Deep learning for nonlinear dimension reduction and applications

Mark Ng and Alexandre Thiery

Department of Statistics and Applied Probability
National University of Singapore
2015/2016

Deep learning for nonlinear dimension reduction and applications

Mark Ng[1] and Alexandre Thiery[2]

Abstract

Deep learning refers to techniques used for learning in neural networks [10, 20]. We begin by explaining fundamental concepts and ideas behind deep feedforward neural networks [20]. In addition to basic dense layer architectures, we supplement this text with a discussion of other types of layers; namely dropout, convolutional and pooling layers [5, 12, 22]. We dedicate a chapter to the discussion of some well-known classification problems and datasets [7, 17, 18]. Some of the main difficulties faced in training deep neural networks are also discussed later on, and we go on to explain some useful and recent practices which are used to overcome such difficulties and improve neural network performance [11, 13, 14]. The next half of the paper focuses on further applications of deep neural networks. One main focus is dimension reduction using traditional autoencoders [1]. Other applications discussed include variational autoencoders (VAEs) [4, 16] and using neural networks to represent uncertainty via Bayesian inference [8, 9]. All code implementations are written in Python 2.7.11 using the Lasagne[3] library.

[1]Undergraduate, National University of Singapore (main author).
[2]Assistant Professor, National University of Singapore.
[3]See http://lasagne.readthedocs.io/en/latest/.

Acknowledgements

I would like to thank Assistant Professor Alexandre Thiery for his helpful supervision throughout the course of this project. I also thank Associate Professor Adrian Roellin and his research student Tang Wai Hoh for the interesting discussions during the final stages of this project.

Contents

1 Introduction

We begin by discussing some basic ideas in machine learning. Deep learning is a subset of machine learning, referring to various techniques used for learning in neural networks. Neural networks can be seen as simply a programming paradigm, applied to solve problems based on the principle of learning from the observational data [20].

1.1 Supervised vs Unsupervised Learning

The terms supervised and unsupervised learning are generally not formally defined. For the purpose of providing sufficient clarity, we provide a rough distinction between the two [10].

Supervised learning is the process whereby each point $\mathbf{x}_i$ in the dataset is associated with a label or target value $\mathbf{y}_i$, and the learning algorithm here is often interested in being able to predict $\mathbf{y}_i$ from $\mathbf{x}_i$. Problems concerning this type of learning are usually classification or regression-based. For example: being able to classify (predict) an associated label $\mathbf{y}_i \in \{0, 1, \ldots, 9\}$ correctly based on an image of a handwritten digit represented by a vector $\mathbf{x}_i \in \mathbb{R}^{28\times 28}$.

Unsupervised learning involves a dataset containing points with no specific label or target value. Thus unsupervised learning algorithms are often used for purposes such as clustering, or more generally pattern-finding algorithms on a given dataset; rather than making predictions from each observation (point) in the dataset. For example: applying dimension reduction techniques on high-dimensional datasets.

Example 1.1.1 (Supervised learning). Using neural networks to classify handwritten digit images, such as those from the MNIST handwritten digit database [18].

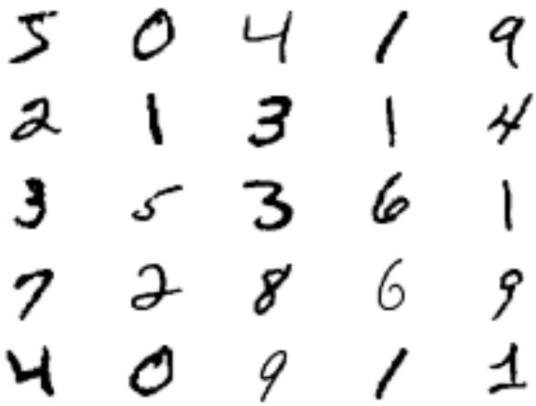


Figure 1.1: MNIST handwritten digit database.

1.2 The Perceptron

As a form of motivation for the next section, we introduce a relatively simple linear classifier known as the *perceptron algorithm* [21]. This algorithm was the building block of research in neural networks. The perceptron is used for binary classification of data, based on a vector of features. In other words, we assume each element in our dataset is of the form $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^d$ represents the features with corresponding categorical label $y_i \in \{-1, +1\}$. We maintain static parameters $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, known as the weight and bias respectively. Let $\mathbf{w} \cdot \mathbf{x}$ denote the usual dot (inner) product in $\mathbb{R}^d$, and define

$$\operatorname{sgn}(x) = \begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0. \end{cases} \tag{1.1}$$

Algorithm 1.1: The Perceptron

Data: features $\mathbf{x} \in \mathbb{R}^d$ and label $y \in \{-1, +1\}$
Result: prediction $\hat{y} \in \{-1, 0, +1\}$
if *weights and bias are not initialised* **then**
  Initialise weight $\mathbf{w} = (0, \ldots, 0)^T \in \mathbb{R}^d$ and bias $b = 0 \in \mathbb{R}$;
`/* compute prediction value */`
$\hat{y} \leftarrow \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$;
if $\hat{y} \neq y$ **then**
  `/* update existing weights and bias */`
  $\mathbf{w} \leftarrow \mathbf{w} + y\mathbf{x}$;
  $b \leftarrow b + y$;

The perceptron is also said to be *online* algorithm, in the sense that its parameters are updated with every prediction (the last step). It is also useful to note that if the dataset is linearly separable, then the weights and bias will eventually converge to some value which correctly classifies all the points in the dataset. This is clear, as we would be able to find some $\mathbf{w}^* \in \mathbb{R}^d$ and $b^* \in \mathbb{R}$ such that $\mathbf{w}^* \cdot \mathbf{x}_i + b^* > 0$ whenever $y_i = +1$ and $\mathbf{w}^* \cdot \mathbf{x}_j + b^* < 0$ whenever $y_j = -1$. On the other hand, the perceptron fails when the dataset is not linearly separable.

A simple example which demonstrates the failure of the perceptron, in the sense that it is unable to classify all the points in the dataset correctly, is the XOR function[4] (notice that the 'input' here is not linearly separable). Note that we may view the perceptron as an approximation $f(\mathbf{x}|\mathbf{w}, b)$ to some function $f^* : \mathbf{X} \to \mathbf{Y}$. Therefore, using the XOR as an example, the function f^* which the perceptron

[4]We take the usual values 1 and 0 to represent True and False respectively.

fails to approximate is

$$f^*(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{x} = (0,1)^T \text{ or } \mathbf{x} = (1,0)^T \\ -1 & \text{if } \mathbf{x} = (0,0)^T \text{ or } \mathbf{x} = (1,1)^T. \end{cases} \tag{1.2}$$

The remainder of this section is slightly theoretical, and can be skipped for those primarily interested in deep learning and applications. We establish an upper bound for the number of mistakes a perceptron can make, under the assumption that the data is linearly separable [21].

Definition 1.2.1. The margin of a single observation $\mathbf{x}_i \in \mathbb{R}^d$ with associated label $y_i \in \{-1, +1\}$ is defined to be

$$\gamma(\mathbf{x}_i, y_i) = y_i(\mathbf{w} \cdot \mathbf{x}_i + b). \tag{1.3}$$

Definition 1.2.2. The margin of an entire set of observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and associated labels $\mathbf{Y} = \{y_1, \ldots, y_n\}$ is

$$\gamma(\mathbf{X}, \mathbf{Y}) = \min_i \gamma(\mathbf{x}_i, y_i). \tag{1.4}$$

The perceptron classifies an observation correctly if and only if the margin of the observation is positive. Furthermore the perceptron eventually classifies all points in the dataset if and only if the margin of the entire dataset is eventually positive; that is: $\gamma(\mathbf{X}, \mathbf{Y}) > 0$ eventually.

Theorem 1.2.3 (Novikoff)**.** Let $R = \max_i \|\mathbf{x}_i\|$, where $\|\cdot\|$ denotes the Euclidean norm. Assume there exists $\mathbf{w}^* \in \mathbb{R}^d, b^* \in \mathbb{R}$ with $\|\mathbf{w}^*\| = 1$, and $\gamma := \gamma(\mathbf{X}, \mathbf{Y}) > 0$. Then, the number of mistakes a perceptron will make it at most

$$\frac{(1 + R^2)(1 + (b^*)^2)}{\gamma^2}. \tag{1.5}$$

Proof. Let $\mathbf{w}_t, b_t$ denote the weight and bias after the t-th update, that is associated with some observation $\mathbf{x}_t$ and label y_t. Note that $\mathbf{w}_0 = (0, \ldots, 0)^T$, $b_0 = 0$ and for all $t > 0$,

$$\begin{aligned} \mathbf{w}_t \cdot \mathbf{w}^* + b_t b^* &= \mathbf{w}_{t-1} \cdot \mathbf{w}^* + b_{t-1} b^* + y_t(\mathbf{x}_t \cdot \mathbf{w}^* + b^*) \\ &\geq \mathbf{w}_{t-1} \cdot \mathbf{w}^* + b_{t-1} b^* + \gamma. \end{aligned} \tag{1.6}$$

By induction, $\mathbf{w}_t \cdot \mathbf{w}^* + b_t b^* \geq t\gamma$. For each t, $y_t(\mathbf{x}_t \cdot \mathbf{w}_{t-1} + b_{t-1}) \leq 0$ since the t-th prediction $\hat{y}_t := \operatorname{sgn}(\mathbf{x}_t \cdot \mathbf{w}_{t-1} + b_{t-1})$ is either zero or of opposite sign. Observe that

$$\begin{aligned} \|\mathbf{w}_t\|^2 + b_t^2 &= \|\mathbf{w}_{t-1} + y_t \mathbf{x}_t\|^2 + (b_{t-1} + y_t)^2 \\ &= \|\mathbf{w}_{t-1}\|^2 + b_{t-1}^2 + y_t^2 \|\mathbf{x}_t\|^2 + 1 + 2y_t(\mathbf{w}_{t-1} \cdot \mathbf{x}_t + b_t) \\ &\leq \|\mathbf{w}_{t-1}\|^2 + b_{t-1}^2 + \|\mathbf{x}_t\|^2 + 1 \\ &\leq \|\mathbf{w}_{t-1}\|^2 + b_{t-1}^2 + R^2 + 1. \end{aligned} \tag{1.7}$$

By induction, $\|\mathbf{w}_t\|^2 + b_t^2 \leq t(R^2+1)$. Finally, the Cauchy-Schwartz inequality gives us the result

$$\begin{aligned} t\gamma &\leq \mathbf{w}_t \cdot \mathbf{w}^* + b_t b^* \leq \sqrt{\|\mathbf{w}\|^2 + b_t^2}\sqrt{\|\mathbf{w}\|^2 + (b^*)^2} \leq \sqrt{t(R^2+1)}\sqrt{1+(b^*)^2} \\ \implies t &\leq \frac{\sqrt{t(R^2+1)}\sqrt{1+(b^*)^2}}{\gamma} \\ \implies t &\leq \frac{(R^2+1)(1+(b^*)^2)}{\gamma^2}. \end{aligned} \tag{1.8}$$

□

2 Deep feedforward neural networks

Deep feedforward neural networks, also referred to as *multilayer perceptrons*, can be seen as the basic idea behind deep learning models [10]. We use this section to explain the essentials.

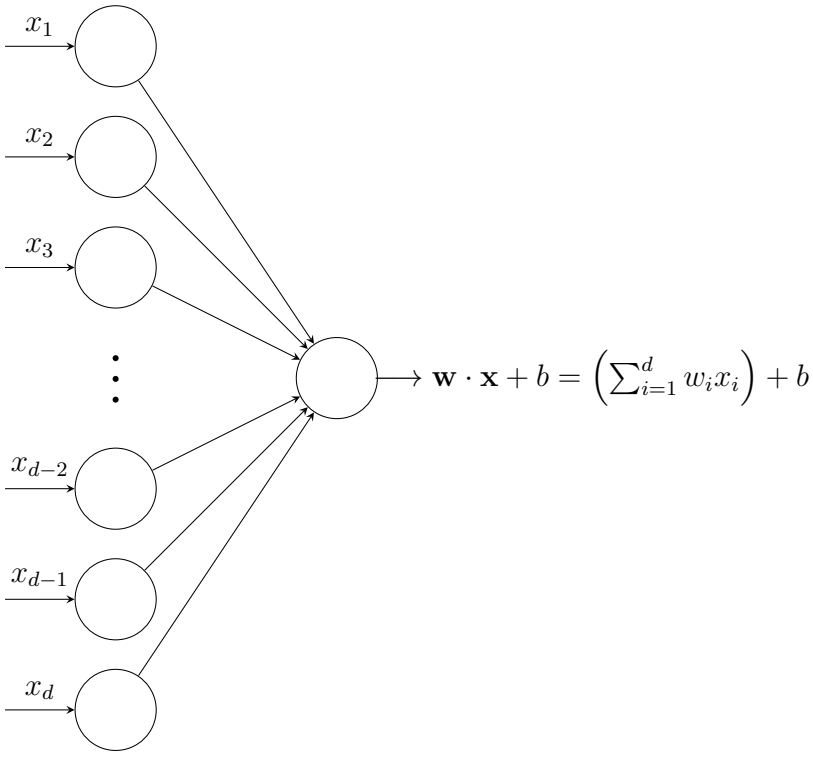
2.1 Activation functions

Recall that a perceptron can be seen as a function $f(\mathbf{x}|\mathbf{w}, b)$ which approximates some function $f^* : \mathbf{X} \to \mathbf{Y}$. Deep feedforward neural networks serve the same purpose. In the case of the perceptron, we have

$$f(\mathbf{x}|\mathbf{w}, b) = \text{sgn}(\mathbf{x} \cdot \mathbf{w} + b) \approx f^*(\mathbf{x}). \tag{2.1}$$

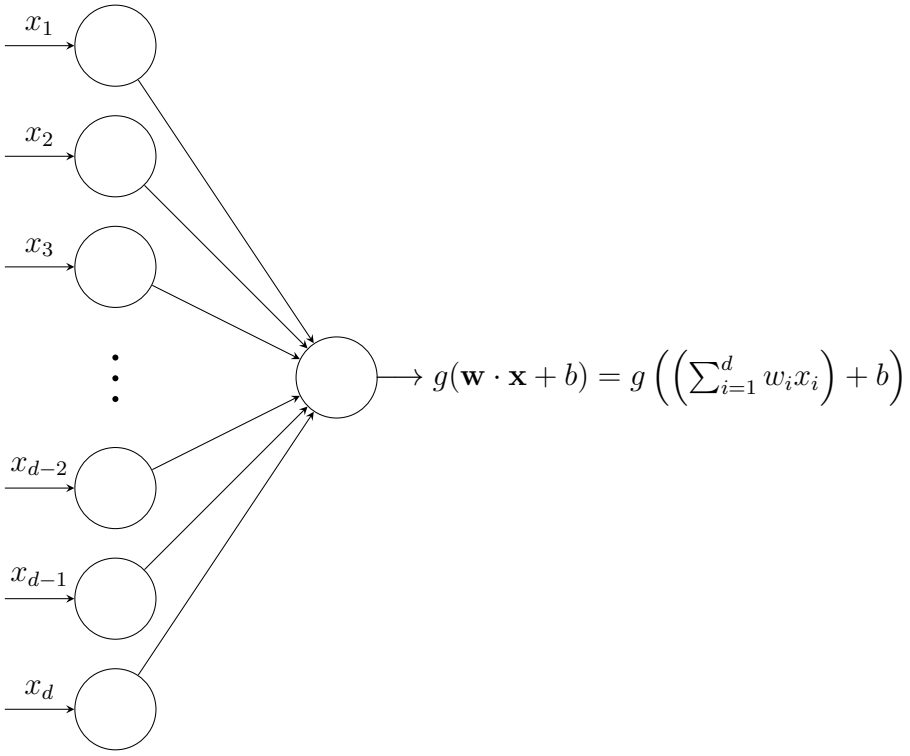
We emphasize that the perceptron simply calculates a linear combination of entries in $\mathbf{x} = (x_1, \ldots, x_d)^T \in \mathbb{R}^d$, with an added constant b, as shown in Figure 2.1 below. We view the sgn function as an intepretation of this 'final' value or output; for the sake of binary classification (negative or positive).

Figure 2.1: Perceptron output.



We begin the construction of a deep feedforward neural network by considering an existing single perceptron and applying what is known as an *activation function* g on $\mathbf{w} \cdot \mathbf{x} + b$, as shown in Figure 2.2 below.

Figure 2.2: Perceptron output (with nonlinearity).



The terms *activation function* and *nonlinearity* are often used interchangeably. This is because g is usually chosen to be nonlinear. Some common examples of g are shown in the list below. Notice that when $g(x) = \sigma(x)$, we obtain the logistic function. Therefore, some aspects of deep learning can be viewed as a generalisation of the logistic regression.

