as a PPR model by combining both equations (6) and (7):

$$g_m(X\omega_m) = \beta_m\sigma(\alpha_{m0} + X\alpha_m) = \beta_m\sigma(\alpha_{m0} + ||\alpha_m||(X\omega_m)) \tag{10}$$

Where $\omega_m = \alpha_m/||\alpha_m||$. Consequently, the linear combinations of derived features are indeed nonlinear functions of linear combinations of original inputs, provided that $\sigma(.)$ is nonlinear.

* Note that $\beta_m \sigma(\alpha_{m0} + ||\alpha_m||(X\omega_m))$ is less complex than a general function $g_m(X\Omega_m)$. Thus, a neural network may combine a large number M of those functions, while PPR generally uses fewer terms.

- **Fitting neural networks:** as discussed above, a single layer neural network has the following parameters, or weights to be estimated:

$$\begin{aligned} &\{(\alpha_{m0}, \alpha_m)\}_{m=1}^{M}, \text{ composing } M(p+1) \text{ weights} \\ &\{(\beta_{k0}, \beta_k)\}_{k=1}^{K}, \text{ composing } K(M+1) \text{ weights} \end{aligned} \tag{11}$$

The estimation of such parameters, gathered into a single vector θ, must minimize a given loss function. For regression, squared-error loss can be used, implying the following error function:

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2 \tag{12}$$

For classification, one alternative is to use cross-entropy (deviance):

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(f_k(x_i)) \tag{13}$$

  - Globally minimizing (12) or (13) may lead to overfitting. Consequently, regularization is required to increase predictive performance over test data, either explicitly, through the inclusion of a penalty term into the error function, or implicitly, through early stopping.
  - The method of minimizing (12) or (13) is **gradient descent**, or **back-propagation**. Assuming for illustration the squared-error loss, and supposing that $f_k(x_i)$ follows equations (6)-(8) with $g_k(T) = g_k(T_k)$, then the derivatives of (12) are:

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i)) g_k'(z_i \beta_k) z_{im} \tag{14}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i)) g_k'(z_i \beta_k) \beta_{km} \sigma'(x_i \alpha_m) x_{il} \tag{15}$$

From the current estimates of $\beta_{km}^{(r)}$ and $\alpha_{ml}^{(r)}$, the gradient descent updates as follows:

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}} \tag{16}$$

$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}} \tag{17}$$

Where γ_r is the **learning rate** parameter. Equations (16) and (17) are simplified by the following definitions:

$$\delta_{ik} = -2(y_{ik} - f_k(x_i))g_k'(z_i\beta_k) \tag{18}$$

$$s_{im} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(z_i\beta_k)\beta_{km}\sigma'(x_i\alpha_m) \tag{19}$$

Therefore, using δ_{ik} and s_{im} in (14) and (15):

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ik} z_{im} \tag{20}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = s_{im} x_{il} \tag{21}$$

– Both terms δ_{ik} and s_{im} guide the updates at the output and hidden layers, respectively. Combining equations (18) and (19), the **back-propagation equations** emerge and are central throughout the estimation procedure:

$$s_{im} = \sigma'(x_i\alpha_m)\sum_{k=1}^{K}\beta_{km}\delta_{ik} \tag{22}$$

Given these equations and initial guesses for $\beta_{km}^{(0)}$ and $\alpha_{ml}^{(0)}$, the *two-pass algorithm* calculates $\hat{f}_k(x_i)$ through (8) in the *forward pass*. Then, $\delta_{ik}^{(1)}$ follows from (18) and $s_{im}^{(1)}$ from (19) in the *backward pass*. Another round of definitions starts with (16), (17), (20) and (21) updating $\beta_{km}^{(1)}$ and $\alpha_{ml}^{(1)}$.

  * Similar results can be obtained with deviance loss function instead of squared-error loss.

– It is worth notice that updates (16) and (17) aggregates all training points at each iteration step, providing a *batch learning*. Differently, learning can occur with each observation at a time, thus updating $\beta_{km}^{(r)}$ and $\alpha_{ml}^{(r)}$ N times when a *training epoch* is finished.

– Concerning the learning rate γ_r, it can be optimized by minimizing the error function at each updating step. Besides, $\gamma_r \to 0$ as $r \to \infty$, configuring a *stochastic approximation*.

– Another fitting procedures may improve computational time as compared to this gradient descent method.

– It should be noticed the resemblance between gradient descent and steepest descent, discussed in the context of gradient boosting models.

- **Starting values:** initial guesses must be provided for β_{km} and α_{ml} in order to implement the two-pass algorithm. The closer the weights are to zero, the more linear the single layer neural network will be, because the lower the activation rate inside the sigmoid $\sigma(.)$, the more linear is the sigmoid function.

  - As a consequence, it is appropriate to define initial guesses close to zero, so that the model becomes increasingly more nonlinear as the weights are updated.
  - Initial weights equal to zero imply no updated as iterations take place. In the other hand, large initial weights may lead to poor solutions.

- **Regularization:** given the large number of parameters estimated by a neural network model, the regularization is particularly relevant for this statistical learning method, since the model will likely overfit at the global minimum of the error function.

  - The **early stopping** regularization method defines criteria for finishing iterations before the global minimum is reached. Considering initial weights close to zero, early stopping will result in a model more linear than that obtained without regularization. This occurs because weights have no time to increase far from zero.

    * Early stopping should be implemented together with cross-validation or validation techniques to define when to stop iterations.

  - An explicit regularization method is given by **weight decay**, which introduces a penalty term into the error function, $R(\theta) + \lambda J(\theta)$, where $J(\theta)$ is similar to ridge penalty:

$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{ml}^2 \tag{23}$$

    The regularization parameter λ can be defined through cross-validation.

    * An option to (23) is provided by the **weight elimination penalty**:

$$J(\theta) = \sum_{km} \frac{\beta_{km}^2}{1+\beta_{km}^2} + \sum_{ml} \frac{\alpha_{ml}^2}{1+\alpha_{ml}^2} \tag{24}$$

      Penalty (24) penalizes more the weights sizes than (23).

    * *Hinton diagrams* produce data visualization to understand patterns on weights of different pairs of hidden layer-output (β_{km}) or of different pairs of input-hidden layer (α_{ml}).

        * The higher the number of hidden units, the higher should be the penalty parameter λ.

- It is crucial to standardize inputs to have mean zero and unit standard deviation before fitting neural networks. With standardized inputs, convenient initial weights are given by the range $[-0.7, 0.7]$.
- In general, the higher the number of hidden units, the better the neural network model, since too few hidden units may not be able to capture all nonlinear patterns in data, while regularization may handle excessive hidden units.
    - It is recommended a number of hidden units varying between 5 and 100, with the chosen number increasing together with the set of original inputs or the number of observations.
    - The number of hidden units M can be set based on cross-validation, even though it may be more appropriate to define a suitable large number for M and use cross-validation for the penalty parameter λ.
        * In general, given these two parameters M and λ, one of them must be defined in such a way that the model is not too much constrained, while the other should be defined through cross-validation.
- The number of hidden layers can be greater than one, which would increase model complexity. The use of multiple hidden layers allows to create hierarchical relationships among derived features.
- Since the error function $R(\theta)$ may be non-convex, this leads to the possibility of non-unique solutions. Consequently, the final estimates for β_{km} and α_{ml} depend on the initial guesses. There are three possibilities for handling this issue:
    - Estimate one model for each set of initial weights, then choosing the set that minimizes the error function.
    - Estimate different models for different sets of initial weights, then choosing the set that minimizes the error function.
    - Implement bagging re-sample technique, which not just estimates different models for different sets of initial weights, but also generates different bootstrap sub-samples from the available dataset.
- **Multiple hidden layers neural network:** one way to model neural networks with more than one hidden layer is to use **local connectivity**, which means that, in order to produce each unit in the

higher hidden layer, only a fraction of units from the lower hidden layer is used. Since several units of the lower hidden layer receive zero weight for the construction of a given unit of the higher hidden layer, local connectivity allows to increase model complexity at the same time that the number of parameters are restricted.

  - *Local connectivity with shared weights:* additional restrictions are imposed when all units from the higher hidden layer that use a given unit from the lower hidden layer are constrained to have the same weight for that unit in their construction. Neural networks of such kind are named *convolutional networks*.

- **Bayesian neural networks:** since neural networks are composed from smooth functions ($\sigma(.)$, $g_k(.)$), they are suitable to receive a Bayesian treatment, in which density functions play a key role.

  - Given training data (X_{tr}, y_{tr}), a data distribution $P(Y|X,\theta)$, and a prior distribution for the parameters $P(\theta)$, the posterior distribution of θ follows the Bayes rule:

$$P(\theta|X_{tr}, y_{tr}) = \frac{P(y_{tr}|X_{tr},\theta)P(\theta)}{\int P(y_{tr}|X_{tr},\theta)P(\theta)d\theta} \tag{25}$$

    For a given test data point X_{test}, the predictive distribution for Y_{test} is given by:

$$P(Y_{test}|X_{test}, X_{tr}, y_{tr}) = \int P(Y_{test}|X_{test},\theta)P(\theta|X_{tr}, y_{tr})d\theta \tag{26}$$

    Since (26) is intractable, it must be approximated. In fact, also (25) can not be directly assessed, thus leading to approximation methods, such as variational approximation. Considering the approximation for (26), Markov Chain Monte Carlo (MCMC) methods are available, and generates several different samples $\hat{\theta}$ for θ from (25) (or, from its variational approximation). Then, an average of $P(Y_{test}|X_{test},\hat{\theta})$ is taken from all different samples $\hat{\theta}$.

  - The Bayesian approach (MCMC, in particular) looks into the model space and averages the predictions of different models, giving more weight to those that are more likely to hold. Bagging and boosting work similarly, but instead of sampling parameter values, bagging re-samples the data and provides different estimates for the parameters, while boosting also holds the data fixed, but estimates models sequentially, adding them to compose an aggregate predictor.

* Bayesian models, bagging and boosting can be summarized by the following expression:

$$\hat{f}(X_{test}) = \sum_{l=1}^{L} w_l E(Y_{test}|X_{test}, \hat{\theta}_l) \tag{27}$$

For Bayesian models, $w_l = 1/L$ and $E(Y_{test}|X_{test}, \hat{\theta}_l)$ follows from (26). For bagging, $w_l = 1/L$ and $\hat{\theta}_l$ are derived from bootstrap estimations. For boosting, $w_l = 1$ and $\hat{\theta}_l$ is estimated in order to sequentially improve the fit.

