Chapter 7

Deep learning

We now begin our study of deep learning. In this set of notes, we give an overview of neural networks, discuss vectorization and discuss training neural networks with backpropagation.

7.1 Supervised learning with non-linear models

In the supervised learning setting (predicting y from the input x), suppose our model/hypothesis is $h_\theta(x)$. In the past lectures, we have considered the cases when $h_\theta(x) = \theta^\top x$ (in linear regression) or $h_\theta(x) = \theta^\top \phi(x)$ (where $\phi(x)$ is the feature map). A commonality of these two models is that they are linear in the parameters θ. Next we will consider learning general family of models that are **non-linear in both** the parameters θ and the inputs x. The most common non-linear models are neural networks, which we will define staring from the next section. For this section, it suffices to think $h_\theta(x)$ as an abstract non-linear model.[1]

Suppose $\{(x^{(i)}, y^{(i)})\}_{i=1}^n$ are the training examples. We will define the nonlinear model and the loss/cost function for learning it.

Regression problems. For simplicity, we start with the case where the output is a real number, that is, $y^{(i)} \in \mathbb{R}$, and thus the model h_θ also outputs a real number $h_\theta(x) \in \mathbb{R}$. We define the least square cost function for the

[1]If a concrete example is helpful, perhaps think about the model $h_\theta(x) = \theta_1^2 x_1^2 + \theta_2^2 x_2^2 + \cdots + \theta_d^2 x_d^2$ in this subsection, even though it's not a neural network.

i-th example $(x^{(i)}, y^{(i)})$ as

$$J^{(i)}(\theta) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2 \,, \tag{7.1}$$

and define the mean-square cost function for the dataset as

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta) \,, \tag{7.2}$$

which is same as in linear regression except that we introduce a constant $1/n$ in front of the cost function to be consistent with the convention. Note that multiplying the cost function with a scalar will not change the local minima or global minima of the cost function. Also note that the underlying parameterization for $h_\theta(x)$ is different from the case of linear regression, even though the form of the cost function is the same mean-squared loss. Throughout the notes, we use the words "loss" and "cost" interchangeably.

Binary classification. Next we define the model and loss function for binary classification. Suppose the inputs $x \in \mathbb{R}^d$. Let $\bar{h}_\theta : \mathbb{R}^d \to \mathbb{R}$ be a parameterized model (the analog of $\theta^\top x$ in logistic linear regression). We call the output $\bar{h}_\theta(x) \in \mathbb{R}$ the logit. Analogous to Section 2.1, we use the logistic function $g(\cdot)$ to turn the logit $\bar{h}_\theta(x)$ to a probability $h_\theta(x) \in [0, 1]$:

$$h_\theta(x) = g(\bar{h}_\theta(x)) = 1/(1 + \exp(-\bar{h}_\theta(x)) \,. \tag{7.3}$$

We model the conditional distribution of y given x and θ by

$$\begin{aligned} P(y = 1 \mid x; \theta) &= h_\theta(x) \\ P(y = 0 \mid x; \theta) &= 1 - h_\theta(x) \end{aligned}$$

Following the same derivation in Section 2.1 and using the derivation in Remark 2.1.1, the negative likelihood loss function is equal to:

$$J^{(i)}(\theta) = -\log p(y^{(i)} \mid x^{(i)}; \theta) = \ell_{\text{logistic}}(\bar{h}_\theta(x^{(i)}), y^{(i)}) \tag{7.4}$$

As done in equation (7.2), the total loss function is also defined as the average of the loss function over individual training examples, $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta)$.

Multi-class classification. Following Section 2.3, we consider a classification problem where the response variable y can take on any one of k values, i.e. $y \in \{1, 2, \ldots, k\}$. Let $\bar{h}_\theta : \mathbb{R}^d \to \mathbb{R}^k$ be a parameterized model. We call the outputs $\bar{h}_\theta(x) \in \mathbb{R}^k$ the logits. Each logit corresponds to the prediction for one of the k classes. Analogous to Section 2.3, we use the softmax function to turn the logits $\bar{h}_\theta(x)$ into a probability vector with non-negative entries that sum up to 1:

$$P(y = j \mid x; \theta) = \frac{\exp(\bar{h}_\theta(x)_j)}{\sum_{s=1}^{k} \exp(\bar{h}_\theta(x)_s)}, \tag{7.5}$$

where $\bar{h}_\theta(x)_s$ denotes the s-th coordinate of $\bar{h}_\theta(x)$.

Similarly to Section 2.3, the loss function for a single training example $(x^{(i)}, y^{(i)})$ is its negative log-likelihood:

$$J^{(i)}(\theta) = -\log p(y^{(i)} \mid x^{(i)}; \theta) = -\log\left(\frac{\exp(\bar{h}_\theta(x^{(i)})_{y^{(i)}})}{\sum_{s=1}^{k} \exp(\bar{h}_\theta(x^{(i)})_s)}\right). \tag{7.6}$$

Using the notations of Section 2.3, we can simply write in an abstract way:

$$J^{(i)}(\theta) = \ell_{\text{ce}}(\bar{h}_\theta(x^{(i)}), y^{(i)}). \tag{7.7}$$

The loss function is also defined as the average of the loss function of individual training examples, $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta)$.

We also note that the approach above can also be generated to any conditional probabilistic model where we have an exponential distribution for y, Exponential-family$(y; \eta)$, where $\eta = \bar{h}_\theta(x)$ is a parameterized nonlinear function of x. However, the most widely used situations are the three cases discussed above.

Optimizers (SGD). Commonly, people use gradient descent (GD), stochastic gradient (SGD), or their variants to optimize the loss function $J(\theta)$. GD's update rule can be written as[2]

$$\theta := \theta - \alpha \nabla_\theta J(\theta) \tag{7.8}$$

where $\alpha > 0$ is often referred to as the learning rate or step size. Next, we introduce a version of the SGD (Algorithm 1), which is lightly different from that in the first lecture notes.

[2]Recall that, as defined in the previous lecture notes, we use the notation "$a := b$" to denote an operation (in a computer program) in which we *set* the value of a variable a to be equal to the value of b. In other words, this operation overwrites a with the value of b. In contrast, we will write "$a = b$" when we are asserting a statement of fact, that the value of a is equal to the value of b.

Algorithm 1 Stochastic Gradient Descent

```
1: Hyperparameter: learning rate $\alpha$, number of total iteration $n_{\text{iter}}$.
2: Initialize $\theta$ randomly.
3: for $i = 1$ to $n_{\text{iter}}$ do
4:     Sample $j$ uniformly from $\{1, \ldots, n\}$, and update $\theta$ by
           $$\theta := \theta - \alpha \nabla_\theta J^{(j)}(\theta) \tag{7.9}$$
```

Oftentimes computing the gradient of B examples simultaneously for the parameter θ can be faster than computing B gradients separately due to hardware parallelization. Therefore, a mini-batch version of SGD is most commonly used in deep learning, as shown in Algorithm 2. There are also other variants of the SGD or mini-batch SGD with slightly different sampling schemes.

Algorithm 2 Mini-batch Stochastic Gradient Descent

```
1: Hyperparameters: learning rate $\alpha$, batch size $B$, # iterations $n_{\text{iter}}$.
2: Initialize $\theta$ randomly
3: for $i = 1$ to $n_{\text{iter}}$ do
4:     Sample $B$ examples $j_1, \ldots, j_B$ (without replacement) uniformly from
       $\{1, \ldots, n\}$, and update $\theta$ by
           $$\theta := \theta - \frac{\alpha}{B} \sum_{k=1}^{B} \nabla_\theta J^{(j_k)}(\theta) \tag{7.10}$$
```

With these generic algorithms, a typical deep learning model is learned with the following steps. 1. Define a neural network parametrization $h_\theta(x)$, which we will introduce in Section 7.2, and 2. write the backpropagation algorithm to compute the gradient of the loss function $J^{(j)}(\theta)$ efficiently, which will be covered in Section 7.4, and 3. run SGD or mini-batch SGD (or other gradient-based optimizers) with the loss function $J(\theta)$.

7.2 Neural networks

Neural networks refer to a broad type of non-linear models/parametrizations $\bar{h}_\theta(x)$ that involve combinations of matrix multiplications and other entry-wise non-linear operations. To have a unified treatment for regression problem and classification problem, here we consider $\bar{h}_\theta(x)$ as the output of the neural network. For regression problem, the final prediction $h_\theta(x) = \bar{h}_\theta(x)$, and for classification problem, $\bar{h}_\theta(x)$ is the logits and the predicted probability will be $h_\theta(x) = 1/(1+\exp(-\bar{h}_\theta(x))$ (see equation 7.3) for binary classification or $h_\theta(x) = \text{softmax}(\bar{h}_\theta(x))$ for multi-class classification (see equation 7.5).

We will start small and slowly build up a neural network, step by step.

A Neural Network with a Single Neuron. Recall the housing price prediction problem from before: given the size of the house, we want to predict the price. We will use it as a running example in this subsection.

Previously, we fit a straight line to the graph of size vs. housing price. Now, instead of fitting a straight line, we wish to prevent negative housing prices by setting the absolute minimum price as zero. This produces a "kink" in the graph as shown in Figure 7.1. How do we represent such a function with a single kink as $\bar{h}_\theta(x)$ with unknown parameter? (After doing so, we can invoke the machinery in Section 7.1.)