(i) Sigmoid,

$$\sigma(x) = \frac{1}{1+e^x}. \tag{2.2}$$

(ii) Tangent hyperbolic,

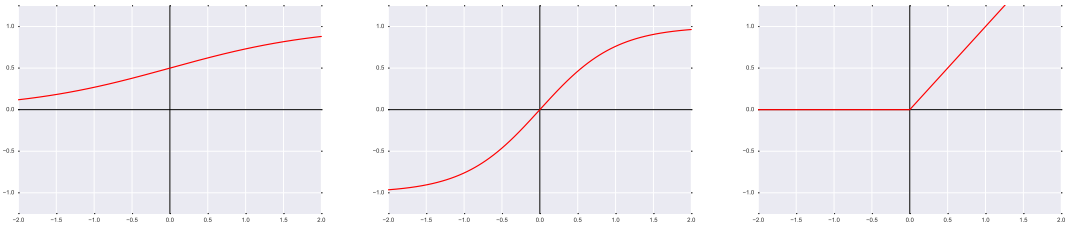
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.3}$$

(iii) Rectified linear unit,

$$\mathrm{ReLU}(x) = \max\{0, x\}. \tag{2.4}$$

Figure 2.3: Activation functions.

(i) Sigmoid (ii) Tangent hyperbolic (iii) Rectified linear unit

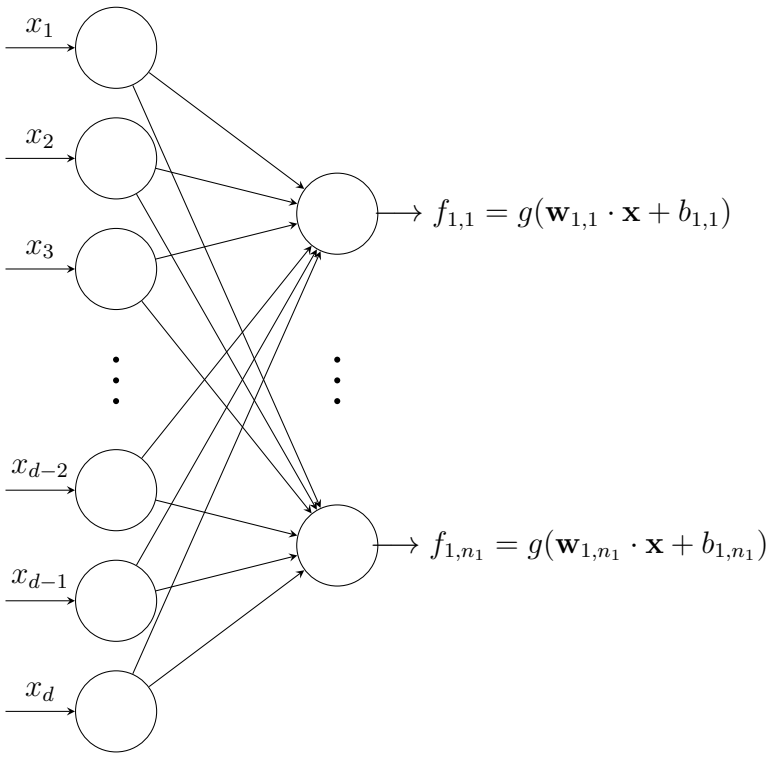


Next, instead of having a single dimensional output, consider one with dimension n_1 and each dimension acts the same way as before but with its own independent weight and bias. To be more precise, denote $f_{1,i} = g(\mathbf{w}_{1,i} \cdot \mathbf{x} + b_{1,i})$ for $1 \leq i \leq n_1$ and let

$$\tilde{\mathbf{f}}_1 = f_1(\mathbf{x}) = \begin{bmatrix} f_{1,1} \\ \vdots \\ f_{1,n_1} \end{bmatrix}. \tag{2.5}$$

The vector $f_1(\mathbf{x})$ represents the first *dense layer* with n_1 neurons after the *input layer* $\mathbf{x}$. A visualisation of this layer is provided in Figure 2.4 below.

Figure 2.4: First dense layer with n_1 neurons.



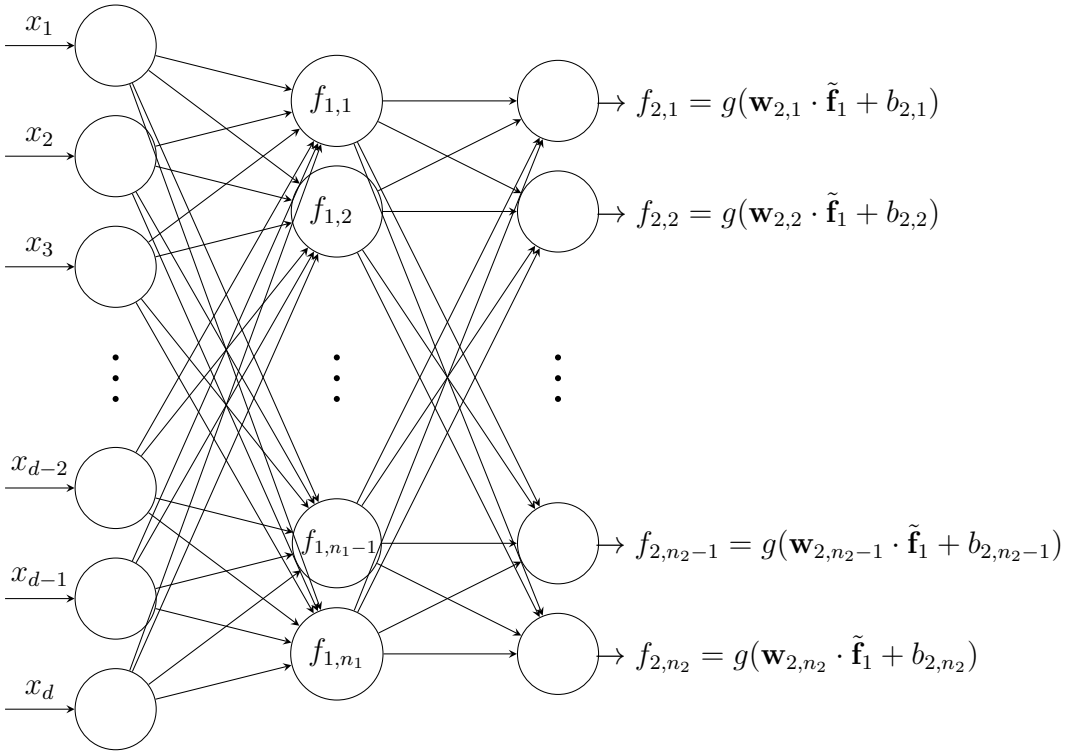
For the sake of elucidation, we apply this step once more before generalizing.

Define a *second* dense layer with n_2 neurons similarly via $f_{2,i} = g(\mathbf{w}_{2,i} \cdot \tilde{\mathbf{f}}_1 + b_{2,i})$ for $1 \leq i \leq n_2$, and similarly

$$\tilde{\mathbf{f}}_2 = f_2 \circ f_1(\mathbf{x}) = \begin{bmatrix} f_{2,1} \\ \vdots \\ f_{2,n_2} \end{bmatrix}. \tag{2.6}$$

The relationship between the layers thus far can be visualised in Figure 2.5 below.

Figure 2.5: Second dense layer with n_2 neurons.

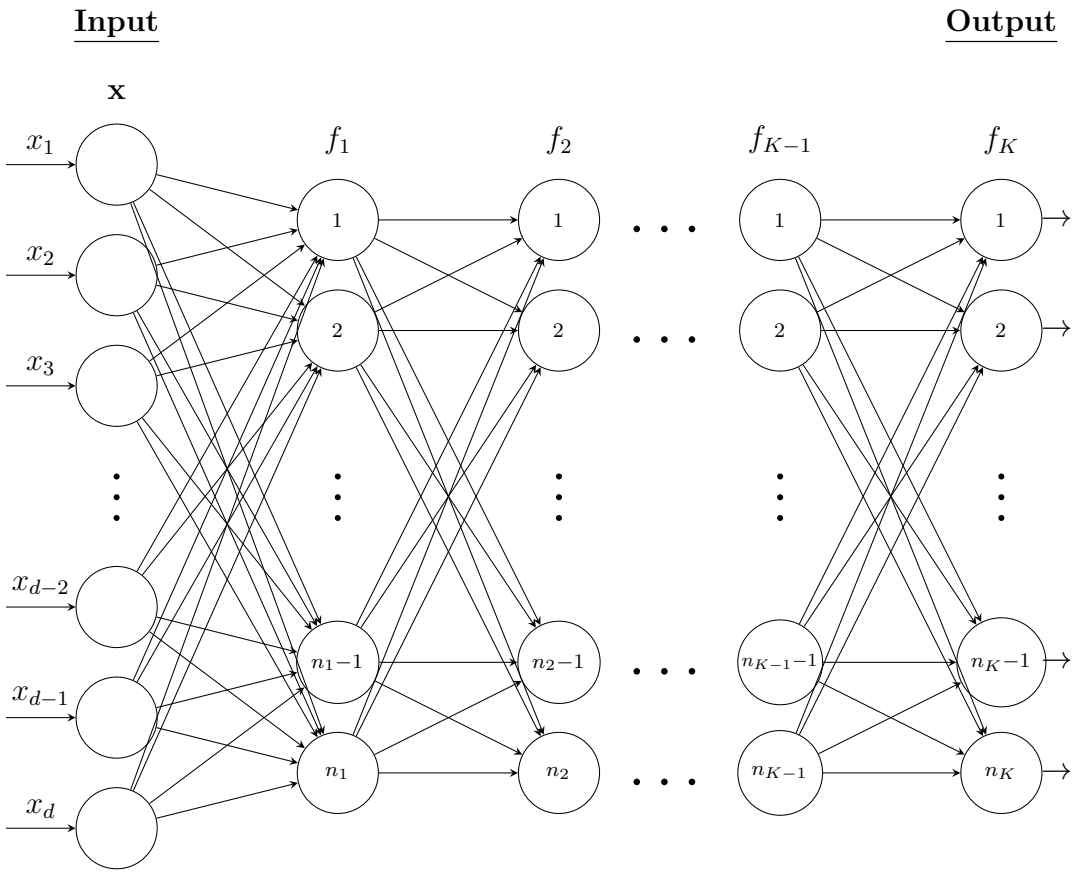


In general, a deep feedforward neural network with K layers can be viewed as a composition of functions $f = f_K \circ \cdots \circ f_1$, whereby $\tilde{\mathbf{f}}_k = f_k \circ \cdots \circ f_1(\mathbf{x})$ represents the k-th dense layer with n_k neurons after the input is given by

$$\tilde{\mathbf{f}}_k = f_k \circ \cdots \circ f_1(\mathbf{x}) = \begin{bmatrix} f_{k,1} \\ \vdots \\ f_{k,n_k} \end{bmatrix} = \begin{bmatrix} g(\mathbf{w}_{k,1} \cdot \tilde{\mathbf{f}}_{k-1} + b_{k,1}) \\ \vdots \\ g(\mathbf{w}_{k,n_k} \cdot \tilde{\mathbf{f}}_{k-1} + b_{k,n_k}) \end{bmatrix}. \tag{2.7}$$

This information is summarized in Figure 2.6.

Figure 2.6: Deep feedforward neural network with K dense layers.



Each neuron of a subsequent layer takes a linear combination of all the neurons in the previous layer (with added bias) and applies a nonlinearity g. This nonlinearity g may in fact vary from layer to layer. The final number of neurons of the *output layer* n_K depends on the purpose of this model.

Unlike the perceptron, which can only be used for binary classification, this model serves much more general purposes such as classication over multiple categories or regression. Common practice for classification is to set n_K equal to the number of distinct categories, apply the *softmax* function

$$\varphi_j(\mathbf{z}) = \frac{e^{z_j}}{\sum_{i=1}^{N_K} e^{z_i}}, \tag{2.8}$$

where $\mathbf{z}$ is the output layer, and then take the predicted value (or label) to be

$$\arg\max_j \{\varphi_j(\mathbf{z}) : 1 \leq j \leq N_K\}. \tag{2.9}$$

If our aim was regression on the other hand, we could simply leave the output layer as a vector in the required dimension.

The overall function representing the networks output is denoted by $f(\mathbf{x}\,|\,\theta)$, where $f = f_K \circ \cdots \circ f_1$ and θ denotes the vector of all the weights and biases involved in the network. Our purpose is to be able to approximate some function $f^* : \mathbf{X} \to \mathbf{Y}$. Note that by virtue (or curse!) of the high-dimensionality of parameters θ, the function f has the ability approximate extremely complex functions. Therefore, the problem reduces to finding an optimal value of θ – which will be discussed in the next subsection.

Remark 2.1.1. For deep feedforward neural networks, the number of layers (depth) K and neurons per layer n_k is usually very large. Naturally, we would expect the dimension of θ to be enormous. The large depth of such networks is the reason for term 'deep learning'.

Remark 2.1.2. The overall structure of a deep feedforward neural network can be modelled as a directed acyclic graph (DAG). Relationships are also easily expressed by matrix notation and applying nonlinearities in a 'vectorized' fashion. This allows for computations to be performed in parallel, hence the important role of GPU/parallel programming in deep learning.

2.2 Loss functions

The network $f(\mathbf{x}\,|\,\theta)$ is capable of expressing a vast amount of complexity, depending on θ. Our goal is to find θ such that our network $f(\mathbf{x}\,|\,\theta)$ achieves an acceptable approximation of $f^* : \mathbf{X} \to \mathbf{Y}$. We start by defining a *training dataset* $\mathcal{D} \subseteq \mathbf{X}$. The idea is to *train* the network such that it *learns* an optimal value of θ based on its experience with $\mathcal{D}$. As means to fulfill our objective, we define a suitable *loss function*[5] over a subset $S \subseteq \mathbf{X}$,

$$\mathcal{L}(\theta, S), \tag{2.10}$$

and define our optimal value of θ to be

$$\theta^* = \arg\min_{\theta}\{\mathcal{L}(\theta, \mathcal{D})\}. \tag{2.11}$$

This optimization problem is nontrivial, and it is standard practice to only offer approximations $\hat{\theta} \approx \theta^*$. This will involve the *backpropagation* optimization algorithm, to be discussed over the next two subsections.

Furthermore, we are usually satisfied with an approximation of some local minima – rather than global. One reason for this is so that we avoid the problem of overfitting with the training data $\mathcal{D}$, as we would like the trained model $f(\mathbf{x}, \hat{\theta})$ to be robust over all observations in $\mathbf{X}$ as well. Such robustness is often measured by observing the models performance over a *test dataset* $\mathcal{T} \subseteq \mathbf{X}$.

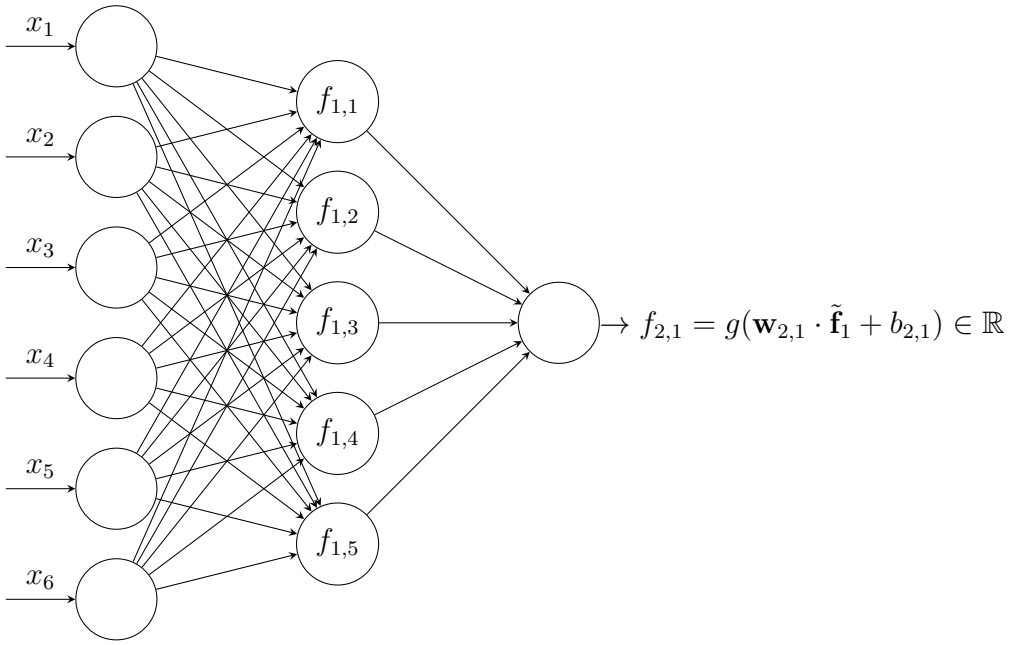
Note that the choice of $\mathcal{L}$ is obviously dependent on the problem at hand (classification or regression) as well as the overall architecture of our network. We

[5]Sometimes referred to as a *cost* function, and can be seen as a measure of the networks performance over some dataset.

end this section by providing some examples of deep feedforward neural networks, with an emphasis on the respective loss functions $\mathcal{L}$ used.

