CS 182/282A Spring 2022 — Designing, Visualizing and Understanding Deep Neural Networks — Marvin Zhang

Discussion 3

This discussion will cover some background on autodiff and practice applying backpropagation.

1 Automatic Differentiation

In this section we will cover some background on different types of differentiation and motivate why we use backward autodiff (instead of forward).

In training neural networks, we are trying to find the model weights $\boldsymbol{\theta}$ that minimize a loss function $\mathcal{L}(\boldsymbol{\theta})$. To do this, recall that we typically use (stochastic) gradient descent as follows

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^{t} - \alpha \boldsymbol{\nabla} L(\boldsymbol{\theta}^{t}).$$

which means it is important to be able to efficiently compute derivatives, especially for large and complex models. Automatic differentiation (autodiff) is a method for computing the derivative of a program-specified function.

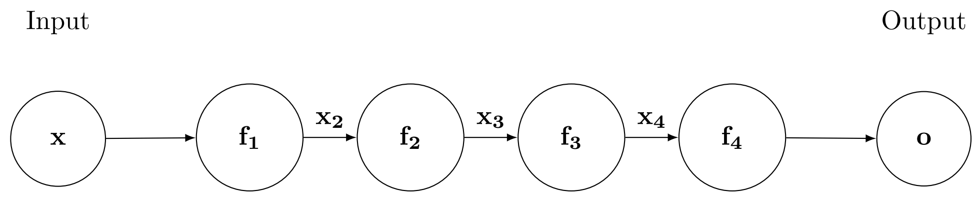
Problem 1: Methods for Differentiation

There are three main methods for differentiation: symbolic differentiation, numerical differentiation, and automatic differentiation. How does each of them work, and what are their pros/cons?

Solution 1: Methods for Differentiation

- *Symbolic Differentiation*: convert the computer program to a single mathematical expression so we can compute exact gradients. While accurate and 'human-readable', this can be difficult or impossible to implement for complex programs.
- *Numerical Differentiation*: compute gradients using finite differences, where we approximate $\frac{\partial f}{\partial x} \approx \frac{f(x+\Delta x)-f(x)}{\Delta x}$ for sufficiently small Δx. While easy to implement, the discretization process and approximation can introduce compounding errors.
- *Automatic Differentiation*: convert the program into a series of primitive operations that we know how to differentiate. Here we get exact derivatives for compositions of defined primitive operations, avoid repetitive calculations of the same values, and the complexity of a derivative-pass through the program is the same as the original program.

Our neural networks are composed of a series of nested functions. Consider the feedforward network below



Here, we have an input $\mathbf{x} \in \mathbb{R}^n$, an output $\mathbf{o} \in \mathbb{R}^m$, and let the Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ be an $m \times n$ dimensional matrix. Note that in the previous discussion, we parametrized the Jacobian with entries $\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{ij} = \frac{\partial f_j}{\partial x_i}$, also known as the denominator layout. In this section, we use the numerator layout instead, which is $\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{ij} = \frac{\partial f_i}{\partial x_j}$.

Explicitly writing out the nested functions, we have

$$\begin{aligned}\mathbf{o} &= \mathbf{f}(\mathbf{x}) \\ &= \mathbf{f_4}(\mathbf{f_3}(\mathbf{f_2}(\mathbf{f_1}(\mathbf{x}))))\end{aligned}$$

Let the dimensionality of the intermediate variables be $\mathbf{x_2} \in \mathbb{R}^{m_1}, \mathbf{x_3} \in \mathbb{R}^{m_2}, \mathbf{x_4} \in \mathbb{R}^{m_3}$. Then by the chain rule, we can write

