CS229 Lecture Notes

Tengyu Ma, Anand Avati, Kian Katanforoosh, and Andrew Ng

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.

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 logistic 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. For simplicity, we start with the case where $y^{(i)} \in \mathbb{R}$ and $h_\theta(x) \in \mathbb{R}$.

Cost/loss function. We define the least square cost function for the i-th example $(x^{(i)}, y^{(i)})$ as

$$J^{(i)}(\theta) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1.1}$$

[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.

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

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J^{(i)}(\theta) \tag{1.2}$$

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.

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]

$$\theta := \theta - \alpha \nabla_\theta J(\theta) \tag{1.3}$$

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.

Algorithm 1 Stochastic Gradient Descent

```
Hyperparameter: learning rate $\alpha$, number of total iteration $n_{\text{iter}}$.
Initialize $\theta$ randomly.
for $i = 1$ to $n_{\text{iter}}$ do
    Sample $j$ uniformly from $\{1, \ldots, n\}$, and update $\theta$ by
        $\theta := \theta - \alpha \nabla_\theta J^{(j)}(\theta)$    (1.4)
```

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.

[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 2 Mini-batch Stochastic Gradient Descent

```
Hyperparameters: learning rate $\alpha$, batch size $B$, # iterations $n_{\text{iter}}$.
Initialize $\theta$ randomly
for $i = 1$ to $n_{\text{iter}}$ do
    Sample $B$ examples $j_1, \ldots, j_B$ (without replacement) uniformly from $\{1, \ldots, n\}$, and update $\theta$ by
        $$\theta := \theta - \frac{\alpha}{B} \sum_{k=1}^{B} \nabla_\theta J^{(j_k)}(\theta) \quad (1.5)$$
```

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 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 3, and 3. run SGD or mini-batch SGD (or other gradient-based optimizers) with the loss function $J(\theta)$.

2 Neural Networks

Neural networks refer to broad type of non-linear models/parametrizations $h_\theta(x)$ that involve combinations of matrix multiplications and other entry-wise non-linear operations. 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 1. How do we represent such a function with a single kink as $h_\theta(x)$ with unknown parameter? (After doing so, we can invoke the machinery in Section 1.)

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

$$h_\theta(x) = \max(wx + b, 0), \text{ where } \theta = (w, b) \in \mathbb{R}^2 \tag{2.1}$$

Here $h_\theta(x)$ 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 $\mathrm{ReLU}(t) \triangleq \max\{t, 0\}$.

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 $h_\theta(x)$ 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

$$h_\theta(x) = \mathrm{ReLU}(w^\top x + b), \text{ where } w \in \mathbb{R}^d,\ b \in \mathbb{R}, \text{ and } \theta = (w, b) \tag{2.2}$$

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 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.

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

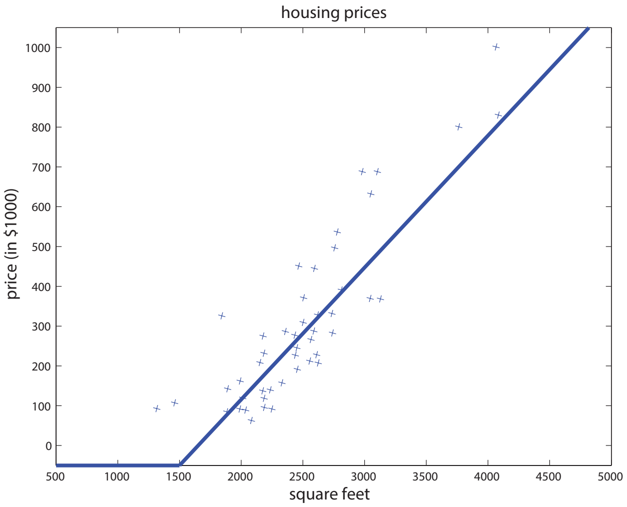

Figure 1: Housing prices with a “kink” in the graph.

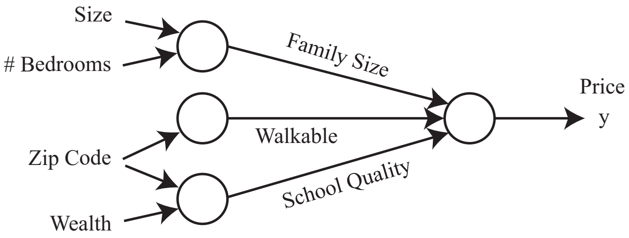

Figure 2: Diagram of a small neural network for predicting housing prices.

