Machine Learning - Michaelmas Term 2016
Lectures 14-16 : Neural Networks

Lecturers: Christoph Haase & Varun Kanade

1 Neural Networks

We've already encountered the perceptron in the very first lecture. A perceptron takes as inputs $x_1, \ldots, x_D$, and then outputs $\text{sign}(b + w_1 x_1 + \cdots + w_D x_D)$, where $w_1, \ldots, w_D$ are the weight parameters of the perceptron and b is the bias term. We can consider more general models of this kind, which are referred to as *artificial neurons*. Let $f : \mathbb{R} \to \mathbb{R}$ be an *activation* function, $x_1, \ldots, x_D$ the inputs, and $w_1, \ldots, w_D$ weights and b the bias. Then an artificial neuron, also known as a unit, with activation function f outputs the following:

$$f(b + w_1 x_1 + \cdots + w_D x_D) \tag{1}$$

We have seen (Problem 4, Sheet 1) that artificial neurons can be used to build "boolean" gates. Thus, in principle, composing artificial neurons is a powerful idea—essentially an artificial neural network can compute any function that a computer can! Of course, one may then wonder why not try to use models that use boolean circuits directly? A neural network composed of units with continuous activation functions is differentiable end-to-end, *i.e.,* a suitably chosen loss function for any fixed input–output pair is a differentiable function of the weight and bias parameters. Thus, in principle, it may be easier to train such networks that boolean ones.[1] Let us look at a few examples of neural networks.

1.1 Logistic Regression

We can view the logistic regression model as a neural network with a single artificial neuron or unit. The activation function used is the sigmoid function, $\sigma(z) = 1/(1 + e^{-z})$. This is shown schematically in Figure 1. The unit is shown as explicitly composed of a "linear function", $b + w_1 x_1 + \cdots + w_D x_D$ followed by a *non-linear* activation function, σ. The output of the model is interpreted as the probability that the observed label is 1. As we've already seen, the logistic regression model for classification results in a linear separating surface. Thus, such a simple neural network cannot represent functions of any greater complexity than other models we've seen so far. In order to exploit the full power of neural networks, we need to have larger and *deeper* "circuits".

1.2 Multilayer Perceptron (MLP)

Neural networks composed of more than one "layer" are called multilayer perceptrons (MLPs). Despite the name, the units in the network do not have to be perceptrons, but can in fact be any kind of artificial neurons. Let us begin by considering a toy example to understand how multilayer perceptrons behave. Let us consider a classification problem, where the data is as shown in Figure 2(a). Clearly, no linear separater can separte the blue points from the red. If

[1] This is just an intuitive explanation and not a formal statement. As of today, there is little theoretical justification for this of claim. However, in practice it is observed that neural networks composed of units with continuous activation functions often perform very well on a range of tasks. Such performance has not been seen with boolean circuits; however, it's fair to say that not as much effort has been spent by the research community trying to train boolean circuits directly. Part of the difficulty is that since they are not differentiable, one has to rely on other heuristic approaches or "genetic algorithms", rather than use gradient-based methods.

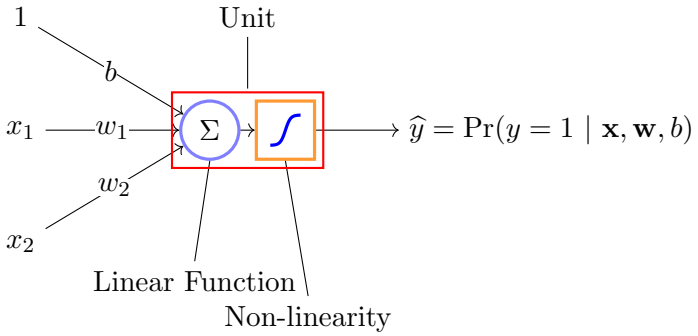


Figure 1: Logistic regression model viewed as a neural network.

we try to train a logistic regression model directly, it fails quite badly, achieving an accuracy not much greater than 50% (see Fig. 2(b)).

We could try and use basis expansion or kernel methods, and they probably would work quite well for this task. Instead, let us consider a simple multilayer perceptron. Let us consider the model show in Fig. 3. There is a lot of notation to unpack in the figure, so let us first go over that. There are *three* layers in the model, the first is referred to as the *input* layer, which just consists of the inputs x_1, x_2. (There are 1s shown as part of input, but essentially they are just accounting for the bias term; in the future we'll drop the 1s and assume that every unit also has a bias term.) The second layer is a *hidden* layer which consists of two hidden units. The layer is said to be hidden because this is not observed as part of the data. Only the inputs and the outputs are observed. In general, it is possible to have more than one hidden layer; indeed, having a large number of hidden layers seems crucial for the success of neural networks on challenging tasks. The third, and in this case final, layer is the output layer. As we're solving a classification problem, we'll have a *sigmoid* activation on the unit in the output layer, so that the output can be interpreted as the probability that the label is 1 (say blue in this case).

Notation: The weight w_{ij}^l is the weight corresponding to the connection that goes from the j^{th} unit in the $(l-1)^{th}$ layer to the i^{th} unit in the l^{th} layer. For example, in Figure 3, w_{21}^2 is the weight on the edge connecting the input x_1 to the second unit in the hidden (second) layer. Succintly, we can represent this by a matrix $\mathbf{W}^l$ of size $n_l \times n_{l-1}$, where n_l denotes the number of units in the l^{th} layer. Similarly, we'll denote the bias term of the i^{th} unit in the l^{th} layer by b_i^l. Succintly the bias terms for an entire layer are denoted by the vector $\mathbf{b}^l$.

Remark 1. *A remark on notation used in this lecture is in order. There are several indices on the parameters which are chosen to make the calculations easier. It is worth emphasising that the superscripts are simply* indices *denoting the layer number, and not a* power*. When we need to use powers, we will use parentheses around the parameter—the l in $w_{i,j}^l$ is simply a superscript indicating that this parameter is a weight corresponding to the connection between the i^{th} unit in the l^{th} layer and j^{th} unit in the $(l-1)^{th}$ layer, while $\left(w_{i,j}^l\right)^d$ is the d^{th} power of that parameter.*

Typically, every unit is a linear function followed by a non-linear activation function (if there is no activation function, we'll assume that the activation function is the identity function). Thus, every unit will have a *pre-activation* value and an *activated* output, sometimes simply referred to as *activation*. For unit i in layer l, we'll use z_i^l to denote its pre-activation, *i.e.,* $z_i^l = b_i^l + w_{i1}^l a_1^{l-1} + \cdots + w_{in_{l-1}}^l a_{n_{l-1}}^{l-1}$, and a_i^l to denote the activation. We'll denote by the vectors $\mathbf{z}^l$ and $\mathbf{a}^l$ the preactivations and activations of all the units in layer l. In most cases, the activation $a_i^l = f(z_i^l)$, where f is the activation function of the unit. However, some activation functions apply to an entire layer, notably the softmax function. Thus, we'll think of activations