Example 2.2.1. Suppose we wish to approximate[6] some function $f^* : \mathbb{R}^6 \to \mathbb{R}$. In other words, our training dataset is some $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^6 \times \mathbb{R}$. One possible[7] configuration is to use a network with *one hidden layer* as shown in Figure 2.7 below.

Figure 2.7: Regression with one hidden layer.



Note that in this example, $\theta = (\mathbf{w}_{1,1}, b_{1,1}, \ldots, \mathbf{w}_{1,5}, b_{1,5}, \mathbf{w}_{2,1}, b_{2,1})$. In hope of achieving a good prediction $f(\mathbf{x} \,|\, \theta) = \hat{y}_{\mathbf{x}} \approx y_{\mathbf{x}} = f^*(\mathbf{x})$ we might propose the *mean squared error* as our loss function,

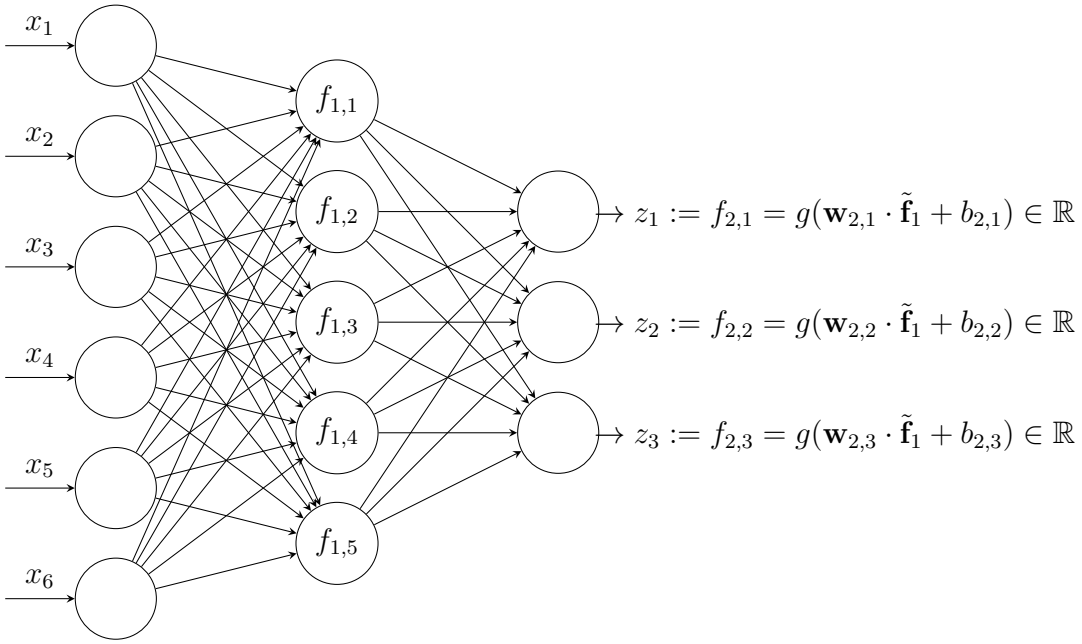
$$\mathcal{L}(\theta, S) = \frac{1}{|S|} \sum_{\mathbf{x} \in S} \mathcal{L}_{\mathbf{x}}(\theta) = \frac{1}{|S|} \sum_{\mathbf{x} \in S} (\hat{y}_{\mathbf{x}} - y_{\mathbf{x}})^2. \tag{2.12}$$

Example 2.2.2. Suppose we wish to instead classify data based on their numerical features $\mathbf{x} \in \mathbb{R}^6$. Assume the classification is 3-way; that is we assume the labels are of the form $y = 1, 2, 3$. Assume a similar configuration as before, but with three dimensional output instead. The specific layout is shown in Figure 2.8.

[6] This can be seen as a regression problem.

[7] This is a very simple example. In practice, networks are much deeper (more layers), and each layer would consist of many neurons.

Figure 2.8: Classifier with one hidden layer.



In practice, each element the output layer $\mathbf{z} = (z_1, z_2, z_3)^T$ is normalized by the *softmax function* φ_j, $j = 1, 2, 3$. This gives us a probability vector since for each j

$$p_{\mathbf{x},j} := \varphi_j(\mathbf{z}) = \frac{e^{z_j}}{e^{z_1} + e^{z_2} + e^{z_3}}. \tag{2.13}$$

The categorical prediction of $y_{\mathbf{x}}$, which we denote by $\hat{y}_{\mathbf{x}}$, is usually taken to be

$$\hat{y}_{\mathbf{x}} = \arg\max_j \{p_{\mathbf{x},j} : j = 1, 2, 3\}. \tag{2.14}$$

The loss function however, is usually based on the softmax output. For example, we might introduce the *categorical cross-entropy*

$$\mathcal{L}(\theta, S) = \frac{1}{|S|} \sum_{\mathbf{x} \in S} \mathcal{L}_{\mathbf{x}}(\theta) = \frac{1}{|S|} \sum_{x \in S} \sum_{j=1}^{3} -t_{\mathbf{x},j} \cdot \log(p_{\mathbf{x},j}), \tag{2.15}$$

where we define the *target variable* $t_{\mathbf{x},j}$ to be

$$t_{\mathbf{x},j} = \begin{cases} 1 & \text{if } y_{\mathbf{x}} = j \\ 0 & \text{otherwise.} \end{cases} \tag{2.16}$$

Remark 2.2.3. One might notice that the mean squared error and categorical cross-entropy share similar traits. For example, they are both nonnegative and tend to zero as the networks improves its predictions. Furthermore they can be also expressed as a mean of loss functions. These are intuitive properties which we should expect from a loss function [20].

2.3 Gradient-based learning

We discuss relevant theory and algorithms for the purpose of finding $\hat{\theta}$ mentioned previously. This optimization process is often referred to as *training* or a *learning process*. The methods we discuss rely heavily on the concept of a gradient (vector), hence the title 'Gradient-based learning'.

Supposed we are given loss function $\mathcal{L}(\theta, \mathcal{D})$ to minimize (with respect to θ). Furthermore, assume θ is a high-dimensional vector[8] – so that even if $\mathcal{L}$ is differentiable, standard analytical methods to find the minimum are not be feasible [20]. As such, we resort to approximate minimization.

The general approach used for this type of problem is via gradient based optimization [10]. For clarity, we denote the parameters involved in our model (weights and biases) to be $\theta = (\theta_1, \ldots, \theta_p)^T \in \mathbb{R}^p$. We start with the *gradient descent* method; which involves computation of the gradient

$$\nabla_\theta \mathcal{L}(\theta, \mathcal{D}) := \begin{bmatrix} \dfrac{\partial}{\partial \theta_1} \mathcal{L}(\theta, \mathcal{D}) \\ \vdots \\ \dfrac{\partial}{\partial \theta_p} \mathcal{L}(\theta, \mathcal{D}) \end{bmatrix}. \tag{2.17}$$

Gradient descent[9], is a iterative method which uses some insight from $\nabla_\theta \mathcal{L}(\theta, \mathcal{D})$ to update terms $\theta^{(k)}$ in a direction which minimizes $\mathcal{L}$. Considering the directional derivative in direction $\mathbf{u}$ defined by $\mathbf{u} \cdot \nabla_\theta \mathcal{L}(\theta, \mathcal{D})$, we have

$$\begin{aligned} \arg \min_{\mathbf{u},\, \|\mathbf{u}\|=1} \{\mathbf{u} \cdot \nabla_\theta \mathcal{L}(\theta, \mathcal{D})\} &= \arg \min_{\mathbf{u},\, \|\mathbf{u}\|=1} \{\|\mathbf{u}\| \|\nabla_\theta \mathcal{L}(\theta, \mathcal{D})\| \cos \alpha\} \\ &= \arg \min_{\mathbf{u},\, \|\mathbf{u}\|=1} \{\cos \alpha\}, \end{aligned} \tag{2.18}$$

where α denotes the angle between $\mathbf{u}$ and $\nabla_\theta \mathcal{L}(\theta, \mathcal{D})$. It is clear that the minimizer for the above expression should be $\mathbf{u} = -c \nabla_\theta \mathcal{L}(\theta, \mathcal{D})$ for some normalization constant $c > 0$.

In other words, we can decrease $\mathcal{L}$ by taking steps in the negative gradient direction. Therefore, the method is carried out by choosing some initial $\theta^{(1)}$ and updating recursively:

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla_\theta \mathcal{L}(\theta^{(k)}, \mathcal{D}). \tag{2.19}$$

In the above expression $\eta_k > 0$ is referred to as the learning rate, possibly set constant for all $k \in \mathbb{N}$. Note that this method will converge once $\nabla_\theta \mathcal{L}(\theta, \mathcal{D}) = \vec{0}$.

[8] Recall that θ is the vector of all weights and biases in the network $f(\mathbf{x} \,|\, \theta)$. In extremely deep networks, the number of weights and biases easily exceeds $1,000,000,000$.

[9] Also known as the *method of steepest descent*.

Algorithm 2.1: Gradient Descent

Data: initial parameters $\theta^{(1)}$, iterations K and learning rate η_k
Result: approximate minimizer $\hat{\theta}$
for $k = 1, \ldots, K-1$ **do**
  compute $\nabla_\theta \mathcal{L}(\theta^{(\theta)}, \mathcal{D})$;
  `/* update parameters */`
  $\theta^{k+1} \leftarrow \theta^k + \eta_k \nabla_\theta \mathcal{L}(\theta^{(k)}, \mathcal{D})$;
$\hat{\theta} \leftarrow \theta^{(K)}$;
return $\hat{\theta}$;

The problem with using 'plain vanilla' gradient descent method is the large size of training dataset $\mathcal{D}$. As noted in the expression $\nabla_\theta \mathcal{L}(\theta, \mathcal{D})$, we are tasked with computing the gradient vector with respect to parameters θ of the loss function over the entire training dataset $\mathcal{D}$. As an example, if $\mathcal{L}$ is taken to be the mean squared error associated with our regression, then time complexity for a single update increases linearly with the dataset size $|\mathcal{D}|$. Optimization occurs very slowly as a result of this [20].

Stochastic gradient descent is employed to speed up this optimization process (or *learning process*). This method is based on the idea of *randomly* generating subsets of the main training dataset $\mathcal{D}$ denoted by $S_1, \ldots, S_m$. These subsets are referred to as *mini-batches* and are usually subject to the condition that they are pairwise disjoint with $S_1 \cup S_2 \cup \cdots \cup S_m = \mathcal{D}$, and each mini-batch size $|S_i|$ is large enough.

One common property of a loss function $\mathcal{L}$, such as the mean squared error or categorical cross entropy, is that they can be written a the mean of loss functions $\mathcal{L}_{\mathbf{x}}$. That is,

$$\mathcal{L}(\theta, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}_{\mathbf{x}}(\theta). \tag{2.20}$$

By invoking this property as well as the assumed large magnitude[10] of $|S_i|$ for each $1 \leq i \leq m$ we obtain

$$\mathcal{L}(\theta, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}_{\mathbf{x}}(\theta) \approx \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathcal{L}_{\mathbf{x}}(\theta) = \mathcal{L}(\theta, S_i). \tag{2.21}$$

Similarly, we obtain

$$\nabla_\theta \mathcal{L}(\theta, \mathcal{D}) \approx \nabla_\theta \mathcal{L}(\theta, \mathcal{S}_i). \tag{2.22}$$

Therefore, in stochastic gradient descent, we perform each update using the gradient $\nabla_\theta \mathcal{L}(\theta, \mathcal{S}_i)$ rather than $\nabla_\theta \mathcal{L}(\theta, \mathcal{D})$. Note that each mini-batch S_i is generally a lot smaller compared to the entire training dataset $\mathcal{D}$, and this allows for much faster updates.

[10] Or simply the law of large numbers (LLN).

During training, an *Epoch* refers to a the process whereby all elements of training dataset $\mathcal{D}$ are exhausted via random mini-batches $S_1, S_2, \ldots, S_m$. The total number of Epochs is a training parameter, denoted by E.

Algorithm 2.2: Stochastic Gradient Descent

Data: initial parameters θ', number of Epochs E and learning rate η_k
Result: approximate minimizer $\hat{\theta}$
for $k = 1, \ldots, E$ **do**
  generate mini-batches $S_1, \ldots, S_m \subseteq \mathcal{D}$;
  for $i = 1, \ldots, m$ **do**
    compute $\nabla_\theta \mathcal{L}(\theta', S_i)$;
    `/* update parameters */`
    $\theta' \leftarrow \theta' + \eta_k \nabla_\theta \mathcal{L}(\theta', S_i)$;
$\hat{\theta} \leftarrow \theta'$;
return $\hat{\theta}$;

2.4 Backpropagation

We are left with the concern of being able to compute the gradient $\nabla_\theta \mathcal{L}(\theta, S_i)$ in a practical manner. *Backpropagation* refers to a fast algorithm used for computing the gradient of $\mathcal{L}(\theta, S_i)$. Justification of the algorithm requires the following concepts and lemmas[11] [20]:

- For neuron j of layer l in the network, its weighted input[12] is denoted by z_j^l.
- For neuron j of layer l in the network, its output[13] is denoted by a_k^l.
- For neuron j of layer l in the network, the associated inputs weights and bias are denoted by w_j^l and b_j^l. Note that w_j^l is a vector, whose k-th element is denoted by w_{jk}^l.
- The *weight matrix* W^l for layer l is a matrix whose columns are w_j^l for all neurons $1 \leq j \leq n_l$ of layer l.

Definition 2.4.1. The error term δ_j^l associated with neuron j of layer l in the network is defined to be

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l}. \tag{2.23}$$

[11] Also known as the *four backpropagation equations*.
[12] Linear combination of the outputs of neurons from the previous layer, plus bias.
[13] We have the relation $a_k^l = g(z_k^l)$ where g is the activation function applied in the layer.

Lemma 2.4.2. For any neuron j of layer l with activation function g,

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial a_j^l} g'(z_j^l). \tag{2.24}$$

Proof. Apply the chain rule and note that $\dfrac{\partial a_k^l}{\partial z_j^l} = 0$ whenever $k \neq j$,

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial a_k^l} \frac{\partial a_k^l}{\partial z_j^l} = \frac{\partial \mathcal{L}}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial \mathcal{L}}{\partial a_j^l} g'(z_j^l). \tag{2.25}$$

□

Lemma 2.4.3. For any layer l,

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot g'(z^l), \tag{2.26}$$

where $\odot$ denotes the *Hadamard* product and

$$\delta^l = \begin{bmatrix} \delta_1^l \\ \vdots \\ \delta_{n_l}^l \end{bmatrix}, \; g'(z^l) = \begin{bmatrix} g'(z_1^l) \\ \vdots \\ g'(z_{n_l}^l) \end{bmatrix}. \tag{2.27}$$

Proof. Apply the chain rule once again to get

$$\delta_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}. \tag{2.28}$$

By definition,

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{1+1} = \sum_j w_{kj}^{l+1} g(z_j^l) + b_k^{1+1} \implies \frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} g'(z_j^l). \tag{2.29}$$

Therefore,

$$\delta_j^l = \sum_k w_{kj}^{l+1} g'(z_j^l) \delta_k^{l+1} \implies \delta^l = ((W^{l+1})^T \delta^{l+1}) \odot g'(z^l). \tag{2.30}$$

□

Lemma 2.4.4. For any neuron j of layer l,

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \delta_j^l. \tag{2.31}$$

Proof. Note that

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \implies \frac{\partial z_j^l}{\partial b_j^l} = 1. \tag{2.32}$$

By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \frac{\partial z_j^l}{\partial b_j^l}\frac{\partial \mathcal{L}}{\partial z_j^l} = \frac{\partial z_j^l}{\partial b_j^l}\delta_j^l = \delta_j^l. \tag{2.33}$$

□

Lemma 2.4.5. For any neuron j of layer l,

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l. \tag{2.34}$$

Proof. Similarly,

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \implies \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}. \tag{2.35}$$

By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^l} = \frac{\partial z_j^l}{\partial w_{jk}^l}\frac{\partial \mathcal{L}}{\partial z_j^l} = \frac{\partial z_j^l}{\partial w_{jk}^l}\delta_j^l = a_k^{l-1}\delta_j^l. \tag{2.36}$$

□

Backpropagation is a direct consequence of Lemma 2.4.2, 2.4.3, 2.4.4 and 2.4.5. Note that because of the backpropagation algorithm we are now able to make practical implementations of the stochastic gradient descent. The exact details can be found in Algorithm 2.3.

Algorithm 2.3: Backpropagation.