“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 $x_1, \dots, x_4$ as inputs. Then as in Figure 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 $h_\theta(x)$ as another linear function with a_1, a_2, a_3 as inputs, and we get[3]

$$h_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12} \tag{2.3}$$

[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.

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 h_θ with the machinery of Section 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 $a_1, \dots, a_m$ 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 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 (2.3) 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 $x_1, \dots, x_4$:

$$\begin{aligned} a_1 &= \mathrm{ReLU}(w_1^\top x + b_1), \text{ where } w_1 \in \mathbb{R}^4 \text{ and } b_1 \in \mathbb{R} \qquad (2.4) \\ a_2 &= \mathrm{ReLU}(w_2^\top x + b_2), \text{ where } w_2 \in \mathbb{R}^4 \text{ and } b_2 \in \mathbb{R} \\ a_3 &= \mathrm{ReLU}(w_3^\top x + b_3), \text{ where } w_3 \in \mathbb{R}^4 \text{ and } b_3 \in \mathbb{R} \end{aligned}$$

We still define $h_\theta(x)$ using equation (2.3) with a_1, a_2, a_3 being defined as above. Thus we have a so-called **fully-connected neural network** as visualized in the dependency graph in Figure 2 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

$$\forall j \in [1, ..., m], \quad z_j = {w_j^{[1]}}^\top x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \qquad (2.5)$$

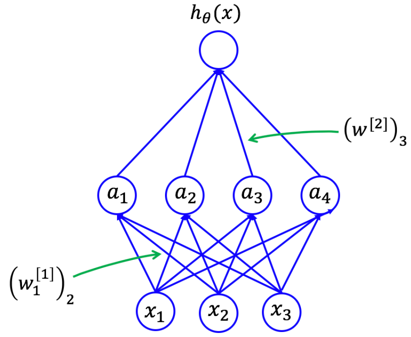

Figure 3: Diagram of a two-layer fully connected neural network. Each edge from node x_i to node a_j indicates that a_j depends on x_i. The edge from x_i to a_j is associated with the weight $(w_j^{[1]})_i$ which denotes the i-th coordinate of the vector $w_j^{[1]}$. The activation a_j can be computed by taking the ReLUof the weighted sum of x_i's with the weights being the weights associated with the incoming edges, that is, $a_j = \text{ReLU}(\sum_{i=1}^d (w_j^{[1]})_i x_i)$.

$$\begin{aligned} a_j &= \text{ReLU}(z_j), \\ a &= [a_1, \dots, a_m]^\top \in \mathbb{R}^m \\ h_\theta(x) &= w^{[2]^\top} a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R}, \end{aligned} \tag{2.6}$$

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 (2.5) 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:

$$W^{[1]} = \begin{bmatrix} \text{—}\, {w_1^{[1]}}^\top \,\text{—} \\ \text{—}\, {w_2^{[1]}}^\top \,\text{—} \\ \vdots \\ \text{—}\, {w_m^{[1]}}^\top \,\text{—} \end{bmatrix} \in \mathbb{R}^{m\times d} \tag{2.7}$$

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

$$\underbrace{\begin{bmatrix} z_1 \\ \vdots \\ \vdots \\ z_m \end{bmatrix}}_{z\in\mathbb{R}^{m\times 1}} = \underbrace{\begin{bmatrix} \text{—}\, {w_1^{[1]}}^\top \,\text{—} \\ \text{—}\, {w_2^{[1]}}^\top \,\text{—} \\ \vdots \\ \text{—}\, {w_m^{[1]}}^\top \,\text{—} \end{bmatrix}}_{W^{[1]}\in\mathbb{R}^{m\times d}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}}_{x\in\mathbb{R}^{d\times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix}}_{b^{[1]}\in\mathbb{R}^{m\times 1}} \tag{2.8}$$

Or succinctly,

$$z = W^{[1]}x + b^{[1]} \tag{2.9}$$

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

$$a = \mathrm{ReLU}(z) \tag{2.10}$$

Define $W^{[2]} = [w^{[2]^\top}] \in \mathbb{R}^{1\times m}$ similarly. Then, the model in equation (2.6) can be summarized as

$$\begin{aligned} a &= \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \\ h_\theta(x) &= W^{[2]}a + b^{[2]} \end{aligned} \tag{2.11}$$