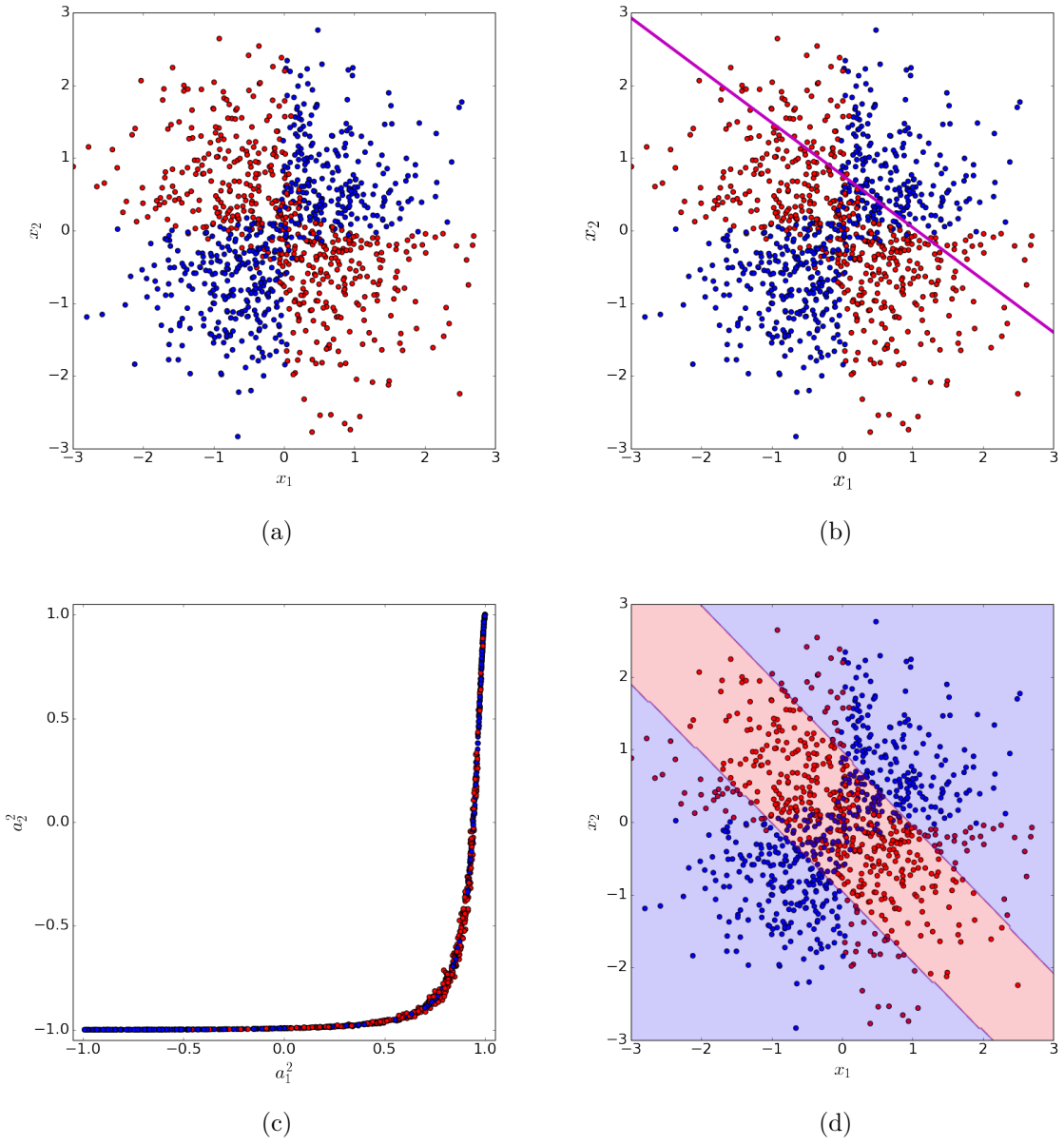


Figure 2: (a) Training data for classification (b) Logistic Regression fit (c) Training data after transformation by one hidden layer; plot of a_1^2 vs a_2^2 (see Fig. 3) (d) Decision boundary using MLP with one hidden layer and two hidden units.

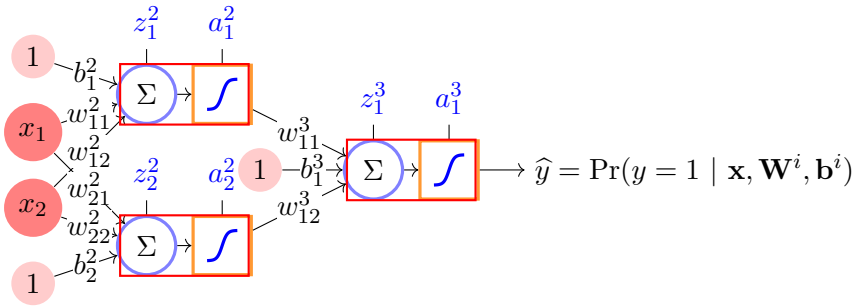


Figure 3: Multilayer Perceptron with one hidden layer and two hidden units.

as a function directly operating on vectors, $\mathbf{a}^l = f_l(\mathbf{z}^l)$, where f_l is the activation function applied to the entire layer l (See Problem 1 on Sheet 4).

For the model shown in Figure 3, let us use the activation function tanh on the hidden layer and sigmoid on the output layer. Then, we can express the model by the following equations:

$$\mathbf{a}^1 = \mathbf{z}^1 = \mathbf{x} \tag{2}$$
$$\mathbf{z}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2 \tag{3}$$
$$\mathbf{a}^2 = \tanh(\mathbf{z}^2) \tag{4}$$
$$\mathbf{z}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3 \tag{5}$$
$$\mathbf{y} = \mathbf{a}^3 = \sigma(\mathbf{z}^3) \tag{6}$$

Let us now look at how the model classifies a datapoint as being red or blue. The inputs x_1, x_2 are transformed into the activations a_1^2, a_2^2 by the hidden layer. These are non-linear and non-local transformations, in that, each of a_1^2 and a_2^2 depends on both x_1 and x_2 and they are non-linear functions of the input. The non-linearity arises due to composition with the hyperbolic tangent function. The final output, is simply a logistic regression model, but on the inputs a_1^2, a_2^2 rather than on x_1, x_2. Figure 2(c) shows a scatter plot of a_1^2 vs a_2^2. Although, the data is still not entirely linearly separable, it is much more linearly separable in terms of a_1^2 and a_2^2 than it is in terms of x_1 and x_2 (see Figure 2(a)). Thus, we can view the hidden layer of the MLP as performing "basis expansion" (or rather transformation in this case, as we have not increased the number of features), however, rather than us designing non-linear features, we allow the features themselves to be "learned" as part of training the neural network.

Let us now turn to training an MLP using the data. Let us suppose that our dataset is $\langle(\mathbf{x}_i, y_i)\rangle_{i=1}^N$. In the toy example, $\mathbf{x}_i \in \mathbb{R}^2$ and $y_i \in \{0, 1\}$, where 0 is red and 1 is blue, say. We will minimise an objective function of the form:

$$\mathcal{L}(\mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3 \ ; \ \mathcal{D}) = \sum_{i=1}^{N} \ell(\mathbf{x}_i, y_i \mid \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3) \tag{7}$$

Above, $\ell(\mathbf{x}_i, y_i \ ; \ \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3)$ denotes the *loss* on a single data point, as a result of the difference between the model prediction and the observed y_i. Let $\widehat{y}_i$ denote the model prediction given the parameters and $\mathbf{x}_i$, which in this case, $\widehat{y}_i \in [0, 1]$ is the probability that the model believes the label should be 1. For classification problems, such as this, we can use the *cross-entropy* loss function,

$$\ell(\mathbf{x}_i, y_i \ ; \ \mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3) = -\left(y_i \log \widehat{y}_i + (1 - y_i) \log(1 - \widehat{y}_i)\right) \tag{8}$$

Let us denote by $\boldsymbol{\theta}$ all the parameters in the model, $\mathbf{W}^2, \mathbf{b}^2, \mathbf{W}^3, \mathbf{b}^3$. Then we wish to find, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. We can express this as,

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{N} \frac{\partial \ell(\mathbf{x}_i, y_i; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{9}$$

Thus, in order to obtain the gradient of the objective function, $\mathcal{L}$, with respect to the parameters, we need to obtain the gradient of the loss function, ℓ, for a single datapoint. To simplify notation, let us refer to this datapoint simply as $(\mathbf{x}, y)$ and avoid the need to keep subscripts. Also, let $\mathbf{W}^{2:3}, \mathbf{b}^{2:3}$ denote all the parameters occurring in this toy model with one hidden layer. Note that other aspects of the model, such as the number of layers, the number of units in each layer and the activation functions are not considered to be parameters for the purposes of training. These could be treated as hyperparameters and model selection performed to select them. However, as neural networks take a rather long time to train, proper model selection is rather costly and some standard default parameters may be used.

