Section 07: Solutions

1. The Chain Rule

(a) Let $f\colon \mathbb{R}^n \to \mathbb{R}^m$, $g\colon \mathbb{R}^\ell \to \mathbb{R}^n$. Write the Jacobian of $f \circ g$ as a matrix in terms of the Jacobian matrix $\frac{\partial f}{\partial y}$ of f and the Jacobian matrix $\frac{\partial g}{\partial x}$ of g. Make sure the matrix dimensions line up. What conditions must hold in order for this formula to make sense?

Solution:

The Chain Rule theorem states that:

$$\frac{\partial (f \circ g)}{\partial x}(x) = \frac{\partial f}{\partial y}\big(g(x)\big) \cdot \frac{\partial g}{\partial x}(x)$$

In order for the dimensions to line up for matrix multiplication, we must have $\frac{\partial f}{\partial y} \in \mathbb{R}^{m \times n}$ and $\frac{\partial g}{\partial x} \in \mathbb{R}^{n \times \ell}$, since $f \circ g\colon \mathbb{R}^\ell \to \mathbb{R}^m$. Note that by this convention, the gradient of a scalar function is:

$$\frac{\partial f}{\partial y}(y) = \begin{bmatrix} \frac{\partial f_1}{\partial y_1}(y) & \cdots & \frac{\partial f_1}{\partial y_n}(y) \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial y_1}(y) & \cdots & \frac{\partial f_m}{\partial y_n}(y) \end{bmatrix}.$$

In order to apply the chain rule, f must be differentiable at $g(x)$ and g must be differentiable at x.

(b) What if instead the input of g is a matrix $W \in \mathbb{R}^{p \times q}$? Can we still represent the derivative $\frac{\partial g}{\partial W}$ of g as a matrix?

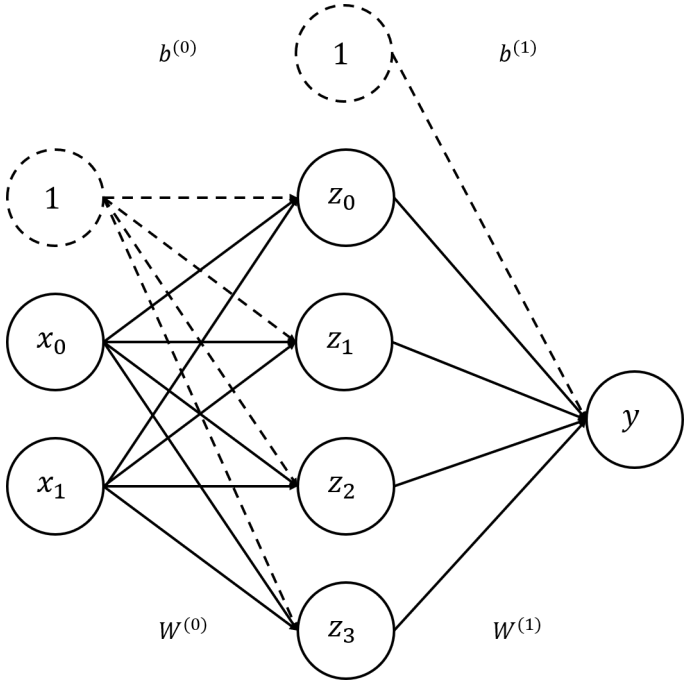
Solution:

No, we cannot. The derivative of $g\colon \mathbb{R}^{p \times q} \to \mathbb{R}^n$ would be represented as a three-dimensional $n \times p \times q$ tensor. In practice, people often *flatten* the input matrix W to a vector $\text{vec}(W) \in \mathbb{R}^{pq}$. Then we can write the derivative of g as a Jacobian matrix, $\frac{\partial g}{\partial \text{vec}(W)} \in \mathbb{R}^{n \times pq}$. Then we must remember to un-flatten the derivative later when we update the matrix W.

2. 1-Hidden-Layer Neural Network Gradients and Initialization

2.1. Forward and Backward pass

Consider a 1-hidden-layer neural network with a single output unit. Formally the network can be defined by the parameters $W^{(0)} \in \mathbb{R}^{h \times d}$, $b^{(0)} \in \mathbb{R}^h$; $W^{(1)} \in \mathbb{R}^{1 \times h}$ and $b^{(1)} \in \mathbb{R}$. The input is given by $x \in \mathbb{R}^d$. We will use sigmoid activation for the first hidden layer z and no activation for the output y. Below is a visualization of such a neural network with $d = 2$ and $h = 4$.



(a) Write out the forward pass for the network using $x, W^{(0)}, b^{(0)}, z, W^{(1)}, b^{(1)}, \sigma$ and y.

Hint: Write $z = \ldots$ and $y = \ldots$

Solution:

$$z = \sigma\left(W^{(0)}x + b^{(0)}\right)$$
$$y = W^{(1)}z + b^{(1)}$$

(b) Find the partial derivatives of the output with respect $W^{(1)}$ and $b^{(1)}$, namely $\frac{\partial y}{\partial W^{(1)}}$ and $\frac{\partial y}{\partial b^{(1)}}$.

