Topics:
  • Jacobians
  • Optimization

CS 4803-DL / 7643-A
ZSOLT KIRA
• **Assignment Due Feb 5th**

• **Resources:**
  - These lectures
  - [Matrix calculus for deep learning](#)
  - [Gradients notes](#) and [MLP/ReLU Jacobian notes](#).
  - Assignment (@41) and matrix calculus (@46)

• **Project:** Teaming thread on piazza

• **Schedule:**

<table>
<thead>
<tr>
<th>Graded Deliverable</th>
<th>Release Date</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of Term</td>
<td>10-Jan</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>14-Jan</td>
<td>5-Feb</td>
</tr>
<tr>
<td>A2</td>
<td>30-Jan</td>
<td>20-Feb</td>
</tr>
<tr>
<td>A3</td>
<td>20-Feb</td>
<td>6-Mar</td>
</tr>
<tr>
<td>Project Proposal</td>
<td>5-Feb</td>
<td>13-Mar</td>
</tr>
<tr>
<td>A4</td>
<td>9-Mar</td>
<td>3-Apr</td>
</tr>
<tr>
<td>Project Report</td>
<td>5-Apr</td>
<td>1-May</td>
</tr>
</tbody>
</table>
To develop a general algorithm for this, we will view the function as a **computation graph**

Graph can be any **directed acyclic graph (DAG)**

- Modules must be differentiable to support gradient computations for gradient descent

A **training algorithm** will then process this graph, **one module at a time**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: Forward Pass

Layer 1 → Layer 2 → Layer 3

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Note that we must store the **intermediate outputs of all layers**!

- This is because we will need them to **compute the gradients** (the gradient equations will have terms with the output values in them)

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Layer 1 → Layer 2 → **→**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Layer 1 → [Green Arrow] → Layer 3
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**
Step 1: Compute Loss on Mini-Batch: **Forward Pass**
Step 2: Compute Gradients wrt parameters: **Backward Pass**
Step 3: Use gradient to update all parameters at the end

Backpropagation is the application of gradient descent to a computation graph via the chain rule!

\[ w_i = w_i - \alpha \frac{\partial L}{\partial w_i} \]
We want to compute: $\left\{ \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \right\}$

We will use the chain rule to do this:

Chain Rule: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$
We will use the chain rule to compute: \( \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \)

- Gradient of loss w.r.t. inputs: \( \frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial h^{\ell-1}} \)
- Gradient of loss w.r.t. weights: \( \frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial W} \)

Given by upstream module (upstream gradient)

Calculated Analytically

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Computing the Gradients of Loss
Conventions:

- Size of derivatives for scalars, vectors, and matrices:
  Assume we have scalar $s \in \mathbb{R}^1$, vector $v \in \mathbb{R}^m$, i.e. $v = [v_1, v_2, \ldots, v_m]^T$ and matrix $M \in \mathbb{R}^{k \times \ell}$

<table>
<thead>
<tr>
<th></th>
<th>$S$</th>
<th>$V$</th>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\frac{\partial s_1}{\partial s_2}$</td>
<td>$\frac{\partial s}{\partial v}$</td>
<td>$\frac{\partial s}{\partial M}$</td>
</tr>
<tr>
<td>$V$</td>
<td>$\frac{\partial v}{\partial s}$</td>
<td>$\frac{\partial v_1}{\partial v_2}$</td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>$\frac{\partial M}{\partial s}$</td>
<td></td>
<td>Tensors</td>
</tr>
</tbody>
</table>
\[ x \in \mathbb{R} \rightarrow z \in \mathbb{R} \rightarrow y \in \mathbb{R} \]

\[ y = g_2(g_1(x)) \]

\[ \frac{dy}{dx} = \frac{dy}{dt} \cdot \frac{dt}{dz} \cdot \frac{dz}{dx} \]