1.2.1 Aside: Computing Derivatives

We'll use the convention that for a vector $\mathbf{z} \in \mathbb{R}^n$ and a function $f : \mathbb{R}^n \to \mathbb{R}$, where f takes values in $\mathbb{R}$, $\frac{\partial f}{\partial \mathbf{w}} \in \mathbb{R}^n$ is the *row vector* given by:

$$\frac{\partial f}{\partial \mathbf{z}} = \left[\frac{\partial f}{\partial z_1}, \ldots, \frac{\partial f}{\partial z_n} \right] \tag{10}$$

If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is a function, with $(\mathbf{f}(\mathbf{z}))_i = f_i(\mathbf{z})$, then, $\frac{\partial \mathbf{f}}{\partial \mathbf{z}}$ is the $m \times n$ Jacobian matrix, given by:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial f_1}{\partial z_1} & \frac{\partial f_1}{\partial z_2} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \frac{\partial f_2}{\partial z_1} & \frac{\partial f_2}{\partial z_2} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial z_1} & \frac{\partial f_m}{\partial z_2} & \cdots & \frac{\partial f_m}{\partial z_n} \end{bmatrix} \tag{11}$$

Finally, if $f : \mathbb{R}^{n \times m} \to \mathbb{R}$, then for $\mathbf{W} \in \mathbb{R}^{n \times m}$, $\frac{\partial f}{\partial \mathbf{W}} \in \mathbb{R}^{n \times m}$ given by

$$\frac{\partial f}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial f}{\partial W_{11}} & \frac{\partial f}{\partial W_{12}} & \cdots & \frac{\partial f}{\partial W_{1m}} \\ \frac{\partial f}{\partial W_{21}} & \frac{\partial f}{\partial W_{22}} & \cdots & \frac{\partial f}{\partial W_{2m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial W_{n1}} & \frac{\partial f}{\partial W_{n2}} & \cdots & \frac{\partial f}{\partial W_{nm}} \end{bmatrix} \tag{12}$$

1.2.2 Computing the Derivatives of the MLP for the Toy Model

Although, we could have computed the partical derivatives with respect to all the model parameters in an *ad hoc* fashion, let's do it using matrix operations and the following chain rule of multivariate calculus: Suppose $f : \mathbb{R}^n \to \mathbb{R}^k$, $g : \mathbb{R}^k \to \mathbb{R}^m$, let $h = g \circ f$, let $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} = f(\mathbf{x})$, then:

$$\frac{\partial h}{\partial \mathbf{x}} = \frac{\partial h}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \tag{13}$$

Note that $\frac{\partial h}{\partial \mathbf{z}}$ is the $m \times k$ Jacobian matrix and $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ is the $k \times n$ Jacobian matrix, the product of which gives the $m \times n$ Jacobian matrix $\frac{\partial h}{\partial \mathbf{x}}$. Computing the derivatives in this manner will set us up for the backpropatation equations, which can be used for more general models.

Let us suppose that we had the partial derivatives, $\frac{\partial \ell}{\partial \mathbf{z}^2}$ and $\frac{\partial \ell}{\partial \mathbf{z}^3}$ computed, although these are not the derivatives we're directly interested in as they are not with respect to the model parameters. Let us look at the partial derivative with respect to the weights w_{ij}^2, where $i \in \{1, 2\}$ and $j \in \{1, 2\}$. We have,

$$z_i^2 = w_{i1}^2 x_1 + w_{i2}^2 x_2 + b_i^2 \tag{14}$$

And thence,

$$\frac{\partial \ell}{\partial w_{ij}^2} = \frac{\partial \ell}{\partial z_i^2} \cdot \frac{\partial z_i^2}{\partial w_{ij}^2} = \frac{\partial \ell}{\partial z_i^2} \cdot x_j \tag{15}$$

More succinctly,

$$\frac{\partial \ell}{\partial \mathbf{W}^2} = \left(\mathbf{x} \frac{\partial \ell}{\partial \mathbf{z}^2}\right)^\mathsf{T} \tag{16}$$

Note that $\mathbf{x}$ is a column vector, or a matrix of size 2×1 and $\frac{\partial \ell}{\partial \mathbf{z}^2}$ is a row vector, or a matrix of size 1×2, so the RHS of (16) is the transpose of the outer product of the vector $\mathbf{x}$ and $\frac{\partial \ell}{\partial \mathbf{z}^2}$. Note that the transpose is necessary, the entry at position ji of $\mathbf{x}\frac{\partial \ell}{\partial \mathbf{z}^2}$ is $\frac{\partial \ell}{\partial w_{ij}^2}$. Similarly, using (14) we have that

$$\frac{\partial \ell}{\partial \mathbf{W}^3} = \left(\mathbf{a}^2 \frac{\partial \ell}{\partial \mathbf{z}^3}\right)^\mathsf{T} \tag{17}$$

Above we are treating $\mathbf{W}^3$ as a 1×2 matrix. Computing the partial derivatives of the bias terms is even simpler. Using (14), we have that:

$$\frac{\partial \ell}{\partial b_i^2} = \frac{\partial \ell}{\partial z_i^2} \cdot 1$$

More succinctly, we can write,

$$\frac{\partial \ell}{\partial \mathbf{b}^2} = \frac{\partial \ell}{\partial \mathbf{z}^2} \tag{18}$$

$$\frac{\partial \ell}{\partial \mathbf{b}^3} = \frac{\partial \ell}{\partial \mathbf{z}^3} \tag{19}$$

Equations (16)-(19) show that it suffices to compute the partial derivatives $\frac{\partial \ell}{\partial \mathbf{z}^2}$ and $\frac{\partial \ell}{\partial \mathbf{z}^3}$ in order to compute the partial derivatives with all the *trainable parameters* of the model. These partial derivatives can be computed using the chain rule, going backward from the output layer towards the input layer. For this reason, the resulting algorithm is called the backpropagation algorithm. In general for the loss function ℓ, we can compute $\frac{\partial \ell}{\partial \mathbf{a}^3}$ directly. To be concrete, we'll use the cross-entropy loss function in this example:

$$\ell(\mathbf{a}^3, y) = -\left(y \log(a_1^3) + (1 - y) \log(1 - a_1^3)\right) \tag{20}$$

Since $\mathbf{a}^3 \in \mathbb{R}^1$, we have,

$$\frac{\partial \ell}{\partial \mathbf{a}^3} = \left[\frac{\partial \ell}{\partial \mathbf{a}_1^3}\right] = \left[\frac{a_1^3 - y}{a_1^3(1 - a_1^3)}\right] \tag{21}$$

As, $\mathbf{a}_1^3 = \sigma(\mathbf{z}_1^3)$, $\frac{\partial \mathbf{a}^3}{\partial \mathbf{z}^3}$ is a 1×1 Jacobian matrix given by,

$$\frac{\partial \mathbf{a}^3}{\partial \mathbf{z}^3} = [\sigma'(z_1^3)] = [a_1^3(1 - a_1^3)] \tag{22}$$

And thence,

$$\frac{\partial \ell}{\partial \mathbf{z}^3} = \frac{\partial \ell}{\partial \mathbf{a}^3}\frac{\partial \mathbf{a}^3}{\partial \mathbf{z}^3} = [y - a_1^3] \tag{23}$$

Let us also compute the partial derivative $\frac{\partial \ell}{\partial \mathbf{z}^2}$; in order to do so, we will use the already computed derivative $\frac{\partial \ell}{\partial \mathbf{z}^3}$. Recall that $\mathbf{z}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3$. Thus,

$$\frac{\partial z_i^3}{\partial a_j^2} = W_{ij}^3$$

Succinctly,

$$\frac{\partial \mathbf{z}^3}{\partial \mathbf{a}^2} = \mathbf{W}^3 \tag{24}$$

Since $a_i^2 = \tanh(z_i^2)$, and $\tanh'(t) = (1 - \tanh^2(t))$, $\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2}$ is a 2×2 Jacobian, given by,

$$\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} = \begin{bmatrix} \frac{\partial a_1^2}{\partial z_1^2} & \frac{\partial a_1^2}{\partial z_2^2} \\ \frac{\partial a_2^2}{\partial z_1^2} & \frac{\partial a_2^2}{\partial z_2^2} \end{bmatrix} = \begin{bmatrix} 1 - (a_1^2)^2 & 0 \\ 0 & 1 - (a_2^2)^2 \end{bmatrix} \tag{25}$$

And thence,

$$\begin{aligned} \frac{\partial \ell}{\partial \mathbf{z}^2} &= \frac{\partial \ell}{\partial \mathbf{z}^3}\frac{\partial \mathbf{z}^3}{\partial \mathbf{a}^2}\frac{\partial \mathbf{a}^2}{\partial \mathbf{z}^2} \\ &= [a_1^3 - y]\mathbf{W}^3 \begin{bmatrix} 1 - (a_1^2)^2 & 0 \\ 0 & 1 - (a_2^2)^2 \end{bmatrix} \end{aligned} \tag{26}$$

This shows how we can compute the partial derivaties of the the loss with respect to the parameters for a single datapoint. If using gradient descent, we need to do this for every datapoint in the training data and then add (or average) the derivatives, before performing a gradient step to update the model paramters. As is more common in the context of neural networks, we average the partial derivatives over a mini-batch rather than the entire training set. (Refer to Lectures 8, 9 for the details of stochastic gradient descent and gradient descent using mini-batches.)