We define a parameterized function $\bar{h}_\theta(x)$ with input x, parameterized by θ, which outputs the price of the house y. Formally, $\bar{h}_\theta : x \to y$. Perhaps one of the simplest parametrization would be

$$\bar{h}_\theta(x) = \max(wx + b, 0), \text{ where } \theta = (w, b) \in \mathbb{R}^2 \tag{7.11}$$

Here $\bar{h}_\theta(x)$ returns a single value: $(wx+b)$ or zero, whichever is greater. In the context of neural networks, the function $\max\{t, 0\}$ is called a ReLU (pronounced "ray-lu"), or rectified linear unit, and often denoted by $\text{ReLU}(t) \triangleq \max\{t, 0\}$.

Generally, a one-dimensional non-linear function that maps $\mathbb{R}$ to $\mathbb{R}$ such as ReLU is often referred to as an **activation function**. The model $\bar{h}_\theta(x)$ is said to have a single neuron partly because it has a single non-linear activation function. (We will discuss more about why a non-linear activation is called neuron.)

When the input $x \in \mathbb{R}^d$ has multiple dimensions, a neural network with a single neuron can be written as

$$\bar{h}_\theta(x) = \text{ReLU}(w^\top x + b), \text{ where } w \in \mathbb{R}^d,\ b \in \mathbb{R}, \text{ and } \theta = (w, b) \tag{7.12}$$

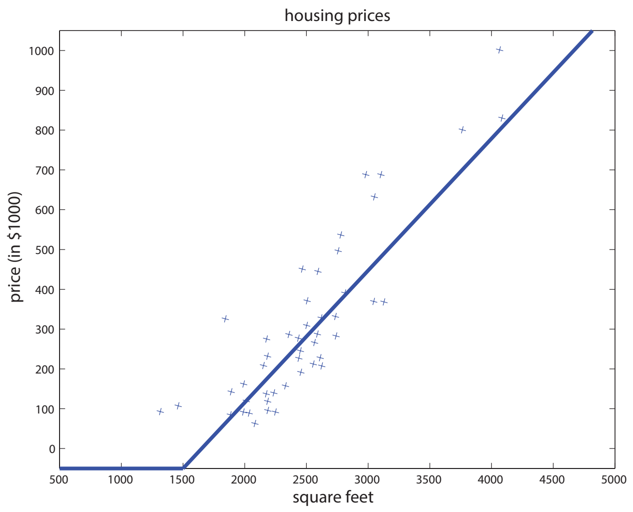

Figure 7.1: Housing prices with a "kink" in the graph.

The term b is often referred to as the "bias", and the vector w is referred to as the weight vector. Such a neural network has 1 layer. (We will define what multiple layers mean in the sequel.)

Stacking Neurons. A more complex neural network may take the single neuron described above and "stack" them together such that one neuron passes its output as input into the next neuron, resulting in a more complex function.

Let us now deepen the housing prediction example. In addition to the size of the house, suppose that you know the number of bedrooms, the zip code and the wealth of the neighborhood. Building neural networks is analogous to Lego bricks: you take individual bricks and stack them together to build complex structures. The same applies to neural networks: we take individual neurons and stack them together to create complex neural networks.

Given these features (size, number of bedrooms, zip code, and wealth), we might then decide that the price of the house depends on the maximum family size it can accommodate. Suppose the family size is a function of the size of the house and number of bedrooms (see Figure 7.2). The zip code may provide additional information such as how walkable the neighborhood is (i.e., can you walk to the grocery store or do you need to drive everywhere). Combining the zip code with the wealth of the neighborhood may predict the quality of the local elementary school. Given these three derived features (family size, walkable, school quality), we may conclude that the price of the home ultimately depends on these three features.

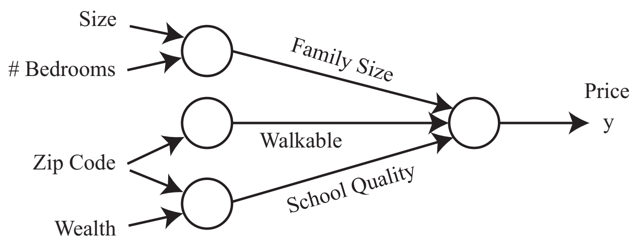

Figure 7.2: Diagram of a small neural network for predicting housing prices.

Formally, the input to a neural network is a set of input features x_1, x_2, x_3, x_4. We denote the intermediate variables for "family size", "walkable", and "school quality" by a_1, a_2, a_3 (these a_i's are often referred to as "hidden units" or "hidden neurons"). We represent each of the a_i's as a neural network with a single neuron with a subset of $x_1, \ldots, x_4$ as inputs. Then as in Figure 7.1, we will have the parameterization:

$$\begin{aligned}
a_1 &= \mathrm{ReLU}(\theta_1 x_1 + \theta_2 x_2 + \theta_3) \\
a_2 &= \mathrm{ReLU}(\theta_4 x_3 + \theta_5) \\
a_3 &= \mathrm{ReLU}(\theta_6 x_3 + \theta_7 x_4 + \theta_8)
\end{aligned}$$

where $(\theta_1, \cdots, \theta_8)$ are parameters. Now we represent the final output $\bar{h}_\theta(x)$ as another linear function with a_1, a_2, a_3 as inputs, and we get[3]

$$\bar{h}_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12} \tag{7.13}$$

where θ contains all the parameters $(\theta_1, \cdots, \theta_{12})$.

Now we represent the output as a quite complex function of x with parameters θ. Then you can use this parametrization $\bar{h}_\theta$ with the machinery of Section 7.1 to learn the parameters θ.

Inspiration from Biological Neural Networks. As the name suggests, artificial neural networks were inspired by biological neural networks. The hidden units $a_1, \ldots, a_m$ correspond to the neurons in a biological neural network, and the parameters θ_i's correspond to the synapses. However, it's unclear how similar the modern deep artificial neural networks are to the biological ones. For example, perhaps not many neuroscientists think biological

[3]Typically, for multi-layer neural network, at the end, near the output, we don't apply ReLU, especially when the output is not necessarily a positive number.

neural networks could have 1000 layers, while some modern artificial neural networks do (we will elaborate more on the notion of layers.) Moreover, it's an open question whether human brains update their neural networks in a way similar to the way that computer scientists learn artificial neural networks (using backpropagation, which we will introduce in the next section.).

Two-layer Fully-Connected Neural Networks. We constructed the neural network in equation (7.13) using a significant amount of prior knowledge/belief about how the "family size", "walkable", and "school quality" are determined by the inputs. We implicitly assumed that we know the family size is an important quantity to look at and that it can be determined by only the "size" and "# bedrooms". Such a prior knowledge might not be available for other applications. It would be more flexible and general to have a generic parameterization. A simple way would be to write the intermediate variable a_1 as a function of all $x_1, \ldots, x_4$:

$$\begin{aligned} a_1 &= \text{ReLU}(w_1^\top x + b_1), \text{ where } w_1 \in \mathbb{R}^4 \text{ and } b_1 \in \mathbb{R} \quad (7.14)\\ a_2 &= \text{ReLU}(w_2^\top x + b_2), \text{ where } w_2 \in \mathbb{R}^4 \text{ and } b_2 \in \mathbb{R}\\ a_3 &= \text{ReLU}(w_3^\top x + b_3), \text{ where } w_3 \in \mathbb{R}^4 \text{ and } b_3 \in \mathbb{R} \end{aligned}$$

We still define $\bar{h}_\theta(x)$ using equation (7.13) with a_1, a_2, a_3 being defined as above. Thus we have a so-called **fully-connected neural network** because all the intermediate variables a_i's depend on all the inputs x_i's.

For full generality, a two-layer fully-connected neural network with m hidden units and d dimensional input $x \in \mathbb{R}^d$ is defined as

$$\begin{aligned} \forall j \in [1, ..., m], \quad z_j &= {w_j^{[1]}}^\top x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \quad (7.15)\\ a_j &= \text{ReLU}(z_j),\\ a &= [a_1, \ldots, a_m]^\top \in \mathbb{R}^m\\ \bar{h}_\theta(x) &= {w^{[2]}}^\top a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R}, \quad (7.16) \end{aligned}$$

Note that by default the vectors in $\mathbb{R}^d$ are viewed as column vectors, and in particular a is a column vector with components $a_1, a_2, ..., a_m$. The indices $^{[1]}$ and $^{[2]}$ are used to distinguish two sets of parameters: the $w_j^{[1]}$'s (each of which is a vector in $\mathbb{R}^d$) and $w^{[2]}$ (which is a vector in $\mathbb{R}^m$). We will have more of these later.

Vectorization. Before we introduce neural networks with more layers and more complex structures, we will simplify the expressions for neural networks with more matrix and vector notations. Another important motivation of vectorization is the speed perspective in the implementation. In order to implement a neural network efficiently, one must be careful when using for loops. The most natural way to implement equation (7.15) in code is perhaps to use a for loop. In practice, the dimensionalities of the inputs and hidden units are high. As a result, code will run very slowly if you use for loops. Leveraging the parallelism in GPUs is/was crucial for the progress of deep learning.

This gave rise to *vectorization*. Instead of using for loops, vectorization takes advantage of matrix algebra and highly optimized numerical linear algebra packages (e.g., BLAS) to make neural network computations run quickly. Before the deep learning era, a for loop may have been sufficient on smaller datasets, but modern deep networks and state-of-the-art datasets will be infeasible to run with for loops.