Data: deep feedforward neural network with L layers and input $\mathbf{x}$.
Result: gradient $\nabla_\theta \mathcal{L}_{\mathbf{x}}(\theta)$.
`/* feedforward */`
for $l = 1, 2, 3, \ldots, L$ **do**
  `/* note: input layer is labelled 0 */`
  Compute $z^l = w^l a^{l-1} + b^l$;
  Compute $a^l = g(z^l)$;
`/* output error */`
Compute $\delta^L = \nabla_a \mathcal{L} \cdot g'(z^L)$;
`/* backpropagate */`
for $l = L-1, L-2, ..., 1$ **do**
  Compute $\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot g'(z^l)$;
  `/* update gradient components */`
  $\frac{\partial \mathcal{L}}{\partial w^l_{jk}} \leftarrow a^{l-1}_k \delta^l_j$;
  $\frac{\partial \mathcal{L}}{\partial b^l_j} \leftarrow \delta^l_j$;
return $\nabla_\theta \mathcal{L}_{\mathbf{x}}(\theta)$

Remark 2.4.6. One might note, that if a neuron outputs 0, then there will be no back propagation on the weights that define this particular neuron. This leads to two main points:

1. Weights are often randomised upon initialisation of the network (before training). Usually, each weight is set to some small, but *non-zero*, random floating point. It is generally a bad idea to set all the weights to zero.

2. Dropout, and other architectures discussed later on rely on this fact.

3 Adding More Layers

In the previous section we introduced the fundamentals of neural networks, but only spoke about dense layers. In this section we will discuss other useful types of layers for deep learning. These layers are generally used in conjuction with one another in deep neural networks.

3.1 Dropout Layers

Dropout is an effective technique used to prevent overfitting in neural networks [12, 22]. For a more detailed description of overfitting, see Section 5.1. In this subsection, we discuss the general structure of a dropout layer. Recall that the output of a single neuron in a dense layer can be represented mathematically via

$$\sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i). \tag{3.1}$$

Here, $\mathbf{x}$ refers to the input (output of the previous layer), $\mathbf{w}_i$ and b_i refers to the weights and bias tied to the neuron, and σ refers to the nonlinearity applied. In a dropout layer, we assign each output with an independent Bernoulli random variable $z_i \sim \text{Bernoulli}(p)$ such that

$$z_i \cdot \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i). \tag{3.2}$$

The general training and testing procedure for neural networks with dropout layers are as follows:

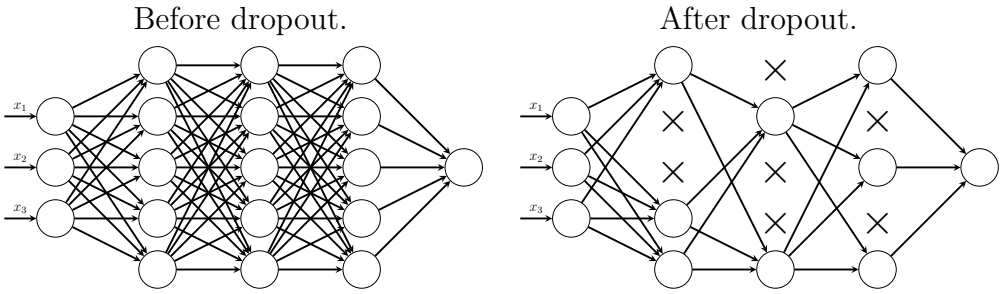
1. We train the network using stochastic gradient descent. During training, for each mini-batch we sample the independent z_i's. This effectively results in a thinned network (see Figure 3.1). The network will learn it's parameters based on this framework.

2. During test time, we deterministically set all $z_i = 1$. In other words, our final network will be a full network utilizing all neurons. Based on the expectation of z_i, we scale the the each output by a factor of p. So the output of each neuron during test time is

$$p \cdot \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i). \tag{3.3}$$

Note that based on the the backpropagation algorithm, any parameter which is not used will result in a zero gradient. Therefore, if a realization of z_i is 0 for a particular neuron then there will be no updates on its weight $\mathbf{w}_i$ and bias b_i. This is akin to randomly 'ignoring' neurons during training.

Figure 3.1: Network with three dropout layers.



3.2 Convolutions and Pooling

Another useful technique in deep learning is the usage of convolutional and pooling layers [5]. This technique is particularly useful for image classification. In this subsection we will explain the basic idea behind these techniques.

First note that, for a given layer (e.g. input layer) we can define an ordering or *location* of each input. For example, if the input is a single vector then we can label each component from top to bottom. Similarly, if the input is a matrix then we can label each component based on some two dimensional coordinate (i, j).

A convolutional layer placed directly after this layer consists of neurons which are not fully connected to all input values, but rather are only connected to a restricted range of inputs within a specified location. This restriction is determined by parameters such as filter (kernel) size, stride, and padding (see [5] for more details on Convolution Arithmetic). Refer to Figure 3.2 for a visualization of this.

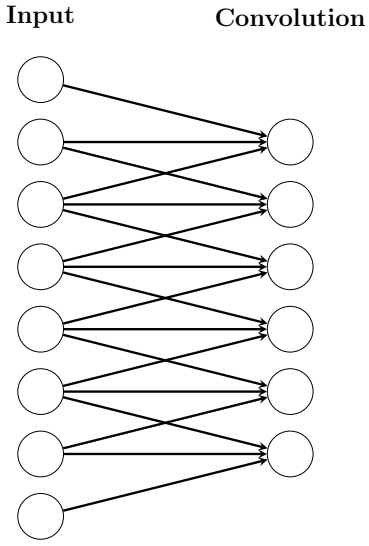


Figure 3.2: Example of 1-dimensional convolution. Notice how this layer is not fully connected with its input.

The idea of pooling is very similar to applying convolutions, and can be found in our Reference [5] as well. Pooling layers operate in a similar fashion to convolutional layers[14]. The only difference is: instead of taking the usual linear combination followed by nonlinearity on the inputs, each neuron calculates a statistic of its input.

For example, we could take the average value of inputs in a given filter (this is known as average-pooling). Another possibility is to take the maximum of values in a given filter (this is known as max-pooling). Refer to Figure 3.3 for a visualization of this procedure.

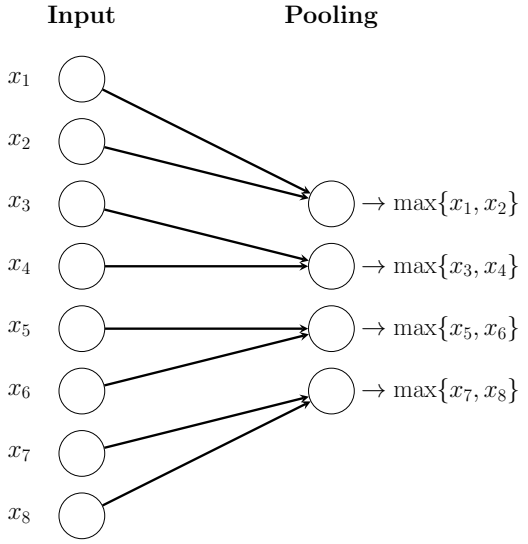


Figure 3.3: Example of a 1-dimensional max-pooling layer.

We end this section with a some remarks on convolutions and pooling. Firstly, the motivation behind these convolutions and pooling is to be able to identify specific features of data. This is done by restricting each neuron to only a subset of the input. Also, note that based on the shape of the input (or previous hidden layer), convolutions can be generalized to n-dimensions.

[14] Although some pooling layers tend to have the property that filters are nonoverlapping. For example, max-pooling.

4 Classification Problems

In this section we discuss some classification problems and some approaches using deep learning.

4.1 Iris Dataset

The Iris dataset consists of 150 instances of species of the Iris flower. Each instance consists of four measurements[15] and the respective class[16] of the flower. There are only 3 classes, and 50 instances of each class. For this example, we show how a neural network can be trained to classify the Iris species based on its four measurements.

We used simple neural network with only one hidden (dense) layer of 50 neurons, with ReLU nonlinearity applied. The training dataset is made up of a random selection of 25 per class, and the test dataset (for validation) consists of the remaining unused data. For training, the standard stochastic gradient descent is applied with mini-batch size of 25 and a total of 300 epochs. The network achieves a final test accuracy of 97.3%.

Although this dataset is fairly simple (and small!), it shows us clearly how neural networks can be applied in other similar pattern recognition problems. For an overview of the model, as well as training and test/validation errors see Figure 4.1. For the code implementation of the network and procedure, see:

https://github.com/mollymr305/iris-classification

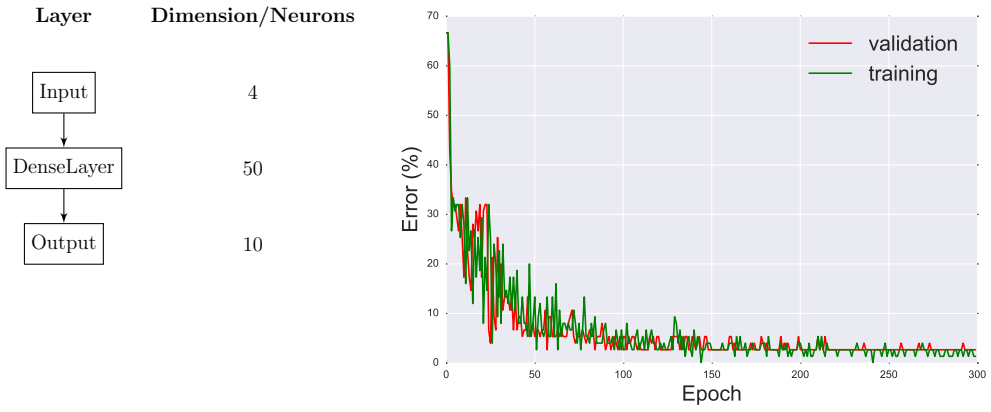


Figure 4.1: Iris classifier using neural networks.

[15] Sepal length, sepal width, petal length, and petal width.
[16] Iris setosa, Iris virginica, Iris versicolor.

4.2 MNIST Handwritten Digit Database

The MNIST Handwritten Digit Database [18] consists of $70,000$ handwritten digit images. Each image is represented by a numeric vector (or matrix) $\mathbf{x} \in \mathbb{R}^{28\times 28}$ and tagged with an integer label $y \in \{0,1,2,3,4,5,6,7,8,9\}$.

Figure 4.2: MNIST Handwritten digit, with $y = 5$.

There have been multiple approaches and types of architectures used to solve this problem [2]. We tested a network consisting of a series of convolutional layers with max pooling, followed by dropout and a dense layer before reaching a softmax output. The learning rate was fixed at 0.001 throughout all Epochs, and we applied the ReLU nonlinearity in the respective hidden layers. The training (optimization) methodology is standard and straightforward:

1. Split the dataset into a training set $\mathcal{D}_{\text{train}}$ of size $50,000$, a validation set $\mathcal{D}_{\text{val}}$ of size $10,000$ and test set $\mathcal{D}_{\text{test}}$ of size $10,000$.

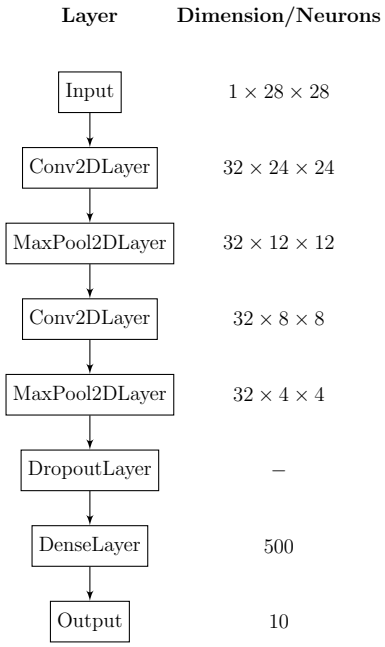
2. Train the neural network over $\mathcal{D}_{\text{train}}$ whilst monitoring its performance over $\mathcal{D}_{\text{val}}$. Use a mini-batch size of 500, and set the total number of Epochs to be $E = 500$.

3. After training, assess the model's accuracy/error on $\mathcal{D}_{\text{test}}$.

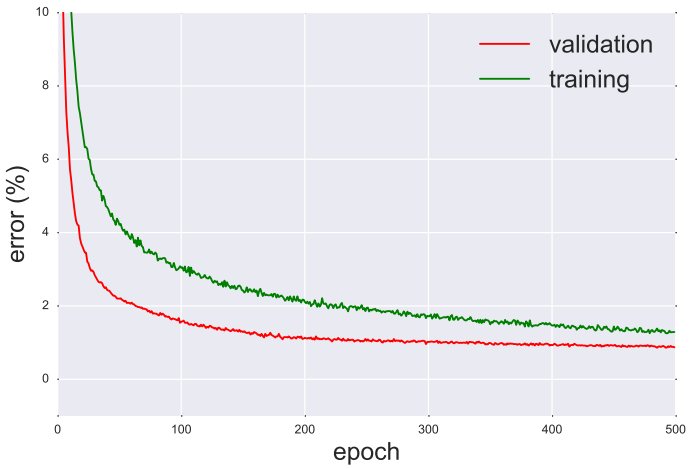
For more information on the model implementation and results[17] see Figure 4.3. The final training, validation and test accuracy after 500 Epochs is shown in Figure 4.4. A confusion matrix is also included as a more detailed report of the networks accuracy; in terms of true versus predicted labels. The full code implementation for this model can be found at:

https://github.com/mollymr305/mnist-classification

[17]Results are based on a single trial and therefore one should expect a small variance in observed results from the exact same model. This is due probabilistic algorithms involved, such as in stochastic gradient descent and applying dropout.



(i) Network configuration.



(ii) Training and validation error.

Figure 4.3: Implementation and results on MNIST data.

Dataset	Accuracy
Training ($\mathcal{D}_{\text{train}}$)	**98.71%**
Validation ($\mathcal{D}_{\text{val}}$)	**99.13%**
Test ($\mathcal{D}_{\text{test}}$)	**99.36%**

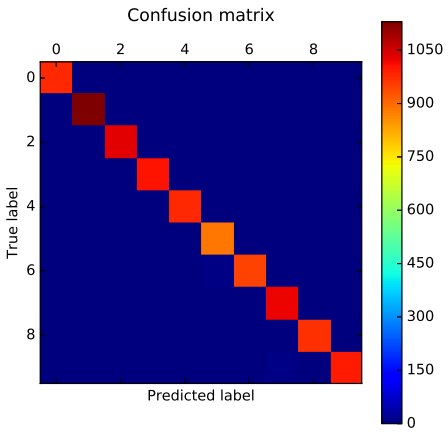


Figure 4.4: Network accuracy and confusion matix.

4.3 CIFAR-10 Dataset

The CIFAR-10 dataset contains a total of 60,000 coloured images, each represented by a vector $\mathbf{x} \in \mathbb{R}^{3\times32\times32}$ and given a label $y = \{1, \ldots, 10\}$ representing a category in which the image belongs to [17]. Note that there are a total of 10 categories

(one example from each category is shown in Figure 4.5) and this is not to be confused with CIFAR-100 which has a total of 100 classes.

Of the 60,000 images: a total of 50,000 make up the training dataset $\mathcal{D}_{\text{train}}$ and 10,000 make up the test dataset $\mathcal{D}_{\text{test}}$. Furthermore the test dataset is evenly distributed in a sense that it contains exactly 1,000 (random) images from each class.

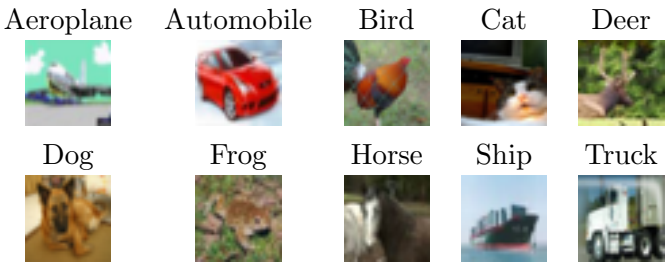


Figure 4.5: One example from each category in the CIFAR-10 dataset.

Readers might note that this classification problem (CIFAR-10) is *a lot* more difficult to solve compared to the classification of MNIST handwritten digits. For example, architectures similar to that in MNIST classification perform poorly on this dataset. One simple reason for this is the larger dimensionality of input data.

In general, best approaches for this type of complex classification task involves much deeper networks, which we will discuss later on [11]. A detailed description and explanation of such an approach will be provided in Section 5.3. We first elaborate on some of the other well-known ways of making improvements to the learning process:

1. For some classes of images, the mirror (or relfection) of each image is again a 'valid' image. Certainly this is not the case for all images, such as MNIST handwritten digit images. For the case of CIFAR-10 images this idea works, and therefore allows us to effectively *double* our training dataset size to 100,000 by adding mirrored images. This allows the network to learn more key features and perform better in classification. Refer to Figure 4.6 for a sample CIFAR-10 image and its mirrored version.

2. Another practice related to data augmentation is the random cropping of training images during each mini-batch generation [15, 23]. This technique generally improves learning by reducing model overfitting.

We postpone the description of the actual neural network model for classifying CIFAR-10 images, and its implementation to Section 5.3.