2 The Backpropagation Algorithm

Let us consider a general neural network with L layers, the first being the inputs, $x_1, \ldots, x_D$ and the last layer being the output. Let $\mathbf{W}^i, \mathbf{b}^i$ for $i = 2, \ldots, L$ denote the weights between layers $i-1$ and i and the bias terms for the units in layer i respectively. Let $\mathbf{W}^{2:L}$ and $\mathbf{b}^{2:L}$ denote all the weights and biases in the neural network. Let us now look at a method to compute the gradient of the loss function $\ell(\mathbf{x}, y \mid \mathbf{W}^{2:L}, \mathbf{b}^{2:L})$ for a single training datapoint $(\mathbf{x}, y)$. Typically, we can think of y as representing a single value, a real number in case of regression, or a class label in the case of classification. However, sometimes for classification problems it may be more convenient to view y as a vector in C (number of classes) dimensions, where the class label is represented using a one-hot encoding.

We will denote the neural network schematically as shown in Figure 4. For now, let us assume that all the layers are *fully connected layers*, *i.e.,* every unit in layer $l-1$ has a connection to every unit layer l. Other architectures are possible and in fact widely used; once one has understood how to derive the forward and backward equations for models with fully connected layers, it is relatively straightforward to generalise to other architectures.

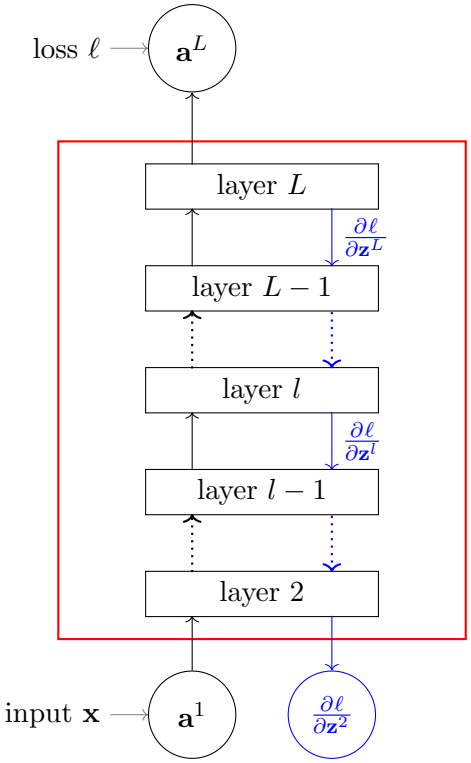


Figure 4: Neural Network. Forward and Backward Propagataion.

2.1 Forward Equations

Let us suppose that we have some value for the parameters $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$ of the model. If we are just beginning the optimisation, these will be initialised randomly. If we've already performed a few iterations of some iterative optimisation algorithm, then the parameters will have been update accordingly. The forward equations show how the predictions are made using the model, *i.e.,* given the model parameters and some input $\mathbf{x}$, the output of the last layer $\mathbf{a}^L$ is used to make the prediction. For regression problems, $\mathbf{a}^L$ will typically be a single real number which is the predicted value. For classification problems, $\mathbf{a}^L$ will typically be a probability distribution over the C classes, with a_c^L representing the probability according to the model that the input $\mathbf{x}$ belongs to class c. (For the special case of binary classification, we'll typically assume that $\mathbf{a}^L$ is a single real number in $[0, 1]$ representing the probability that the input $\mathbf{x}$ has label 1 according to the model.)

The input $\mathbf{x}$ is considered to be layer 1. Note that since there are no linear combinations and activations at the input layer, for notational convenience, we'll set $\mathbf{x} = \mathbf{z}^1 = \mathbf{a}^1$. Every other layer first computes a linear function of the outputs (activations) of the previous layer to obtain the pre-activations $\mathbf{z}^l$, and then applies a non-linear function, f_l to obtain activation $\mathbf{a}^l$. The weight parameters for the connections between layer $l-1$ and layer l are represented by the matrix $\mathbf{W}^l$ of size $n_l \times n_{l-1}$, where n_l is the number of units in layer l. The biases for the units in layer l are represented by the vector $\mathbf{b}^l$ of size n_l. The preactivations $\mathbf{z}^l$ are simply affine functions of the activations of the previous layer. The activations $\mathbf{a}^l$ (outputs of layer l) are then computed by applying the activation function f_l.

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{FE1}$$

$$\mathbf{a}^l = f(\mathbf{z}^l) \tag{FE2}$$

In most cases, the activation function is applied at the unit level, for *e.g.,* if z_i^l is the pre-activation of the i^{th} unit in layer l, then $a_i^l = f_l(z_i^l)$. In such a case, $\mathbf{a}^l = f_l(\mathbf{z}^l)$ simply applies

the activation function to each unit separately. However, sometimes, f_l may map the entire vector $\mathbf{z}^l \mapsto \mathbf{a}^l$ directly. The most common case is when f_l is the softmax function, for $\mathbf{z} \in \mathbb{R}^n$,

$$\text{softmax}(\mathbf{z}) = \left[\frac{e^z_1}{\sum_{j=1}^n e^{z_j}}, \frac{e^z_2}{\sum_{j=1}^n e^{z_j}}, \ldots, \frac{e^z_n}{\sum_{j=1}^n e^{z_j}} \right]^\mathsf{T}$$

This is usually only applied to the output layer; for other layers the non-linearities are usually applied at the unit level, though the forward and backward equations we derive will be general enough to allow layerwise activation functions. Note that using Equations (FE1)-(FE2), we can compute the output $\mathbf{a}^L$ of the neural network and hence make a prediction according to the model.

2.2 Backward Equations

When the model is being trained using some gradient based method, the parameters of the model are updated using the gradients of the loss function on the training data. For any individual datapoint $(\mathbf{x}, y)$, we'll denote the loss on that datapoint using $\ell(\mathbf{x}, y \mid \mathbf{W}^{2:L}, \mathbf{b}^{2:L})$ to emphasise that the loss under consideration is for the single datapoint and is affected by the model parameters. Typically, given the model parameters, $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$ producing a prediction $\mathbf{a}^L$, on some input $\mathbf{x}$ with target output y, the loss is a function of $\mathbf{a}^L$ and y. For example, when $\mathbf{a}^L \in \mathbb{R}$ and using the squared error as a loss function, $\ell(\mathbf{x}, y \mid \mathbf{W}^{2:L}, \mathbf{b}^{2:L}) = (\mathbf{a}^L - y)^2$. For classification problems with $y \in \{1, \ldots, C\}$, the cross entropy loss function is given by:

$$\ell(\mathbf{x}, y \,; \mathbf{W}^{2:L}, \mathbf{b}^{2:L}) = -\sum_{c=1}^{C} \mathbb{1}(y = c) \log(a^L_c)$$

Given training data $\mathcal{D} = \langle (\mathbf{x}_i, y_i) \rangle_{i=1}^N$, the objective function we want to minimise as part of the training procedure is:

$$\mathcal{L}(\mathbf{W}^{2:L}, \mathbf{b}^{2:L} \;;\; \mathcal{D}) = \sum_{i=1}^{N} \ell(\mathbf{x}_i, y_i \;;\; \mathbf{W}^{2:L}, \mathbf{b}^{2:L}) \tag{27}$$

For the optimisation algorithm, we need to compute the gradient of $\mathcal{L}$ with respect to the parameters $(\mathbf{W}^{2:L}, \mathbf{b}^{2:L})$,

$$\nabla_{(\mathbf{W}^{2:L}, \mathbf{b}^{2:L})} \mathcal{L} = \sum_{i=1}^{N} \nabla_{(\mathbf{W}^{2:L}, \mathbf{b}^{2:L})} \ell(\mathbf{x}_i, y_i \;;\; \mathbf{W}^{2:L}, \mathbf{b}^{2:L}) \tag{28}$$

Thus the key step in computing the gradient above is computing the gradient (partial derivatives) of $\ell(\mathbf{x}, y \;;\; \mathbf{W}^{2:L}, \mathbf{b}^{2:L})$ for a single datapoint; this is what the backpropagation algorithm does.