- Neural networks are more effective for problems when prediction without interpretation is needed, and for the empirical contexts when there is a high signal-to-noise ratio.

Part II

Neural Networks and Deep Learning, Michael Nielsen

2 Using neural nets to recognize handwritten digits (Chapter 1)

- The most simple neural network is composed from a single and basic **perceptron**, which, in its turn, has the following components: *inputs* (x_1, x_2, ..., x_p), the combination of them, and an output. The combination of inputs is linear and based on *weights* w_1, w_2, ..., w_p, one for each input. The *output* is defined through a trigger given by the comparison between the linear combination of inputs and a *threshold* t:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq t \\ 1, & \text{if } \sum_j w_j x_j > t \end{cases} \tag{28}$$

- Equation (28) may be simplified by replacing the threshold by a *bias term* b, defined similarly to the weights as a parameter of the perceptron model:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j + b \leq 0 \\ 1, & \text{if } \sum_j w_j x_j + b > 0 \end{cases} \tag{29}$$

- Neural networks are composed from such perceptrons. Figure 2 illustrates a neural network with two (hidden) layers of perceptrons: a first that receives (original) inputs and whose outputs serve a second layer, and a second whose inputs are outputs from previous perceptrons and whose outputs ultimately lead to an observed output.

- See page 6 from chapter one for a discussion over perceptrons and NAND gates.

- In order to a neural network to learn from data so as it can predict response variables for test data points, the definition of output from linear combination of inputs should be smoother than (29), so that small changes in w_j or in b produce small changes in output. From such small changes, those parameters may adjust themselves so that accurate predictions can be made from original inputs, i.e., so that the neural network may learn from training data.

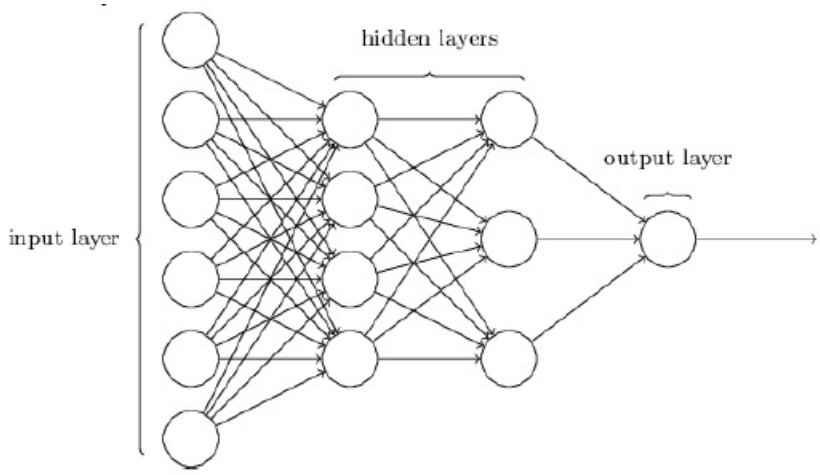

Figure 2.1: Neural network

  - Instead of perceptrons, another kind of artificial neurons are used for learning from data, the **sigmoid neurons**. Output from these neurons follows:

$$\text{output} = \begin{cases} 0, & \text{if } \sigma(\sum_j w_j x_j + b) \leq 0 \\ 1, & \text{if } \sigma(\sum_j w_j x_j + b) > 0 \end{cases} \tag{30}$$

    Where $\sigma(z) = 1/(1 + \exp(-z))$ is the sigmoid function.

  - Actually, sigmoid is just one alternative among a collection of different **activation functions**.

- As discussed and presented by figure 2, neural networks are composed from an input layer, hidden layers (one or more), and an output layer. This design of layers distinguishes **feedforward neural networks** from those who display *feedback loops*, in which the output of a layer serves as input for a predecessor layer, thus feeding the layer that had previously produced it. Neural networks with loops are named **recurrent neural networks**.

  - If the number of units entering into input and output layers depends on how much original inputs and outputs there are, the number of hidden layer units should be defined when modeling the neural network.

- As with any other statistical learning method, neural networks also learn from training data, while the resultant estimated model is expected to predict values of response variable y for test data points x, thus leading to the notation $y(x)$ to indicate the dependence of y to x.

– If $y(x) \in \mathbb{R}^K$, and denoting as $a \in \mathbb{R}^K$ the vector of outputs that follows from weights w and biases b, then the **cost function** $C(w, b)$ can be quadratically defined as:

$$C(w, b) = \frac{1}{2N} \sum_x ||y(x) - a||^2 \tag{31}$$

A neural network whose output is given by a learns when weights w and biases b are defined so as to minimize the cost function (31).

– Differentiating (31) with respect to w and b, setting the derivatives to zero, and analytically finding solutions is unfeasible when w and b are defined in high dimensional spaces. The learning algorithm used to minimize (31) is **gradient descent**, a method that also uses derivatives to try to reach a global minimum for the objective function $C(w, b)$.

– For a given layer l, changing slightly weights $\{w^l_{jk}\}^{J_l}_{j=1}$ (Δw^l_k, for a given neuron k of previous layer $l-1$) or biases $\{b^l_j\}^{J_l}_{j=1}$ produces a proportionate change in cost function ΔC, where this change depends on the gradient vector ∇C and it is approximated by:

$$\Delta C \approx \nabla C \cdot \Delta w^l_k = \frac{\partial C}{\partial w^l_{1k}}.\Delta w^l_{1k} + \frac{\partial C}{\partial w^l_{2k}}.\Delta w^l_{2k} + ... + \frac{\partial C}{\partial w^l_{J_l k}}.\Delta w^l_{J_l k} \tag{32}$$

$$\Delta C \approx \nabla C \cdot \Delta b = \frac{\partial C}{\partial b_1}.\Delta b_1 + \frac{\partial C}{\partial b_1}.\Delta b_2 + ... + \frac{\partial C}{\partial b_{J_l}}.\Delta b_{J_l} \tag{33}$$

Where (32) summarizes weights w^l_{jk} for a hidden unit j of hidden layer l, while (33) indicates bias b^l_j for hidden unit j of hidden layer l.

– The objective is to produce variations Δw and Δb so that $\Delta C < 0$. Consequently, a good guess is to induce:

$$\Delta w^l_{jk} = -\eta \frac{\partial C}{\partial w^l_{jk}} \tag{34}$$

$$\Delta b^l_j = -\eta \frac{\partial C}{\partial b^l_j} \tag{35}$$

Where $\eta > 0$ is small and defined as the **learning rate**. From (32) and (33), follows the *gradient descent update rule*:

$$w^l_{jk} \to w^{l'}_{jk} = w^l_{jk} - \eta \frac{\partial C}{\partial w^l_{jk}} \tag{36}$$

$$b^l_j \to b^{l'}_j = b^l_j - \eta \frac{\partial C}{\partial b^l_j} \tag{37}$$

- For fixed small step sizes $||\Delta w_k^l|| = \epsilon > 0$, $\Delta w_k^l = \eta \nabla C$ with $\eta = \epsilon / ||\nabla C||$ produces the highest possible (approximated) decreasing in C.
  - Therefore, gradient descent produces small variations in the directions of w_k and b that reduce C the most.
- Some modifications of gradient descent use hessian matrices of C in order to improve the minimization process.
- Since from (31) $C = (1/N) \sum_x C_x$, as the number of training data points increases, the complexity of updating w_{kj} and b_k with (36) and (37) also increases, and actually can become very costly to compute. This because the gradient of C is given by:

$$\nabla C = \frac{\sum_x \nabla C_x}{N} = \Big((1/N) \sum_x \frac{\partial C_x}{\partial w_{1k}^l}, (1/N) \sum_x \frac{\partial C_x}{\partial w_{2k}^l}, ..., (1/N) \sum_x \frac{\partial C_x}{\partial w_{J_l k}^l} \Big) \tag{38}$$

  Consequently, for each vector of p weights, N derivatives must be calculated at each iteration.
- **Stochastic gradient descent** randomly picks S training data points to estimate the gradient $\nabla C = (\partial C / \partial w_{1k}^l, ..., \partial C / \partial w_{J_l k}^l)$. Thus, if $\nabla C = (1/N) \sum_x \nabla C_x$, then:

$$\nabla C = \frac{\sum_x \nabla C_x}{N} \approx \frac{\sum_x \nabla C_x}{S} \tag{39}$$

  This random sample $\{x_1, x_2, ..., x_S\}$ from training data is known as a *mini-batch*.
- Given (39), update rules (36) and (37) change to:

$$w_{jk}^l \to {w_{jk}^l}' = w_{jk}^l - \frac{\eta}{S} \sum_x \frac{\partial C_x}{\partial w_{jk}^l} \tag{40}$$

$$b_j^l \to b_j^{l\prime} = b_j^l - \frac{\eta}{S} \sum_x \frac{\partial C_x}{\partial b_j^l} \tag{41}$$

- When all training data points are used after successive random picked mini-batches, an *epoch* of training is finished.
- It should be noticed that dividing terms by N or S is not necessary. Indeed, to implement *online learning*, in which parameters are updated at real time as new observations come in, there is no way how to precise the number of training data points, thus eliminating N and S scale components.

- From the above discussion and from code development towards neural network design, it has emerged the following set of **hyper-parameters**

- Learning rate η.
- Number of hidden layers H.
- Number of neurons in each hidden layer, J_h for $h \in \{1, 2, ..., H\}$.
- Number of training epochs.
- Size of mini-batches.

While the first three concern the neural network architecture, the last two are related to the fitting procedure. All of them, however, are named hyper-parameters since they must be specified previously to the estimation of weights and biases, these known as *parameters*.

- **Deep learning:** a neural network composed from multiple hidden layers is called *deep neural network*. If a single layer neural network can be understand as decomposing a problem into several different sub-problems, each focusing on some aspects of the data, then a deep neural network can be seen as decomposing each sub-problem into a new collection of sub-problems, thus increasing the level of complexity and abstraction in inputs processing.
  - Fitting deep neural networks involves techniques improved upon stochastic gradient descent and backpropagation.