We vectorize the two-layer fully-connected neural network as below. We define a weight matrix $W^{[1]}$ in $\mathbb{R}^{m\times d}$ as the concatenation of all the vectors $w_j^{[1]}$'s in the following way:

$$W^{[1]} = \begin{bmatrix} — {w_1^{[1]}}^\top — \\ — {w_2^{[1]}}^\top — \\ \vdots \\ — {w_m^{[1]}}^\top — \end{bmatrix} \in \mathbb{R}^{m\times d} \tag{7.17}$$

Now by the definition of matrix vector multiplication, we can write $z = [z_1, \dots, z_m]^\top \in \mathbb{R}^m$ as

$$\underbrace{\begin{bmatrix} z_1 \\ \vdots \\ \vdots \\ z_m \end{bmatrix}}_{z\in\mathbb{R}^{m\times 1}} = \underbrace{\begin{bmatrix} — {w_1^{[1]}}^\top — \\ — {w_2^{[1]}}^\top — \\ \vdots \\ — {w_m^{[1]}}^\top — \end{bmatrix}}_{W^{[1]}\in\mathbb{R}^{m\times d}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}}_{x\in\mathbb{R}^{d\times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix}}_{b^{[1]}\in\mathbb{R}^{m\times 1}} \tag{7.18}$$

Or succinctly,

$$z = W^{[1]}x + b^{[1]} \tag{7.19}$$

We remark again that a vector in $\mathbb{R}^d$ in this notes, following the conventions previously established, is automatically viewed as a column vector, and can also be viewed as a $d \times 1$ dimensional matrix. (Note that this is different from numpy where a vector is viewed as a row vector in broadcasting.)

Computing the activations $a \in \mathbb{R}^m$ from $z \in \mathbb{R}^m$ involves an element-wise non-linear application of the ReLU function, which can be computed in parallel efficiently. Overloading ReLU for element-wise application of ReLU (meaning, for a vector $t \in \mathbb{R}^d$, $\mathrm{ReLU}(t)$ is a vector such that $\mathrm{ReLU}(t)_i = \mathrm{ReLU}(t_i)$), we have

$$a = \mathrm{ReLU}(z) \tag{7.20}$$

Define $W^{[2]} = [w^{[2]^\top}] \in \mathbb{R}^{1\times m}$ similarly. Then, the model in equation (7.16) can be summarized as

$$\begin{aligned} a &= \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \\ \bar{h}_\theta(x) &= W^{[2]}a + b^{[2]} \end{aligned} \tag{7.21}$$

Here θ consists of $W^{[1]}, W^{[2]}$ (often referred to as the weight matrices) and $b^{[1]}, b^{[2]}$ (referred to as the biases). The collection of $W^{[1]}, b^{[1]}$ is referred to as the first layer, and $W^{[2]}, b^{[2]}$ the second layer. The activation a is referred to as the hidden layer. A two-layer neural network is also called one-hidden-layer neural network.

Multi-layer fully-connected neural networks. With this succinct notations, we can stack more layers to get a deeper fully-connected neural network. Let r be the number of layers (weight matrices). Let $W^{[1]}, \ldots, W^{[r]}, b^{[1]}, \ldots, b^{[r]}$ be the weight matrices and biases of all the layers. Then a multi-layer neural network can be written as

$$\begin{aligned} a^{[1]} &= \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \\ a^{[2]} &= \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]}) \\ &\cdots \\ a^{[r-1]} &= \mathrm{ReLU}(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \\ \bar{h}_\theta(x) &= W^{[r]}a^{[r-1]} + b^{[r]} \end{aligned} \tag{7.22}$$

We note that the weight matrices and biases need to have compatible dimensions for the equations above to make sense. If $a^{[k]}$ has dimension m_k, then the weight matrix $W^{[k]}$ should be of dimension $m_k \times m_{k-1}$, and the bias $b^{[k]} \in \mathbb{R}^{m_k}$. Moreover, $W^{[1]} \in \mathbb{R}^{m_1\times d}$ and $W^{[r]} \in \mathbb{R}^{1\times m_{r-1}}$.

The total number of neurons in the network is $m_1 + \cdots + m_r$, and the total number of parameters in this network is $(d+1)m_1 + (m_1+1)m_2 + \cdots + (m_{r-1}+1)m_r$.

Sometimes for notational consistency we also write $a^{[0]} = x$, and $a^{[r]} = h_\theta(x)$. Then we have simple recursion that

$$a^{[k]} = \mathrm{ReLU}(W^{[k]}a^{[k-1]} + b^{[k]}), \forall k = 1, \ldots, r-1 \tag{7.23}$$

Note that this would have be true for $k = r$ if there were an additional ReLU in equation (7.22), but often people like to make the last layer linear (aka without a ReLU) so that negative outputs are possible and it's easier to interpret the last layer as a linear model. (More on the interpretability at the "connection to kernel method" paragraph of this section.)

Other activation functions. The activation function ReLU can be replaced by many other non-linear function $\sigma(\cdot)$ that maps $\mathbb{R}$ to $\mathbb{R}$ such as

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{(sigmoid)} \tag{7.24}$$

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \text{(tanh)} \tag{7.25}$$

$$\sigma(z) = \max\{z, \gamma z\}, \gamma \in (0,1) \qquad \text{(leaky ReLU)} \tag{7.26}$$

$$\sigma(z) = \frac{z}{2}\left[1 + \mathrm{erf}(\frac{z}{\sqrt{2}})\right] \qquad \text{(GELU)} \tag{7.27}$$

$$\sigma(z) = \frac{1}{\beta}\log(1 + \exp(\beta z)), \beta > 0 \qquad \text{(Softplus)} \tag{7.28}$$

The activation functions are plotted in Figure 7.3. Sigmoid and tanh are less and less used these days partly because their are bounded from both sides and the gradient of them vanishes as z goes to both positive and negative infinity (whereas all the other activation functions still have gradients as the input goes to positive infinity.) Softplus is not used very often either in practice and can be viewed as a smoothing of the ReLU so that it has a proper second order derivative. GELU and leaky ReLU are both variants of ReLU but they have some non-zero gradient even when the input is negative. GELU (or its slight variant) is used in NLP models such as BERT and GPT (which we will discuss in Chapter 14.)

Why do we not use the identity function for $\sigma(z)$? That is, why not use $\sigma(z) = z$? Assume for sake of argument that $b^{[1]}$ and $b^{[2]}$ are zeros.

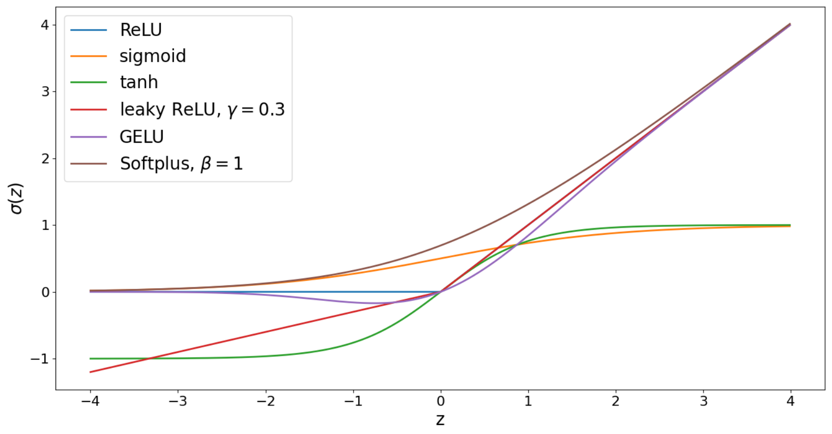

Figure 7.3: Activation functions in deep learning.

Suppose $\sigma(z) = z$, then for two-layer neural network, we have that

$$\begin{aligned}
\bar{h}_\theta(x) &= W^{[2]} a^{[1]} && && (7.29)\\
&= W^{[2]} \sigma(z^{[1]}) && \text{by definition} && (7.30)\\
&= W^{[2]} z^{[1]} && \text{since } \sigma(z) = z && (7.31)\\
&= W^{[2]} W^{[1]} x && \text{from Equation (7.18)} && (7.32)\\
&= \tilde{W} x && \text{where } \tilde{W} = W^{[2]} W^{[1]} && (7.33)
\end{aligned}$$

Notice how $W^{[2]}W^{[1]}$ collapsed into $\tilde{W}$.

This is because applying a linear function to another linear function will result in a linear function over the original input (i.e., you can construct a $\tilde{W}$ such that $\tilde{W}x = W^{[2]}W^{[1]}x$). This loses much of the representational power of the neural network as often times the output we are trying to predict has a non-linear relationship with the inputs. Without non-linear activation functions, the neural network will simply perform linear regression.

Connection to the Kernel Method. In the previous lectures, we covered the concept of feature maps. Recall that the main motivation for feature maps is to represent functions that are non-linear in the input x by $\theta^\top \phi(x)$, where θ are the parameters and $\phi(x)$, the feature map, is a handcrafted function non-linear in the raw input x. The performance of the learning algorithms can significantly depends on the choice of the feature map $\phi(x)$. Oftentimes people use domain knowledge to design the feature map $\phi(x)$ that suits the particular applications. The process of choosing the feature maps is often referred to as **feature engineering**.

We can view deep learning as a way to automatically learn the right feature map (sometimes also referred to as "the representation") as follows. Suppose we denote by β the collection of the parameters in a fully-connected neural networks (equation (7.22)) except those in the last layer. Then we can abstract right $a^{[r-1]}$ as a function of the input x and the parameters in β: $a^{[r-1]} = \phi_\beta(x)$. Now we can write the model as

$$\bar{h}_\theta(x) = W^{[r]}\phi_\beta(x) + b^{[r]} \tag{7.34}$$

When β is fixed, then $\phi_\beta(\cdot)$ can viewed as a feature map, and therefore $\bar{h}_\theta(x)$ is just a linear model over the features $\phi_\beta(x)$. However, we will train the neural networks, both the parameters in β and the parameters $W^{[r]}, b^{[r]}$ are optimized, and therefore we are not learning a linear model in the feature space, but also learning a good feature map $\phi_\beta(\cdot)$ itself so that it's possible to predict accurately with a linear model on top of the feature map. Therefore, deep learning tends to depend less on the domain knowledge of the particular applications and requires often less feature engineering. The penultimate layer $a^{[r]}$ is often (informally) referred to as the learned features or representations in the context of deep learning.