Figure 4.6: Original and mirrored image example from CIFAR-10. Both images are, obviously valid images of an automobile.

5 Improvements

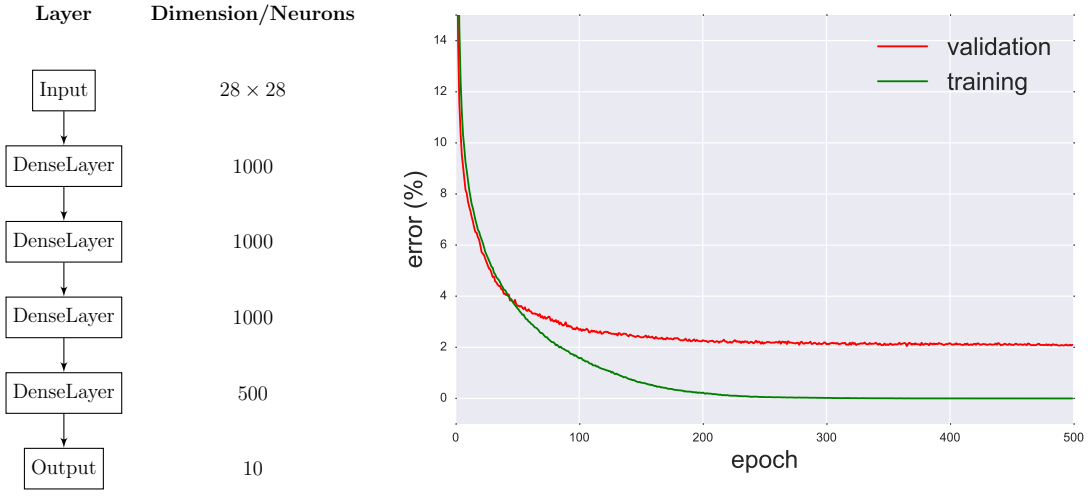
In this section, we discuss relevant challenges faced when designing neural networks - and more importantly, practical methods of overcoming such challenges.

5.1 Regularization

Overfitting[18] refers to the problem whereby during the training period, a neural network shows improvement over training dataset but not over the test or validation dataset [20]. In other words, the network is unable to *generalize* what is has learnt from the training dataset to other datasets.

This problem is often associated with the more complex, deep networks. This is not surprising, as such networks would naturally have an extremely large number of free parameters.

As a more concrete example of overfitting: we trained a multilayer perceptron model with a significant number of parameters on the MNIST dataset. We used ReLU activation functions throughout, and softmax output. The error (%) on training and validation data, as well as more details of the network configuration, are shown in Figure 5.1.



(i) Network configuration.

(ii) Training and validation error.

Figure 5.1: A clear case of overfitting on MNIST data.

Notice that after epoch ≈ 50, the training error continues to decrease towards zero; whereas the validation error remains constant. The model does not seem to make any improvements in predicting validation data, depsite the significant improvement over training data.

[18] Also known as *overtraining*.

Regularization refers to a set of techniques which can reduce the problem of overfitting [20]. Intuitively, we want to prevent ourselves from entering a situation whereby the our network is unable to generalise because it is *too complex*. We list two methods of regularization in this subsection:

1. The first method for preventing overfitting is dropout [12, 22], as discussed earlier. Note that the MNIST Classifier discussed in Section 4.2 uses dropout, and has shown little overfitting as compared to the model in Figure 5.1.

2. Another common technique would be to apply a penalty based on the norm of parameters (weights) used. For example, given an existing loss function $\mathcal{L}(\theta)$ we could append a slight modification to it so that our new loss function $\mathcal{L}_{\text{new}}$ is
$$\mathcal{L}_{\text{new}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2, \tag{5.1}$$
where $\lambda > 0$ is some constant known as the weight decay and $\|\cdot\|_2$ is the usual ℓ^2-norm on $\mathbb{R}^P$ where P is the dimension of θ. Note that some implementations involve replacing the ℓ^2-norm with the ℓ^1-norm, although they both clearly serve the purpose of regularization as they both penalize large weights. Disproportionately large weights are often viewed as a cause for overfitting.

5.2 Batch Normalisation

Another difficulty faced in training deep neural networks is *internal covariate shift*. This refers to changes in distribution of network activations due to training updates, and often results in slow convergence during training (sometimes referred to as the saturation problem). The standard practice in addressing this problem is the usage of ReLU activations and small learning rates.

In a 2015 paper discussing this problem [14]; a method known as *batch normalization* is presented as a technique for reducing internal covariate shift in deep networks. Batch normalization is carried out as follows: for each mini-batch, we transform each coordinate of the input[19] $\mathbf{x} = (x_1, \ldots, x_n)$ via
$$\hat{x}_i := \frac{x_i - \mathbf{E}[x_i]}{\sqrt{\mathbf{Var}(x_i)}}. \tag{5.2}$$

We may also view batch normalization as a layer in its own right. Furthermore, some batch normalization layers include a scaling and shifting parameter to be learned over training. This method has shown to improve neural network models [14], and are also applied multiple times over many layers in some deep network models [11]. Although the purpose of batch normalization is somewhat clear, we

[19] Or more generally: the previous layer, since batch normalization layers can be placed between two hiddem layers.

state some methods which allow us to take full advantage of batch normalization, as provided in [14].

1. The normalization procedure is time-consuming. Therefore, the runtime over a single epoch is expected to increase if we were to apply batch normalization. A common solution for this is to increase the learning rate.

2. For some models, removal of dropout and weight regularization tends to improve accuracy of the overall network with existing batch normalization.

3. Thorough shuffling of training data has also led to more better results with batch normalization.

5.3 Residual Networks with Stochastic Depth

Previously in Section 4.3, we introduced the CIFAR-10 image dataset and discussed some preliminary tactics of data augmentation for better learning (e.g. expanding the overall dataset by including mirrored images). In this subsection we will introduce the concept of deep residual networks [11] and stochastic depth [13] in classification of CIFAR-10 images.

For the model itself, we start by introducing the concept of a deep residual network. A deep residual network is a neural network consisting of multiple *blocks of layers* in sequence, where each block is known as a residual block (or ResBlock). A single ResBlock can be written as,

$$H_l = \texttt{ReLU}(\texttt{id}(H_{l-1}) + f_l(H_{l-1})) \tag{5.3}$$

where H_{l-1} refers to the output of the previous $(l-1)$-th layer, id refers to the identity and $f_l(H_{l-1})$ refers to the following sequence of layers and element-wise nonlinearities applied on H_{l-1}:

$$\texttt{Convolution} \rightarrow \texttt{BatchNorm} \rightarrow \texttt{ReLU} \rightarrow \texttt{Convolution} \rightarrow \texttt{BatchNorm}. \tag{5.4}$$

In other words, we split the input towards two functions (identity, and a sequence of layers). Then, we take an element-wise sum of the two function outputs and apply the ReLU nonlinearity to the result. The architecture of a single ResBlock is summarized in Figure 5.2 for clarity.

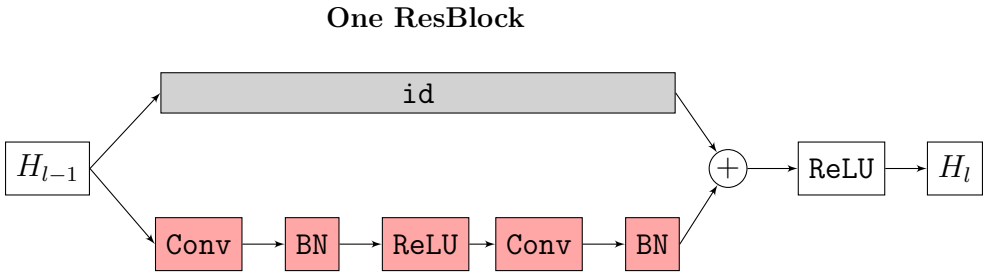


Figure 5.2: Diagram of a single residual block (ResBlock). The identity function id is shaded gray, whereas the function f_l is shaded red. `Conv`, `BN` and `ReLU` denote a convolutional layer, batch normalization layer, and element-wise ReLU respectively.

A deep residual network (sometimes referred to as a ResNet) consists of multiple ResBlocks and is parameterized by a positive integer n. We explain how n influences the depth of the network in our implementation as follows:

1. The input (a CIFAR-10 image) first passes through a mandatory convolutional layer, denoted `Conv2DLayer`, followed by a batch normalization layer, denoted `BatchNorm` with a resulting output dimension of $16 \times 32 \times 32$.

2. Pass this output through a sequence of n ResBlocks, denoted `ResBlock(1,i)` where $1 \leq \texttt{i} \leq \texttt{n}$; each with equal output dimension $16 \times 32 \times 32$.

3. Pass this output through a sequence of n ResBlocks, denoted `ResBlock(2,i)` where $1 \leq \texttt{i} \leq \texttt{n}$; each with equal output dimension $32 \times 16 \times 16$.

4. Pass this output through a sequence of n ResBlocks, denoted `ResBlock(3,i)` where $1 \leq \texttt{i} \leq \texttt{n}$; each with equal output dimension $64 \times 8 \times 8$.

5. Apply a global pooling layer, denoted `GlobalPoolLayer`. This pooling layer takes the mean across all trailing dimensions for a fixed first coordinate.

6. Output layer is a dense layer consisting of 10 neurons, with softmax applied.

For an overview of the ResNet architecture, see Figure 5.3. Each increase in *channels* (16 to 32, and 32 to 64 as stated in 3. and 4. above) requires slight modification of the beginning ResBlock, so that both functions f_l and id fit the required dimensions. It is carried out as prescribed in [11] by:

- Doubling the number of filters in the initial convolutional layer.

- Applying zero-padding to the identity portion such that the dimensions fit as required[20].

[20] See Option A in [11]. Also note that there are alternatives to zero-padding, such as applying a projection as in Option B, C.

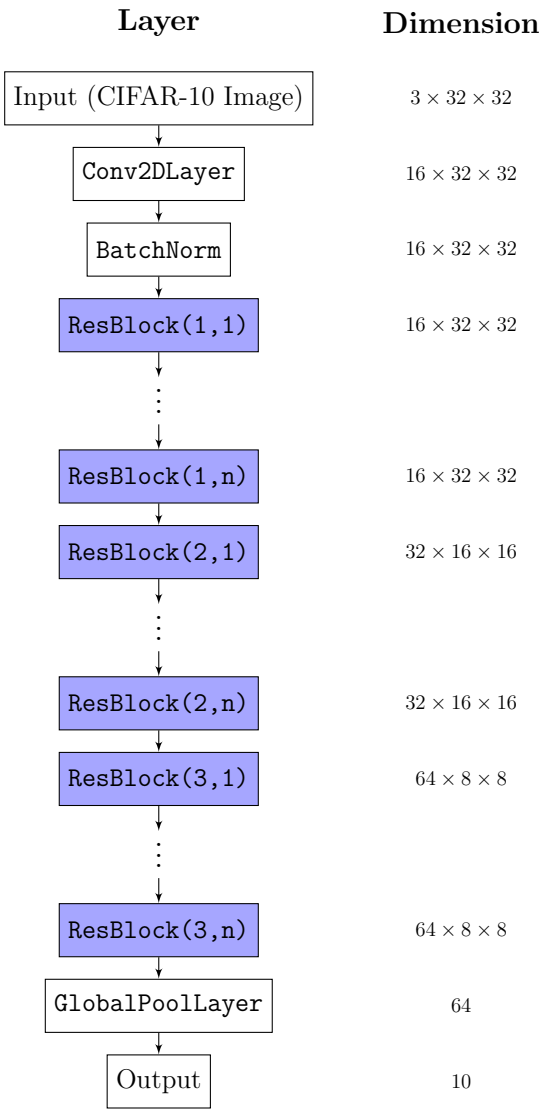


Figure 5.3: Overall structure of a deep residual network (ResNet). The portion shaded in blue represents the 'pile' of ResBlocks which make up most of the network.

The loss function for this network is taken to be the sum of the categorical cross-entropy and ℓ^2 weight regularization with weight-decay $\lambda = 10^{-4}$. Training is carried out over a total of 500 Epochs, and the learning rate is initialised at 0.1 and divided by a factor of 10 after Epochs 250 and 375 as prescribed by [13].

For the purpose of classifying CIFAR-10 images, this architecture has been known to perform well [11] with well over 90% accuracy on test data after training (this accuracy value varies with n, of course). However, a major setback from this architecture is the time taken to train the model (although still considered feasible). This fact is simply due to the inherent depth of residual networks. For example, setting $n = 18$ results in a 110-layer neural network!

A recent paper discussed a useful technique which can be applied on ResNets, known as stochastic depth. Stochastic depth allows us to not only reduce the time taken to train ResNets, but also experience a slight improvement in accuracy [13]. It makes only one modification to the current ResNet model: for each l-th ResBlock, a bernoulli random variable $z_l \sim \text{Bernoulli}(p_l)$ is assigned so that the block now becomes

$$H_l = \texttt{ReLU}(\texttt{id}(H_{l-1}) + z_l \cdot f_l(H_l)). \tag{5.5}$$

The value of p_l for a ResNet with a total of L ResBlocks is such that $p_0 = 1.0$, $p_L = 0.5$ and p_l decreases linearly from p_0 to p_L. This is also referred to as *linear decay* in the main reference [13]. Note that when a realization of z_l is 0, we obtain

$$H_l = \texttt{ReLU}(\texttt{id}(H_{l-1})) = \texttt{id}(H_{l-1}). \tag{5.6}$$

The last equality is true since, the previous input H_l is always nonnegative (due to previous ReLU activations). This effectively reduces the depth of the overall ResNet architecture since we are ignoring an entire block of layers. By sampling b_l's during training time we can shorten the total time taken to train our network, due to the shorter *expected* depth. During test time, we turn on all ResBlocks (by setting $b_l = 1$ deterministically for all l) and make use of the entire ResNet architecture.

We experimented and trained a few ResNets with stochastic depth (varying the depth parameter n) and managed to replicate similar results as in [13]. For a summary of our observed test errors for various ResNets see Figure 5.4. The 32-layer model ($n = 5$) achieved a **93.91%** test accuracy after training. The 56-layer model ($n = 9$) achieved a **94.55%** test accuracy after training. For details on our code implementation, see:

https://github.com/mollymr305/stochastic-depth

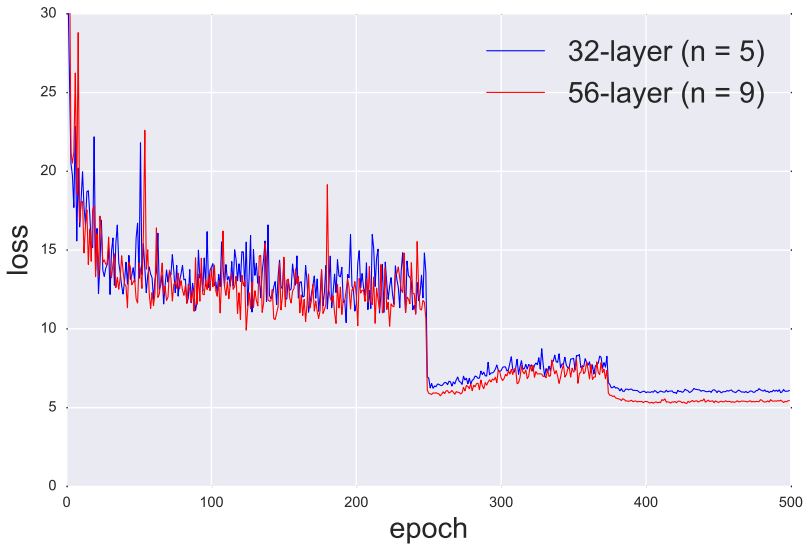


Figure 5.4: Networks with Stochastic Depth: error over test data.

6 Dimension Reduction

In this section we introduce the general idea behind dimension reduction, and how deep learning has resulted in useful dimension reduction techniques. In particular, we discuss traditional autoencoders with examples.

6.1 Motivation for Dimension Reduction

In terms of learning algorithms, training over high-dimensional data poses many challenges. This is because, in order for a model to recognize (or learn) key features of high dimensional data, it will likely require exponentially more training examples as compared to learning from data of lower dimensionality [1]. It is therefore useful to ask, if such data can be represented in a lower dimension.

An oversimplified example: suppose we were taskes to classify data based on features $\mathbf{x} = (x_1, x_2, x_3) \in \mathbb{R}^3$, but we know that $\|\mathbf{x}\| = 1$ for all data points. In other words, all data points exist on the unit sphere $\mathbb{S}^2$. It is clear that this dataset can be represented in two dimensions rather than three (noting that the unit sphere is indeed a 2-dimensional manifold). In fact, we can reduce the dimensionality of our data simply by applying stereographic projection.

For more "realistic" examples however, such as classes of images for classification, it is hard to visualize this problem and solve it by some 'known formula'. This is where autoencoders come into the picture, and provide some practical means for dimension reduction.