Remark: If we use a mini-batch approach instead of taking the gradient over the entire dataset, we'll replace N in (28) by B, where B is the size of the mini-batch. It is important to shuffle the data before using batches and then cycle over the batches. Refer to Lectures 8, 9 and (Murphy, 2012, Chap 8) for more details.

2.2.1 Backpropagation

In order to compute the gradient $\nabla_{(\mathbf{W}^{2:L}, \mathbf{b}^{2:L})} \ell(\mathbf{x}, y \;;\; \mathbf{W}^{2:L}, \mathbf{b}^{2:L})$, we need to compute the partial derivatives $\frac{\partial \ell}{\partial \mathbf{W}^i}$ and $\frac{\partial \ell}{\partial \mathbf{b}^i}$ for $i = 2, \ldots, L$. (We'll assume that the loss function refers to a single datapoint $(\mathbf{x}, y)$ and the model parameter $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$ for the rest of this section.) As

an intermediate step, we'll compute the partial derivatives, $\frac{\partial \ell}{\partial \mathbf{z}^l}$. In the first step, we compute the partial derivative, $\frac{\partial \ell}{\partial \mathbf{a}^L}$. Note that this can be computed directly using the form of the loss function. Using this, we can compute, $\frac{\partial \ell}{\partial \mathbf{z}^L}$ as follows:

$$\frac{\partial \ell}{\partial \mathbf{z}^L} = \frac{\partial \ell}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} \tag{BE1}$$

In the above equation, if $\mathbf{z}^L$ and $\mathbf{a}^L$ are of size n_L, then $\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}$ is the $n_L \times n_L$ Jacobian matrix. So, we are able to compute $\frac{\partial \ell}{\partial \mathbf{z}^L}$ at the output layer. Using this we can compute $\frac{\partial \ell}{\partial \mathbf{z}^{L-1}}$ and so on, backward through the layers. Suppose, we have $\frac{\partial \ell}{\partial \mathbf{z}^{l+1}}$ computed. Then, we can compute $\frac{\partial \ell}{\partial \mathbf{z}^l}$ as follows:

$$\begin{aligned}\frac{\partial \ell}{\partial \mathbf{z}^l} &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l} \\ &= \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \frac{\partial \mathbf{z}^{l+1]}}{\partial \mathbf{a}^l} \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\end{aligned}$$

As $\mathbf{z}^{l+1} = \mathbf{W}^{l+1}\mathbf{a}^l + \mathbf{b}^{l+1}$, $\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} = \mathbf{W}^{l+1}$. Further, $\frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}$ is an $n_l \times n_l$ Jacobian matrix, depending on the activation function f_l. If the activation function f_l is applied unit by unit, then this matrix is diagonal, and computational savings are achieved by not computing it explicitly. However, if f_l applies to the layer as a whole, as in the case of softmax, the entire Jacobian matrix may need to be computed. Thus,

$$\frac{\partial \ell}{\partial \mathbf{z}^l} = \frac{\partial \ell}{\partial \mathbf{z}^{l+1}} \mathbf{W}^{l+1} \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l} \tag{BE2}$$

Using Equations (BE1) and (BE2), we can compute $\frac{\partial \ell}{\partial \mathbf{z}^l}$ for $l = L, L-1, \ldots, 2$. It still remains to compute $\frac{\partial \ell}{\partial \mathbf{W}^l}$ and $\frac{\partial \ell}{\partial \mathbf{b}^l}$. However, this is relatively simple. Recall that:

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{29}$$

Using the above, we get

$$\frac{\partial \ell}{\partial W^l_{ij}} = \frac{\partial \ell}{\partial z^l_i} \cdot a^{l-1}_j$$

We can put the above in matrix form and succintly express this as,

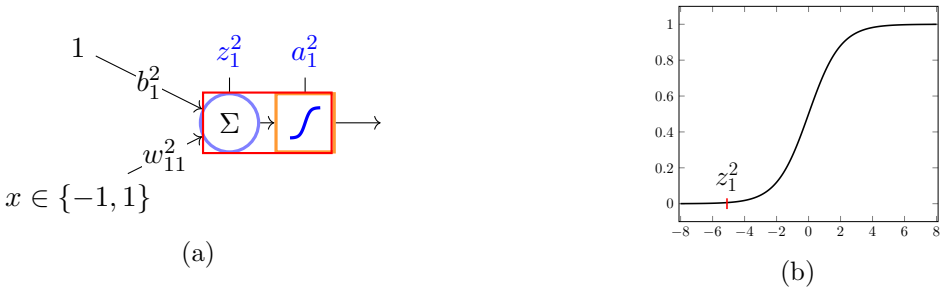
$$\frac{\partial \ell}{\partial \mathbf{W}^l} = \left(\mathbf{a}^{l-1} \frac{\partial \ell}{\partial \mathbf{z}^l}\right)^\mathsf{T} \tag{BE3}$$

Above the RHS is the transpose of the outer product of the column vector $\mathbf{a}^{l-1}$ and the row vector $\frac{\partial \ell}{\partial \mathbf{z}^l}$. Using (29), we also immediately get,

$$\frac{\partial \ell}{\partial \mathbf{b}^l} = \frac{\partial \ell}{\partial \mathbf{z}^l} \tag{BE4}$$

2.2.2 Implementing Backpropagation

The backpropagation algorithm, implemented by using the forward equations (FE1)-(FE2) and the backward equations (BE1)-(BE4), is nothing but an application of the chain rule of multivariate calculus. If we represent the loss as a function of the weight parameters, there are many paths going from a weight W^l_{ij} to the loss function, through the various layers. The chain rule requires us to multiply all the partial derivatives across each of these paths and sum them up. The backpropagation algorithm is a way of managing these computations efficiently, by storing repeatedly used partial derivatives, *i.e.,* the terms $\frac{\partial \ell}{\partial \mathbf{z}^l}$. Thus, we reduce the running time of the naïve algorithm to compute the gradient at the cost of some extra space.



(a) (b)

Figure 5: (a) A toy problem demonstrating the saturation effect. (b) Sigmoid function with $z_1^2 \approx -5$.

The dominant term in the running time of the backpropagation algorithm is the matrix multiplications that are required in the backward and forward equations. In addition to storing all the model parameters, $\mathbf{W}^{2:L}, \mathbf{b}^{2:L}$, we also need to store the preactivations $\mathbf{z}^l$ and activations $\mathbf{a}^l$ at each layer.

When actually implementing the algorithm, rather than perform the gradient computation for the loss of a single datapoint, you would do it for a batch of examples together. The matrix operations in the above equations then need to be replaced by suitable tensor operations, where the extra dimension represents the different datapoints. While there is a saving to be obtained by processing multiple datapoints together, the batch size should be chosen to be suitably small so that all the computations can still be performed on the memory available on the GPU; otherwise the advantage of processing multiple examples simultaneously will be lost.