In the example of house price prediction, a fully-connected neural network does not need us to specify the intermediate quantity such "family size", and may automatically discover some useful features in the last penultimate layer (the activation $a^{[r-1]}$), and use them to linearly predict the housing price. Often the feature map / representation obtained from one datasets (that is, the function $\phi_\beta(\cdot)$ can be also useful for other datasets, which indicates they contain essential information about the data. However, oftentimes, the neural network will discover complex features which are very useful for predicting the output but may be difficult for a human to understand or interpret. This is why some people refer to neural networks as a *black box*, as it can be difficult to understand the features it has discovered.

7.3 Modules in Modern Neural Networks

The multi-layer neural network introduced in equation (7.22) of Section 7.2 is often called multi-layer perceptron (MLP) these days. Modern neural networks used in practice are often much more complex and consist of multiple building blocks or multiple layers of building blocks. In this section, we will introduce some of the other building blocks and discuss possible ways to combine them.

First, each matrix multiplication can be viewed as a building block. Consider a matrix multiplication operation with parameters (W, b) where W is the weight matrix and b is the bias vector, operating on an input z,

$$\mathrm{MM}_{W,b}(z) = Wz + b\,. \tag{7.35}$$

Note that we implicitly assume all the dimensions are chosen to be compatible. We will also drop the subscripts under MM when they are clear in the context or just for convenience when they are not essential to the discussion.

Then, the MLP can be written as as a composition of multiple matrix multiplication modules and nonlinear activation modules (which can also be viewed as a building block):

$$\mathrm{MLP}(x) = \mathrm{MM}_{W^{[r]},b^{[r]}}(\sigma(\mathrm{MM}_{W^{[r-1]},b^{[r-1]}}(\sigma(\cdots \mathrm{MM}_{W^{[1]},b^{[1]}}(x)))). \tag{7.36}$$

Alternatively, when we drop the subscripts that indicate the parameters for convenience, we can write

$$\mathrm{MLP}(x) = \mathrm{MM}(\sigma(\mathrm{MM}\sigma(\cdots \mathrm{MM}(x)))). \tag{7.37}$$

Note that in this lecture notes, by default, all the modules have different sets of parameters, and the dimensions of the parameters are chosen such that the composition is meaningful.

Larger modules can be defined via smaller modules as well, e.g., one activation layer σ and a matrix multiplication layer MM are often combined and called a "layer" in many papers. People often draw the architecture with the basic modules in a figure by indicating the dependency between these modules. E.g., see an illustration of an MLP in Figure 7.4, Left.

Residual connections. One of the very influential neural network architecture for vision application is ResNet, which uses the residual connections that are essentially used in almost all large-scale deep learning architectures these days. Using our notation above, a very much simplified residual block can be defined as

$$\mathrm{Res}(z) = z + \sigma(\mathrm{MM}(\sigma(\mathrm{MM}(z)))). \tag{7.38}$$

A much simplified ResNet is a composition of many residual blocks followed by a matrix multiplication,

$$\text{ResNet-S}(x) = \mathrm{MM}(\mathrm{Res}(\mathrm{Res}(\cdots \mathrm{Res}(x)))). \tag{7.39}$$

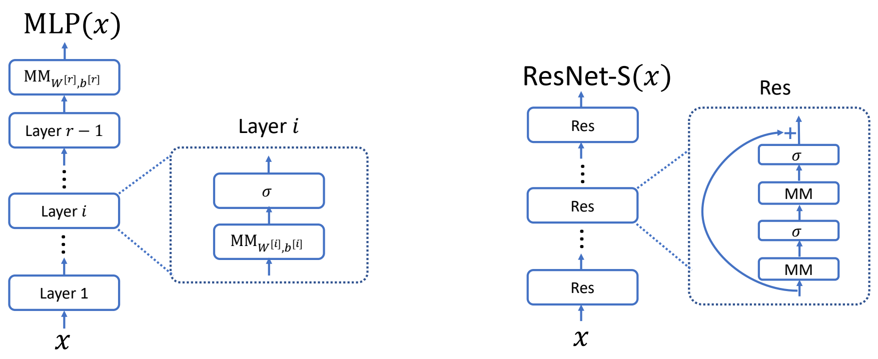

Figure 7.4: Illustrative Figures for Architecture. **Left**: An MLP with r layers. **Right**: A residual network.

We also draw the dependency of these modules in Figure 7.4, Right.

We note that the ResNet-S is still not the same as the ResNet architecture introduced in the seminal paper [He et al., 2016] because ResNet uses convolution layers instead of vanilla matrix multiplication, and adds batch normalization between convolutions and activations. We will introduce convolutional layers and some variants of batch normalization below. ResNet-S and layer normalization are part of the Transformer architecture that are widely used in modern large language models.

Layer normalization. Layer normalization, denoted by LN in this text, is a module that maps a vector $z \in \mathbb{R}^m$ to a more normalized vector $\text{LN}(z) \in \mathbb{R}^m$. It is oftentimes used after the nonlinear activations.

We first define a sub-module of the layer normalization, denoted by LN-S.

$$\text{LN-S}(z) = \begin{bmatrix} \frac{z_1-\hat{\mu}}{\hat{\sigma}} \\ \frac{z_2-\hat{\mu}}{\hat{\sigma}} \\ \vdots \\ \frac{z_m-\hat{\mu}}{\hat{\sigma}} \end{bmatrix}, \tag{7.40}$$

where $\hat{\mu} = \frac{\sum_{i=1}^m z_i}{m}$ is the empirical mean of the vector z and $\hat{\sigma} = \sqrt{\frac{\sum_{i=1}^m (z_i-\hat{\mu}^2)}{m}}$ is the empirical standard deviation of the entries of z.[4] Intuitively, LN-S(z) is a vector that is normalized to having empirical mean zero and empirical standard deviation 1.

[4]Note that we divide by m instead of $m-1$ in the empirical standard deviation here because we are interested in making the output of LN-S(z) have sum of squares equal to 1 (as opposed to estimating the standard deviation in statistics.)

Oftentimes zero mean and standard deviation 1 is not the most desired normalization scheme, and thus layernorm introduces to parameters learnable scalars β and γ as the desired mean and standard deviation, and use an affine transformation to turn the output of LN-S(z) into a vector with mean β and standard deviation γ.

$$\text{LN}(z) = \beta + \gamma \cdot \text{LN-S}(z) = \begin{bmatrix} \beta + \gamma \left(\frac{z_1 - \hat{\mu}}{\hat{\sigma}}\right) \\ \beta + \gamma \left(\frac{z_2 - \hat{\mu}}{\hat{\sigma}}\right) \\ \vdots \\ \beta + \gamma \left(\frac{z_m - \hat{\mu}}{\hat{\sigma}}\right) \end{bmatrix}. \tag{7.41}$$

Here the first occurrence of β should be technically interpreted as a vector with all the entries being β. in We also note that $\hat{\mu}$ and $\hat{\sigma}$ are also functions of z and shouldn't be treated as constants when computing the derivatives of layernorm. Moreover, β and γ are learnable parameters and thus layernorm is a parameterized module (as opposed to the activation layer which doesn't have any parameters.)

Scaling-invariant property. One important property of layer normalization is that it will make the model invariant to scaling of the parameters in the following sense. Suppose we consider composing LN with $\text{MM}_{W,b}$ and get a subnetwork $\text{LN}(\text{MM}_{W,b}(z))$. Then, we have that the output of this subnetwork does not change when the parameter in $\text{MM}_{W,b}$ is scaled:

$$\text{LN}(\text{MM}_{\alpha W, \alpha b}(z)) = \text{LN}(\text{MM}_{W,b}(z)), \forall \alpha > 0. \tag{7.42}$$

To see this, we first know that LN-S$(\cdot)$ is scale-invariant

$$\text{LN-S}(\alpha z) = \begin{bmatrix} \frac{\alpha z_1 - \alpha\hat{\mu}}{\alpha\hat{\sigma}} \\ \frac{\alpha z_2 - \alpha\hat{\mu}}{\alpha\hat{\sigma}} \\ \vdots \\ \frac{\alpha z_m - \alpha\hat{\mu}}{\alpha\hat{\sigma}} \end{bmatrix} = \begin{bmatrix} \frac{z_1 - \hat{\mu}}{\hat{\sigma}} \\ \frac{z_2 - \hat{\mu}}{\hat{\sigma}} \\ \vdots \\ \frac{z_m - \hat{\mu}}{\hat{\sigma}} \end{bmatrix} = \text{LN-S}(z). \tag{7.43}$$

Then we have

$$\begin{aligned} \text{LN}(\text{MM}_{\alpha W, \alpha b}(z)) &= \beta + \gamma \text{LN-S}(\text{MM}_{\alpha W, \alpha b}(z)) && (7.44) \\ &= \beta + \gamma \text{LN-S}(\alpha \text{MM}_{W,b}(z)) && (7.45) \\ &= \beta + \gamma \text{LN-S}(\text{MM}_{W,b}(z)) && (7.46) \\ &= \text{LN}(\text{MM}_{W,b}(z)). && (7.47) \end{aligned}$$

Due to this property, most of the modern DL architectures for large-scale computer vision and language applications have the following scale-invariant property w.r.t all the weights that are not at the last layer. Suppose the network f has last layer' weights W_{last}, and all the rest of the weights are denote by W. Then, we have $f_{W_{\text{last}},\alpha W}(x) = f_{W_{\text{last}},W}(x)$ for all $\alpha > 0$. Here, the last layers weights are special because there are typically no layernorm or batchnorm after the last layer's weights.