6.2 Traditional Autoencoders

A traditional autoencoder is an unsupervised learning model, using deep neural networks. The idea is simple; given some domain X we construct a deep neural network $f : X \to X$. This neural network f has the property that one of it's hidden layers has output in a (significantly) lower dimensional space S. This leads to two implicit functions,

$$\varphi : X \to S, \text{ and } \psi : S \to X, \tag{6.1}$$

whose composition is f. That is,

$$f \equiv \psi \circ \varphi : X \to X \tag{6.2}$$

We refer to φ as the encoder, and ψ as the decoder. The training is done as usual, and we normally implement the mean squared error as the loss function. Mathematically, we are just optimizing over the networks paramters to find:

$$\arg\max_{\varphi,\psi} \{\|\psi \circ \varphi(X) - X\|\}. \tag{6.3}$$

The neural network is essentially trying to learn the identity function, with some restriction to the dimensions of some hidden layer(s). While this might seem fairly

trivial, there are some challenges faced in this procedure[21]. What is more important is the implications of a successful autoencoder: if an autoencoder performs well, this means that we have managed to find a lower dimension representation of input data via the encoding $\varphi(X)$. Of course, this encoding (or compression) is *lossy* in the sense that we will not be able to recover 100% of the input by using the decoder ψ. A general overview of the autoencoder architecture can be found in Figure 6.1.

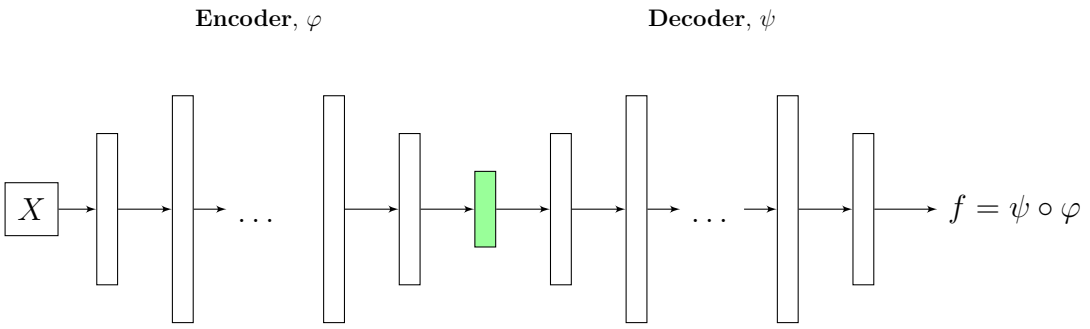


Figure 6.1: General autoencoder architecture. The shaded green layer represents the code layer, which can be seen as a compression of the input X.

6.3 Examples

The code implementation for the autoencoders in this entire subsection can be found at:

https://github.com/mollymr305/mnist-autoencoder

6.3.1 Compressing Handwritten Digit Images

The purpose of this autoencoder is to compress MNIST Handwritten Digit Images into 4×4 pixel images (or equivalently a vector in $\mathbb{R}^{16}$). We train an autoencoder consisting of one convolutional layer, and multiple dense layers. The code layer consists of 16 neurons, reshaped into a 4×4 pixel image output for visualization. All nonlinearities used are ReLU, except for the code layer and final output (image of φ and ψ) in which we used the sigmoid function (for imaging purposes). The network ended training with a mean squared error of approximately **0.01968** on the test dataset. The training and validation loss can be seen in Figure 6.2.

[21] More details of the challenges faced are provided in the last part of this section

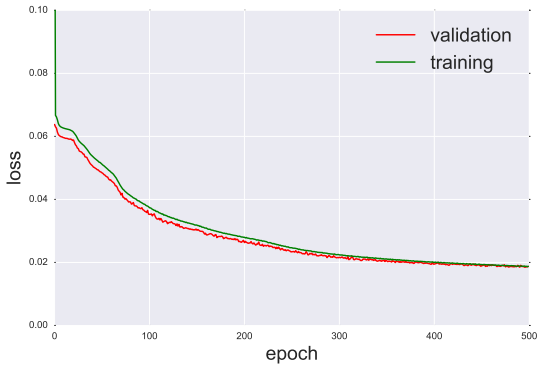


Figure 6.2: Training and validation loss.

To see what is happening in this model, refer to Figure 6.3. Handwritten digit images are compressed (encoded) into 4×4 pixel images. The original image is then recovered (decoded) from this, with a small amount of loss.

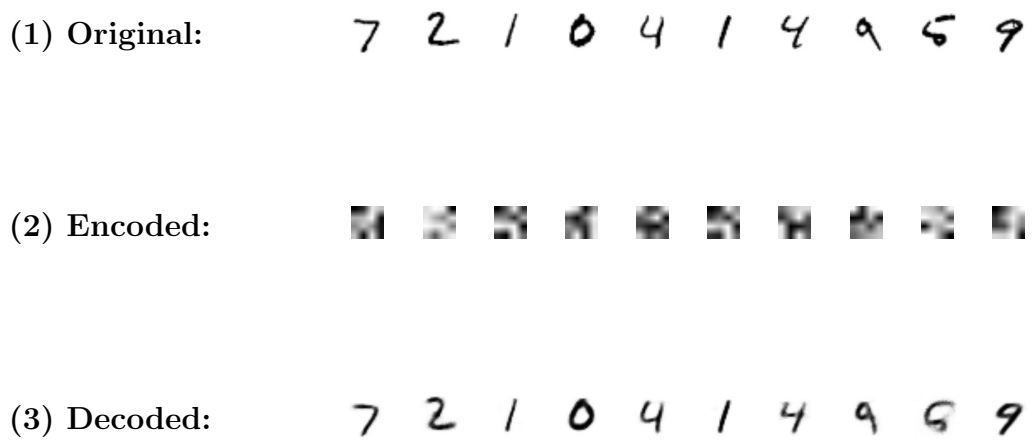


Figure 6.3: (1) Original image, (2) Encoded representation, (3) Decoded image recovered from encoded representation. Notice how both '1' Digits have similar encoding.

6.3.2 Visualization of Dimension Reduction

For the sake of visualization, we reuse the previous model but restrict the code layer to only 3 neurons. Furthermore, we set the activation function in the code layer to be the identity, for maximum range. This allows us to visualize (heuristically) the process of dimension reduction due to the autoencoder. The training and validation loss is shown in Figure 6.4.

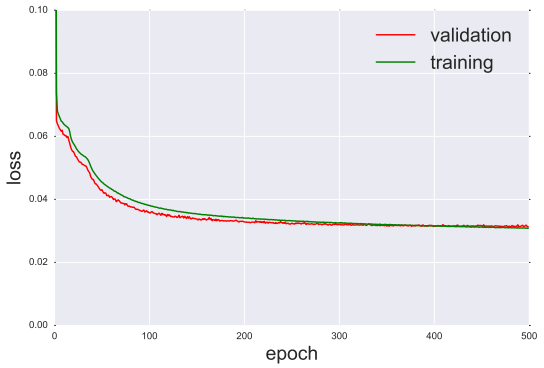


Figure 6.4: Training and validation loss.

Note that this autoencoder does not perform as well as the previous, with a final mean squared error of approximately **0.03322** on the test dataset. Furthermore, the decoded output for the same digits have a greater loss, as shown in Figure 6.5.

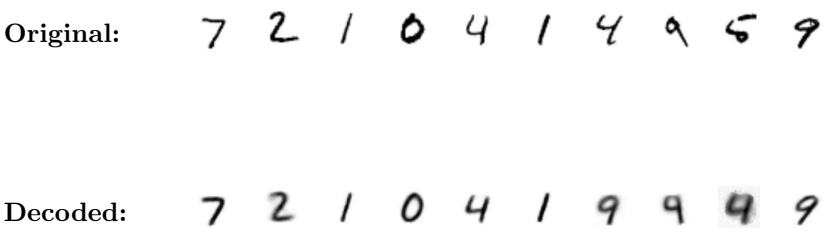


Figure 6.5: Original Image and (Decoded) Output. Notice the '5' Digit on the right is poorly reconstructed.

We can visualise the effects of dimension reduction by taking the test dataset as input and plotting the output of the code layer in $\mathbb{R}^3$. The code layer visualization before training is shown in Figure 6.6. The code layer visualization after training is shown in Figure 6.7.

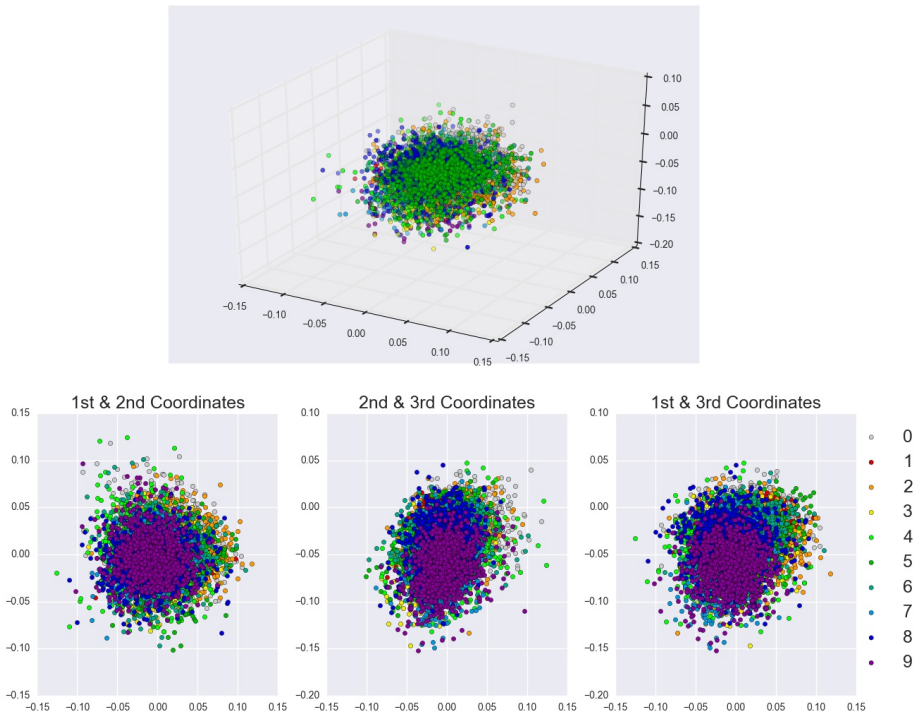


Figure 6.6: 3D and 2D visualization of code layer: before training.

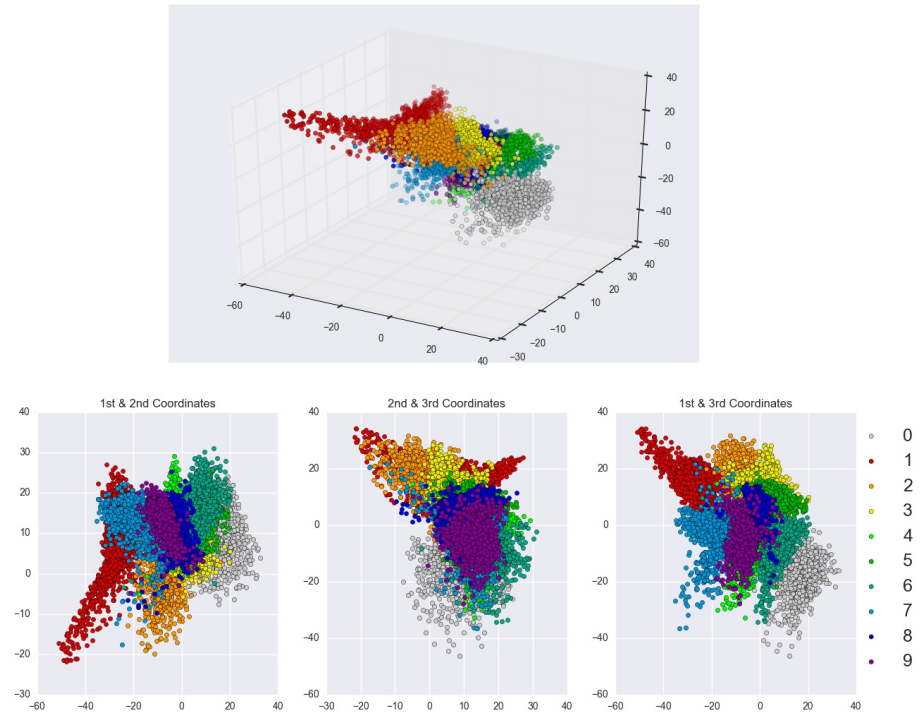


Figure 6.7: 3D and 2D visualization of code layer: after training.

6.4 Useful Remarks

We end this section with some remarks regarding traditional autoencoders in general. Firstly, there are some challenges faced in implementation. Minimizing the mean squared-error (loss function) is tricky. Oftentimes, the local minimum reached is an "average" of all training inputs. For visualization of this problem, refer to Figure 6.8.

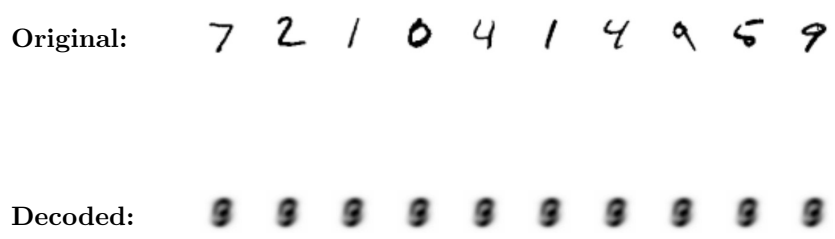


Figure 6.8: Original Image and (Decoded) Output after training. Notice how all the outputs look the same, resembling an "average" of all learned digits.

We found that this "averaging" problem can be overcome by:

- Introducing convolutional layers into the encoder section.
- Increasing the depth of the network.
- Assigning some hidden layers with large range activation functions. For example: using ReLU or identity, instead of sigmoid or tangent hyperbolic.
- Removing biases.

Finally, we consider an interesting application (or consequence) of autoencoders as a generative model. Suppose the code layer outputs a number in $\mathbb{R}$. Then we can sample a real-valued random variable, perhaps from a standard normal distribution, which we then could use as input for the decoder: generating random handwritten digits!

Of course, this idea is 'oversimplified' and does not work very well using the framework we have discussed so far. We will be discussing better approaches for this purpose, in the following section.

7 Variational Autoencoders

Variational autoencoders (VAE) [16] are significantly different from the traditional autoencoders we described previously. In this section we explain fundamental concepts involved in building VAEs, with examples and applications.

7.1 Basic Theory

One main difference is that they result in a type of generative model [4], in a sense that they deal with the inference of probability distributions and being able to randomly generate samples under such distributions. The main purpose of a VAE is to *learn* a distribution $P(\mathbf{x})$ such that P is as similar as possible with an unknown "ground-truth" distribution P_{gt}.

Oftentimes, the dimension of each datapoint $\mathbf{x}$ is high. For example, we could be dealing with a images of handwritten digits $\mathbf{x}$ and therefore learning some complex probability distribution[22] of images $P(\mathbf{x})$. This would allow us to randomly generate similar images to our observations $\mathbf{x}$. In this introduction, we develop some theory and motivation before discussing deep learning architectures for the VAE.

We begin by introducing the *Kullback-Leibler divergence*. This function can be seen as a measure of how far (or different) two probability distributions are.