3 Training Neural Networks

The backpropagation algorithm gives the gradient of the loss function with respect to a single datapoint. The gradient of the objective function can be computed by summing this over the entire dataset (or a minibatch). As with any other machine learning method, neural networks trained using backpropagation may be prone to overfitting. This is especially the case with very large neural networks, in which the number of parameters is typically much greater than the number of training examples available. Methods such as ℓ_2 and ℓ_1 regularisation which we used in the context of linear regression, logistic regression, *etc.* can also be used when training neural networks. In addition, there are a few other aspects of the training that are more specific to neural networks. Many of these ideas are to be viewed as 'known hacks', or current best practice, rather than exact science. Training neural networks is an active area or research and these ideas may evolve over time. The main difficulty is that the objective function being minimised when training neural networks is not a convex function of the parameters, and hence the training procedure may at times feel more art than science. Before we discuss some methods to improve the training of neural networks, let us understand some of the problems that arise when attempting to train neural networks.

3.1 Difficulties in Training Neural Networks

3.1.1 Saturation

Let us consider an extremely simple problem. Let's suppose that the training data is of the form (x, y), where $x \in \{-1, 1\}$ and $y = (1 - x)/2$, *i.e.,* if $x = 1$, the target is $y = 0$ and if $x = -1$, the target is $y = 1$. Thus, we are essentially being asked to implement a simple NOT gate. Suppose we wish to train a neural network with a single unit and sigmoid activation function. The network is shown in Figure 5. The only parameters in the model are w_{11}^2 and b_1^2. Suppose we initialised with $w_{11}^2 = -5$ and $b_1^2 = 0$. Also, suppose we use the squared loss

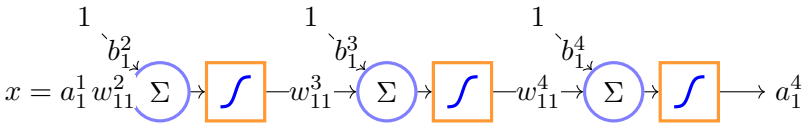


Figure 6: Demonstrating the vanishing gradient problem.

function, $\ell(\mathbf{x}, y; w^2_{11}, b^2_1) = (a^2_1 - y)^2$. Then, computing the derivative,

$$\frac{\partial \ell}{\partial z^2_1} = 2(a^2 - y)\sigma'(z^2_1) \tag{30}$$

Now, if $x = -1$, (the target is $y = 1$), $z^2_1 = -5$ and $a^2_1 \approx 0$. Thus, although $(a^2_1 - y)^2$ is very large, the derivative $\frac{\partial \ell}{\partial z^2_1} \approx 0$, as $\sigma'(z^2_1) \approx 0$. This happens because the sigmoid function is very flat at $z^2_1 = -5$, even though the prediction is off by a lot. The pre-activations being in the range where the activation function is very flat is referred to as *saturation*. When the neural networks are saturated, gradient steps may not make much progress (as the gradient is very small), even though the loss is large.

In this case, we can get around this problem by using the cross entropy loss function instead. For cross entropy, it is easy to show that $\frac{\partial \ell}{\partial z^2_1} = (a^2_1 - y)$, so if $|a^2_1 - y|$ is large, then the magnitude of the gradient will be large as well.

3.1.2 Vanishing Gradient

Let us consider the same problem considered in Section 3.1.1. However, in this case, we'll try to build a network with three hidden layers. Of course, the main purpose of these examples is to demonstrate the difficulties arising in training neural networks. (You would never use these kinds of networks for such simple problems.) Let us look at the the derivative, $\frac{\partial \ell}{\partial z^2_1}$, obtained by expanding out all of the intermediate derivatives.

$$\begin{aligned}
\frac{\partial \ell}{\partial z^2_1} &= \frac{\partial \ell}{\partial z^3_1} \cdot \frac{\partial z^3_1}{\partial z^2_1} \\
&= \frac{\partial \ell}{\partial z^4_1} \cdot \frac{\partial z^4_1}{\partial z^3_1} \cdot \frac{\partial z^3_1}{\partial z^2_1} \\
&= (a^4_1 - y) \cdot w^4_1 \cdot \sigma'(z^3_1) \cdot w^3_1 \cdot \sigma'(z^2_1)
\end{aligned} \tag{31}$$

We've seen that $\sigma'(t) = \sigma(t)(1 - \sigma(t)) \leq \frac{1}{4}$.[2] Observing (31), we see that the derivative is a product of terms containing $\sigma'(z^3_1)$ and $\sigma'(z^4_1)$. For a network with more layers there will be more such terms. The product of several numbers, each less than 1/4, approaches 0 rather quickly. Thus, the derivative may *vanish*, this is referred to as the vanishing gradient problem. It may be that the product of the weights w^4_1 and w^3_1 counteracts this effect, but unless these products exactly cancel each other, we may get a gradient that either vanishes or explodes. These problems are referred to as the *exploding* and *vanishing* gradient problems respectively.

3.2 Tricks/Ideas to Improve Training and Reduce Overfitting

Let us now discuss some good practice to avoid the problems discussed in the previous section and other problems that may arise when training neural networks. These and other tricks are discussed in much greater detail in (Goodfellow et al., 2016).

3.2.1 Avoiding Saturation

The problem of saturation is inherent to many of the activations functions used in neural networks, such as sigmoid, tanh, etc. One activation function that has been widely used in the

[2] For two numbers, $a, b \in [0, 1]$ such that $a + b = 1$, the maximum possible value for ab is 1/4.

last few years, is the rectifier, $f(z) = \max(0, z)$. The corresponding unit is called a rectified linear unit, or ReLU for short. The rectifier activation function has the advantage that it only saturates on one side. Unless all the datapoints result in a negative preactivation, at least for some datapoints, the rectifier will always have derivative 1, and so to some extent the saturation, and hence vanishing gradient problem can be avoided.

Initialisation is also important to make sure that the network is not in a saturated state at the beginning of optimisation. Suppose $w_1, \ldots, w_D$ are the weights going into a sigmoid unit, then it is usually a good idea to set the initial values drawn randomly from $\mathcal{N}(0, \sigma^2)$, $\sigma^2 = 1/D$, assuming that the inputs x_i, themselves satisfy, $\mathbb{E}\left[x_i^2\right] \approx 1$. For sigmoid units the bias term can be set to 0, or some small random number (positive for negative). For rectified linear units, it's a good idea to set the bias to be a small positive number, so that most units are not saturated to begin with. The weights for the rectified linear unit can be set as in the case of sigmoid units.

It is important to stress that the randomness in the initialisations plays an important role in symmetry breaking. All the units in a given layer typically are symmetric to begin with. If all the weights are intialised identically, then there will be nothing to differentiate them during training. This is very bad because all the units are computing exactly the same thing and it is unlikely that such a neural network would have good performance.

3.3 Avoiding Overfitting

The number of parameters in neural networks used in practice can be pretty large. For example, for the problems related to MNIST on Sheet 4, have networks with approximately 2 million parameters, whereas the training set only has 50,000 images. The network used by Krizhevsky et al. (2012), which won the Imagenet competition in 2012 has around 60 million parameters, whereas their dataset had only 1.2 million training images. It is not the surprise then that these networks overfit unless specific techniques are used to prevent it.

Classical methods of regularisation such as an ℓ_1 or ℓ_2 penalty can be used. There are a few other approaches specific to neural networks that we'll discuss here.

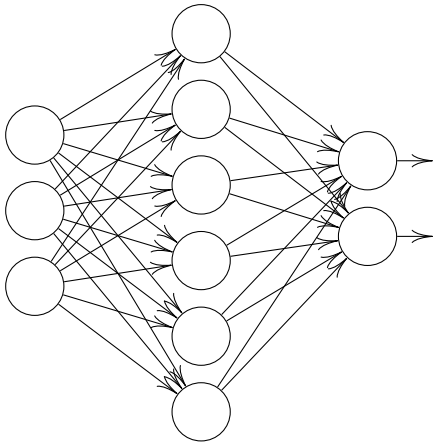
Early Stopping

Almost always, neural networks are trained using iterative optimisation methods. One way to reduce overfitting is to run the optimisation for a relatively small number of steps. This is referred to as *early stopping*. A principled way to decide when to stop is to keep aside a validation set and measure the performance of the classifier after each gradient step on it. Of course, the optimisation algorithm should only use the training set, not the validation set. Once the performance on the validation set starts plateauing, the optimisation procedure can be stopped.