3 How the backpropagation algorithm works (Chapter 2)

- Backpropagation provides a way of computing $\partial C/\partial w$ and $\partial C/\partial b$ so that weights and biases can be adjusted appropriately, following gradient descent update rule that tries to minimize overall cost function. Therefore, backpropagation and gradient descent compose the core of neural network fitting.
- **Matrix notation:** w^l_{jk} denotes the weight connecting neurons k from layer $l-1$ and j from layer l. Similarly, b^l_j expresses the bias of neuron j in layer l. Given an activation function $\sigma(.)$, a^l_j is named activation of neuron j from layer l and follows:

$$a^l_j = \sigma(z^l_j) = \sigma\Big(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\Big) \tag{42}$$

$$a_l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l) \tag{43}$$

Where z_j^l is the weighted input, k refers to neurons in layer $l-1$, w^l is a J_lxK_{l-1} matrix of weights, a^{l-1} is a vector K_{l-1}x1, and b^l a J_lx1 vector of biases.

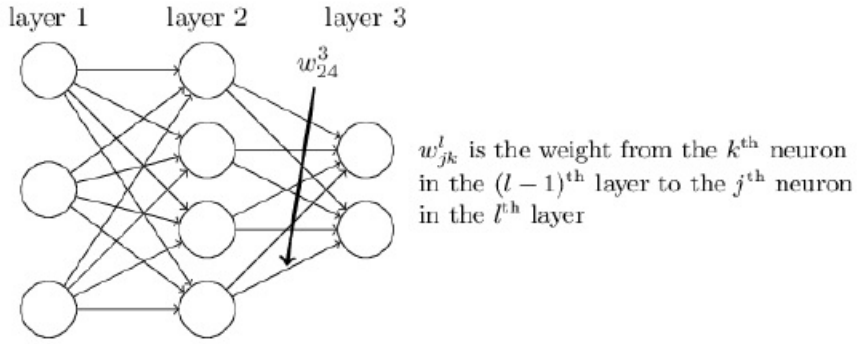

Figure 3.1: Weight w^l_{jk}

- For deriving backpropagation, the cost function should satisfy two assumptions: first, the overall function C is supposed to be an average of cost functions for data points C_x, consequently, $C = (1/N)\sum_x C_x$; second, overall and individual costs should be able to be expressed as a function of activation of output layer $C = C(a^L)$.

- **Fundamental equations for backpropagation:** for each neuron and each layer, an intermediate quantity δ_j^l can be defined to help deriving the fundamental equations:

$$\delta_j^l = \frac{\partial C_x}{\partial z_j^l} \tag{44}$$

  Thus, δ_j^l indicates how changing weighted inputs impacts cost function.

  – First equation defines δ_j^L for output layer: from definition (44):

$$\delta_j^L = \frac{\partial C_x}{\partial z_j^l} = \sum_k \frac{\partial C_x}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}$$

  Where above k refers to neurons in output layer L, then, since weighted input z_j^L only affects a_j^L:

$$\delta_j^L = \frac{\partial C_x}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

$$\delta_j^L = \frac{\partial C_x}{\partial a_j^L}\sigma'(z_j^L) \tag{45}$$

In matrix-form:

$$\delta^L = \nabla_a C_x \circ \sigma'(z^L) \tag{46}$$

– Second equation defines δ_j^l for non-output layers as a function of δ_k^{l+1}: also following definition (44):

$$\delta_j^l = \frac{\partial C_x}{\partial z_j^l} = \sum_k \frac{\partial C_x}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Where k refers to layer $l+1$ and j to layer l. From definition (42):

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

Thus:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

Consequently:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{47}$$

In matrix-form:

$$\delta^l = (w^{l+1})^T \delta^{l+1} \circ \sigma'(z^l) \tag{48}$$

Since w^{l+1} has as many rows as there are neurons in layer $l+1$ and one column for each neuron in layer l, then (47) in matrix-form requires the transposition of w^{l+1}.

– Third equation defines partial derivatives of C_x with respect to biases of layer l as a function of δ^l. From (42):

$$\frac{\partial C_x}{\partial b_j^l} = \frac{\partial C_x}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l}$$

Therefore:

$$\frac{\partial C_x}{\partial b_j^l} = \delta_j^l \tag{49}$$

– Fourth and last fundamental equation defines partial derivatives of C_x with respect to weights of layer l as a function of both δ^l and a^{l-1}. From (42):

$$\frac{\partial C_x}{\partial w_{jk}^l} = \frac{\partial C_x}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l}$$

Then:

$$\frac{\partial C_x}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \tag{50}$$

- Equation (50) indicates that if a_k^{l-1} is small, then for any neuron j in layer l its weight will have minor effects on cost C_x. In this case, w_{jk}^l will learn slowly, since it changes only a few at each update (36) or (40).
- Equations (47) and (50) suggest that slow learning also applies when δ_j^l is near zero, which may be due to $\sigma(z_j^l)$ approaching 1 or 0 and to $\sigma(.)$ having a format similar to that of sigmoid. The same applies for bias learning, replacing (49) by (50).
- In particular, (45) and (50) reveal that slow learning may occur when output neuron is saturated, meaning either that $\sigma(z_j^L)$ is near 1 or 0.
- Intuitively, what comments above mean is that when a_j^l approaches either 1 or 0, then w_{jk}^l, for any k from layer $l-1$, has little information from which to learn what direction to take and then updates its value. In the same way, if a_k^{l-1} is small enough, then w_{jk}^l, for all j in layer l, will also have few information to be updated.
- By designing activation functions to have a shape different from that of sigmoid, slow learning may be avoided. Rectified-linear unit (ReLU) is an alternative for neurons in hidden layers, since such function will not saturate, preventing slow learning.
- In short, backpropagation is constructed upon equations (45), (47), (49) and (50).
- Note: equations (45) and (47) are structurally the same as (18) and (19), respectively.

- **Backpropagation algorithm:**

1. *Inputs*: from input x, compute activation a^1 of input layer.
2. *Feedforward*: for each $l \in \{2, 3, ..., L\}$, calculate weighted inputs $z^l = w^l a^{l-1} + b^l$ and activations $a^l = \sigma(z^l)$.
3. *Output error* δ^L: define it through (45), $\delta^L = \nabla_a C \circ \sigma'(z^L)$.
4. *Backpropagation of error*: for each $l \in \{L-1, L-2, ..., 2\}$, compute $\delta^l = (w^{l+1})^T \delta^{l+1} \circ \sigma'(z^l)$ as given by equation (47).
5. *Gradient of cost function*: for all weights and biases, define $\partial C_x / \partial w_{jk}^l = a_k^{l-1} \delta_j^l$ and $\partial C_x / \partial b_j^l = \delta_j^l$ as given by equations (49) and (50).

- Step 4 that defines a backward movement is a consequence of the assessment of how changes in w_{jk}^l or b_j^l affect z_j^l and, then, a_j^l. These changes, in their turn, affect $z_{j'}^{l+1}$ and $a_{j'}^{l+1}$ for all

neurons j' in layer $l+1$, which requires a proper adjustment of $w^{l+1}_{j'j}$ and $b^{l+1}_{j'}$. Backpropagation stars from the end of this chained relationship and moves backwards until the input layer is reached.

- Steps above hold for a given data point x, but can be appropriately modified to apply for data batches or mini-batches.

1. Consider a set of data points x.
2. For each input x:
   (a) *Input layer activation*: calculate $a^{x,1}$.
   (b) *Feedforward*: for each $l \in \{2, 3, ..., L\}$, compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.
   (c) *Output error*: from equation (45), $\delta^{x,L} = \nabla_a C_x \circ \sigma'(z^{x,L})$.
   (d) *Backpropagation*: for each $l \in \{L-1, L-2, ..., 2\}$ and from (47), calculate:

$$\delta^{x,l} = (w^{l+1})^T \delta^{x,l+1} \circ \sigma'(z^{x,l}) \tag{51}$$

3. *Gradient descent*: for each $l \in \{L, L-1, ..., 2\}$, apply update rules using equations (49) and (50):

$$w^l \to w^l - \frac{\eta}{S} \sum_x \delta^{x,l} (a^{x,l-1})^T \tag{52}$$

$$b^l \to b^l - \frac{\eta}{S} \sum_x \delta^{x,l} \tag{53}$$

- In addition to this algorithm, two others must be added to implement neural network fitting: one that loops over mini-batches and a second that loops over epochs of training.

- Backpropagation not only presents a relatively fast way of estimating all weights and biases for a neural network, but also is equivalent to an intuitive way of calculating the effect on cost function $C(a)$ of a small change Δw^l_{jk} in any weight w^l_{jk} (or bias).

  - Given the neural network structure, the small change Δw^l_{jk} is expected to produce a change in activation a^l_j, Δa^l_j, which is approximated by:

$$\Delta a^l_j \approx \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk} \tag{54}$$

– Then, all activations in the next layer $l+1$ may also change. Focusing on a given neuron whose activation is a_q^{l+1}:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l \tag{55}$$

Combining both (54) and (55):

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l \tag{56}$$

– Since that change Δa_q^{l+1} is expected to produce further changes in all activations from layer $l+2$, it is possible to resume a given path of such chained perturbations all the way to the final cost function $C(a)$ through the following expression:

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l \tag{57}$$

– Summing over all possible paths of how a change in w_{jk}^l may ultimately affect C:

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l \tag{58}$$

Considering the indirect way of expressing the total change in C caused by a change in w_{jk}^l:

$$\Delta C \approx \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l \tag{59}$$

Then:

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \tag{60}$$

– Therefore, the overall rate of change in C caused by a change in w_{jk}^l is equal to the sum of rates of change in C for all paths through which the change in w_{jk}^l may affect C.

– It turns out that backpropagation is a short and fast way of computing (60), the overall paths rate of change in C given a small change in w_{jk}^l.

4 Improving the way neural networks learn (Chapter 3)

- From a basic setting where all neurons of layer $l-1$ enter into the weighted inputs of layer l and where quadratic cost function is used, neural networks may be improved by: changing cost function to cross-entropy (or to log-likelihood with softmax output layer); by implementing regularization; by wisely initializing weights; or by choosing good hyper-parameters.