Definition 7.1.1. Let Q and P be probability density (or mass) functions. The Kullback-Leibler (KL) divergence of Q from P is defined to be

$$\mathrm{KL}\left[P \,\|\, Q\right] = \mathop{\mathbf{E}}_{\mathbf{x}\sim P} \log \frac{P(\mathbf{x})}{Q(\mathbf{x})}. \tag{7.1}$$

Remark 7.1.2. When P, Q are both discrete, Equation 7.1 reduces to

$$\mathrm{KL}\left[P \,\|\, Q\right] = \sum_k P_k \log \frac{P(k)}{Q(k)}. \tag{7.2}$$

On the other hand, when P, Q are both continuous with densities p, q respectively we have

$$\mathrm{KL}\left[P \,\|\, Q\right] = \int p(x) \log \frac{p(x)}{q(x)} \mathrm{d}x. \tag{7.3}$$

Suppose we are given some dataset $\mathcal{D} = \{X_1, \ldots, X_N\}$ consisting of high-dimensional datapoints. Define z to be a vector of *latent* variables easily sampled by some density $P(z)$. Let $f(z, \theta)$ denote some function parametrized by θ, which we may assume to be some neural network for example. We wish to be able to generate samples X^* *resembling* those from $\mathcal{D}$ (but not necessarily identical) via sampling $z \sim P(z)$ and then computing $f(z, \theta) = X^*$. One possibility is to sample

[22] We would only expect the model to be able to compute $P(\mathbf{x})$ numerically, of course.

a vector $z \sim \mathcal{N}(0, I)$ such that $f(z) := f(z, \theta)$ generates similar data as in $\mathcal{D}$. This is done by optimizing θ, under the framework that

$$P(X) = \int P(X|z, \theta)P(z)\,\mathrm{d}z. \tag{7.4}$$

In order to set up an objective for our optimization, we consider the KL divergence of $P(z|X)$ from $Q(z)$. Here, $Q(z)$ refers to an arbitrary density from which we could sample z, and $P(z|X)$ can be seen as possible values of z which could have resulted in X.

$$\mathrm{KL}\left[Q(z) \,\|\, P(z|X)\right] = \mathbf{E}_{z\sim Q}\left[\log \frac{Q(z)}{P(z|X)}\right] = \mathbf{E}_{z\sim Q}\left[\log Q(z) - \log P(z|X)\right] \tag{7.5}$$

By applying Bayes rule and noting that $P(X)$ is independent of z, we get

$$\mathrm{KL}\left[Q(z) \,\|\, P(z|X)\right] = \mathbf{E}_{z\sim Q}\left[\log Q(z) - \log P(X|z) - \log P(z)\right] + \log P(X). \tag{7.6}$$

Rearrangement the terms in the last equation gives

$$\begin{aligned}\log P(X) - \mathrm{KL}\left[Q(z) \,\|\, P(z|X)\right] &= \mathbf{E}_{z\sim Q}[\log P(X|z)] - \mathbf{E}_{z\sim Q}[\log Q(z) - \log P(z)] \\ &= \mathbf{E}_{z\sim Q}[\log P(X|z)] - \mathrm{KL}\left[Q(z) \,\|\, P(z)\right].\end{aligned} \tag{7.7}$$

We write $Q(z) = Q(z|X)$ since Q will be constructed using X, so our equation becomes

$$\log P(X) - \mathrm{KL}\left[Q(z|X) \,\|\, P(z|X)\right] = \mathbf{E}_{z\sim Q}[\log P(X|z)] - \mathrm{KL}\left[Q(z|X) \,\|\, P(z)\right]. \tag{7.8}$$

If we take the left-hand side as the objective for optimization, this intuitively means that we are trying to maximize the loglikelihood $\log P(X)$ and minimize the divergence of $P(z|X)$ from $Q(z|X)$. Note that the right-hand expression is much easier to compute and optimize via SGD, as we will see later.

At this point we set $Q(z|X)$ to be $\mathcal{N}(z|\mu(X), \Sigma(X))$, since this is the usual choice in practice [4]. In other words, we define a deterministic mean and covariance function μ and Σ (with trainable parameters) on observations of X. This is so that we can sample a normal vector z defined from these functions. Furthermore, we set $P(z) = \mathcal{N}(z|0, I)$ so that the KL divergence on the right-hand expression is simply a KL divergence between two multivariate Gaussians. Note that this admits a closed form expression [4]; in general for two multivariate Gaussians $\mathcal{N}(\mu_0, \Sigma_0)$ and $\mathcal{N}(\mu_1, \Sigma_1)$

$$\begin{aligned}&\mathrm{KL}\left[\mathcal{N}(\mu_0, \Sigma_0) \,\|\, \mathcal{N}(\mu_1, \Sigma_1)\right] \\ =&\frac{1}{2}\left(\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) + (\mu_1 - \mu_0)^T\Sigma_1^{-1}(\mu_1 - \mu_0) - k + \log(\frac{\det \Sigma_1}{\det \Sigma_0})\right).\end{aligned} \tag{7.9}$$

Here k refers to the dimension of the multivariate Gaussian distributions. For our case of $Q(z|X) = \mathcal{N}(z|\mu(X), \Sigma(X))$ and $P(z) = \mathcal{N}(z|0, I)$, we obtain

$$\mathrm{KL}\left[\mathcal{N}(\mu(X), \Sigma(X)) \,\|\, \mathcal{N}(0, I)\right] = \frac{1}{2}\left(\mathrm{tr}(\Sigma(X)) + \mu(X)^T\mu(X) - k - \log\det\Sigma(X)\right). \tag{7.10}$$

As with SGD, we will be interested in optimising (with some slight abuse of notation; since we are writing $\mathcal{D}$ as the sample distribution of X as well as the dataset)

$$\begin{aligned} &\mathbf{E}_{X\sim\mathcal{D}}\left[\log P(X) - \mathrm{KL}\left[Q(z|X) \,\|\, P(z|X)\right]\right] \\ =&\mathbf{E}_{X\sim\mathcal{D}}\left[\mathbf{E}_{z\sim Q}[\log P(X|z)] - \mathrm{KL}\left[Q(z|X) \,\|\, P(z)\right]\right]. \end{aligned} \tag{7.11}$$

Consider the gradient of the following expression

$$\log P(X|z) - \mathrm{KL}\left[Q(z|X) \,\|\, P(z)\right]. \tag{7.12}$$

By averaging this gradient over many samples of $X \sim \mathcal{D}$ and $z \sim Q$, the result will converge to the gradient we are interested in (Equation 7.11). It is worthy to note that the gradient operator can be moved inside the expectation for Equation 7.11.

The last challenge with optimisation is being able to properly execute back-propagation. This is difficult because we eventually have to go through a layer with outputs $z \sim Q$, which has no meaningful gradient. This is overcome by applying a useful "reparameterization trick" [16] where we sample $z \in Q$ via $z = \mu(X) + \Sigma^{1/2}(X) * \varepsilon$, with $\varepsilon \sim \mathcal{N}(0, I)$. The final expression in which we will be optimising and taking the gradient of is therefore

$$\mathbf{E}_{X\sim\mathcal{D}}\left[\mathbf{E}_{\varepsilon\sim\mathcal{N}(0,I)}[\log P(X|z)] - \mathrm{KL}\left[Q(z|X) \,\|\, P(z)\right]\right], \tag{7.13}$$

where $z = \mu(X) + \Sigma^{1/2}(X) * \varepsilon$.

7.2 Typical Architecture

In this section, we fix $Q(z|X) = \mathcal{N}(z|\mu(X), \Sigma(X))$ and $P(z) = \mathcal{N}(0, I)$ as discussed above and proceed to describe some architectures for such VAEs presented in literature [4, 16]. The idea is simple:

1. The encoder section consists of a multilayer perceptron which takes an input X and outputs two vectors $\mu(X)$ and $\Sigma(X)$.

2. We sample $\varepsilon_1, \ldots, \varepsilon_n \sim \mathcal{N}(0, I)$ and apply the "reparameterization trick" discussed earlier. In other words we obtain, for $1 \leq i \leq n$,

$$z = \mu(X) + \Sigma^{1/2}(X) * \varepsilon_i. \tag{7.14}$$

3. The decoder section consists of another multilayer perceptron which takes in z and outputs a (learned) reconstructed imaged of X, denoted by $f(z)$.

Readers can refer to Figure 7.1 for a summary of this procedure. Note that the overall loss function for this VAE consists the (expected) sum of two loss functions, namely:

1. The KL divergence, $\mathrm{KL}\left[Q(z|X) \,\|\, P(z)\right]$, extracted from the output of the encoder section.

2. The loglikelihood expression, $\log P(X|z)$, extracted from the output of the decoder section.

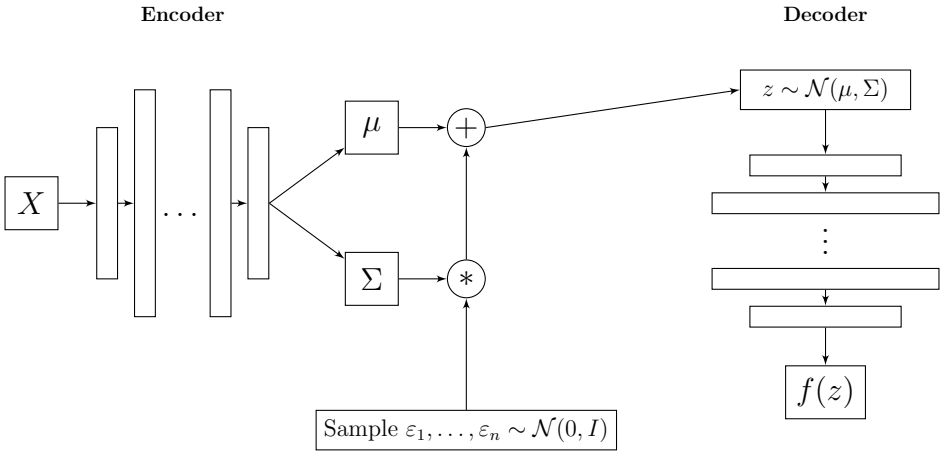


Figure 7.1: Overall architecture using the reparameterization trick.

7.3 Example: Generating Random Handwritten Digits

The results in this section make use of a variational autoencoders code implementation from our reference [19].

For sake of providing a concrete example of a VAE, we consider inputs X taken from the MNIST Handwritten Digits. The architecture is similar to Figure 7.1. We simply set the latent variable space to be two-dimensional. In other words, the encoder outputs $\mu(X)$ and $\Sigma(X)$ are two-dimensional vectors[23], and we sample *two* i.i.d. random variables with every feedforward (for the reparameterization trick):

$$\varepsilon_1, \varepsilon_2 \sim \mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right). \tag{7.15}$$

After training, we take evenly distributed points $z \in \mathbb{R}^2$ and observe the resultant output $(f(z))$ after feeding z into the decoder. Refer to Figure 7.2 for this result.

[23]For $\Sigma(X)$, this is because we take it to be a 2×2 diagonal matrix. Also, there might be some concerns regarding the actual output as it the components of $\Sigma(X)$ are supposed to be strictly postive. In order to overcome this in practice, we can take the exponential of the (real-valued) outputs.

We can therefore see, how the decoder can be used as a generative model for handwritten digit images: simply sample a two-dimensional normal random vector z and use the decoder as a transformation to obtain a random handwritten digit!

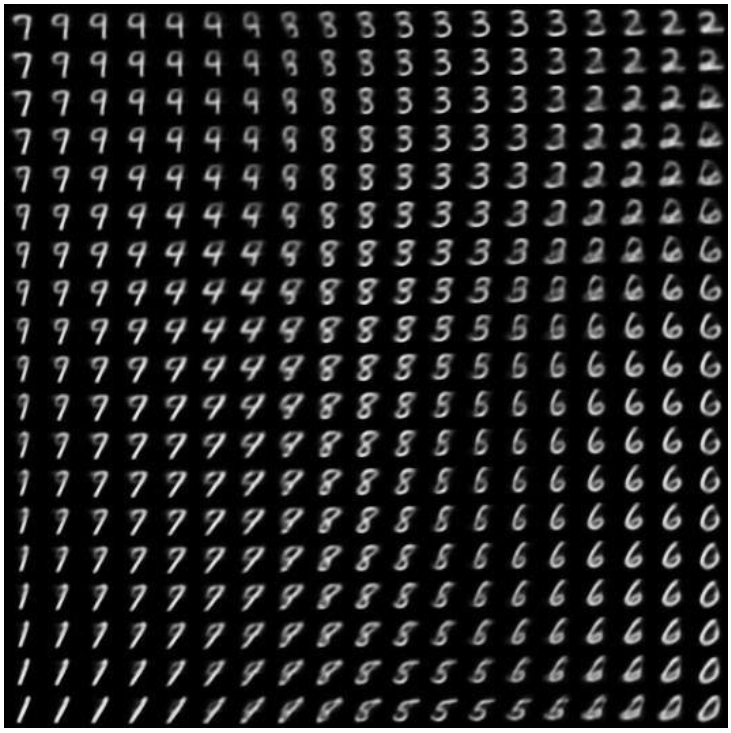


Figure 7.2: Visualization of the 2-dimensional latent variable space. This result is generated by the code implementation from [19].

8 Bayesian Inference in Neural Networks

This section is devoted to recent studies concerning representation of *uncertainty* in neural networks. For example, rather that give a predictive output for classification; it might be more attractive for our network report a (theoretically-sound) categorical distribution representing the probabilities of each class being the 'correct output'. Similarly, for regression we would like our network to output a *distribution* rather than a prediction. This is where Bayesian inference comes in the picture. We begin by introducing important prerequisites for such analysis.

8.1 Bayesian Regression and Gaussian Processes

Given a training dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$ we attempt to find a relationship which alows us to make meaningful predictive distributions on some test dataset. In standard linear regression, we can take θ to be a vector of parameters (to be learned) and iid $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$ for $1 \leq i \leq m$, and write this relationship as

$$\begin{aligned} & y_i = \theta^T \mathbf{x}_i + \varepsilon_i \\ \iff & y_i - \theta^T \mathbf{x}_i = \varepsilon_i \\ \iff & p(y_i|\mathbf{x}_i, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(y_i - \theta^T \mathbf{x}_i)^2}{2\sigma^2}\right\}. \end{aligned} \tag{8.1}$$

In the Bayesian approach [3], we first assume some *prior* density on θ. For example $\theta \sim \mathcal{N}(0, \sigma^2 I)$. Then by Bayes' Theorem and assumption of independence we obtain

$$\begin{aligned} p(\theta|\mathcal{D}) &= \frac{p(\theta)p(\mathcal{D}|\theta)}{\int_{\theta_0} p(\theta_0)p(\mathcal{D}|\theta_0)\,\mathrm{d}\theta_0} \\ &= \frac{p(\theta)\prod_{i=1}^m p(y_i|\mathbf{x}_i, \theta)}{\int_{\theta_0} p(\theta_0)\prod_{i=1}^m p(y_i|\mathbf{x}_i, \theta_0)\,\mathrm{d}\theta_0}. \end{aligned} \tag{8.2}$$

Now, given a test datapoint $(y_*, \mathbf{x}_*)$, the predictive distribution given by

$$p(y_*|\mathbf{x}_*, \mathcal{D}) = \int_\theta p(y_*|x_*, \theta)p(\theta|\mathcal{D})\,\mathrm{d}\theta. \tag{8.3}$$

Equations 8.2 and 8.3 are often not easily computed. However, if θ follows a multivariate Gaussian then $p(\theta|\mathcal{D})$ and $p(y_*|\mathbf{x}_*, \mathcal{D})$ is tractable (can be computed via formula).

We now discuss some basic theory behind Gaussian Processes (GP) [3], and how an application of GPs can be seen as a form of Bayesian inference; similar to what we have just gone through.

Definition 8.1.1. A Gaussian Process (GP) on a set S is a collection of random variables $\{Z_t : t \in S\}$ such that for any given finite number of points $t_1, \dots, t_n \in S$ the vector $(Z_{t_1}, \dots, Z_{t_n})^T$ follows a multivariate Gaussian distribution.

In order to describe a GP, we often associate it with a mean function $\mu : S \to \mathbb{R}$ and covariance function[24] $k : S \times S \to \mathbb{R}$. In other words, for the finite points $t_1, \cdots, t_n \in S$, we have

$$\begin{bmatrix} Z_{t_1} \\ \vdots \\ Z_{t_n} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(t_1) \\ \vdots \\ \mu(t_n) \end{bmatrix}, \begin{bmatrix} k(t_1,t_1) & \cdots & k(t_1,t_n) \\ \vdots & \ddots & \vdots \\ k(t_n,t_1) & \cdots & k(t_n,t_n) \end{bmatrix} \right). \tag{8.4}$$

We adopt the notation $(Z_t)_{t\in S} \sim \mathcal{GP}(\mu(\cdot), k(\cdot,\cdot))$ for such a Gaussian process. Suppose we are given the same test data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ as previously discussed. Then, the GP regression model is written as

$$y_i = f(x_i) + \varepsilon_i, \tag{8.5}$$

where we have iid $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$ and $f \sim \mathcal{GP}(0, k(\cdot,\cdot))$ is our *prior* distribution on the space of functions. For simplicity we write $\vec{y} = \vec{f} + \vec{\varepsilon}$, where

$$\vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \vec{f} = \begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_m) \end{bmatrix}, \vec{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_m \end{bmatrix}. \tag{8.6}$$

It is clear that $\vec{f}$ follows some multivariate Gaussian, whose covariance matrix which we will denote by $K(X,X)$ is determined by its prescribed covariance function $k(\cdot,\cdot)$. Similarly, $\vec{y}$ follows a multivariate Gaussian with covariance matrix $\sigma^2 I + K(X,X)$.