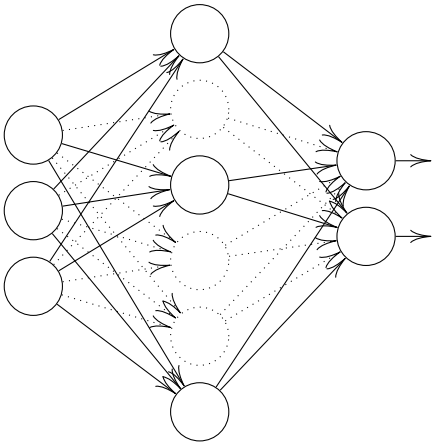
One thing to bear in mind is that because this is a non-convex optimisation problem, plateauing does not necessarily mean being close to optimality. This may be especially the case for really large networks and if the landscape of the objective function itself has plateaus. This may simply mean that we are stuck at a "local minimum", rather than indicating overfitting. Changing the learning rate in such cases can be used as a way to see if the training procedure can escape the local minimum. A detailed discussion of optimization landscapes is beyond the scope of this course and the reader is referred to (Goodfellow et al., 2016) as a starting reference point to investigate this area. There is also some recent theoretical work indicating that early stopping does indeed reduce overfitting (Hardt et al., 2016).

Adding Data

The most obvious method to reduce overfitting is to increase the quantity of training data. Of course, this is easier said than done. Obtaining additional data may be at best expensive and



(a) Neural Network



(b) Neural Network with Dropout

at worst impossible. In some domains, tricks can be used that allow us to add 'fake' data based on the existing training data. For example, when using neural networks for object detection, minor rotations, translations, *etc.* do not affect whether or not an image contains a coffee cup, or a dog, or any other thing. Thus, we may get more data by merely modifying existing training data. An extreme example of this is a Google *translation app* that is trained using entirely fake data; the only requirement in this case was to recognise letters of the alphabet and they started by generating such images and modifying them through simple transformations.[3]. This simple approach of augmenting data does not work when we have no obvious elementary transformations that we can apply to data and be sure that the targets remain unchanged.

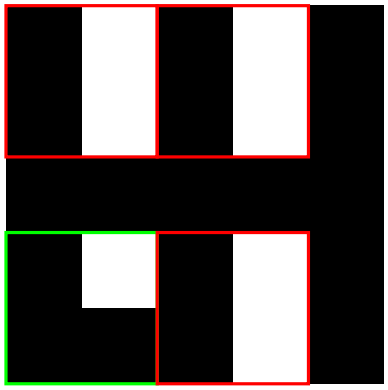
Dropout

Dropout is a way to reduce overfitting in neural networks developed by Srivastava et al. (2014). The details of this method are beyond the scope of this course. The basic idea is the following. During each training step (*i.e.,* gradient update step), a fraction (typically half) of the units in specified hidden layers are "dropped". Thus, only the weights and biases related to units that are not dropped are udpated in this gradient step. However, the choice of units to be dropped is random and different at each gradient update step. The intuition behind this is that it prevents co-adaptation among different neurons. At test time, the entire network is used. For this reason, all the weights need to be scaled appropriately before using the model (the weights need to be halved, if the dropout rate was 1/2).

The dropout approach can be viewed as a form of model averaging. In this sense, it is connected to an old idea from statistics, called bootstrap aggregation, or *bagging* Breiman (1996). In bagging the idea is the following: let $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^N$ be given. We consider k different datasets, $\mathcal{D}_1, \ldots, \mathcal{D}_k$, each of size N, where each $\mathcal{D}_l$ is obtained from $\mathcal{D}$ by sampling *with replacement*. The *with replacement* is crucial, otherwise, since $\mathcal{D}_l$ has the same size as the original dataset, we would have got $\mathcal{D}_l = \mathcal{D}$ (possibly a permutation). The with replacement sampling ensures that some datapoints from $\mathcal{D}$ appear multiple times (typically twice) in $\mathcal{D}_l$, while a few others do not appear at all. Thus, all of the datasets $\mathcal{D}_1, \ldots, \mathcal{D}_k$, although very similar as they are sampled from the same dataset $\mathcal{D}$, are slightly different. We train k models, $f_1, \ldots, f_k$, using each of these datasets. The final model is some "average" of the k models, *e.g.,* for classification, we simply can use the majority (or plurality) label, for regression, we can use the average prediction. The idea behind bagging (and dropout) is that it reduces the *variance*,[4] and hence reduces overfitting.

[3]Refer to the Google Research Blog for further details, `https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html`

[4]Recall our discussion on bias-variance in Lecture 6.



(a) Simple black and white image

Filter 1	$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$	-1	ReLU
Filter 2	$\begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix}$	0	ReLU

(b) Filter 1 detects black-white edges, Filter 2 detects black L shapes

Figure 8: Convolution operations on a simple black an white image.

Other Ideas

Several other ideas are employed to reduce overfitting in neural networks, particularly of the deep variety. These ideas include adding hard constraints on every individual weight in the model, rather than the more soft version penalising the sum of the squares or absolute values of all weights. If the gradient has large magnitude, it is "clipped" to lie in a certain range. Noise can be injected in the system before computing the gradient; the idea being that the model will only place weight on features that are truly relevant to avoid simply fitting noise. All of these ideas are discussed in greater detail in (Goodfellow et al., 2016, Chap 8).

4 Convolutional Neural Networks

Convolutional neural networks—*convnets* for short—are a class of neural networks that have enjoyed great practical success in recent years. Convolutional neural networks can be viewed as a way of reducing overfitting by introducing weight-tying that exploits the geometry. The first and most successful application of convolutional neural networks was in the context of image data, however, of late they have been employed successfully on audio, video and even text data. We'll begin by describing the convolution operation. In this lecture, we'll focus on the case where we have image data and so mostly deal with 2-dimensional convolutions; however, for different types of data, 1-dimensional (audio) and 3-dimensional (video) convolutions may make sense.

4.1 The Convolution Operation

Let us suppose that our input is structured as a 2-dimensional tensor of shape $m \times n$,[5] where each element of the tensor is a number between 0 and 1, indicating the normalised pixel intensity of a grey-scale image. The total number of pixels in the image is mn.

A convolutional filter (sometimes called kernel) is a smaller tensor, in this case say of size $W \times H$. Each number in this matrix is some real number. The convolutional filter is applied to the image by taking the dot product of the filter tensor and the image tensor for certain $W \times H$ patches in the image, *i.e.,* the corresponding WH entries (in the filter and the $W \times H$ image patch) are multiplied element-wise and added together; further, a bias term and a *non-linear* activation function may be added. Which patches to consider are decided by *stride* and *padding* parameters. In the case of 2-dimensional convolution, the stride parameters controls the step

[5] In this section, we use a d-dimensional tensor of shape $n_1 \times n_2 \times \cdots \times n_d$ to denote a table with each entry indexed by a d-tuple.

in each direction; (the stride in each dimension may be different). For example, starting with a 100×100 image (tensor) and a convolutional fiter of size 5×5, with a stride 2 in both directions, we get a new matrix of size 48×48. Starting from the $(1, 1)$ position in the input x, we first slide the filter in the x-direction (increasing columns), first to $(1, 3)$ and so on up to position $(1, 95)$. Subsequently, the filter would be set at the position $(3, 1)$ and then moved again horizontally to $(3, 95)$ and so on. Notice that we can't quite move the filter to position $(1, 97)$ or $(3, 97)$ as the 5×5 filter would then go outside the range of the input matrix. However, this also means that the 100^{th} column of the matrix is completely ignored. This is not typically a problem for large images, but if we want to include it, we can pad the image with zeros around the boundary (as may rows and columns of zeros as needed), so that every part of the original image is included in at least one convolution operation.[6]

To make things more concrete, let us consider a simple black and white image (where pixel values are either 0 or 1) and see how a 2×2 tensor (matrix) can be used to detect black-white vertical edges. Consider the filter:

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

We will use a bias term of -1 and use a rectifier, $f(x) = \max\{0, x\}$ as a non-linearity. We notice that the output of this filter (for any 2×2 image patch) is 1 precisely if this patch has a black-white edge and 0 otherwise. This is shown in Figure 8. Simliarly, Figure 8 also shows a filter that can detect black L patterns in 2×2 patches. Thus, we can understand how convolutional filters can be useful in various machine learning tasks on image data. When actually using convolutional neural networks, we will not design filters by hand but treat the elements of the filter as parameters and learn them through back propagation. We describe this in greater detail below.