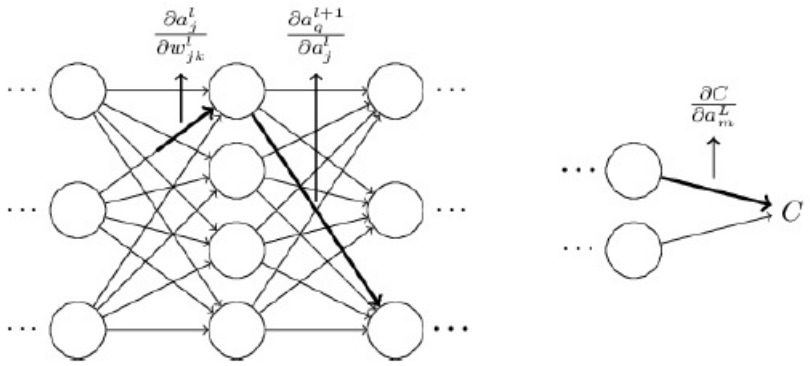

Figure 3.2: Single path of changes from w_{jk}^l to C

- **Cross-entropy cost function:** it was previously discussed how backpropagation may lead to slow learning, also named *learning slowdown*, which happens when a neuron saturates, so that $\sigma(z_j^l) \to 1$ or $\sigma(z_j^l) \to 0$, and depending on the shape of $\sigma(.)$, this imply $\sigma'(z_j^l) \to 0$. Considering the activation of an output neuron a_j^L, learning slowdown constitutes an effective issue when the saturation is based on a wrong value for the response variable.

  – Using equations (45) and (50):

  $$\frac{\partial C_x}{\partial w_{jk}^L} = \frac{\partial C_x}{\partial a_j^L} a_k^{L-1} \sigma'(z_j^L) \tag{61}$$

  And considering a quadratic cost function $C_x = (y(x) - a(x))^2/2$:

  $$\frac{\partial C_x}{\partial w_{jk}^L} = (a_j^L - y_j) a_k^{L-1} \sigma'(z_j^L) \tag{62}$$

  Then, instead of a rate of learning just proportionate to the error, with quadratic cost function the rate of learning also depends on $\sigma'(z_j^L)$, which may converge to zero as output neuron saturates.

  – Consequently, with quadratic cost function, large errors may not lead to fast learning, which is more desirable than a slow learning precisely when the neural network has a lot of information from what to learn and correct its weights and biases.

  – Given a neural network with a unique output neuron whose activation is $a^L = \sigma(z^L)$, then the *cross-entropy cost function* is given by:

  $$C = -\frac{1}{N} \sum_x [y \log(a^L) + (1 - y) \log(1 - a^L)] \tag{63}$$

Cross-entropy can be understood as a cost function since $C \geq 0$. Besides, with an activation a^L close to actual output y for most training data points, cross-entropy will converge to zero.

– Cross-entropy cost function avoids the occurrence of learning slowdown as a result of the absence of $\sigma'(z_j^L)$ in the expression for partial derivative of C with respect to weights and biases. For single output neuron:

$$\frac{\partial C}{\partial w_k^L} = -\frac{1}{N}\sum_x \Big(\frac{y}{\sigma(z^L)}\sigma'(z^L)a_k^{L-1} - \frac{1-y}{1-\sigma(z^L)}\sigma'(z^L)a_k^{L-1}\Big)$$

$$\frac{\partial C}{\partial w_k^L} = -\frac{1}{N}\sum_x \Big(\frac{y}{\sigma(z^L)} - \frac{1-y}{1-\sigma(z^L)}\Big)\sigma'(z^L)a_k^{L-1}$$

$$\frac{\partial C}{\partial w_k^L} = -\frac{1}{N}\sum_x \Big(\frac{(1-\sigma(z^L))y - (1-y)\sigma(z^L)}{\sigma(z^L)(1-\sigma(z^L))}\Big)\sigma'(z^L)a_k^{L-1}$$

$$\frac{\partial C}{\partial w_k^L} = -\frac{1}{N}\sum_x \Big(\frac{y-\sigma(z^L)}{\sigma(z^L)(1-\sigma(z^L))}\Big)\sigma'(z^L)a_k^{L-1}$$

$$\frac{\partial C}{\partial w_k^L} = \frac{1}{N}\sum_x (\sigma(z^L) - y)a_k^{L-1} \tag{64}$$

Expression (64) does not depend on $\sigma'(z^L)$, and thus a single output neural network does not suffer from learning slowdown (at least with respect to weights and biases from output layer). Moreover, (64) denotes that the rate at which the neural network learns is directly related to the error $\sigma(z^L) - y$. Similarly, for the bias of single output neuron:

$$\frac{\partial C}{\partial b^L} = \frac{1}{N}\sum_x (\sigma(z^L) - y) \tag{65}$$

– For multiple output neurons, cross-entropy cost function is given by:

$$C = -\frac{1}{N}\sum_x \sum_j [y_j \log(a_j^L) + (1-y_j)\log(1-a_j^L)] \tag{66}$$

Consequently:

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{N}\sum_x (\sigma(z_j^L) - y_j)a_k^{L-1} \tag{67}$$

$$\frac{\partial C}{\partial b_j^L} = \frac{1}{N}\sum_x (\sigma(z_j^L) - y_j) \tag{68}$$

– Therefore, when using sigmoid activation function, or any other activation function whose shape leads to the possibility of output neurons saturation, cross-entropy is a better choice than quadratic cost function.

    - A main source of saturation causing learning slowdown is bad initial guesses for weights and biases. This would lead to output activations far from actual outputs, then driving partial derivatives of cost function to zero when $\sigma'(z_j^L)$ is present and when $\sigma(z_j^L)$ has a shape as that of sigmoid function.
  - It can be shown that expressions like (67) and (68) are inherently related to a cost function with the cross-entropy form.

- **Softmax function:** another option to avoid learning slowdown is to use *softmax activation function* for neurons in output layer:

$$a_j^L = \frac{\exp(z_j^L)}{\sum_k \exp(z_k^L)} \tag{69}$$

  - As expression (69) shows, softmax function for output neurons makes possible to understand their activations as composing a probability distribution, since $a_j^L \geq 0$ and $\sum_j a_j^L = 1$. Consequently, for classification problems, a_j^L may be understood as the probability that a given data point belongs to class j.
  - When softmax function is used as activation for output neurons together with *log-likelihood cost function*, then again learning slowdown can be addressed leading to expressions for partial derivatives equivalent to (67) and (68).
    - If y is the correct label for a categorical response variable, then a_y^L denotes the activation corresponding to that class, and then the log-likelihood cost function is given by:

$$C = -\log(a_y^L) \tag{70}$$

  - The initialization of weights and biases for neurons in a softmax layer need not follow any randomization procedure, neither the first proposed or the alternative below, which considers characteristics of sigmoid function. Rather, the initialization of parameters for softmax neurons may define them as all equal to zero.

- **Overfitting:** overfitting occurs when the model fits very well to the training data but fails to generalize to new data points. The larger the number of free parameters a model has the more likely it is to suffer from overfitting.
  - Overfitting should be identified by checking test set accuracy, since this is the reference metric for predictive tasks. This detection is reinforced by comparing it against training data accuracy.

- When test data accuracy no longer improves, further epochs of training are unnecessary, irrespective of training data accuracy continuing to increase (evidence of overfitting) or not.
- **Early stopping:** at the final of each epoch of training, predictive accuracy is assessed on *validation data* held out apart from training and test data. If predictive accuracy saturates, training stops even before all predefined epochs of training are executed.
    - Saturation may be defined as predictive accuracy not improving more than some specified threshold for more than a given number of training epochs. Alternatively, saturation may be seen as predictive accuracy not improving over the best so far after a given number of training epochs.
- **Hold out method:** available data is split into training, validation and test data.
    - While the first is used for estimating all parameters of a given model, validation data is applied for trying out different sets of hyper-parameters of the estimation method, from which that set associated with best validation data performance is chosen.
    - Finally, test data predictive accuracy points to how well the model is expected to perform on independent new data points. Therefore, the choice of hyper-parameters is not affected by test data patterns, as it is the case when actual generalization is made.
    - Performance on test data may also be used for comparing different estimation methods, sets of original inputs, transformations on data, and so on.
    - When data length is not enough to generate training, validation and test data with sufficient number of observations, cross-validation with train-test split is an alternative.
- In a first-best scenario, overfitting is reduced by enlarging training dataset. When this is unfeasible, corrective methods should be implemented in addition to early stopping.

- **Regularization:** overfitting is caused by models which explores complexities of training data to such extent that they generalize poorly on data points not used during model estimation. Therefore, one possibility to solve the overfitting problem is to reduce the size of the neural network, either reducing the number of hidden layers or by excluding some of their neurons. Since this would affect negatively the predictive accuracy, it would be optimal to keep untouched the model structure. Regularization serves to accomplish this precise goal.
    - **Weight decay**, or **L2 regularization** modify cost function to penalize the size of weights,

working in a continuous manner to reduce how training data affect final outputs:

$$C = -\frac{1}{N}\sum_{xj}[y_j \log(a_j^L) + (1-y_j)\log(1-a_j^L)] + \frac{\lambda}{2N}\sum_w w^2 \tag{71}$$

Where $\lambda > 0$ is the penalization parameter. Equation (71) makes the weight estimation to consider how to minimize original cost without increasing excessively overall weights size.

– If C_0 is the unregularized cost function, then partial derivatives of C are given by:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{N}w \tag{72}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b} \tag{73}$$

Where $\partial C_0/\partial w$ and $\partial C_0/\partial b$ can be calculated through backpropagation following (45)-(50). Consequently, update rules for stochastic gradient descent changes to:

$$\begin{aligned} w &\to w - \eta\frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{N}w \\ w &\to \Big(1 - \frac{\eta\lambda}{N}\Big)w - \frac{\eta}{S}\sum_x \frac{\partial C_x}{\partial w} \end{aligned} \tag{74}$$

$$b \to b - \frac{\eta}{S}\sum_x \frac{\partial C_x}{\partial b} \tag{75}$$

  * Equation (74) shows that update under L2 regularization occurs for a shrinked version of weights, since $1 - \eta\lambda/N$ will generally be smaller than 1, thus justifying the name *weight decay*.
  * If N is increased, for a same penalization it is necessary to increase λ proportionally.