Other normalization layers. There are several other normalization layers that aim to normalize the intermediate layers of the neural networks to a more fixed and controllable scaling, such as batch-normalization [?], and group normalization [?]. Batch normalization and group normalization are more often used in computer vision applications whereas layer norm is used more often in language applications.

Convolutional Layers. Convolutional Neural Networks are neural networks that consist of convolution layers (and many other modules), and are particularly useful for computer vision applications. For the simplicity of exposition, we focus on 1-D convolution in this text and only briefly mention 2-D convolution informally at the end of this subsection. (2-D convolution is more suitable for images which have two dimensions. 1-D convolution is also used in natural language processing.)

We start by introducing a simplified version of the 1-D convolution layer, denoted by Conv1D-S$(\cdot)$ which is a type of matrix multiplication layer with a special structure. The parameters of Conv1D-S are a filter vector $w \in \mathbb{R}^k$ where k is called the filter size (oftentimes $k \ll m$), and a bias scalar b. Oftentimes the filter is also called a kernel (but it does not have much to do with the kernel in kernel method.) For simplicity, we assume $k = 2\ell + 1$ is an odd number. We first pad zeros to the input vector z in the sense that we let $z_{1-\ell} = z_{1-\ell+1} = .. = z_0 = 0$ and $z_{m+1} = z_{m+2} = .. = z_{m+\ell} = 0$, and treat z as an $(m + 2\ell)$-dimension vector. Conv1D-S outputs a vector of dimension $\mathbb{R}^m$ where each output dimension is a linear combination of subsets of z_j's with coefficients from w,

$$\text{Conv1D-S}(z)_i = w_1 z_{i-\ell} + w_2 z_{i-\ell+1} + \cdots + w_{2\ell+1} z_{i+\ell} = \sum_{j=1}^{2\ell+1} w_j z_{i-\ell+(j-1)}. \tag{7.48}$$

Therefore, one can view Conv1D-S as a matrix multiplication with shared parameters: Conv1D-S$(z) = Qz$, where

$$Q = \begin{bmatrix}
w_{\ell+1} & \cdots & w_{2\ell+1} & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
w_{\ell} & \cdots & w_{2\ell} & w_{2\ell+1} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
\vdots & & & & & & & & & & & \\
w_1 & \cdots & w_{\ell+1} & \cdots & \cdots & \cdots & w_{2\ell+1} & 0 & \cdots & \cdots & \cdots & 0 \\
0 & w_1 & \cdots & \cdots & \cdots & \cdots & w_{2\ell} & w_{2\ell+1} & 0 & \cdots & \cdots & 0 \\
\vdots & & & & & & & & & & & \\
\vdots & & & & & & & & & & & \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & w_1 & \cdots & & \cdots & w_{2\ell+1} \\
\vdots & & & & & & & & & & & \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & w_1 & \cdots & w_{\ell+1}
\end{bmatrix}. \tag{7.49}$$

Note that $Q_{i,j} = Q_{i-1,j-1}$ for all $i, j \in \{2, \ldots, m\}$, and thus convoluation is a matrix multiplication with parameter sharing. We also note that computing the convolution only takes $O(km)$ times but computing a generic matrix multiplication takes $O(m^2)$ time. Convolution has k parameters but generic matrix multiplication will have m^2 parameters. Thus convolution is supposed to be much more efficient than a generic matrix multiplication (as long as the additional structure imposed does not hurt the flexibility of the model to fit the data).

We also note that in practice there are many variants of the convolutional layers that we define here, e.g., there are other ways to pad zeros or sometimes the dimension of the output of the convolutional layers could be different from the input. We omit some of this subtleties here for simplicity.

The convolutional layers used in practice have also many "channels" and the simplified version above corresponds to the 1-channel version. Formally, Conv1D takes in C vectors $z_1, \ldots, z_C \in \mathbb{R}^m$ as inputs, where C is referred to as the number of channels. In other words, the more general version, denoted by Conv1D, takes in a matrix as input, which is the concatenation of $z_1, \ldots, z_C$ and has dimension $m \times C$. It can output C' vectors of dimension m, denoted by Conv1D$(z)_1, \ldots,$ Conv1D$(z)_{C'}$, where C' is referred to as the output channel, or equivalently a matrix of dimension $m \times C'$. Each of the output is a sum of the simplified convolutions applied on various channels.

$$\forall i \in [C'], \text{Conv1D}(z)_i = \sum_{j=1}^{C} \text{Conv1D-S}_{i,j}(z_j). \tag{7.50}$$

Note that each Conv1D-S$_{i,j}$ are modules with different parameters, and thus the total number of parameters is k (the number of parameters in a Conv1D-S) $\times CC'$ (the number of Conv1D-S$_{i,j}$'s) $= kCC'$. In contrast, a generic linear mapping from $\mathbb{R}^{m \times C}$ and $\mathbb{R}^{m \times C'}$ has m^2CC' parameters. The parameters can also be represented as a three-dimensional tensor of dimension $k \times C \times C'$.

2-D convolution (brief). A 2-D convolution with one channel, denoted by Conv2D-S, is analogous to the Conv1D-S, but takes a 2-dimensional input $z \in \mathbb{R}^{m \times m}$ and applies a filter of size $k \times k$, and outputs Conv2D-S$(z) \in \mathbb{R}^{m \times m}$. The full 2-D convolutional layer, denoted by Conv2D, takes in a sequence of matrices $z_1, \ldots, z_C \in \mathbb{R}^{m \times m}$, or equivalently a 3-D tensor $z = (z_1, \ldots, z_C) \in \mathbb{R}^{m \times m \times C}$ and outputs a sequence of matrices, $\text{Conv2D}(z)_1, \ldots, \text{Conv2D}(z)_{C'} \in \mathbb{R}^{m \times m}$, which can also be viewed as a 3D tensor in $\mathbb{R}^{m \times m \times C'}$. Each channel of the output is sum of the outcomes of applying Conv2D-S layers on all the input channels.

$$\forall i \in [C'], \text{Conv2D}(z)_i = \sum_{j=1}^{C} \text{Conv2D-S}_{i,j}(z_j). \tag{7.51}$$

Because there are CC' number of Conv2D-S modules and each of the Conv2D-S module has k^2 parameters, the total number of parameters is $CC'k^2$. The parameters can also be viewed as a 4D tensor of dimension $C \times C' \times k \times k$.

7.4 Backpropagation

In this section, we introduce backpropgation or auto-differentiation, which computes the gradient of the loss $\nabla J(\theta)$ efficiently. We will start with an informal theorem that states that as long as a *real-valued function* f can be efficiently computed/evaluated by a differentiable network or circuit, then its gradient can be efficiently computed in a similar time. We will then show how to do this concretely for neural networks.

Because the formality of the general theorem is not the main focus here, we will introduce the terms with informal definitions. By a differentiable circuit or a differentiable network, we mean a composition of a sequence of differentiable arithmetic operations (additions, subtraction, multiplication, divisions, etc) and elementary differentiable functions (ReLU, exp, log, sin, cos, etc.). Let the size of the circuit be the total number of such operations and elementary functions. We assume that each of the operations and functions, and their derivatives or partial derivatives ecan be computed in $O(1)$ time.

Theorem 7.4.1: *[backpropagation or auto-differentiation, informally stated] Suppose a differentiable circuit of size N computes a real-valued function*

$f : \mathbb{R}^\ell \to \mathbb{R}$. Then, the gradient ∇f can be computed in time $O(N)$, by a circuit of size $O(N)$.[5]

We note that the loss function $J^{(j)}(\theta)$ for j-th example can be indeed computed by a sequence of operations and functions involving additions, subtraction, multiplications, and non-linear activations. Thus the theorem suggests that we should be able to compute the $\nabla J^{(j)}(\theta)$ in a similar time to that for computing $J^{(j)}(\theta)$ itself. This does not only apply to the fully-connected neural network introduced in the Section 7.2, but also many other types of neural networks that uses more advance modules.

We remark that auto-differentiation or backpropagation is already implemented in all the deep learning packages such as tensorflow and pytorch, and thus in practice, in most of cases a researcher does not need to write their backpropagation algorithms. However, understanding it is very helpful for gaining insights into the working of deep learning.

Organization of the rest of the section. In Section 7.4.1, we will start reviewing the basic Chain rule with a new perspective that is particularly useful for understanding backpropgation. Section 7.4.2 will introduce the general strategy for backpropagation. Section 7.4.2 will discuss how to compute the so-called backward function for basic modules used in neural networks, and Section 7.4.4 will put everything together to get a concrete backprop algorithm for MLPs.

7.4.1 Preliminaries on partial derivatives

Suppose a *scalar variable* J depend on some variables z (which could be a scalar, matrix, or high-order tensor), we write $\frac{\partial J}{\partial z}$ as the partial derivatives of J w.r.t to the variable z. We stress that the convention here is that $\frac{\partial J}{\partial z}$ has exactly the same dimension as z itself. For example, if $z \in \mathbb{R}^{m\times n}$, then $\frac{\partial J}{\partial z} \in \mathbb{R}^{m\times n}$, and the (i,j)-entry of $\frac{\partial J}{\partial z}$ is equal to $\frac{\partial J}{\partial z_{ij}}$.