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 $W^{[1]}, \dots, W^{[r]}, b^{[1]}, \dots, b^{[r]}$ be the weight matrices and biases of all the layers. Then a multi-layer neural network can be written as

$$\begin{aligned} a^{[1]} &= \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \\ a^{[2]} &= \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]}) \\ &\cdots \\ a^{[r-1]} &= \mathrm{ReLU}(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \\ h_\theta(x) &= W^{[r]}a^{[r-1]} + b^{[r]} \end{aligned} \tag{2.12}$$

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

$$a^{[k]} = \mathrm{ReLU}(W^{[k]}a^{[k-1]} + b^{[k]}), \forall k = 1, \dots, r-1 \tag{2.13}$$

Note that this would have be true for $k = r$ if there were an additional ReLU in equation (2.12), 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

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{(sigmoid)} \tag{2.14}$$

$$\sigma(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \qquad \text{(tanh)} \tag{2.15}$$

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

$$\begin{aligned}
h_\theta(x) &= W^{[2]}a^{[1]} && && (2.16)\\
&= W^{[2]}\sigma(z^{[1]}) && \text{by definition} && (2.17)\\
&= W^{[2]}z^{[1]} && \text{since } \sigma(z) = z && (2.18)\\
&= W^{[2]}W^{[1]}x && \text{from Equation (2.8)} && (2.19)\\
&= \tilde{W}x && \text{where } \tilde{W} = W^{[2]}W^{[1]} && (2.20)
\end{aligned}$$

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 (2.12)) 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

$$h_\theta(x) = W^{[r]}\phi_\beta(x) + b^{[r]} \tag{2.21}$$

When β is fixed, then $\phi_\beta(\cdot)$ can viewed as a feature map, and therefore $h_\theta(x)$ 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 $a^{[r-1]}$ 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.

3 Backpropagation

In this section, we introduce backpropgation or auto-differentiation, which computes the gradient of the loss $\nabla J^{(j)}(\theta)$ 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 fully-connected 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 can be computed in $O(1)$ time in the computer.

Theorem 3.1 : *[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 $O(N)$.*[4]

We note that the loss function $J^{(j)}(\theta)$ for the 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 $\nabla J^{(j)}(\theta)$ 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 Section 2, but also many other types of neural networks.

In the rest of the section, we will showcase how to compute the gradient of the loss efficiently for fully-connected neural networks using backpropagation. Even though auto-differentiation or backpropagation is implemented in all the deep learning packages such as TensorFlow and PyTorch, understanding it is very helpful for gaining insights into the workings of deep learning.

3.1 Preliminary: chain rule

We first recall the chain rule in calculus. Suppose the variable J depends on the variables $\theta_1, \dots, \theta_p$ via the intermediate variables $g_1, \dots, g_k$:

$$g_j = g_j(\theta_1, \dots, \theta_p), \forall j \in \{1, \cdots, k\} \tag{3.1}$$

[4] 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 $N \leq \ell$, we can compute the gradient in $O(N)$ time, which might be potentially even less than ℓ.

$$J = J(g_1, \ldots, g_k) \tag{3.2}$$

Here we overload the meaning of g_j's: they denote both the intermediate variables but also the functions used to compute the intermediate variables. Then, by the chain rule, we have that $\forall i$,

$$\frac{\partial J}{\partial \theta_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial g_j} \frac{\partial g_j}{\partial \theta_i} \tag{3.3}$$

For the ease of invoking the chain rule in the following subsections in various ways, we will call J the output variable, $g_1, \ldots, g_k$ intermediate variables, and $\theta_1, \ldots, \theta_p$ the input variables in the chain rule.

3.2 Backpropagation for two-layer neural networks

Now we consider the two-layer neural network defined in equation (2.11). Our general approach is to first unpack the vectorized notation to scalar form to apply the chain rule, but as soon as we finish the derivation, we will pack the scalar equations back to a vectorized form to keep the notations succinct.