$$\begin{aligned}\frac{\partial \mathbf{o}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{o}}{\partial \mathbf{x_4}}\frac{\partial \mathbf{x_4}}{\partial \mathbf{x_3}}\frac{\partial \mathbf{x_3}}{\partial \mathbf{x_2}}\frac{\partial \mathbf{x_2}}{\partial \mathbf{x}} \\ &= \frac{\partial \mathbf{f_4}(\mathbf{x_4})}{\partial \mathbf{x_4}}\frac{\partial \mathbf{f_3}(\mathbf{x_3})}{\partial \mathbf{x_3}}\frac{\partial \mathbf{f_2}(\mathbf{x_2})}{\partial \mathbf{x_2}}\frac{\partial \mathbf{f_1}(\mathbf{x})}{\partial \mathbf{x}} \\ &= \underbrace{J_{\mathbf{f_4}}(\mathbf{x_4})}_{m \times m_3}\underbrace{J_{\mathbf{f_3}}(\mathbf{x_3})}_{m_3 \times m_2}\underbrace{J_{\mathbf{f_2}}(\mathbf{x_2})}_{m_2 \times m_1}\underbrace{J_{\mathbf{f_1}}(\mathbf{x})}_{m_1 \times n} \\ &= J_{\mathbf{f}}(\mathbf{x})\end{aligned}$$

Recall that $\frac{\partial \mathbf{f_i}}{\partial x_j}$ is the ith row and jth column of $J_{\mathbf{f}}(\mathbf{x})$. To build the Jacobian, we can use the **Jacobian-vector product** (JVP) or **vector-Jacobian product** (VJP) to build up the full Jacobian up column-wise or row-wise, respectively.

Forward Differentiation. The JVP is the right-multiplication of the Jacobian with a vector $\mathbf{v} \in \mathbb{R}^n$. To find $\frac{\partial \mathbf{f}}{\partial x_j}$, we take the JVP with $\mathbf{e}_j \in \mathbb{R}^n$, where $\mathbf{e}_i$ is a column vector that is 1 at index i and all zeros otherwise.

$$\frac{\partial \mathbf{f}}{\partial x_1} = J_{\mathbf{f}}(\mathbf{x})\mathbf{e_1}, \frac{\partial \mathbf{f}}{\partial x_2} = J_{\mathbf{f}}(\mathbf{x})\mathbf{e_2}, \cdots, \frac{\partial \mathbf{f}}{\partial x_n} = J_{\mathbf{f}}(\mathbf{x})\mathbf{e_n}$$

Thus computing a gradient of $\mathbf{o}$ with respect to $\mathbf{x}$ requires n JVPs with $\mathbf{e_1}, \cdots, \mathbf{e_n}$. In total, this requires $O(n(mm_3 + m_3m_2 + m_2m_1 + m_1n))$.

Backward Differentiation. The VJP is the left-multiplication of a vector $\mathbf{u} \in \mathbb{R}^m$, and the Jacobian. To find $\nabla \mathbf{f}_i(\mathbf{x})$, we take the VJP with $\mathbf{e}_j \in \mathbb{R}^m$.

$$\nabla \mathbf{f_1}(\mathbf{x}) = \mathbf{e}_1^\top J_{\mathbf{f}}(\mathbf{x}), \nabla \mathbf{f_2}(\mathbf{x}) = \mathbf{e}_2^\top J_{\mathbf{f}}(\mathbf{x}), \cdots, \nabla \mathbf{f_m}(\mathbf{x}) = \mathbf{e}_m^\top J_{\mathbf{f}}(\mathbf{x})$$

Thus computing a gradient of $\mathbf{o}$ with respect to $\mathbf{x}$ requires m VJPs with $\mathbf{e_1}, \cdots, \mathbf{e_m}$. In total, this requires $O(m(mm_3 + m_3m_2 + m_2m_1 + m_1n))$. In practice, what this may look like is

Algorithm 1 Backward Autodiff

```
Require: $\mathbf{x} \in \mathbb{R}^n$
  $\mathbf{x_1} \leftarrow \mathbf{x}$, $\mathbf{u}_i \leftarrow \mathbf{e}_i \in \mathbb{R}^m, i \in \{1, ..., m\}$
  for $k = 1...K$ do
    $\mathbf{x}_{k+1} \leftarrow \mathbf{f}_k(\mathbf{x}_k)$        ▷ Forward pass to get the outputs at each layer (which we store)
  end for
  for $k = K...1$ do
    $\mathbf{u}_i^\top \leftarrow \mathbf{u}_i^\top J_{\mathbf{f}_k}(\mathbf{x}_k), i \in \{1, ..., m\}$        ▷ Backward pass to build the Jacobian
  end for
     return $\mathbf{o} = \mathbf{x}_{K+1}, J_{\mathbf{f}}(\mathbf{x})_i = \mathbf{u}_i^\top$
```

Problem 2: Motivation for Backwards Autodiff

Looking at the computational costs above, why do we use backward autodiff in machine learning?