So far, we've assumed that the input is 2-dimensional. However, colour images will typically be 3-dimensional, *e.g.,* using an RGB encoding. We will still consider 2-dimensional convolutions, in that the filter can only be moved in two of the three dimensions. As a starting point if the input image is a tensor with shape $m \times n \times 3$, we can consider filters with the shape $W \times H \times 3$, where W and H represent the height and width of the filter. The "3" that is common in both the shape of the input and the shape of the filter is not accidental, it is for this reason we refer to this as 2-dimensional convolution, as the filter can move across the input in only two dimensions. In general, we may want to apply more than one convolutional filter—each filter results in a 2-dimensional tensor and so the combined output of a convolutional layer can be viewed as a 3-dimensional tensor. Formally, suppose the "input" (which might be the original input or some hidden layer) of a convolutional layer with shape $m \times n \times c$, where c is the number of "channels" (*e.g.,* 3 in the case of colour images), and m and n are the width and height respectively, then applying c' filters each of shape $W \times H \times c$ results in an output of shape $m' \times n' \times c'$. Notice that c' results from the number of filters used, not from the 3-dimensional nature of the input. We can write the forward and backward equations in the same manner as for fully-connected layers; however, we have to bear in mind that the weights used in the convolutional filters are shared as the filter is moved over the input layer. To be concrete, consider a convolutional layer between layer l and $l + 1$, suppose the output (after applying activations) of the l^{th} layer is of shape $m_l \times n_l \times F_l$ and we will index the elements of this tensor as $a^l_{i,j,f}$. Suppose we apply F_{l+1} filters each of shape $W_f \times H_f \times F_l$, then we can write the entries $z^{l+1}_{i',j',f'}$ in this layer as follows (we assume no zero-padding and stride of 1 in

[6]For further details about stride and zero-padding, the reader is encouraged to look at the tensorflow documentation for conv2d (`https://www.tensorflow.org/api\_docs/python/tf/nn/conv2d`) and follow the links therein.

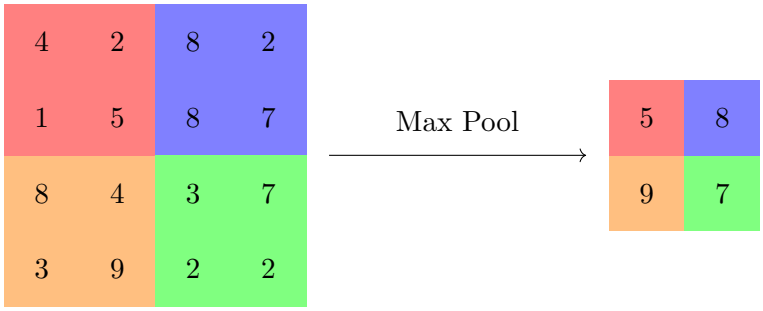


Figure 9: Max pool operation. Pool size is 2x2 and stride is (2, 2).

each direction):

$$z^{l+1}_{i',j',f'} = b^{l+1,f'} + \sum_{i=1}^{W_{f'}} \sum_{j=1}^{H_{f'}} \sum_{f=1}^{F_l} a^l_{i'+i-1,j'+j-1,f} w^{l+1,f'}_{i,j,f} \tag{32}$$

Above $w^{l+1,f'}_{i,j,f}$ is the parameter for the $f'th$ filter between layers l and $l+1$ indexed by (i,j,f). Assuming we've already computed the derivative $\frac{\partial \ell}{\partial z^{l+1}_{i',j',f'}}$ of the loss function (for a single datapoint), we can compute the gradients with respect to the parameters $w^{l+1,f'}_{i,j,f}$ as follows:

$$\frac{\partial z^{l+1}_{i',j',f'}}{\partial w^{l+1,f'}_{i,j,f}} = a^l_{i'+i-1,j'+j-1,f} \tag{33}$$

$$\frac{\partial \ell}{\partial w^{l+1,f'}_{i,j,f}} = \sum_{i',j'} \frac{\partial \ell}{\partial z^{l+1}_{i',j',f'}} \cdot a^l_{i'+i-1,j'+j-1,f} \tag{34}$$

The derivatives with respect to the bias terms, $b^{l+1,f'}$ can be computed similarly.

For backpropagation, we need to show how to compute $\frac{\partial \ell}{\partial z^l_{i,j,f}}$ given that we have already computed $\frac{\partial \ell}{\partial z^{l+1}_{i',j',f'}}$ for all possible value of (i',j',f'). Of course, we can compute $\frac{\partial a^l_{i,j,f}}{\partial z^l_{i,j,f}}$ simply using the derivative of the non-linearity used. We have the following:

$$\frac{\partial z^{l+1}_{i',j',f'}}{\partial a^l_{i,j,f}} = w^{l+1,f'}_{i-i'+1,j-j'+1,f} \tag{35}$$

$$\frac{\partial \ell}{\partial a^l_{i,j,f}} = \sum_{i',j',f'} \frac{\partial \ell}{\partial z^{l+1}_{i',j',f'}} \cdot w^{l+1,f'}_{i-i'+1,j-j'+1,f} \tag{36}$$

$$\frac{\partial \ell}{\partial z^l_{i,j,f}} = f'(z^l_{i,j,f}) \cdot \sum_{i',j',f'} \frac{\partial \ell}{\partial z^{l+1}_{i',j',f'}} \cdot w^{l+1,f'}_{i-i'+1,j-j'+1,f} \tag{37}$$

A convolutional neural network will typically have several convolutional layers starting from the input and then a few more fully connected layers towards the output. In addition to convolution operations, it is common to apply pooling operations in a convolutional neural network which are described below.

4.2 Pooling Operations

Convolutional neural networks frequently make use of a pooling operation after convolution layers. The motivation for using pooling layer is a following. Let us consider our example of detecting edges, but think of larger filters and images, then if we detect an edge in one part of the image, it is quite likely that an "edge" will be detected in the immediately neighbouring parts

of the image. However, this is of course the same edge. So in order to avoid this redundancy, the max-pool operation looks at a small matrix in the input (which may be some intermediate layer) and chooses the largest value of the entries . As in the case of convolutional fitlers, max-pool layers can be used using different sizes of the pooling and strides. Somtimes, average pooling or other pooling operations may be used, instead of max pooling. Figure 9 shows the application of a max-pool operation on a simple 2-dimensional input. Mathematically, we can express the forward and backward equations for pooling operations as follows ($\Omega(i', j')$ denotes the set of all indices input which contribute to the specific patch where the pool operation is applied, *e.g.,* in Figure 9 $\Omega(1,1) = \{(1,1),(1,2),(2,1),(2,2)\}$):

$$s^{l+1}_{i',j'} = \max_{i,j \in \Omega(i',j')} a^l_{i,j}$$

$$\frac{\partial s^{l+1}_{i',j'}}{\partial a^l_{i,j}} = \mathbb{1}\left((i,j) = \operatorname*{argmax}_{\widetilde{i},\widetilde{j} \in \Omega(i',j')} a^l_{\widetilde{i},\widetilde{j}}\right)$$

These local equations can be easily integrated into the general forward and backward equations for the whole network. The reader is encouraged to do this as practice.

References

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996. ISSN 1573-0565. doi: 10.1007/BF00058655. URL `https://doi.org/10.1007/BF00058655`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1225–1234, New York, New York, USA, 20–22 Jun 2016. PMLR. URL `http://proceedings.mlr.press/v48/hardt16.pdf`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

Kevin P. Murphy. *Machine Learning : A Probabilistic Perspective*. MIT Press, 2012.

Michael Nielsen. *Neural Networks and Deep Learning*. 2015. Online Book available at: `http://neuralnetworksanddeeplearning.com/`.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=2627435.2670313`.