Recall the following equations are used for the computation of the loss J:

$$\begin{aligned}
z &= W^{[1]}x + b^{[1]} \\
a &= \mathrm{ReLU}(z) \\
h_\theta(x) \triangleq o &= W^{[2]}a + b^{[2]} \\
J &= \frac{1}{2}(y - o)^2
\end{aligned} \tag{3.4}$$

Recall that $W^{[1]} \in \mathbb{R}^{m \times d}$, $W^{[2]} \in \mathbb{R}^{1 \times m}$, and $b^{[1]}, z, a \in \mathbb{R}^m$, and $o, y, b^{[2]} \in \mathbb{R}$. Recall that a vector in $\mathbb{R}^d$ is automatically interpreted as a column vector (like a matrix in $\mathbb{R}^{d \times 1}$) if need be.[5]

Computing $\frac{\partial J}{\partial W^{[2]}}$. Suppose $W^{[2]} = [W_1^{[2]}, \ldots, W_m^{[2]}]$. We start by computing $\frac{\partial J}{\partial W_i^{[2]}}$ using the chain rule (3.3) with o as the intermediate variable.

$$\frac{\partial J}{\partial W_i^{[2]}} = \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial W_i^{[2]}}$$

[5] We also note that even though this is the convention in math, it's different from the convention in `numpy` where an one dimensional array will be automatically interpreted as a row vector.

$$\begin{aligned} &= (o - y) \cdot \frac{\partial o}{\partial W_i^{[2]}} \\ &= (o - y) \cdot a_i && \text{(because } o = \textstyle\sum_{i=1}^m W_i^{[2]} a_i + b^{[2]}\text{)} \end{aligned}$$

Vectorized notation. The equation above in vectorized notation becomes

$$\frac{\partial J}{\partial W^{[2]}} = (o - y) \cdot a^\top \in \mathbb{R}^{1\times m} \tag{3.5}$$

Similarly, we leave the reader to verify that

$$\frac{\partial J}{\partial b^{[2]}} = (o - y) \in \mathbb{R} \tag{3.6}$$

Clarification for the dimensionality of the partial derivative notation. We will use the notation $\frac{\partial J}{\partial A}$ frequently in the rest of the lecture notes. We note that here we only use this notation for the case when J is a **real-valued** variable,[6] but A can be a vector or a matrix. Moreover, $\frac{\partial J}{\partial A}$ has the same dimensionality as A. For example, when A is a matrix, the (i, j)-th entry of $\frac{\partial J}{\partial A}$ is equal to $\frac{\partial J}{\partial A_{ij}}$. If you are familiar with the notion of total derivatives, we note that the convention for dimensionality here is different from that for total derivatives.

Computing $\frac{\partial J}{\partial W^{[1]}}$. Next we compute $\frac{\partial J}{\partial W^{[1]}}$. We first unpack the vectorized notation: let $W_{ij}^{[1]}$ denote the (i, j)-the entry of $W^{[1]}$, where $i \in [m]$ and $j \in [d]$. We compute $\frac{\partial J}{\partial W_{ij}^{[1]}}$ using chain rule (3.3) with z_i as the intermediate variable.

$$\begin{aligned} \frac{\partial J}{\partial W_{ij}^{[1]}} &= \frac{\partial J}{\partial z_i} \cdot \frac{\partial z_i}{\partial W_{ij}^{[1]}} \\ &= \frac{\partial J}{\partial z_i} \cdot x_j && \text{(because } z_i = \textstyle\sum_{k=1}^d W_{ik}^{[1]} x_k + b_i^{[1]}\text{)} \end{aligned}$$

Vectorized notation. The equation above can be written compactly as

$$\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial z} \cdot x^\top \tag{3.7}$$

[6] There is an extension of this notation to vector or matrix variable J. However, in practice, it's often impractical to compute the derivatives of high-dimensional outputs. Thus, we will avoid using the notation $\frac{\partial J}{\partial A}$ for J that is not a real-valued variable.

We can verify that the dimensions match: $\frac{\partial J}{\partial W^{[1]}} \in \mathbb{R}^{m\times d}$, $\frac{\partial J}{\partial z} \in \mathbb{R}^{m\times 1}$ and $x^\top \in \mathbb{R}^{1\times d}$.