Solution 2: Motivation for Backwards Autodiff

Note that backward differentiation with VJPs is more efficient when $m \leq n$. In machine learning, typically we want to minimize the loss function, which is a scalar. This corresponds to the $m = 1$ setting where backward differentiation is more efficient. However, note that in backward differentiation we need to keep track of all the intermediate computations, as the VJP is a multiplication from left to right (i.e. multiply by $J_{\mathbf{f_n}}(\mathbf{x_n})$ first, then $J_{\mathbf{f_{n-1}}}(\mathbf{x_{n-1}})$, etc.).

2 Mechanical Backpropagation

In this section, we will work through some calculations used during backpropagation.

Recall the softmax function $\mathbf{p} : \mathbb{R}^k \to \mathbb{R}^k$, with entries given by

$$p_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}.$$

Each entry p_i corresponds to the probability assigned to the label i. We derived in Discussion 1 that the partial derivatives of $p_i(\mathbf{z})$ for each entry of $\mathbf{z}$ is given by,

$$\begin{aligned}\frac{\partial p_i(\mathbf{z})}{\partial z_j} &= \begin{cases} p_i(\mathbf{z})(1 - p_j(\mathbf{z})) & \text{if } i = j \\ -p_i(\mathbf{z})p_j(\mathbf{z}) & \text{if } i \neq j \end{cases} \\ &= p_i(\mathbf{z})(\delta_{ij} - p_j(\mathbf{z})).\end{aligned}$$

We can then concisely write the full gradient with respect to $\mathbf{z}$ as

$$\nabla p_i(\mathbf{z}) = p_i(\mathbf{z})(\mathbf{e}_i - \mathbf{p}(\mathbf{z})),$$

where $\mathbf{e}_i$ is the unit vector with 1 at index i and 0 elsewhere.

In this example, we will maximize the log-likelihood of the given labels in our dataset, which motivates the following loss for a multiclass logistic regression model.

$$L(\mathbf{x}, y, W, \mathbf{b}) = -\log p_y(W\mathbf{x} + \mathbf{b}).$$

Problem 3: Gradient with respect to linear layer parameters

Utilize the chain rule to compute the gradient of $L(\mathbf{x}, y, W, \mathbf{b})$ with respect to W and $\mathbf{b}$.

Solution 3: Gradient with respect to linear layer parameters

Let $\mathbf{z} = W\mathbf{x} + \mathbf{b}$. Given we already know $\boldsymbol{\nabla} p_i(\mathbf{z})$, we first compute $\boldsymbol{\nabla} \log p_i(\mathbf{z})$ as

$$\begin{aligned}\boldsymbol{\nabla} \log p_i(\mathbf{z}) &= \frac{\boldsymbol{\nabla} p_i(\mathbf{z})}{p_i(\mathbf{z})} \\ &= \mathbf{e}_i - \mathbf{p}(\mathbf{z}).\end{aligned}$$

We note that since our loss is the *negative* log likelihood, we will need to flip the sign of our gradient. We first consider the gradient of the loss with respect to the bias $\mathbf{b}$. Notice that,

$$\frac{\partial z_i}{\partial b_j} = \delta_{ij},$$

and so the Jacobian $\frac{\partial \mathbf{z}}{\partial \mathbf{b}}$ is simply the identity matrix. Utilizing the chain rule to compute the gradient with respect to the bias, we have

$$\begin{aligned}\boldsymbol{\nabla}_{\mathbf{b}} L &= -\frac{\partial \mathbf{z}}{\partial \mathbf{b}} \boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \\ &= -I \boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \\ &= -\boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \\ &= \mathbf{p}(\mathbf{z}) - \mathbf{e}_y.\end{aligned}$$