– Besides attenuating overfitting and increasing predictive accuracy, L2 regularization may prevent stochastic gradient descent to find poor local minima and, therefore, to generate unstable results. With large values for w, any slight change around a (poor) local minimum (thus, $\partial C_x/\partial w$ small) will produce negligible changes in w.

– It is a premise, although not always true, that between a complex model and a simpler one, where both have similar assessed predictive accuracy, the simpler model should be preferred.

  * In this sense, a neural network with reduced weights is less complex, and thus should be expected to generalize better.

* If both a simple and a complex model fits well to the data, then it is reasonable to expect that the complex model is just learning local noise, and therefore would generalize poorly to new data points.
* In the context of neural networks, small weights, as those obtained when using regularized cost functions, imply that small changes in inputs will not affect notably the outputs from the model. Consequently, this means that the estimated model will be less sensitive to local noise, attenuating the effects of overfitting.
* To put it shortly, overfitting occurs when the model is capturing excessively patterns that may only apply to training data. Regularization mitigates overfitting and improves predictive performance on test data by forcing the model to focus on patterns whose frequency on training data is such that they are more likely to also hold for new data points.
* Finally, it is important to highlight that simplicity not necessarily guarantees that a model generalizes well.

– **L1 regularization:** instead of L2 norm, the L1 norm for calculating the length of a vector is used in the penalty term:

$$C = C_0 + \frac{\lambda}{N} \sum_w |w| \tag{76}$$

Partial derivatives with respect to weights now change to:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{N} sign(w) \tag{77}$$

And the update rule now is:

$$w \to w - \frac{\eta\lambda}{N} sign(w) - \frac{\eta}{S} \sum_x \frac{\partial C_x}{\partial w} \tag{78}$$

* Differently from L2 regularization, where shrinkage occurs proportionally to w, as (74) shows, with L1 regularization w reduces linearly and may even become zero when w assumes a small value.
* At the same time, large weights are more reduced with L2 regularization, given its proportional decay.
* Therefore, L1 regularization concentrates most relevant weights with non-zero values, while others are driven toward zero.

* Since the derivative of module function is not defined in 0, the convention of assuming $sign(0) = 0$ is adopted. Thus, when a weight $w \approx 0$, unregularized update rule for stochastic gradient descent is used (40) instead of (78).

– **Dropout layers:** this regularization method applies for a given hidden layer which has a random sample of its neurons temporarily deleted from the neural network. More specifically, at each mini-batch of data, a share of randomly selected neurons of a dropout layer has its activations set to zero, as if those units did not exist. In the next mini-batch, another sample of neurons is disregarded during update, and so on.

* If l is a dropout layer, then neurons $\{1, 2, ..., J_l^d\}$ being dropped out mean not only that their activations a_j^l are set to zero, but also that biases b_j^l, weights w_{jk}^l and $w_{j'j}^{l+1}$ (for $j \in \{1, 2, ..., J_l^d\}$) will not be updated in a given mini-batch iteration.

* Dropout provides regularization as a consequence of its procedure working similarly as if a set of different (complete) neural networks were estimated, and then as if their results were averaged.

* Averaging relates to regularization since it puts less relevance to each individual estimation that may be influenced by specificities of training data.

* This because, with dropout layers, at each mini-batch a different set of weights and biases are actually updated. Therefore, both models with the same architecture, but where to only one of them it is applied dropout layers, will likely to provide somewhat different estimated weights and biases. Moreover, the dropout layers neural network will be constructed upon updates of different sets of weights and biases.

* Dropout may imply in particularly improved results for deep networks, for which a huge set of parameters is estimated, thus making the model more likely to suffer from overfitting.

– A final approach to reduce the effects of overfitting is to artificially expand training data by creating new observations from randomly disturbing existent data points. Such data creation, however, should mimic variation as the data generation process would originally imply.

- **Weight initialization:** one standard way to randomly define initial values for weights and biases is to extract them from independent normal distributions with zero mean and standard deviation equal to 1.

- Although functional, this method makes the weighted input constructed from this sort of weights and biases to have a normal distribution with an extremely large variance. This because the weighted input for the j-th neuron from l-th hidden layer can be thought of approximating a sum of J_{l-1} standard normal distributions, where J_{l-1} is the number of neurons in the $l-1$-th hidden layer. Therefore, it is pretty likely that $\sigma(z) \to 1$ or $\sigma(z) \to 0$. In either case, $\sigma'(z) \to 0$, if $\sigma(.)$ is the sigmoid function or has a similar shape.
- Therefore, not only output neurons are likely to saturate, but also it is any neuron from hidden layers. Consider a given neuron in the first hidden neuron, if the weights connecting it to the input layer come from $N(0,1)$, then small changes in those weights will produce negligible changes in $\sigma(z_j^2)$. Besides, all neurons from further layers will also poorly respond to such changes, finally creating only a miniscule change in cost function.
- This represents a situation of learning slowdown. The strategy to mitigate its effects used previously, that constitute in changing cost function, only applies for correcting saturation of output neurons. Concerning saturation of neurons from hidden layers, modifying weights initialization is a way of speeding up learning.
- If the problem of extracting initial weights w_{jk}^l and biases b_j^l from $N(0,1)$ was the production of a weighted input with excessively large variance, it is possible to change the normal distribution of reference for w_{jk}^l to $N(0, 1/\sqrt{J_{l-1}})$, where J_{l-1} is the number of neurons in the predecessor layer.
  - As a consequence, the weighted input is still random, but its smaller variance reduce the probability of getting an extremely large value for $|z|$ that would lead to neuron saturation.
  - Since, for a given neuron j in layer l, there are multiple weights w_{jk}^l for just one bias b_j^l, there is no need for changing initialization of biases in order to correct for neurons saturation.
- It is important to notice that, in general, improving weight initialization will only benefit neural network estimation by reducing running time. Even so, in some contexts also the predictive accuracy may be affected.

- **Choosing hyper-parameters:** learning methods such as neural network that dispose of a large set of hyper-parameters involve a lot of work on specifying the best options available. Even more

difficult than defining appropriate values for each of them is to find a set of hyper-parameters that is the best among a huge collection of alternatives.

- **Broad strategy:** the first challenge when dealing with highly complex learning methods is to get a predictive performance better than chance. In order to achieve this, the first step is simplifying the learning problem, by reducing the sample of early estimations or by keeping aside some of the validation data points.
  - Note that by reducing the number of training data, the regularization parameter should be proportionally changed.
  - Another convenient simplification involves the overall architecture of the learning method, here depicted by the number of hidden layers and the amount of their neurons.
  - Increasing the frequency of results monitoring from the entire training epoch to only some share of training data.
  - A first hyper-parameter to freely explore is the learning rate η. Once a promising value is found, trials should move to the regularization parameter λ. After this, the architecture can be modified in order to improve the performance even further. Then, η and λ should be properly adjusted, and so on.
  - The general strategy, therefore, is to simplify as much as possible the learning task so that rapidly feedbacks can be obtained. Given that, whenever a signal of improvement is found, this should be more carefully explored.
- **Learning rate:** large values of η are likely to produce a series of training costs that highly oscillate across epochs. This because, at each update, changes in weights and biases are too high to reduce cost, thus increasing it instead. Small values of η are less prone to lead to such *overshooting*, however, this slows down learning, since at each update the weights and biases are only slightly changed.
  - Therefore, a strategy for finding an appropriate value for η involves defining a threshold η^* for which training cost immediately decreases, instead of displaying an oscillating behaviour.
  - In practice, for a given initial value for η, if training cost systematically decreases in early epochs of training, then try out some larger values until training cost starts to oscillate.

* If training cost oscillates during first epochs of training, then trial smaller values until training cost definitively decreases in first epochs.
* The chosen learning rate should not be larger than the threshold, and actually would rather be somewhat smaller than it (a fraction of 2, for instance), since this can allow for more steadily decrease in training cost along a higher number of training epochs that should take place when effective estimation is implemented.
* Training cost is suitable for defining the learning rate, since it is designed for adjusting learning from training data. Notwithstanding, performance metrics on validation data may also be used for specifying it.

– **Number of training epochs:** early stopping is a very convenient way to define this hyper-parameter, since it allows training as long as prediction accuracy on validation data is able to be increased. Once improvements in performance stop, also the estimation comes to an end, which not just helps to specify this hyper-parameter, but also prevents overfitting.
* Note that early stopping is appropriate only after the other hyper-parameters have already been defined.
* In the context of neural network modeling, the understand of predictive accuracy on validation data no longer increasing may be as follows: performance has stopped improving in the last T training epochs.
* It is recommended to define smaller values for T during initial experimentation (like $T = 10$). If performance should be optimized, than higher values must be adopted ($T > 20$).

– **Learning rate schedule:** η need not to be constant throughout epochs of training. Indeed, on early epochs, when the neural network is getting too wrong, a high learning rate seems a good choice for rapidly adjusting weights and biases.
* A promising strategy is to define a large value of η during initial epochs, and then to reduce it as validation accuracy starts to slow down. This can be repeated until $\eta' \approx \eta/1000$.
* Again, learning rate schedule should be avoided during initial experiments, and only be implemented to optimize performance.

– **Regularization parameter:** when first defining the learning rate, $\lambda = 0$ is more appropriate. Once some suitable value for η is found, λ can be defined based on validation accuracy. Then, η can be adjusted, which may eventually lead to new trials for λ.

- **Mini-batch size:** $S = 1$ represents online learning, and appears to be convenient since this leads to more recurrent updates for weights and biases. Even though, using $S \gg 1$ makes it possible to apply matrix techniques that result in returns of scale with respect to running time, given that calculating gradients for $S > 1$ data points turns to be faster than $S > 1$ estimations of single gradients.
    * Since optimizing running time through the best choice of mini-batch size is relatively not related with other hyper-parameters, a good strategy is to use reasonable values for them and then to plot validation accuracy against running time for different values of S. As a result, the best choice for S will be that value leading to the most rapid improvement in performance. Finally, use this mini-batch size when finding the best values for the other hyper-parameters.
- **Automated techniques:** all strategies discussed above may lead to systematic approaches such as *grid search*, when a list of different values for one or more hyper-parameters is sequentially trialed, and the combination with the highest validation accuracy should be chosen.
    * Another promising strategy is the Bayesian optimization of hyper-parameters.
    * Note that grid search, for instance, do not contradict the ideas mentioned for each hyper-parameter specification. Indeed, they can be used either to compose good lists of values for the hyper-parameters or to refine the search.
    * Grid search testing for different combinations of multiple hyper-parameters accomplishes the fact that there are not predictable relationships among the hyper-parameters, which helps achieving better performances than what would follow from partially defining each one of them.
    * If there is no definite way for defining hyper-parameters of neural networks, or any other learning method, the crucial turns out to be the development of workflows for both rapidly finding appropriate values for hyper-parameters and further improving performance by fine-tuning them.