Abstraction: For future usage, the computations for $\frac{\partial J}{\partial W^{[1]}}$ and $\frac{\partial J}{\partial W^{[2]}}$ above can be abstractified into the following claim:

Claim 3.2 : Suppose J is a real-valued output variable, $z \in \mathbb{R}^m$ is the intermediate variable, and $W \in \mathbb{R}^{m\times d}, u \in \mathbb{R}^d, b \in \mathbb{R}^m$ are the input variables, and suppose they satisfy the following:

$$z = Wu + b \tag{3.8}$$

$$J = J(z) \tag{3.9}$$

Then $\frac{\partial J}{\partial W}$ and $\frac{\partial J}{\partial b}$ satisfy:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial z} \cdot u^\top \tag{3.10}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial z} \tag{3.11}$$

Computing $\frac{\partial J}{\partial z}$. Equation (3.7) tells us that to compute $\frac{\partial J}{\partial W^{[1]}}$, it suffices to compute $\frac{\partial J}{\partial z}$, which is the goal of the next few derivations.

We invoke the chain rule with J as the output variable, a_i as the intermediate variable, and z_i as the input variable,

$$\begin{aligned}\frac{\partial J}{\partial z_i} &= \frac{\partial J}{\partial a_i}\frac{\partial a_i}{\partial z_i}\\ &= \frac{\partial J}{\partial a_i} \cdot 1\{z_i \geq 0\}\end{aligned}$$

Vectorization and abstraction. The computation above can be summarized into:

Claim 3.3: Suppose the real-valued output variable J and vectors $z, a \in \mathbb{R}^m$ satisfy the following:

$$\begin{aligned}&a = \sigma(z), \text{ where } \sigma \text{ is an element-wise activation, } z, a \in \mathbb{R}^m\\ &J = J(a)\end{aligned}$$

Then, we have that

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial a} \odot \sigma'(z) \tag{3.12}$$

where $\sigma'(\cdot)$ is the element-wise derivative of the activation function σ, and $\odot$ denotes the element-wise product of two vectors of the same dimensionality.

Computing $\frac{\partial J}{\partial a}$. Now it suffices to compute $\frac{\partial J}{\partial a}$. We invoke the chain rule with J as the output variable, o as the intermediate variable, and a_i as the input variable,

$$\begin{aligned}\frac{\partial J}{\partial a_i} &= \frac{\partial J}{\partial o}\frac{\partial o}{\partial a_i} \\ &= (o-y)\cdot W_i^{[2]} && \text{(because } o = \textstyle\sum_{i=1}^m W_i^{[2]} a_i + b^{[2]}\text{)}\end{aligned}$$

Vectorization. In vectorized notation, we have

$$\frac{\partial J}{\partial a} = W^{[2]^\top}\cdot(o-y) \tag{3.13}$$

Abstraction. We now present a more general form of the computation above.

Claim 3.4 : Suppose J is a real-valued output variable, $v \in \mathbb{R}^m$ is the intermediate variable, and $W \in \mathbb{R}^{m\times d}, u \in \mathbb{R}^d, b \in \mathbb{R}^m$ are the input variables, and suppose they satisfy the following:

$$\begin{aligned}v &= Wu + b \\ J &= J(v)\end{aligned}$$

Then,

$$\frac{\partial J}{\partial u} = W^\top \frac{\partial J}{\partial v} \tag{3.14}$$

Summary for two-layer neural networks. Now combining the equations above, we arrive at Algorithm 3 which computes the gradients for two-layer neural networks.

3.3 Multi-layer neural networks

In this section, we will derive the backpropagation algorithms for the model defined in (2.12). With the notation $a^{[0]} = x$, recall that we have

$$\begin{aligned}a^{[1]} &= \mathrm{ReLU}(W^{[1]}a^{[0]} + b^{[1]}) \\ a^{[2]} &= \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]}) \\ &\cdots \\ a^{[r-1]} &= \mathrm{ReLU}(W^{[r-1]}a^{[r-2]} + b^{[r-1]})\end{aligned}$$

Algorithm 3 Back-propagation for two-layer neural networks

.

```
Compute the values of $z \in \mathbb{R}^m$, $a \in \mathbb{R}^m$, and $o \in \mathbb{R}$
Compute
    $\delta^{[2]} \triangleq \frac{\partial J}{\partial o} = (o - y) \in \mathbb{R}$
    $\delta^{[1]} \triangleq \frac{\partial J}{\partial z} = ({W^{[2]}}^\top (o - y)) \odot 1\{z \geq 0\} \in \mathbb{R}^{m \times 1}$
        (by eqn. (3.12) and (3.13))
Compute
    $\frac{\partial J}{\partial W^{[2]}} = \delta^{[2]} a^\top \in \mathbb{R}^{1 \times m}$    (by eqn. (3.5))
    $\frac{\partial J}{\partial b^{[2]}} = \delta^{[2]} \in \mathbb{R}$    (by eqn. (3.6))
    $\frac{\partial J}{\partial W^{[1]}} = \delta^{[1]} x^\top \in \mathbb{R}^{m \times d}$    (by eqn. (3.7))
    $\frac{\partial J}{\partial b^{[1]}} = \delta^{[1]} \in \mathbb{R}^m$    (as an exercise)
```

$$\begin{aligned} a^{[r]} &= z^{[r]} = W^{[r]}a^{[r-1]} + b^{[r]} \\ J &= \frac{1}{2}(a^{[r]} - y)^2 \end{aligned}$$