Given some test dataset $\mathcal{T} = \{(\mathbf{x}_1^*, y_1^*), \ldots, (\mathbf{x}_t^*, y_t^*)\}$ we can write $\vec{y}^* = \vec{f}^* + \vec{\varepsilon}^*$ similarly, where

$$\vec{y}^* = \begin{bmatrix} y_1^* \\ \vdots \\ y_t^* \end{bmatrix}, \vec{f}^* = \begin{bmatrix} f^*(\mathbf{x}_1) \\ \vdots \\ f^*(\mathbf{x}_t) \end{bmatrix}, \vec{\varepsilon}^* = \begin{bmatrix} \varepsilon_1^* \\ \vdots \\ \varepsilon_t^* \end{bmatrix}. \tag{8.7}$$

We write the covariance matrix for $\vec{f}^*$ as $K(X^*, X^*)$. Applying the GP property gives:

$$\begin{bmatrix} \vec{f} \\ \vec{f}^* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X) & K(X,X^*) \\ K(X^*,X) & K(X^*,X^*) \end{bmatrix}\right). \tag{8.8}$$

This means that

$$\begin{bmatrix} \vec{y} \\ \vec{y}^* \end{bmatrix} = \begin{bmatrix} \vec{f} \\ \vec{f}^* \end{bmatrix} + \begin{bmatrix} \vec{\varepsilon} \\ \vec{\varepsilon}^* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X) + \sigma^2 I & K(X,X^*) \\ K(X^*,X) & K(X^*,X^*) + \sigma^2 I \end{bmatrix}\right). \tag{8.9}$$

[24] Also known as a symmetric positive semidefinite kernel, or simply kernel. We assume the matrix $K = (k(t_i,t_j))_{1\le i,j\le n}$ is symmetric and positive semidefinite.

Finally, by using a well-known formula [3] for the marginal of a multivariate Gaussian we can obtain the condition distribution $p(\vec{y}^*|\vec{y}, X, X^*)$:

$$\vec{y}^*|\vec{y}, X, X^* \sim \mathcal{N}(\mu^*, \Sigma^*), \tag{8.10}$$

where

$$\begin{aligned} \mu^* &= K(X^*, X)[K(X, X) + \sigma^2 I]^{-1} \cdot \vec{y} \\ \Sigma^* &= K(X^*, X^*) + \sigma^2 I - K(X^*, X)[K(X, X) + \sigma^2 I]^{-1} K(X, X^*). \end{aligned} \tag{8.11}$$

The similarities between Bayesian and GP regression are clear. Before discussing how neural networks relates to these frameworks, we make some remarks.

1. Although the marginal of a multivariate Gaussian is tractable – it is computationally expensive. This is because it involves inverting a *large* matrix, which generally holds a time complexity of $\mathcal{O}(N^3)$ [8]. In practice it is common to use variational approximations to GP.

2. A similar concept (using GPs) can be applied for classification tasks [6]. In particular we can view f as a latent function, in which we apply the sigmoid or softmax function (or more generally, some suitable transformation $\pi : \mathbb{R} \to [0, 1]$) to obtain the respective "class probabilities".

8.2 Dropout as Bayesian Approximation

In 2015, it was shown that deep neural networks with dropout applied before every weight layer are mathematically equivalent to approximate variational inference in deep Gaussian processes [8, 9]. In this subsection we briefly explain some of the theory.

In our explanation, we assume for the sake of simplicity; a single-layer neural network with a one-dimensional output layer tasked for regression. For a more thorough extension of this idea to general neural networks see [8, 9]. Note that this assumption allows us to express the network as a simple function

$$f(\mathbf{x}, \mathbf{w}, b) = \sigma(\mathbf{w} \cdot \mathbf{x} + b). \tag{8.12}$$

In the equation above, σ denotes the nonlinearity applied on the layer and $\mathbf{w}, b$ denotes the weight vector and bias respectively. The GP model of interest uses the following covariance function[25]:

$$\mathbf{K}(\mathbf{x}, \mathbf{y}) = \int p(\mathbf{w})p(b)\sigma(\mathbf{w}^T\mathbf{x} + b)\sigma(\mathbf{w}^T\mathbf{y} + b)\mathrm{d}\mathbf{w}\mathrm{d}b, \tag{8.13}$$

[25] We take for granted that $\mathbf{K}$ is indeed a valid covariance function, although this can be shown mathematically.

Here, $p(\mathbf{w})$ is a standard multivariate normal distribution on the vector of weights and $p(b)$ is some arbitrary distribution on the bias. By Monte Carlo integration with K terms as approximation for $\mathbf{K}$ we "switch" our covariance function to

$$\hat{\mathbf{K}}(\mathbf{x}, \mathbf{y}) = \frac{1}{K} \sum_{k=1}^{K} \sigma(\mathbf{w}_k^T \mathbf{x} + b_k)\sigma(\mathbf{w}_k^T \mathbf{y} + b_k). \tag{8.14}$$

Write $\mathbf{W} = (\mathbf{w}_k)_{k=1}^K$, $\mathbf{b} = (b_k)_{k=1}^K$. Note that we wish to obtain the predictive distribution given by

$$p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{X}, \mathbf{Y}) = \int p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{W}) p(\mathbf{W}|\mathbf{X}, \mathbf{Y}) \, \mathrm{d}\mathbf{W}. \tag{8.15}$$

Here, we used the notation $\mathbf{X}, \mathbf{Y}$ for training inputs and outputs respectively, as well as $\mathbf{x}^*, \mathbf{y}^*$ for a single test input and output. Since the posterior $p(\mathbf{W}|\mathbf{X}, \mathbf{Y})$ is difficult to compute in general, we use an approximating variational distribution $q(\mathbf{W})$ instead [9]. The distribution $q(\mathbf{W})$ is given by

$$\mathbf{W} = \mathbf{M} \cdot \mathrm{diag}([z_j]_{j=1}^K), \tag{8.16}$$

Where each $z_j \sim \mathrm{Bernoulli}(p_0)$ and $\mathbf{M}$ is a variational parameter matrix. Note that $z_j = 0$ corresponds to a j-th neuron being "dropped out". Optimization under this variational framework involves minimizing the following loss function [9]

$$-\int q(\mathbf{W}) \log p(\mathbf{Y}|\mathbf{X}, \mathbf{W}) \, \mathrm{d}\mathbf{W} + \mathrm{KL}\left[q(\mathbf{W}) \,\|\, p(\mathbf{W})\right]. \tag{8.17}$$

This loss can be rewritten as

$$-\left(\sum_{i=1}^{N} \int q(\mathbf{W}) \log p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{W}) \, \mathrm{d}\mathbf{W}\right) + \mathrm{KL}\left[q(\mathbf{W}) \,\|\, p(\mathbf{W})\right]. \tag{8.18}$$

And finally, it can be further shown [8,9] that we can approximately minimize the loss function above via minimizing

$$-\frac{1}{N} \sum_{n=1}^{N} \frac{\log p(\mathbf{y}_n|\mathbf{x}_n, \mathbf{W}_n)}{\tau} + \frac{p_0 \ell^2}{2\tau N} \|\mathbf{M}\|_2^2 + \frac{\ell^2}{2\tau N} \|b\|_2^2 \tag{8.19}$$

where $\mathbf{W}_n$ are samples under the variational distribution, and τ, ℓ are some hyperparameters[26].

At this point, we show the similarity between this expression and the loss function of a similar network with dropout. Suppose we add a dropout layer

[26] In particular τ is the precision parameter of the Gaussian distribution $p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{W})$, and ℓ is known as the *length-scale* of the prior of the weights [8].

between our input and output layers, and use a loss consisting of the mean squared-error and weight regularisation with decay λ. Then the loss function is simply,

$$\mathcal{L}_{\text{dropout}} = \frac{1}{N}\sum_{i=1}^{N} |\mathbf{y}_i - \hat{\mathbf{y}}_i|^2 + \lambda(\|\mathbf{W}\|_2^2 + \|b\|_2^2). \tag{8.20}$$

And it is clear that Equation 8.19 and Equation 8.20 are equivalent, with appropriate setting of τ and ℓ. Each sample $\mathbf{W}_n$ is simply the realization of Bernoulli random variables in dropout. This idea also extends to classification tasks (e.g. using categorical crossentropy as loss instead) and does not impose much restriction on the nature of our neural network. For example, the general result [8, 9] also applies to networks with convolutions and arbitrary depth.

8.3 Applications

The main application is representing model uncertainty, using a method referred to as *MC dropout* [8, 9]. Given an test input $\mathbf{x}^*$ we apply a total of T stochastic feedforwards[27], obtaining a sample distribution of 'predictions' denoted $\hat{\mathbf{y}}_1^*, \ldots, \hat{\mathbf{y}}_T^*$.

- For regression tasks, we measure uncertainty by taking the sample mean and variance of the T outputs as an estimate of true mean and variance of $\mathbf{y}^*$ under the variational approximation $q(\mathbf{y}^*|\mathbf{x}^*)$.
- For the case of categorical outputs, the sample $\hat{\mathbf{y}}_1^*, \ldots, \hat{\mathbf{y}}_T^*$ can be used to estimate predictive distribution (probability mass function) and infer uncertainty levels.

One application example is the following: we trained a "MC Dropout" modified version of the MNIST classifier introduced in Section 4.2, where we simple applied dropout layer before each hidden layer.

We set the Bernoulli parameter to $p = 0.5$ throughout, and trained the model over 3000 epochs. For details on training and validation errors, see Figure 8.2. For testing, we ran a total of $T = 100$ stochastic feedforwards per test input to obtain the sample distributions. We present some interesting outputs from the resulting network based on the test data. See Figure 8.1 for a visualization of these test data cases. The full code implementation of this procedure can be found at:

https://github.com/mollymr305/mnist-mc-dropout

[27] Feedforward with dropout left "on".

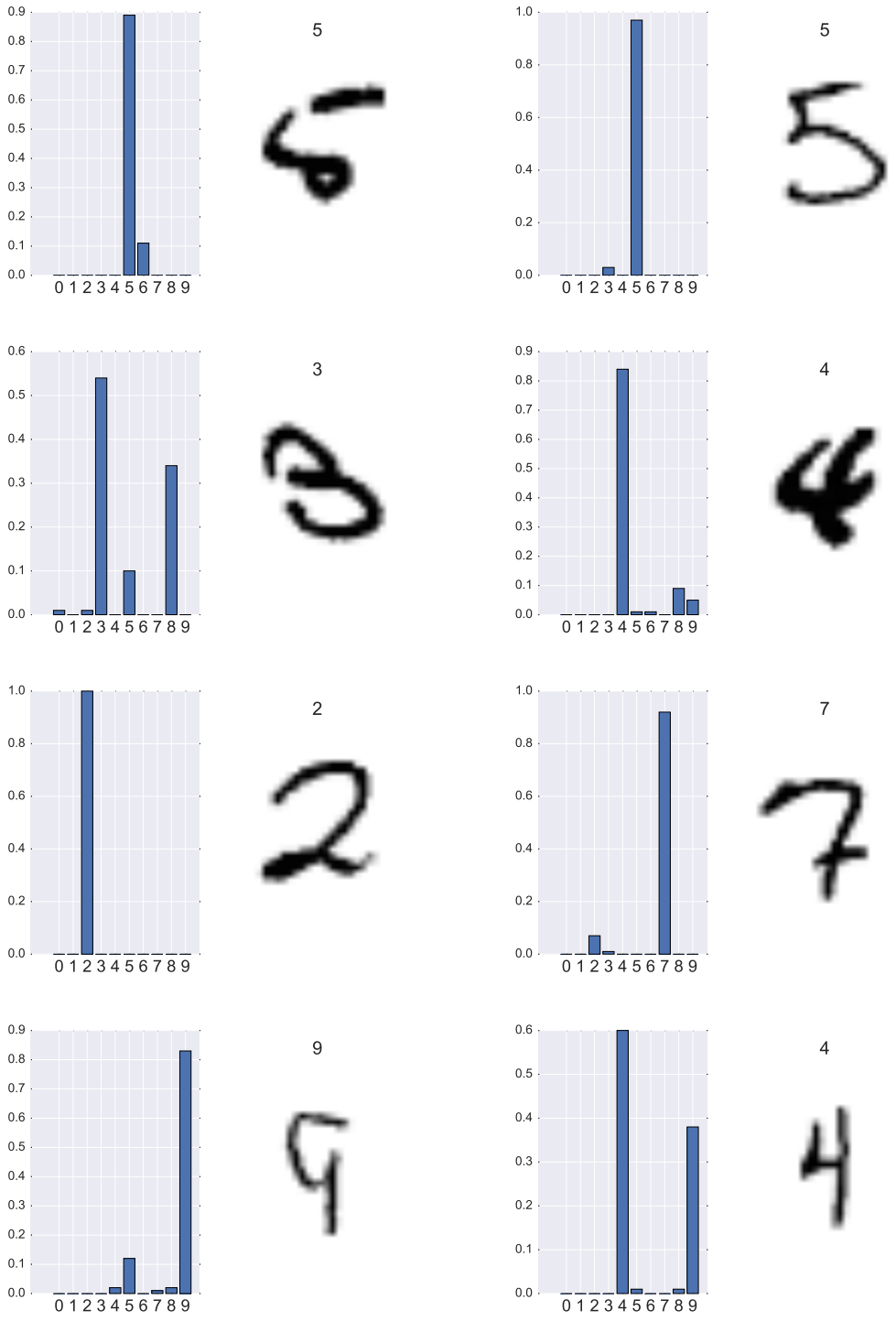


Figure 8.1: Output results (sample distribution and input image with label) for selected MNIST test data. Notice how the model displays larger levels of uncertainty for 'poorly written' digits. On the other hand, 'well-written' digits such as the digit '2' above are classified with high-certainty.

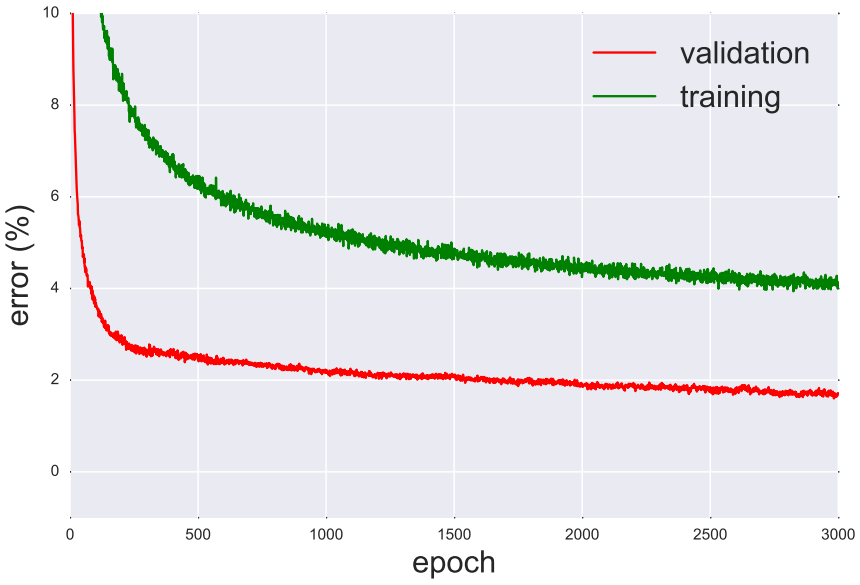


Figure 8.2: Training and validation errors for the MNIST MC Dropout model.

References

[1] Buduma, N. The curse of dimensionality and the autoencoder. http://nikhilbuduma.com/2015/03/10/the-curse-of-dimensionality/.

[2] Deng, L. The mnist database of handwritten digit images for machine learning. In *IEEE Signal Processing Magazine* (Nov 2012).

[3] Do, C. B. Gaussian processes. Stanford University CS229 Lecture Notes, Nov 2008.

[4] Doersch, C. Tutorial on variational autoencoders, 2016.

[5] Dumoulin, V., and Visin, F. A guide to convolution arithmetic for deep learning, 2016.

[6] Ebden, M. Gaussian processes: A quick introduction, 2015.

[7] Fisher, R. A. The use of multiple measurements in taxonomic problems. *Annual Eugenics, 7, Part II* (1936).

[8] Gal, Y., and Ghahramani, Z. Dropout as a bayesian approximation: Appendix, 2015.

[9] Gal, Y., and Ghahramani, Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2015.

[10] Goodfellow, I., Bengio, Y., and Courville, A. Deep learning. Book in preparation for MIT Press, 2016.

[11] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

[12] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[13] Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Deep networks with stochastic depth, 2016.

[14] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[15] Kim, J.-H., Karray, F., Jo, J., Sincak, P., and Myung, H. *Robot Intelligence Technology and Applications 4: Results from the 4th International Conference on Robot Intelligence Technology and Applications*. Springer, 2016.

[16] Kingma, D. P., and Welling, M. Auto-encoding variational bayes, 2013.

[17] Krizhevsky, A., and Hinton, G. Learning multiple layers of features from tiny images.

[18] LeCun, Y., and Cortes, C. MNIST handwritten digit database.

[19] Lee, T. Implementation of variational autoencoder (aevb) algorithm. https://github.com/Lasagne/Recipes/blob/master/examples/variational_autoencoder/variational_autoencoder.py.

[20] Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

[21] Smola, A., and Vishwanathan, S. *Introduction to Machine Learning*. Cambridge Univeristy Press, 2010.

[22] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res. 15*, 1 (Jan. 2014), 1929–1958.

[23] Øygard, A. M., and Schlüter, J. Lasagne implementation of cifar-10 examples from "deep residual learning for image recognition". https://github.com/Lasagne/Recipes/blob/master/papers/deep_residual_learning/Deep_Residual_Learning_CIFAR-10.py.