Solution:

$$\frac{\partial y}{\partial W^{(1)}} = z$$
$$\frac{\partial y}{\partial b^{(1)}} = 1$$

(c) Now find the partial derivative of the output with respect to the output of the hidden layer z, that is $\frac{\partial y}{\partial z}$

Solution:

$$\frac{\partial y}{\partial z} = W^{(1)}$$

(d) Finally find the partial derivatives of the output with respect to $W^{(0)}$ and $b^{(0)}$, that is $\frac{\partial y}{\partial W^{(0)}}$ and $\frac{\partial y}{\partial b^{(0)}}$.

Hint: First find $\frac{\partial z_i}{\partial W_i^{(0)}}$ and $\frac{\partial z_i}{\partial b_i^{(0)}}$, where $W_i^{(0)}$ denotes the i-th row of $W^{(0)}$. Then note that $\frac{\partial y}{\partial W_i^{(0)}} = \sum_{j=1}^{h} \frac{\partial y}{\partial z_j}\frac{\partial z_j}{\partial W_i^{(0)}} = \frac{\partial y}{\partial z_i}\frac{\partial z_i}{\partial W_i^{(0)}}$ and $\frac{\partial y}{\partial b_i^{(0)}} = \sum_{j=1}^{h} \frac{\partial y}{\partial z_j}\frac{\partial z_j}{\partial b_i^{(0)}} = \frac{\partial y}{\partial z_i}\frac{\partial z_i}{\partial b_i^{(0)}}$ using the chain rule for multi-variate functions.

Solution:

$$\frac{\partial z_i}{\partial W_i^{(0)}} = z_i(1-z_i)x^\top \in \mathbb{R}^d$$
$$\frac{\partial y}{\partial W_i^{(0)}} = \frac{\partial y}{\partial z_i}\frac{\partial z_i}{\partial W_i^{(0)}} = W_i^{(1)} \cdot z_i(1-z_i)x^\top \in \mathbb{R}^d$$
$$\frac{\partial y}{\partial W^{(0)}} = \left[W^{(1)} \circ z \circ (1-z)\right] x^\top \in \mathbb{R}^{h \times d},$$

$$\frac{\partial z_i}{\partial b_i^{(0)}} = z_i(1-z_i) \in \mathbb{R}$$
$$\frac{\partial y}{\partial b_i^{(0)}} = \frac{\partial y}{\partial z_i}\frac{\partial z_i}{\partial b_i^{(0)}} = W_i^{(1)} \cdot z_i(1-z_i) \in \mathbb{R}$$
$$\frac{\partial y}{\partial b^{(0)}} = W^{(1)} \circ z \circ (1-z) \in \mathbb{R}^h.$$

We have provided the shapes of the matrix representations of derivatives. Try to reason about why it is of the given shape.

2.2. Weight initialization

Suppose we initialize all weights and biases in the network to 0 before performing gradient descent.

(a) For all $x \in \mathbb{R}^d$, find z and y after the forward pass.

Solution:

$$z_i = \sigma(W_i^{(0)}x + b^{(0)_i}) = \sigma(\mathbf{0}x + 0) = \sigma(0) = \frac{1}{2}$$
$$y = W^{(1)}z + b^{(1)} = \mathbf{0} \cdot \frac{\mathbf{1}}{\mathbf{2}} + 0 = 0$$

(b) Now find the values of the gradients $\frac{\partial y}{\partial W^{(1)}}$, $\frac{\partial y}{\partial b^{(1)}}$, $\frac{\partial y}{\partial W^{(0)}}$ and $\frac{\partial y}{\partial b^{(0)}}$. Note that some of the gradients will be in terms of x.

Solution:

$$\frac{\partial y}{\partial W^{(1)}} = z = \frac{\mathbf{1}}{\mathbf{2}}$$
$$\frac{\partial y}{\partial b^{(1)}} = 1$$
$$\frac{\partial y}{\partial W^{(0)}} = \left[W^{(1)} \circ z \circ (1-z)\right] x^\top$$
$$= (\mathbf{0} \circ \frac{\mathbf{1}}{\mathbf{2}} \circ \frac{\mathbf{1}}{\mathbf{2}})x^\top = \mathbf{0}$$
$$\frac{\partial y}{\partial b^{(0)}} = W^{(1)} \circ z \circ (1-z)$$
$$= \mathbf{0} \circ \frac{\mathbf{1}}{\mathbf{2}} \circ \frac{\mathbf{1}}{\mathbf{2}} = \mathbf{0}\,.$$

(c) Observe the values of each z_i and observe each $\frac{\partial y}{\partial W_i^{(1)}}$ and $\frac{\partial y}{\partial b_i^{(1)}}$. What do you notice? And what does this imply for the expressiveness of the network? (Note that there is nothing special about the value 0 here, it just

simplifies the calculations. The same can be shown for initialization with any constant c)

Solution:

The key insight is that if we initialize the weights to all have the same value, all z_i are the same. Similarly all $W_i^{(l)}$ and $b_i^{(l)}$ are the same too and so the output y could be expressed with just a single z_i instead of h. Thus the neural network boils down to just having a single hidden unit. The same holds for the gradients, so during a step of gradient descent, $W_i^{(l)}$ and $b_i^{(l)}$ are updated in the same way. Thus after a step of gradient descent, all $W_i^{(l)}$ and $b_i^{(l)}$ are still the same. By induction, the same holds after an arbitrary number of steps of gradient descent.