Remark 7.4.2: When both J and z are not scalars, the partial derivatives of J w.r.t z becomes either a matrix or tensor and the notation becomes somewhat tricky. Besides the mathematical or notational challenges in dealing

[5] We note if the output of the function f does not depend on some of the input coordinates, then we set by default the gradient w.r.t that coordinate to zero. Setting to zero does not count towards the total runtime here in our accounting scheme. This is why when $N \le \ell$, we can compute the gradient in $O(N)$ time, which might be potentially even less than ℓ.

with these partial derivatives of multi-variate functions, they are also expensive to compute and store, and thus rarely explicitly constructed empirically. The experience of authors of this note is that it's generally more productive to think only about derivatives of scalar function w.r.t to vector, matrices, or tensors. For example, in this note, we will not deal with derivatives of multi-variate functions.

Chain rule. We review the chain rule in calculus but with a perspective and notions that are more relevant for auto-differentiation.

Consider a scalar variable J which is obtained by the composition of f and g on some variable z,

$$\begin{aligned} & z \in \mathbb{R}^m \\ & u = g(z) \in \mathbb{R}^n \\ & J = f(u) \in \mathbb{R}\,. \end{aligned} \tag{7.52}$$

The same derivations below can be easily extend to the cases when z and u are matrices or tensors; but we insist that the final variable J is a scalar. (See also Remark 7.4.2.) Let $u = (u_1, \ldots, u_n)$ and let $g(z) = (g_1(z), \cdots, g_n(z))$. Then, the standard chain rule gives us that

$$\forall i \in \{1, \ldots, m\}, \quad \frac{\partial J}{\partial z_i} = \sum_{j=1}^{n} \frac{\partial J}{\partial u_j} \cdot \frac{\partial g_j}{\partial z_i}\,. \tag{7.53}$$

Alternatively, when z and u are both vectors, in a vectorized notation:

$$\frac{\partial J}{\partial z} = \begin{bmatrix} \frac{\partial g_1}{\partial z_1} & \cdots & \frac{\partial g_n}{\partial z_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial z_m} & \cdots & \frac{\partial g_n}{\partial z_m} \end{bmatrix} \cdot \frac{\partial J}{\partial u}\,. \tag{7.54}$$

In other words, the backward function is always a linear map from $\frac{\partial J}{\partial u}$ to $\frac{\partial J}{\partial z}$, though note that the mapping itself can depend on z in complex ways. The matrix on the RHS of (7.54) is actually the transpose of the Jacobian matrix of the function g. However, we do not discuss in-depth about Jacobian matrices to avoid complications. Part of the reason is that when z is a matrix (or tensor), to write an analog of equation (7.54), one has to either flatten z into a vector or introduce additional notations on tensor-matrix product. In this sense, equation (7.53) is more convenient and effective to use in all cases. For example, when $z \in \mathbb{R}^{r \times s}$ is a matrix, we can easily rewrite equation (7.53) to

$$\forall i,k, \quad \frac{\partial J}{\partial z_{ik}} = \sum_{j=1}^{n} \frac{\partial J}{\partial u_j} \cdot \frac{\partial g_j}{\partial z_{ik}}\,. \tag{7.55}$$

which will indeed be used in some of the derivations in Section 7.4.3.

Key interpretation of the chain rule. We can view the formula above (equation (7.53) or (7.54)) as a way to compute $\frac{\partial J}{\partial z}$ from $\frac{\partial J}{\partial u}$. Consider the following abstract problem. Suppose J depends on z via u as defined in equation (7.52). However, suppose the function f is not given or the function f is complex, but we are given the value of $\frac{\partial J}{\partial u}$. Then, the formula in equation (7.54) gives us a way to compute $\frac{\partial J}{\partial z}$ from $\frac{\partial J}{\partial u}$.

$$\frac{\partial J}{\partial u} \quad \xRightarrow[\text{only requires info about } g(\cdot) \text{ and } z]{\text{chain rule, formula (7.54)}} \quad \frac{\partial J}{\partial z}\,. \tag{7.56}$$

Moreover, this formula only involves knowledge about g (more precisely $\frac{\partial g_j}{\partial z_i}$). We will repeatedly use this fact in situations where g is a building blocks of a complex network f.

Empirically, it's often useful to modularized the mapping in (7.53) or (7.54) into a black-box, and mathematically it's also convenient to define a notation for it.[6] We use $\mathcal{B}[g,z]$ to define the function that maps $\frac{\partial J}{\partial u}$ to $\frac{\partial J}{\partial z}$, and write

$$\frac{\partial J}{\partial z} = \mathcal{B}[g,z]\left(\frac{\partial J}{\partial u}\right)\,. \tag{7.57}$$

We call $\mathcal{B}[g,z]$ the **backward function** for the module g. Note that when z is fixed, $\mathcal{B}[g,z]$ is merely a linear map from $\mathbb{R}^n$ to $\mathbb{R}^m$. Using equation (7.53), we have

$$(\mathcal{B}[g,z](v))_i = \sum_{j=1}^{m} \frac{\partial g_j}{\partial z_i} \cdot v_j\,. \tag{7.58}$$

Or in vectorized notation, using (7.54), we have

$$\mathcal{B}[g,z](v) = \begin{bmatrix} \frac{\partial g_1}{\partial z_1} & \cdots & \frac{\partial g_n}{\partial z_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_1}{\partial z_m} & \cdots & \frac{\partial g_n}{\partial z_m} \end{bmatrix} \cdot v\,. \tag{7.59}$$

[6]e.g., the function is the .backward() method of the module in pytorch.

and therefore $\mathcal{B}[g, z]$ can be viewed as a matrix. However, in reality, z will be changing and thus the backward mapping has to be recomputed for different z's while g is often fixed. Thus, empirically, the backward function $\mathcal{B}[g, z](v)$ is often viewed as a function which takes in z (=the input to g) and v (=a vector that is supposed to be the gradient of some variable J w.r.t to the output of g) as the inputs, and outputs a vector that is supposed to be the gradient of J w.r.t to z.

7.4.2 General strategy of backpropagation

We discuss the general strategy of auto-differentiation in this section to build a high-level understanding. Then, we will instantiate the approach to concrete neural networks. We take the viewpoint that neural networks are complex compositions of small building blocks such as MM, σ, Conv2D, LN, etc., defined in Section 7.3. Note that the losses (e.g., mean-squared loss, or the cross-entropy loss) can also be abstractly viewed as additional modules. Thus, we can abstractly write the loss function J (on a single example (x, y)) as a composition of many modules:[7]

$$J = M_k(M_{k-1}(\cdots M_1(x)))\,. \tag{7.60}$$

For example, for a binary classification problem with a MLP $\bar{h}_\theta(x)$ (defined in equation (7.36) and (7.37)), the loss function has ber written in the form of equation (7.60) with $M_1 = \text{MM}_{W^{[1]},b^{[1]}}$, $M_2 = \sigma$, $M_3 = \text{MM}_{W^{[2]},b^{[2]}}$, $\ldots$, and $M_{k-1} = \text{MM}_{W^{[r]},b^{[r]}}$ and $M_k = \ell_{\text{logistic}}$.

We can see from this example that some modules involve parameters, and other modules might only involve a fixed set of operations. For generality, we assume that eachj M_i involves a set of parameters $\theta^{[i]}$, though $\theta^{[i]}$ could possibly be an empty set when M_i is a fixed operation such as the nonlinear activations. We will discuss more on the granularity of the modularization, but so far we assume all the modules M_i's are simple enough.

We introduce the intermediate variables for the computation in (7.60).

[7]Technically, we should write $J = M_k(M_{k-1}(\cdots M_1(x)), y)$. However, y is treated as a constant for the purpose of computing the derivatives w.r.t to the parameters, and thus we can view it as part of M_k for the sake of simplicity of notations.

Let

$$\begin{aligned} u^{[0]} &= x \\ u^{[1]} &= M_1(u^{[0]}) \\ u^{[2]} &= M_2(u^{[1]}) \\ &\vdots \\ J = u^{[k]} &= M_k(u^{[k-1]})\,. \end{aligned} \tag{F}$$

Backpropgation consists of two passes, the forward pass and backward pass. In the forward pass, the algorithm simply computes $u^{[1]}, \dots, u^{[k]}$ from $i = 1, \dots, k$, sequentially using the definition in (F), and **save all the intermediate variables** $u^{[i]}$'s in the memory.

In the **backward pass**, we first compute the derivatives w.r.t to the intermediate variables, that is, $\frac{\partial J}{\partial u^{[k]}}, \dots, \frac{\partial J}{\partial u^{[1]}}$, sequentially in this backward order, and then compute the derivatives of the parameters $\frac{\partial J}{\partial \theta^{[i]}}$ from $\frac{\partial J}{\partial u^{[i]}}$ and $u^{[i-1]}$. These two type of computations can be also interleaved with each other because $\frac{\partial J}{\partial \theta^{[i]}}$ only depends on $\frac{\partial J}{\partial u^{[i]}}$ and $u^{[i-1]}$ but not any $\frac{\partial J}{\partial u^{[k]}}$ with $k < i$.