- **Variations on stochastic gradient descent:** while gradient descent mainly focuses on the speed of changes in cost function, alternative approaches also consider how the velocity itself is changing.
    - **Hessian technique:** begins by defining the cost function through a Taylor series:

$$C(w+\Delta w) = C(w) + \sum_j \frac{\partial C}{\partial w_j}\Delta w_j + \frac{1}{2}\sum_{jk}\Delta_j \frac{\partial^2}{\partial w_j w_k}\Delta_k + ...$$

$$C(w+\Delta w) = C(w) + \nabla C.\Delta w + \frac{1}{2}\Delta w^T H w + ... \tag{79}$$

Where H is the hessian matrix of C with respect to weights and biases evaluated at w. What minimizes the second-order approximation of (79) via Δw is:

$$\Delta w = -H^{-1}\nabla C \tag{80}$$

Therefore, in order to minimize C, one may:

i. Initialize weights and biases w.

ii. Update w following the rule $w' = w - \eta H^{-1}\nabla C$, where η is the learning rate.

iii. Repeat i. and ii. until all training data points have been used, for one or multiple epochs of training.

* The Hessian technique considers not only the first order change in cost function, but also the second order, thus refining the information of how cost changes with weights and biases.
* Hessian optimization can reach a minimum faster than stochastic gradient descent. In the other hand, the need for calculating a hessian matrix makes calculation too complex.

– **Momentum-based gradient descent:** this methods also takes into account a second-order information of how cost changes, but avoids the need for calculating the hessian matrix.

* Instead of updating weights and biases based solely on gradient of cost, a new variable called *velocity* is inserted into the update rule, controlling the amount of change in w:

$$v \to v' = \mu v - \eta\nabla C \tag{81}$$

$$w \to w' = w + v' \tag{82}$$

* The hyper-parameter μ can be understood as a parameter of friction. If $\mu = 1$, then there is no friction, since changes in weights and biases will fully accumulate across iterations. If $\mu = 0$, then weights and biases will change with no velocity, as occurs with standard gradient descent.

* In the case of $\mu \in (0, 1)$, however, the update will be more intense than with standard gradient descent, but will be less prone to suffer from overshooting.
* The *momentum coefficient* μ can be defined by validation methods, in a similar fashion as for η and λ.
* Momentum-based gradient descent can be implemented through backpropagation algorithm with only minor changes in the code for gradient descent.

- **Other models of artificial neurons:** hidden layer or output neurons need not to be activated through sigmoid function, $\sigma(z) = 1/(1 + \exp(-z))$. Indeed, some alternatives may provide faster learning or better generalization. Two of main activation functions are *tanh neuron* and *rectified linear unit.*

  – **Tanh neuron:** uses the hyperbolic tangent function:

  $$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \tag{83}$$

  Since $\sigma(z) = (1 + \tanh(z/2))/2$, the tanh neuron is a rescaled version of sigmoid neuron, and the plot of $\tanh(z)$ as a function of z has a very similar shape of that for sigmoid function. However, while $\sigma(z) \in (0, 1)$, $\tanh(z) \in (-1, 1)$.

  * The fact that $\tanh(z) \in (-1, 1)$ implies that not all weights w^l_{jk} from a same neuron j and layer l will increase or decrease together (see equation (52)). Therefore, tanh neurons are more flexible in their weights update.

  – **Rectified linear unit:** follows the function:

  $$relu(z) = \max(0, z) \tag{84}$$

  This neuron model can not suffer from saturation, since the derivative of $relu(z)$ does not converge to zero, but instead remains constant as z increases. At the same time, as z decreases towards $-\infty$, $relu(z)$ becomes zero, thus making the weights and biases to stop learning at all.

5 A visual proof that neural nets can compute any function (Chapter 4)

- The **universal approximation theorem** indicates that neural networks can compute any function. In fact, whatever function $f : \mathbb{R}^p \to \mathbb{R}^m$ can be approximated with an arbitrary level of closeness

by neural networks with just a single hidden layer.

- Empirically this result is of special interest in conjunction with the fact that most real world processes can be translated into mathematical functions. Therefore, neural networks can provide a source of predictions for various types of empirical process.

- The approximation to a function $f(x)$ implied by neural networks is equivalent to finding an output function $g(x)$ such that $|f(x) - g(x)| < \epsilon$ for some $\epsilon > 0$ and for any input vector x. It is important to notice that the original function $f(x)$ should be continuous.

- Chapter 4 of the web-book presents a graphical proof of the universality theorem considering first a real valued function $f : \mathbb{R} \to \mathbb{R}$, and then considering any function $f : \mathbb{R}^p \to \mathbb{R}^m$ approximated by a neural network with two hidden layers.

- The strategy adopted in the book begins with sigmoid neurons, or any activation function with shape similar to that of sigmoid function. Since such functions converge to 1 as its argument increases indefinitely, if weights assume values excessively high, then the output from activation as a function of inputs will approximate a *step function*. The position of the step, in its turn, is defined by the bias, following $s = -b/w$.

  - The step function is approximated by using only one hidden neuron, and considering the weights connecting it to the inputs.
  - As long as two hidden neurons are used, with weights connecting them to the output given by $w_1 = h$ and $w_2 = -h$, then it emerges a *bump function*, this time defined as a function of the weighted inputs instead of activation output.
  - Applying k pairs of hidden neurons produces k different bump functions.
  - The graphical proof follows a procedure somewhat similar to Riemann sum used for calculating definite integrals, or the area under a curve. This because, by employing any number of pairs of hidden neurons, changing their positions via changes in bias terms, and finally regulating the height of the bumps, an approximation to any arbitrary function can be conceived.
  - Precisely, given a function $f(x)$, since the procedure creates an output as a function of weighted inputs, what effectively should be approximated is $\sigma^{-1} \circ f(x)$, so that applying the activation function $\sigma(.)$ would ultimately approximate $f(x)$.

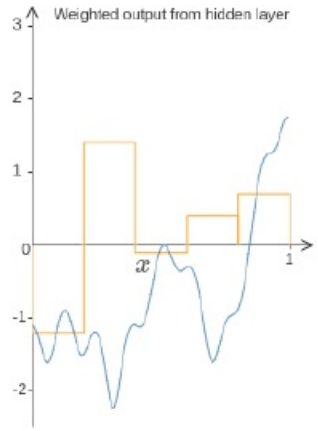

Figure 5.1: Approximating a function $f : \mathbb{R} \to \mathbb{R}$

  - In terms of neural networks parameters, the approximation considers very high weights for the hidden neurons, connecting them to input neurons. The bias terms for the hidden neurons are adjusted following $s = -b/w$, where s is the position of the step function (weighted input after applying the activation function having as argument the inputs). Hidden neurons should be defined in pairs with weights values are h and $-h$, and which connect them to the output. Finally, the bias term for the output neuron should be equal to 0.

- Chapter 4 presents an analysis for the multiple inputs case based on just two inputs. Analogously to the one input case, when the weight connecting a hidden neuron to any input neuron increases arbitrarily, and considering the weight for the other input being equal to zero, the activation of that hidden neuron as a function of the input again approaches the step function, this time defined on $\mathbb{R}^2$. Again, the bias for the hidden neuron controls the position of the step.

- When there are two hidden neurons, each connected to the same input neuron and having weights given by h and $-h$ connecting to the output neuron, a three-dimensional bump function is created returning the weighted input as a function of inputs. Adding a new pair of hidden neurons, another bump function can be generated, either on the same or in the other direction.

  - Since the combination of those two bump functions do not produce a single combined bump, or "tower" for the output of the network, both pairs of hidden neurons should be adjusted in order to have a large absolute height h, while the bias must accommodate such height with an

appropriate value (around -3h/2) so that a single tower emerges.

– Combining more than one of such towers, and plotting the weight output for the second hidden layer, which connects the neurons set as describe above, then any three-dimensional function can be approximated through the same way for the one input case, resembling the argument of Riemann sum.

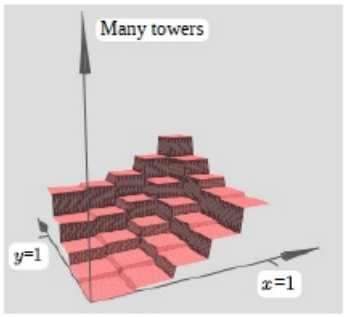

Figure 5.2: Approximating a function $f : \mathbb{R} \to \mathbb{R}$

- Functions with more than two inputs works exactly the same, but the bias for the output neuron (or any neuron in the second hidden layer, if it exists) should equal $b = (-p + 1/2)h$ to produce the towers.

- Functions with more than one output can be approximated using the procedure above several times, one for each function composing the vector of outputs.

- Not just neural networks with sigmoid neurons can approximate any function with an arbitrary level of accuracy. In fact, the universality of neural networks applies for any activation function $s(z)$ such that $\lim_{z\to-\infty} s(z)$ and $\lim_{z\to\infty} s(z)$ are well defined, i.e, such that these limits exist. A second necessary condition relating with activation functions demands that those limits are different from each other.

- The approximations to step functions provided by the output of hidden neurons are not necessarily as good as possible near the step position.

  – The solution proposed on chapter 4 considers the M splits of the function to be approximated, $f(x)$. Then, M approximations to each split should be constructed and then added to produce

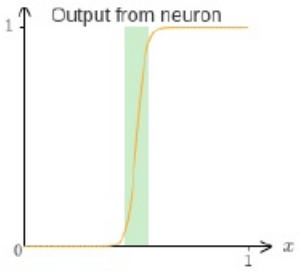

Figure 5.3: Window of failure

an approximation to the entire function $f(x)$. By doing this, each window of failure has smaller height than that produced by a single approximation. In some sense, this split reasoning leads to more smoothness for the approximation.