3. (Optional) Shapes in Convolutional Neural Networks

When designing a convolutional neural network, it's important to think about the shape of the data flowing through the network. In this problem you will get gain experience with thinking about the shapes in a neural network and a better intuition for why convolutional neural networks require so few parameters compared to fully connected layers.

Shape of a convolutional layer / maxpooling output: For a $n \times n$ input, $f \times f$ filter, padding p and stride s, the output size is $o \times o$ where:

$$o = \frac{n - f + 2p}{s} + 1$$

We will use Pytorch Conv2d to represent a 2D convolution, and Pytorch MaxPool2d to represent a 2D max pooling. Take a look at the documentation on Conv2d and MaxPool2d.

(a) Assume your input is a batch of N 64×64 RGB images. The input tensor your neural network receives will have shape $(N, 3, 64, 64)$. For each of the following operations, determine the new shape of the tensor as it flows through the network. Note that activations are omitted since they don't change the shape of the data as they act coordinate-wise.

1. `Conv2D(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)`

   Solution:

   $(N, 16, 64, 64)$

2. `MaxPool2d(kernel_size=2, stride=2, padding=0)`

   Solution:

   $(N, 16, 32, 32)$

3. `Conv2D(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=0)`

   Solution:

   $(N, 32, 30, 30)$

4. `MaxPool2d(kernel_size=2, stride=2, padding=1)`

Solution:

$$(N, 32, 16, 16)$$

5. `Conv2D(in_channels=32, out_channels=8, kernel_size=1, stride=1, padding=0)`

Solution:

$$(N, 8, 16, 16)$$

6. `Conv2D(in_channels=8, out_channels=4, kernel_size=5, stride=1, padding=0)`

Solution:

$$(N, 4, 12, 12)$$

7. `Flatten`

Solution:

$$(N, 576)$$

8. `Linear(in_features=576, out_features=10)`

Solution:

$$(N, 10)$$

(b) Again assume your input is a batch of N 64×64 RGB images. Now compute the number of parameters that each layers has. For the convolutional layers, also compute the number of parameters a fully connected layer mapping from the flattened input channels to the flattened output channels would have. It is okay to leave the number of parameters as products and additions such as $64 \cdot 32 + 16$.

1. `Conv2D(in_channels=3, out_channels=16, kernel_size=3, stride=1, padding=1)`

Solution:

Conv: $3 \cdot 16 \cdot 3 \cdot 3 + 16 = 448$

Fully connected: $3 \cdot 64 \cdot 64 \cdot 16 \cdot 64 \cdot 64 + 16 \cdot 64 \cdot 64 = 805306512$

2. `MaxPool2d(kernel_size=2, stride=2, padding=0)`

Solution:

$$0$$

3. `Conv2D(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=0)`

Solution:

Conv: $16 \cdot 32 \cdot 3 \cdot 3 + 32 = 4640$

Fully connected: $16 \cdot 32 \cdot 32 \cdot 32 \cdot 30 \cdot 30 + 32 \cdot 30 \cdot 30 = 471888000$

4. `MaxPool2d(kernel_size=2, stride=2, padding=1)`

Solution:

0

5. `Conv2D(in_channels=32, out_channels=8, kernel_size=1, stride=1, padding=0)`

Solution:

Conv: $32 \cdot 8 + 8 = 264$

Fully connected: $32 \cdot 16 \cdot 16 \cdot 8 \cdot 16 \cdot 16 + 8 \cdot 16 \cdot 16 = 16779264$

6. `Conv2D(in_channels=8, out_channels=4, kernel_size=5, stride=1, padding=0)`

Solution:

Conv: $8 \cdot 4 \cdot 5 \cdot 5 + 4 = 804$

Fully connected: $8 \cdot 16 \cdot 16 \cdot 4 \cdot 12 \cdot 12 + 4 \cdot 12 \cdot 12 = 1180224$

7. `Flatten`

Solution:

0

8. `Linear(in_features=576, out_features=10)`

Solution:

$576 \cdot 10 + 10 = 5770$

4. Additional Resources

Many people have written tutorials about backpropagation. So if you want to review the concepts from a different perspective, check out the following resources:

(a) We particularly recommend slides 24-36 of Joseph Redmon's Deep Learning slides 03

(b) Joseph Redmon's Deep Learning slides 02

(c) Section 8 notes from 20wi

(d) colah's blog: Calculus on Computational Graphs: Backpropagation

This resource from CS231n goes over the high-level intuition for transfer learning and practical tips for implemention: https://cs231n.github.io/transfer-learning/.

Here are resources to review CNNs from Joseph Redmon's Deep Learning class:

(a) Convolutional neural networks lecture from 20au with slides

(b) Image classification lecture from 20au with overview of AlexNet in slides 28 - 30