Here we define both $a^{[r]}$ and $z^{[r]}$ as $h_\theta(x)$ for notational simplicity.

First, we note that we have the following local abstraction for $k \in \{1, \dots, r\}$:

$$\begin{aligned} z^{[k]} &= W^{[k]}a^{[k-1]} + b^{[k]} \\ J &= J(z^{[k]}) \end{aligned}$$

Invoking Claim 3.2, we have that

$$\begin{aligned} \frac{\partial J}{\partial W^{[k]}} &= \frac{\partial J}{\partial z^{[k]}} \cdot a^{[k-1]^\top} \\ \frac{\partial J}{\partial b^{[k]}} &= \frac{\partial J}{\partial z^{[k]}} \end{aligned} \tag{3.15}$$

Therefore, it suffices to compute $\frac{\partial J}{\partial z^{[k]}}$. For simplicity, let's define $\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}}$. We compute $\delta^{[k]}$ from $k = r$ to 1 inductively. First we have that

$$\delta^{[r]} \triangleq \frac{\partial J}{\partial z^{[r]}} = (z^{[r]} - y) \tag{3.16}$$

Next for $k \leq r - 1$, suppose we have computed the value of $\delta^{[k+1]}$, then we will compute $\delta^{[k]}$. First, using Claim 3.3, we have that

$$\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}} = \frac{\partial J}{\partial a^{[k]}} \odot \mathrm{ReLU}'(z^{[k]})$$

Then we note that the relationship between $a^{[k]}$ and $z^{[k+1]}$ can be abstractly written as

$$z^{[k+1]} = W^{[k+1]}a^{[k]} + b^{[k+1]} \tag{3.17}$$

$$J = J(z^{[k+1]}) \tag{3.18}$$

Therefore by Claim 3.4 we have that

$$\frac{\partial J}{\partial a^{[k]}} = W^{[k+1]^\top} \frac{\partial J}{\partial z^{[k+1]}} \tag{3.19}$$

It follows that

$$\begin{aligned} \delta^{[k]} &= \left(W^{[k+1]^\top} \frac{\partial J}{\partial z^{[k+1]}} \right) \odot \mathrm{ReLU}'(z^{[k]}) \\ &= \left(W^{[k+1]^\top} \delta^{[k+1]} \right) \odot \mathrm{ReLU}'(z^{[k]}) \end{aligned}$$

Algorithm 4 Back-propagation for multi-layer neural networks.

.

```
Compute and store the values of $a^{[k]}$'s and $z^{[k]}$'s for $k = 1, \ldots, r$, and $J$.
.                                        ▷ This is often called the "forward pass"
.
for $k = r$ to 1 do                      ▷ This is often called the "backward pass"
    if $k = r$ then
        compute $\delta^{[r]} \triangleq \frac{\partial J}{\partial z^{[r]}}$
    else
        compute
            $$\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}} = \left(W^{[k+1]^\top} \delta^{[k+1]}\right) \odot \mathrm{ReLU}'(z^{[k]})$$
    Compute
            $$\frac{\partial J}{\partial W^{[k]}} = \delta^{[k]} a^{[k-1]^\top}$$
            $$\frac{\partial J}{\partial b^{[k]}} = \delta^{[k]}$$
```

4 Vectorization Over Training Examples

As we discussed in Section 1, in the implementation of neural networks, we will leverage the parallelism across 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:

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{d\times 3} \tag{4.1}$$

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

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]} \tag{4.2}$$

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 $\tilde{b}^{[1]} \in \mathbb{R}^{4\times 3}$:

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \tag{4.3}$$

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 (4.2), 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.[7] 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

$$Z^{[1]} = XW^{[1]} + b^{[1]} \in \mathbb{R}^{3\times m} \tag{4.4}$$

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