Now, we consider the partial derivatives of $\mathbf{z}$ with respect to W.

$$\frac{\partial z_k}{\partial W_{ij}} = \delta_{ik} x_j.$$

Noting that the derivative with respect to W_{ij} depends only on the i'th entry of $\mathbf{z}$, we compute

$$\begin{aligned}\frac{\partial L}{\partial W_{ij}} &= \frac{\partial L}{\partial z_i} x_j \\ \frac{\partial L}{\partial W} &= -\boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z}) \mathbf{x}^T \\ &= (\mathbf{p}(\mathbf{z}) - \mathbf{e}_y)\mathbf{x}^T.\end{aligned}$$

Suppose now that we had a multilayer neural network and $W, \mathbf{b}$ were the the parameters of the last layer of the network. To compute gradients of the earlier parameters of the network with backpropagation, we also need to compute the gradient of the loss with respect to $\mathbf{x}$ and pass it backwards.

Problem 4: Gradient with respect to input

Utilize the chain rule to compute the gradient of $L(\mathbf{x}, y, W, \mathbf{b})$ with respect to $\mathbf{x}$.

Solution 4: Gradient with respect to input

We can again compute

$$\begin{aligned}\frac{\partial L}{\partial x_j} &= \sum_{i=1}^{k} \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial x_j} \\ &= \sum_{i=1}^{k} \frac{\partial L}{\partial z_i} W_{ij}.\end{aligned}$$

Vectorizing, we obtain

$$\frac{\partial L}{\partial \mathbf{x}} = -W^T \boldsymbol{\nabla}_{\mathbf{z}} \log p_y(\mathbf{z})$$

Having computed these, one then simply needs to also compute the backwards pass through the chosen activation function to able backpropagate through fully-connected feedforward networks!

We'll now move on to a slightly more complicated example of backpropagation involving a *skip connection*, which you'll see again when we cover ResNets.

Problem 5: Gradient in a nonlinear computation graph

Suppose we have $\mathbf{y} = W_2\sigma(W_1\mathbf{x}) + \mathbf{x}$, where σ is the ReLU activation. Letting $\delta_{\mathbf{y}}$ denote the gradient of the loss with respect to $\mathbf{y}$, compute the gradient of the loss with respect to $\mathbf{x}$.

Solution 5: Gradient in a nonlinear computation graph

Note here that $\mathbf{x}$ now contributes to $\mathbf{y}$ both through the $W_2\sigma(W_1\mathbf{x})$ (the usual feedforward pass) and the $\mathbf{x}$ term (the skip connection).
We'll first go through the contribution to the gradient from the feedforward pass. Let $\mathbf{z} = W_1\mathbf{x}$ and $\mathbf{a} = \sigma(W_1\mathbf{x})$. From our earlier calculations, we see that

$$\frac{\partial L}{\partial \mathbf{a}} = W_2^T \delta_{\mathbf{y}}.$$

We now need to backpropagate through the ReLU activation. Let $R(\mathbf{z})$ denote the diagonal matrix such that $\mathbb{R}(\mathbf{z})_{ii} = 1$ if $\mathbf{z}_i > 0$ and 0 otherwise. We see that the backward pass through the ReLU activation simply multiplies the post-activation gradient by $R(\mathbf{z})$, and so we have

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}} &= R(\mathbf{z})\frac{\partial L}{\partial \mathbf{a}} \\ &= R(\mathbf{z})W_2^T\delta_{\mathbf{y}}.\end{aligned}$$

Finally, we can compute the feedforward pass's contribution to $\frac{\partial L}{\partial \mathbf{x}}$ as

$$W_1^T\frac{\partial L}{\partial \mathbf{z}} = W_1^T R(W_1\mathbf{x})W_2^T\delta_{\mathbf{y}}.$$

To handle the two different paths through to the output, we simply need to sum each gradient contribution, so our final gradient is

$$\frac{\partial L}{\partial \mathbf{x}} = W_1^T R(W_1\mathbf{x})W_2^T\delta_{\mathbf{y}} + \delta_{\mathbf{y}}.$$