- Although for the multiple input case the book has proved the universality using two hidden layers, with just one the theorem also holds. Despite of their higher complexity, deep neural networks produce a hierarchical structure that is found to be useful in empirical applications.

6 Why are deep neural networks hard to train? (Chapter 5)

- Neural networks with more than one hidden layer reproduce the principle under which humans solve tasks (computational or not) by dividing the main problem into several and successive sub-problems. Consequently, a deep neural network is able to learn highly complex target functions.

- Even though deep neural networks can be trained using gradient-based techniques and the back-propagation algorithm, this approach is not efficient for learning.

- **Vanishing gradient problem:** to measure how fast weights and biases from a given layer are learning, i.e., the rate of change in cost with respect to them, the third and the fourth fundamental equations of backpropagation point to $\partial C/\partial b^l_j$ as the main quantity controlling the speed of learning. This follows from the fact that both $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$ are a linear function of δ^l_j. Therefore, the higher $\partial C/\partial b^l_j = \delta^l_j$, the faster w^l_{jk} and b^l_j will learn from the data.

  - Considering the speed of learning for an entire hidden layer l, the length of δ^l, $||\delta^l|| = \sqrt{\delta^{12}_1 + \delta^{22}_2 + ... + \delta^{J_l 2}_{J_{l-1}}}$, indicates an aggregate measure of speed.

- The vanishing gradient problem characterizes the fact that neurons in early hidden layers are likely to learn much more slowly than neurons in later layers.
- If not suffering from learning slowdown, weights and biases from early hidden layers may present the opposite, as they learn much faster than those from later layers - *exploding gradient problem.*
- Thus, more generally, deep networks trained using gradient descent will have their neurons from different hidden layers learning at very different speeds.
- To see how bad it is to have early neurons learning very slowly, it is crucial to remember that random initialization may lead to bad results at the same time weights and biases are adjusting themselves too few at each mini-batch and at each epoch. So, it will take too long for the early neurons to capture patterns in training data. Even more, this learning slowdown means that information are being lost as they are not effectively being propagated to neurons in later layers.

- A convenient expression for $\partial C/\partial b_j^l$ can be first derived considering a neural network with only a single neuron per hidden layer. The definition of partial derivative defines that:

$$\frac{\partial C}{\partial b^1} \approx \frac{\Delta C}{\Delta b^1} \tag{85}$$

Changing b^1 produces a variation on a^1 approximated by:

$$\Delta a^1 \approx \frac{\partial \sigma(a^0 w^1 + b^1)}{\partial b^1} \Delta b^1 = \sigma'(z^1)\Delta b^1 \tag{86}$$

The change in a^1, in its turn, produces a variation on z^2:

$$\Delta z^2 \approx \frac{\partial z^2}{\partial a^1} \Delta a^1 = w^2 \Delta a^1 = \sigma'(z^1) w^2 \Delta b^1 \tag{87}$$

Again, Δz^2 leads to a change on a^2:

$$\Delta a^2 \approx \frac{\partial a^2}{\partial z^2} \Delta z^2 = \sigma'(z^2)\Delta z^2 = \sigma'(z^1) w^2 \sigma'(z^2) \Delta b^1 \tag{88}$$

Continuing forward, the effect on cost is assessed:

$$\Delta C \approx \sigma'(z^1) w^2 \sigma'(z^2) ... \sigma'(z^L) \frac{\partial C}{\partial a^L} \Delta b^1 \tag{89}$$

Consequently:

$$\frac{\partial C}{\partial b^1} = \sigma'(z^1) w^2 \sigma'(z^2) ... \sigma'(z^L) \frac{\partial C}{\partial a^L} \tag{90}$$

Note that in (90) $l = 1$ does not refer to the input layer, neither $l = L$ to the output layer.

- Except for the first and the last terms, all components in (90) are equal to $w^l\sigma'(z^l)$. The maximum value of $\sigma'(z)$ is $1/4$, while w^l being initialized through a standard Gaussian distribution means that $|w^l\sigma'(z^l)| < 1/4$. Therefore, the more terms in $\partial C/\partial b^l$, the smaller it will be. Moreover, if w^l is initialized from $N(0, 1/\sqrt{J_{l-1}})$, when more than one neuron is present in the $l-1$-th layer, than the expected value of random values for w^l is even lower.

  - Comparing (90) for b^1 with the analogous expression for b^3:

$$\frac{\partial C}{\partial b^3} = \sigma'(z^3)w^4\sigma'(z^4)...\sigma'(z^L)\frac{\partial C}{\partial a^L} \tag{91}$$

    Therefore, the fact that (90) is likely to be pretty smaller than (91) gives ground for the **vanishing gradient problem**.

- If weights w^l get sufficiently high during training, then (91) may become higher than (90), which will accelerate the learning of early neurons relative to later neurons. Therefore, it emerges an opposite issue, **exploding gradient problem**.

- More generally, expressions (90) and (91) will differ from each other, thus implying in neurons from different hidden layers learning at very different speeds, the problem of **unstable gradient problem**.

- The vanishing gradient problem is more likely to happen than exploding gradient descent, since even with large weights the gradient $\partial C/\partial b^l$ is likely to be smaller than 1, as a consequence of the bell shape of $\sigma'(z)$, which makes $\sigma'(z)$ small when z is large.

- The expression (90) for hidden layers with multiple neurons is given by:

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T\Sigma'(z^{l+1})(w^{l+2})^T...\Sigma'(z^L)\nabla_a C \tag{92}$$

  Where l is a hidden layer and $\Sigma'(z)$ is a diagonal matrix whose entries are $\sigma'(z)$ values for each neuron in the hidden layer.

- Not just gradient-based learning techniques may lead to problems for training deep networks. The sigmoid activation function can cause neurons in the final hidden layer to saturate, causing learning lowdown, for instance. The weight initialization and the specification of parameters for momentum-based stochastic gradient descent make possible to obtain very unstable results, which also applies for other hyper-parameters and for the network architecture.

7 Deep learning (Chapter 6)

- Concerning the estimation of deep neural networks, chapter 5 presented the unstable gradient problem and two of its main important aspects, vanishing and exploding gradient problems. The difficulty involved in deep learning derives fundamentally from gradient descent method. Even so, some additional techniques may help fitting neural networks with many hidden layers without excessive modifications in the learning algorithm.

- **Convolutional networks** introduce a different kind of hidden layers that saves the estimation of a large number of parameters. The main novelty with respect to standard neural nets is the replacement of *fully connected layers* by partially connected ones.

  – A fully connected layer l has its neurons connected to all neurons in layers $l-1$ and $l+1$. Therefore, there are *a priori* non-zero and different biases b_j^l, for each $j \in \{1, 2, ..., J_l\}$ and weights w_{jk}^l, for each $k \in \{1, 2, ..., J_{l-1}\}$, and $w_{j'j}^{l+1}$, for each $j' \in \{1, 2, ..., J_{l+1}$ and for each $j \in \{1, 2, ..., J_l\}$.

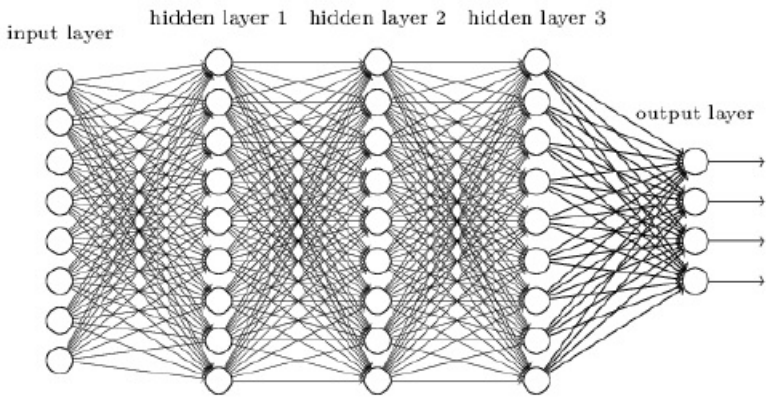

Figure 7.1: Fully connected layers

  – Fully connected layers may not be appropriate for exploring local patterns in the features space. In terms of image recognition, fully connected layers are not suited for capturing spatial structure of the images. The relationship between pixels far from each other is modeled in the same way as for those pixels that rely closely.

  – **Convolutional layers** explicitly try to capture local or spatial structures. Besides of that,

a neural network disposing of convolutional layers is faster to train than standard networks, which helps the fit of models with many hidden layers. A convolutional layer is defined on the basis of three main concepts: local receptive fields, shared weights, and pooling.

– **Local receptive fields:** starting from the input layer, a first convolutional layer will have neurons connected only to a few input units that are nearly localized. The image recognition task is appropriate for changing the input layer representation from a stack to a matrix form. Consequently, if there are 28 x 28 pixels in each image, then a local receptive field is a region in the matrix representation containing, for instance, only 5 x 5 input units. Each neuron in the first convolutional layer will be connected to a different local receptive field in the input layer.

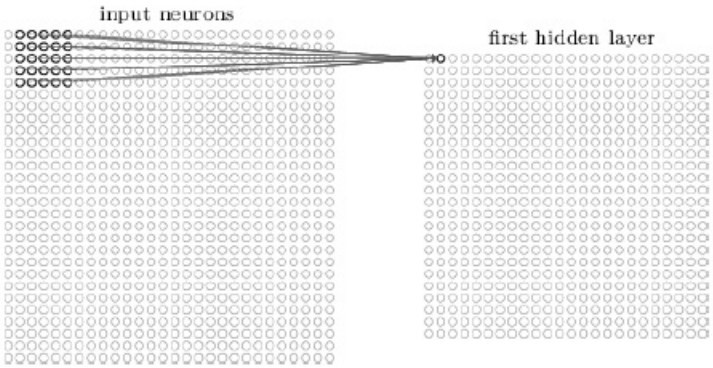

Figure 7.2: Local receptive field

  * Considering a **stride length** equal to 1, then two neighbors neurons in the convolutional layer will share all input units but 5, either if the slide is over columns or rows. The stride length may be chosen through validation or cross-validation procedures. The same applies for choosing the size of local receptive fields.
  * As before, the connection of each input unit in a local receptive field to a hidden neuron in the convolutional layer implies in a weight. Additionally, each hidden neuron has a bias term composing the weighted inputs over which the activation function should be applied.