We first see why $\frac{\partial J}{\partial u^{[i-1]}}$ can be computed efficiently from $\frac{\partial J}{\partial u^{[i]}}$ and $u^{[i-1]}$ by invoking the discussion in Section 7.4.1 on the chain rule. We instantiate the discussion by setting $u = u^{[i]}$ and $z = u^{[i-1]}$, and $f(u) = M_k(M_{k-1}(\cdots M_{i+1}(u^{[i]})))$, and $g(\cdot) = M_i(\cdot)$. Note that f is very complex but we don't need any concrete information about f. Then, the conclusive equation (7.56) corresponds to

$$\frac{\partial J}{\partial u^{[i]}} \xRightarrow[\text{only requires info about } M_i(\cdot) \text{ and } u^{[i-1]}]{\text{chain rule}} \frac{\partial J}{\partial u^{[i-1]}}. \tag{7.61}$$

More precisely, we can write, following equation (7.57)

$$\frac{\partial J}{\partial u^{[i-1]}} = \mathcal{B}[M_i, u^{[i-1]}]\left(\frac{\partial J}{\partial u^{[i]}}\right). \tag{B1}$$

Instantiating the chain rule with $z = \theta^{[i]}$ and $u = u^{[i]}$, we also have

$$\frac{\partial J}{\partial \theta^{[i]}} = \mathcal{B}[M_i, \theta^{[i]}]\left(\frac{\partial J}{\partial u^{[i]}}\right). \tag{B2}$$

See Figure 7.5 for an illustration of the algorithm.

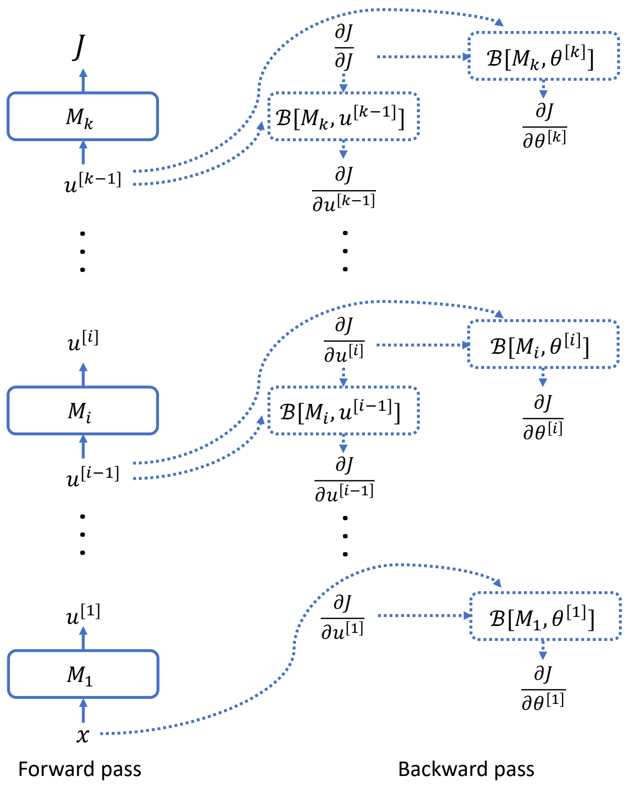
J
M_k
$u^{[k-1]}$
$\frac{\partial J}{\partial J}$
$\mathcal{B}[M_k, \theta^{[k]}]$
$\mathcal{B}[M_k, u^{[k-1]}]$
$\frac{\partial J}{\partial \theta^{[k]}}$
$\frac{\partial J}{\partial u^{[k-1]}}$
$u^{[i]}$
M_i
$u^{[i-1]}$
$\frac{\partial J}{\partial u^{[i]}}$
$\mathcal{B}[M_i, \theta^{[i]}]$
$\mathcal{B}[M_i, u^{[i-1]}]$
$\frac{\partial J}{\partial \theta^{[i]}}$
$\frac{\partial J}{\partial u^{[i-1]}}$
$u^{[1]}$
M_1
x
$\frac{\partial J}{\partial u^{[1]}}$
$\mathcal{B}[M_1, \theta^{[1]}]$
$\frac{\partial J}{\partial \theta^{[1]}}$
Forward pass
Backward pass

Figure 7.5: Back-propagation.

Remark 7.4.3: [Computational efficiency and granularity of the modules] The main underlying purpose of treating a complex network as compositions of small modules is that small modules tend to have efficiently implementable backward function. In fact, the backward functions of all the atomic modules such as addition, multiplication and ReLU can be computed as efficiently as the the evaluation of these modules (up to multiplicative constant factor). Using this fact, we can prove Theorem 7.4.1 by viewing neural networks as compositions of many atomic operations, and invoking the backpropagation discussed above. However, in practice, it's oftentimes more convenient to modularize the networks using modules on the level of matrix multiplication, layernorm, etc. As we will see, naive implementation of these operations' backward functions also have the same runtime as the evaluation of these functions.

7.4.3 Backward functions for basic modules

Using the general strategy in Section 7.4.2, it suffices to compute the backward function for all modules M_i's used in the networks. We compute the backward function for the basic module MM, activations σ, and loss functions in this section.

Backward function for MM. Suppose $\mathrm{MM}_{W,b}(z) = Wz + b$ is a matrix multiplication module where $z \in \mathbb{R}^m$ and $W \in \mathbb{R}^{n\times m}$. Then, using equation (7.59), we have for $v \in \mathbb{R}^n$

$$\mathcal{B}[\mathrm{MM}, z](v) = \begin{bmatrix} \frac{\partial(Wz+b)_1}{\partial z_1} & \cdots & \frac{\partial(Wz+b)_n}{\partial z_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial(Wz+b)_1}{\partial z_m} & \cdots & \frac{\partial(Wz+b)_n}{\partial z_m} \end{bmatrix} v \,. \tag{7.62}$$

Using the fact that $\forall i \in [m], j \in [n], \frac{\partial(Wz+b)_j}{\partial z_i} = \frac{\partial b_j + \sum_{k=1}^m W_{jk} z_k}{\partial z_i} = W_{ji}$, we have

$$\mathcal{B}[\mathrm{MM}, z](v) = W^\top v \in \mathbb{R}^m \,. \tag{7.63}$$

In the derivation above, we have treated MM as a function of z. If we treat MM as a function of W and b, then we can also compute the backward function for the parameter variables W and b. It's less convenient to use equation (7.59) because the variable W is a matrix and the matrix in (7.59) will be a 4-th order tensor that is challenging for us to mathematically write down. We use (7.58) instead:

$$(\mathcal{B}[\mathrm{MM}, W](v))_{ij} = \sum_{k=1}^m \frac{\partial(Wz+b)_k}{\partial W_{ij}} \cdot v_k = \sum_{k=1}^m \frac{\partial \sum_{s=1}^m W_{ks} z_s}{\partial W_{ij}} \cdot v_k = v_i z_j \,. \tag{7.64}$$

In vectorized notation, we have

$$\mathcal{B}[\mathrm{MM}, W](v) = v z^\top \in \mathbb{R}^{n \times \times m} \,. \tag{7.65}$$

Using equation (7.59) for the variable b, we have,

$$\mathcal{B}[\mathrm{MM}, b](v) = \begin{bmatrix} \frac{\partial(Wz+b)_1}{\partial b_1} & \cdots & \frac{\partial(Wz+b)_n}{\partial b_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial(Wz+b)_1}{\partial b_n} & \cdots & \frac{\partial(Wz+b)_n}{\partial b_n} \end{bmatrix} v = v \,. \tag{7.66}$$

Here we used that $\frac{\partial (Wz+b)_j}{\partial b_i} = 0$ if $i \neq j$ and $\frac{\partial (Wz+b)_j}{\partial b_i} = 1$ if $i = j$.

The computational efficiency for computing the backward function is $O(mn)$, the same as evaluating the result of matrix multiplication up to constant factor.

Backward function for the activations. Suppose $M(z) = \sigma(z)$ where σ is an element-wise activation function and $z \in \mathbb{R}^m$. Then, using equation (7.59), we have

$$\begin{aligned}
\mathcal{B}[\sigma, z](v) &= \begin{bmatrix} \frac{\partial \sigma(z_1)}{\partial z_1} & \cdots & \frac{\partial \sigma(z_m)}{\partial z_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial \sigma(z_1)}{\partial z_m} & \cdots & \frac{\partial \sigma(z_m)}{\partial z_m} \end{bmatrix} v && (7.67) \\
&= \operatorname{diag}(\sigma'(z_1), \cdots, \sigma'(z_m)) v && (7.68) \\
&= \sigma'(z) \odot v \in \mathbb{R}^m \,. && (7.69)
\end{aligned}$$

Here, we used the fact that $\frac{\partial \sigma(z_j)}{\partial z_i} = 0$ when $j \neq i$, $\operatorname{diag}(\lambda_1, \dots, \lambda_m)$ denotes the diagonal matrix with $\lambda_1, \dots, \lambda_m$ on the diagonal, and $\odot$ denotes the element-wise product of two vectors with the same dimension, and $\sigma'(\cdot)$ is the element-wise application of the derivative of the activation function σ.

Regarding computation efficiency, we note that at the first sight, equation (7.67) appears to indicate the backward function takes $O(m^2)$ time, but equation (7.69) shows that it's implementable in $O(m)$ time (which is the same as the time for evaluating of the function.) We are not supposed to be surprised by that the possibility of simplifying equation (7.67) to (7.69)—if we use smaller modules, that is, treating the vector-to-vector nonlinear activation as m scalar-to-scalar non-linear activation, then it's more obvious that the backward pass should have similar time to the forward pass.

Backward function for loss functions. When a module M takes in a vector z and outputs a scalar, by equation (7.59), the backward function takes in a scalar v and outputs a vector with entries $(\mathcal{B}[M, z](v))_i = \frac{\partial M}{\partial z_i} v$. Therefore, in vectorized notation, $\mathcal{B}[M, z](v) = \frac{\partial M}{\partial z} \cdot v$.

Recall that squared loss $\ell_{\text{MSE}}(z, y) = \frac{1}{2}(z - y)^2$. Thus, $\mathcal{B}[\ell_{\text{MSE}}, z](v) = \frac{\partial \frac{1}{2}(z-y)^2}{\partial z} \cdot v = (z - y) \cdot v$.