– **Shared weights and biases:** a local receptive field with 5 x 5 neurons is connected to a neuron in the convolutional layer through a 5 x 5 matrix whose entries are weights connecting the respective neurons to the convolutional layer unit. The development of convolutional

networks assumes, in general, that such a matrix is equal across all 24 x 24 hidden neurons in the convolutional layer, which also applies to each of their bias terms.

* This structure with shared weights and biases implies that all convolutional layer units capture the same *feature* from the input image (considering a first convolutional layer), representing that a given shared set of characteristics of the image is being detected by all hidden neurons. However, since each one of them refers to different locations in the image, this same feature is successively analyzed for different positions.
* If a shared matrix of weights works for capturing a given feature from the input layer for different positions in the image, then another *feature map* connecting the input layer to the convolutional layer using a different shared matrix of weights will provide the detection of a new pattern in input units.
* Since it is natural to explore more than one feature map, a convolutional layer is defined by several different feature maps.

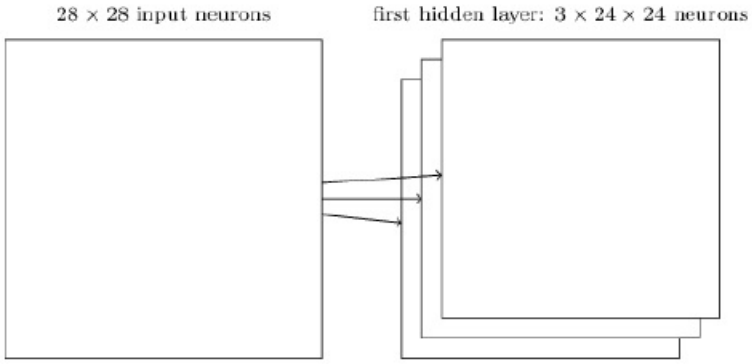

Figure 7.3: Feature maps

* One main learning advantage of convolutional layers is the smaller quantity of parameters to be estimated in comparison to fully connected layers. With 28 x 28 = 784 input units, and supposing a hidden layer with 30 neurons, there are 784 x 30 = 23550 weights plus 30 biases. A convolutional layer with local receptive fields of size 5 x 5 and 20 feature maps will have only 500 weights and 20 biases.

– **Pooling layers:** a pooling layer follows immediately after a convolutional layer, and works for simplifying the information contained on it. In the same way as before, a neuron in the

pooling layer will combine the activation of neighbor units in the convolutional layer, again capturing spatial patterns.

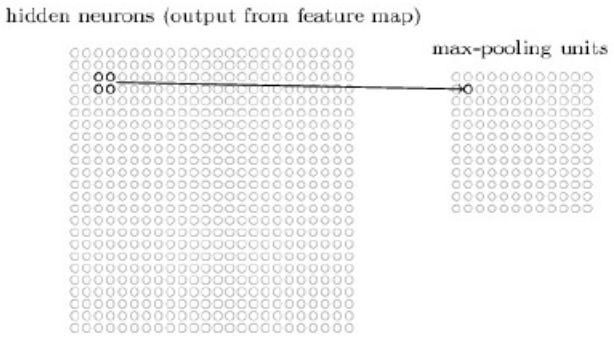

Figure 7.4: Pooling layer

* A **max-pooling layer** takes the information from 2 x 2 neurons in the convolutional layer, for instance, and defines the value of a pooling layer unit as the maximum value across those 2 x 2 neurons. Using this number of neurons, and having 24 x 24 convolutional layer units, there will be 12 x 12 units in the pooling layer.
* A pooling layer will be composed by connections with consecutive feature maps in the convolutional layer.

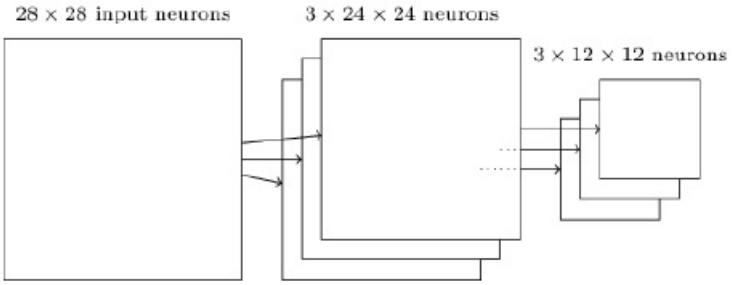

Figure 7.5: Pair of convolutional and pooling layers

* If a fully connected hidden layer is to be added next to a pooling layer, this will demand fewer weights to perform the connection than would be necessary in the absence of a pooling layer, since this reduces the number of units from the convolutional layer.
* An alternative to max-pooling is **L2-pooling**, which takes 2 x 2 units from convolutional layer, for instance, and outputs the square-root of the sum of squares of their activations.

– Finishing the design of a convolutional network, the last layer should be a fully connected output layer. Yet, many more pairs of convolutional and pooling layers may be added, besides of any number of fully connected hidden layers.

  * Later fully connected layers capture more abstract, global patterns in data, integrating information which was previously integrated in a local level.
  * Adding a second pair of convolutional and pooling layers refines the exploration of local patterns in data, since a first pair still keeps track of spatial structure.
  * The first convolutional layer is usually composed by more than one feature map, and this is reflected in the subsequent pooling layer. Adding a second convolutional layer implies that each of its neurons is connected to all feature maps of the predecessor pooling layer, but always following the same neuron-specific local receptive field.
  * As before, all neurons in the second convolutional layer will share a same matrix of weights to connect them to the neurons in the predecessor pooling layer of their own local receptive field and for all feature maps.

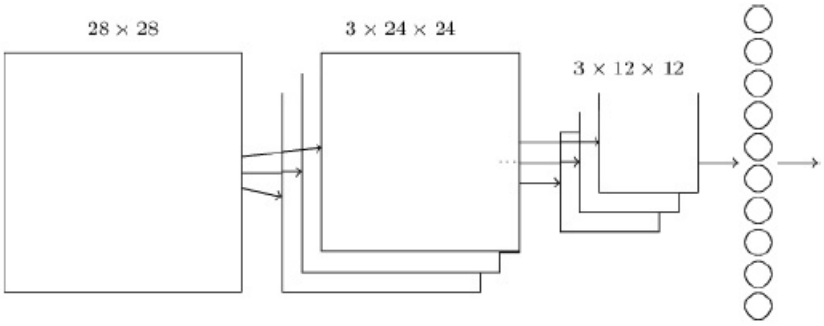

Figure 7.6: Convolutional network

– Even that the design of a convolutional network is somewhat different and more complex than a standard network, either with a single or multiple hidden layers, its functioning is as usual, with information from inputs being propagated throughout several different simple units connected by weights w^l_{jk} and activated using bias terms b^l_j and activation functions $\sigma(z)$. Additionally, the same learning techniques, such as stochastic gradient descent and backpropagation algorithm, may be used for estimation, requiring only small changes.

• In the experiments presented in the book, the use of two pairs of convolutional and pooling layers,

besides of a last fully connected layer sequentially improved the classification performance. Another techniques that increased predictive accuracy were the use of rectified-linear unit (ReLU) activation function and the artificial expansion of training data. Further improvements followed the inclusion of a second fully connected hidden layer and, mainly, the applying of dropout for the fully connected layers (hidden and output). A final improvement may be obtained through the use of an ensemble of networks.

  - Feature maps, also known as convolutional filters, provide a source of overfitting reduction, since they are forced to look over the entire instance, instead of focusing on specific parts of it.
  - The use of GPU instead of CPU is useful for improving training, since this speeds up learning, allowing the implementation of more sophisticated techniques.

- The empirical applications of the books were based on the MNIST dataset, which has as its task the recognition of handwritten digits. Neural networks can and have been successful in many other image recognition problems.

  - The last three sections of chapter 6 of the book present several major papers which applies deep neural networks to different image recognition tasks. Besides of the varying application frameworks, those studies reveal the way of how standard ideas can be further enhanced to produce extremely sophisticated models.
  - Additionally, some problems that may affect neural networks predictive capacity were discussed. Despite of them, neural networks have a extremely high generalization power, and thus those problem mainly reinforce how this learning approach needs to be more deeply understood.

- The final part of chapter 6 also presents different approaches to deep neural networks. The feedforward network developed in the book is not the only structure for deep learning.

  - A first class of an alternative construction is **recurrent neural networks (RNNs)**, for which neurons in a given layer depend not only on the activations of neurons from predecessor layers, but also on activations of early models (either those from neurons in predecessor layers, or even those from their own early activations).

    * Therefore, RNNs are very suited for capturing patterns that evolve over time.

    * Interesting applications of RNNs are such that a neural network may be trained to infer what algorithms to use for solving a given problem. This may combine characteristics of both conventional algorithms and neural networks.
    * Despite of their different structure, RNNs may be trained using variations of stochastic gradient descent and the backpropagation algorithm, besides of many other techniques used with feedforward networks.
    * See this further reference for more on RNNs.
  - Standard feedforward neural networks are likely to suffer from learning slowdown, under which neurons in early layers learn at very slower rates than those neurons in later layers. This problem tends to be enlarged with RNNs, since the same applies for all parameters in early models, which will probably learn slower than later models. When **long short-term memory units (LSTMs)** are introduced into RNNs, they help this class of networks to overcome learning slowdown.
  - Another alternative to feedforward networks are **deep belief networks (DBNs)**, which in turn consist on an instance of **generative models**. A DBN model is such that, when learning from data, it automatically creates artificial inputs to expand training data. A second interest characteristic of DBNs is that they can implement unsupervised and semi-supervised learning, i.e., inputs without label can also help the network to learn from the data.
    * Check this reference for more on DBNs.
  - A last extension of neural networks is the combination of convolutional networks with techniques of **reinforcement learning**.

- The last section of chapter 6 presents a very interesting discussion on the future of neural networks, mainly of their applications for developing general artificial intelligence. Adapting the Conway's law - under which the development of any system reflects the communications structure that controls its design - to science, the author sums up to a relevant argument: the knowledge structure shapes the social structure of science, and this in turn rules how the knowledge itself will evolve. Therefore, the future of neural networks and their uses towards artificial intelligence would depend on more advances on theoretical understanding of such models, thus allowing a more complex social structure that could sustain a vigorous development of new and deeper ideas.