For logistics loss, by equation (2.6), we have

$$\mathcal{B}[\ell_{\text{logistic}}, t](v) = \frac{\partial \ell_{\text{logistic}}(t, y)}{\partial t} \cdot v = (1/(1 + \exp(-t)) - y) \cdot v \,. \tag{7.70}$$

For cross-entropy loss, by equation (2.17), we have

$$\mathcal{B}[\ell_{\text{ce}}, t](v) = \frac{\partial \ell_{\text{ce}}(t, y)}{\partial t} \cdot v = (\phi - e_y) \cdot v\,, \tag{7.71}$$

where $\phi = \text{softmax}(t)$.

7.4.4 Back-propagation for MLPs

Given the backward functions for every module needed in evaluating the loss of an MLP, we follow the strategy in Section 7.4.2 to compute the gradient of the loss w.r.t to the hidden activations and the parameters.

We consider the an r-layer MLP with a logistic loss. The loss function can be computed via a sequence of operations (that is, the forward pass),

$$\begin{aligned}
z^{[1]} &= \text{MM}_{W^{[1]}, b^{[1]}}(x), \\
a^{[1]} &= \sigma(z^{[1]}) \\
z^{[2]} &= \text{MM}_{W^{[2]}, b^{[2]}}(a^{[1]}) \\
a^{[2]} &= \sigma(z^{[2]}) \\
&\vdots \\
z^{[r]} &= \text{MM}_{W^{[r]}, b^{[r]}}(a^{[r-1]}) \\
J &= \ell_{\text{logistic}}(z^{[r]}, y)\,.
\end{aligned} \tag{7.72}$$

We apply the backward function sequentially in a backward order. First, we have that

$$\frac{\partial J}{\partial z^{[r]}} = \mathcal{B}[\ell_{\text{logistic}}, z^{[r]}]\left(\frac{\partial J}{\partial J}\right) = \mathcal{B}[\ell_{\text{logistic}}, z^{[r]}](1)\,. \tag{7.73}$$

Then, we iteratively compute $\frac{\partial J}{\partial a^{[i]}}$ and $\frac{\partial J}{\partial z^{[i]}}$'s by repeatedly invoking the chain rule (equation (7.58)),

$$\begin{aligned}
\frac{\partial J}{\partial a^{[r-1]}} &= \mathcal{B}[\text{MM}, a^{[r-1]}]\left(\frac{\partial J}{\partial z^{[r]}}\right) \\
\frac{\partial J}{\partial z^{[r-1]}} &= \mathcal{B}[\sigma, z^{[r-1]}]\left(\frac{\partial J}{\partial a^{[r-1]}}\right) \\
&\vdots \\
\frac{\partial J}{\partial z^{[1]}} &= \mathcal{B}[\sigma, z^{[1]}]\left(\frac{\partial J}{\partial a^{[1]}}\right)\,.
\end{aligned} \tag{7.74}$$

Numerically, we compute these quantities by repeatedly invoking equations (7.69) and (7.63) with different choices of variables.

We note that the intermediate values of $a^{[i]}$ and $z^{[i]}$ are used in the back-propagation (equation (7.74)), and therefore these values need to be stored in the memory after the forward pass.

Next, we compute the gradient of the parameters by invoking equations (7.65) and (7.66),

$$\begin{aligned}
\frac{\partial J}{\partial W^{[r]}} &= \mathcal{B}[\mathrm{MM}, W^{[r]}]\left(\frac{\partial J}{\partial z^{[r]}}\right) \\
\frac{\partial J}{\partial b^{[r]}} &= \mathcal{B}[\mathrm{MM}, b^{[r]}]\left(\frac{\partial J}{\partial z^{[r]}}\right) \\
&\vdots \\
\frac{\partial J}{\partial W^{[1]}} &= \mathcal{B}[\mathrm{MM}, W^{[1]}]\left(\frac{\partial J}{\partial z^{[1]}}\right) \\
\frac{\partial J}{\partial b^{[1]}} &= \mathcal{B}[\mathrm{MM}, b^{[1]}]\left(\frac{\partial J}{\partial z^{[1]}}\right) .
\end{aligned} \tag{7.75}$$

We also note that the block of computations in equations (7.75) can be interleaved with the block of computation in equations (7.74) because the $\frac{\partial J}{\partial W^{[i]}}$ and $\frac{\partial J}{\partial b^{[i]}}$ can be computed as soon as $\frac{\partial J}{\partial z^{[i]}}$ is computed.

Putting all of these together, and explicitly invoking the equations (7.72), (7.74) and (7.75), we have the following algorithm (Algorithm 3).

Algorithm 3 Back-propagation for multi-layer neural networks.

```
1: Forward pass. Compute and store the values of $a^{[k]}$'s, $z^{[k]}$'s, and $J$ using the equations (7.72).
2: Backward pass. Compute the gradient of loss $J$ with respect to $z^{[r]}$:
       $$\frac{\partial J}{\partial z^{[r]}} = \mathcal{B}[\ell_{\text{logistic}}, z^{[r]}](1) = \big(1/(1+\exp(-z^{[r]})) - y\big)\,. \tag{7.76}$$
3: for $k = r-1$ to 0 do
4:     Compute the gradient with respect to parameters $W^{[k+1]}$ and $b^{[k+1]}$.
           $$\frac{\partial J}{\partial W^{[k+1]}} = \mathcal{B}[\text{MM}, W^{[k+1]}]\left(\frac{\partial J}{\partial z^{[k+1]}}\right) = \frac{\partial J}{\partial z^{[k+1]}} a^{[k]\top}. \tag{7.77}$$
           $$\frac{\partial J}{\partial b^{[k+1]}} = \mathcal{B}[\text{MM}, b^{[k+1]}]\left(\frac{\partial J}{\partial z^{[k+1]}}\right) = \frac{\partial J}{\partial z^{[k+1]}}\,. \tag{7.78}$$
5:     When $k \geq 1$, compute the gradient with respect to $z^{[k]}$ and $a^{[k]}$.
           $$\frac{\partial J}{\partial a^{[k]}} = \mathcal{B}[\sigma, a^{[k]}]\left(\frac{\partial J}{\partial z^{[k+1]}}\right) = W^{[k+1]\top}\frac{\partial J}{\partial z^{[k+1]}}\,. \tag{7.79}$$
           $$\frac{\partial J}{\partial z^{[k]}} = \mathcal{B}[\sigma, z^{[k]}]\left(\frac{\partial J}{\partial a^{[k]}}\right) = \sigma'(z^{[k]}) \odot \frac{\partial J}{\partial a^{[k]}}\,. \tag{7.80}$$
```

7.5 Vectorization over training examples

As we discussed in Section 7.1, in the implementation of neural networks, we will leverage the parallelism across the multiple examples. This means that we will need to write the forward pass (the evaluation of the outputs) of the neural network and the backward pass (backpropagation) for multiple training examples in matrix notation.

The basic idea. The basic idea is simple. Suppose you have a training set with three examples $x^{(1)}, x^{(2)}, x^{(3)}$. The first-layer activations for each example are as follows:

$$\begin{aligned}
z^{[1](1)} &= W^{[1]}x^{(1)} + b^{[1]} \\
z^{[1](2)} &= W^{[1]}x^{(2)} + b^{[1]} \\
z^{[1](3)} &= W^{[1]}x^{(3)} + b^{[1]}
\end{aligned}$$

Note the difference between square brackets $[\cdot]$, which refer to the layer number, and parenthesis $(\cdot)$, which refer to the training example number. Intuitively, one would implement this using a for loop. It turns out, we can vectorize these operations as well. First, define:

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{d\times 3} \tag{7.81}$$

Note that we are stacking training examples in columns and *not* rows. We can then combine this into a single unified formulation:

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]} \tag{7.82}$$

You may notice that we are attempting to add $b^{[1]} \in \mathbb{R}^{4\times 1}$ to $W^{[1]}X \in \mathbb{R}^{4\times 3}$. Strictly following the rules of linear algebra, this is not allowed. In practice however, this addition is performed using *broadcasting.* We create an intermediate $\tilde{b}^{[1]} \in \mathbb{R}^{4\times 3}$:

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \tag{7.83}$$

We can then perform the computation: $Z^{[1]} = W^{[1]}X + \tilde{b}^{[1]}$. Often times, it is not necessary to explicitly construct $\tilde{b}^{[1]}$. By inspecting the dimensions in (7.82), you can assume $b^{[1]} \in \mathbb{R}^{4\times 1}$ is correctly broadcast to $W^{[1]}X \in \mathbb{R}^{4\times 3}$.

The matricization approach as above can easily generalize to multiple layers, with one subtlety though, as discussed below.

Complications/Subtlety in the Implementation. All the deep learning packages or implementations put the data points in the rows of a data matrix. (If the data point itself is a matrix or tensor, then the data are concentrated along the zero-th dimension.) However, most of the deep learning papers use a similar notation to these notes where the data points are treated as column vectors.[8] There is a simple conversion to deal with the mismatch: in the implementation, all the columns become row vectors, row vectors become column vectors, all the matrices are transposed, and the orders of the matrix multiplications are flipped. In the example above, using the row major convention, the data matrix is $X \in \mathbb{R}^{3\times d}$, the first layer weight matrix has dimensionality $d \times m$ (instead of $m \times d$ as in the two layer neural net section), and the bias vector $b^{[1]} \in \mathbb{R}^{1\times m}$. The computation for the hidden activation becomes

$$Z^{[1]} = XW^{[1]} + b^{[1]} \in \mathbb{R}^{3\times m} \tag{7.84}$$

[8]The instructor suspects that this is mostly because in mathematics we naturally multiply a matrix to a vector on the left hand side.