Scalar mult
Vector Case

\[\mathbf{x} \in \mathbb{R}^d \rightarrow \mathbf{R}^m \rightarrow \mathbf{y} \in \mathbb{R}^c\]

\[g_1(\mathbf{x}) \rightarrow g_1(\mathbf{y}) \quad \text{and} \quad g_2(\mathbf{x}) \rightarrow g_2(\mathbf{y})\]

\[
\begin{bmatrix}
\frac{\partial y_1}{\partial x}
\frac{\partial y_2}{\partial x}
\end{bmatrix} =
\begin{bmatrix}
\frac{\partial y_1}{\partial x_1}
\frac{\partial y_2}{\partial x_2}
\end{bmatrix}
\begin{bmatrix}
\frac{\partial y_1}{\partial x_1}
\frac{\partial y_2}{\partial x_2}
\end{bmatrix}
\]

\[J_{g_2 \circ g_1} = J_{g_2} \cdot J_{g_1}\]
Jacobian View of Chain Rule
Graphical View of Chain Rule
Chain Rule: Cascaded
Input: \( x \in \mathbb{R}^D \)

Binary label: \( y \in \{-1, +1\} \)

Parameters: \( w \in \mathbb{R}^D \)

Output prediction: \( p(y = 1|x) = \frac{1}{1 + e^{-w^T x}} \)

Loss: \( L = \frac{1}{2} ||w||^2 - \lambda \log(p(y|x)) \)

Log Loss

Adapted from slide by Marc'Aurelio Ranzato
We have discussed **computation graphs for generic functions**

Machine Learning functions *(input \(\to\) model \(\to\) loss function)* is also a computation graph

We can use the **computed gradients from backprop/automatic differentiation** to update the weights!

\[
\begin{align*}
\mathbf{w}^T \mathbf{x} & \rightarrow u = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} \\
& \rightarrow p = -\log(p) \\
& \rightarrow L
\end{align*}
\]
Automatic differentiation:
- Carries out this procedure for us on arbitrary graphs
- Knows derivatives of primitive functions
- As a result, we just define these (forward) functions and don’t even need to specify the gradient (backward) functions!

Example Gradient Computations

\[ L = 1 \]
\[ \bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p} \]

where \( p = \sigma(w^T x) \) and \( \sigma(x) = \frac{1}{1+e^{-x}} \)

\[ \bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \bar{p} \sigma(1 - \sigma) \]

\[ \bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \frac{\partial u}{\partial w} = \bar{u} x^T \]

We can do this in a combined way to see all terms together:

\[ \bar{w} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)} \sigma(w^T x)(1 - \sigma(w^T x))x^T \]

\[ = - \left(1 - \sigma(w^T x)\right)x^T \]

This effectively shows gradient flow along path from \( L \) to \( w \)
The chain rule can be computed as a series of scalar, vector, and matrix linear algebra operations. Extremely efficient in graphics processing units (GPUs).

\[ \bar{w} = -\frac{1}{\sigma(w^T x)} \sigma(w^T x)(1 - \sigma(w^T x)) x^T \]
Different Views of Equivalent Ideas

Computation Graph / Global View of Chain Rule

\[
L = \frac{1}{1 + e^{-u}}
\]

\[
\frac{\partial L}{\partial p} = -\frac{1}{p}
\]

where \( p = \sigma(w^T x) \) and \( \sigma(x) = \frac{1}{1 + e^{-x}} \)

\[
\bar{u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \hat{p} \sigma(1 - \sigma)
\]

\[
\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \frac{\partial u}{\partial w} = \bar{u} x^T
\]

We can do this in a combined way to see all terms together:

\[
\bar{w} = \frac{\partial L}{\partial w} = \frac{1}{\sigma(w^T x)} \sigma(w^T x)(1 - \sigma(w^T x)) x^T
\]

This effectively shows gradient flow along path from \( L \) to \( w \)

Computational / Tensor View

Graph View

Backpropagation View (Recursive Algorithm)

We want to compute:

\[
\begin{align*}
\frac{\partial L}{\partial h^{l-1}}, & \quad \frac{\partial L}{\partial h^l}, \\
\frac{\partial L}{\partial h^l}, & \quad \frac{\partial L}{\partial W}
\end{align*}
\]
Define:
\[ h_i = w_i^T h^{\ell-1} \]

\[ h^{\ell} = W h^{\ell-1} \]

\[ \begin{bmatrix} |h^{\ell}| \times 1 \\ |h^{\ell}| \times |h^{\ell-1}| \\ |h^{\ell-1}| \times 1 \end{bmatrix} \]
\begin{align*}
  h^\ell &= Wh^{\ell-1} \\
  \frac{\partial h^\ell}{\partial h^{\ell-1}} &= W \\
  \text{Define:} & & h_i &= w_i^T h^{\ell-1} \\
  \frac{\partial h^\ell_i}{\partial w_i} &= h^{(\ell-1),T} \left( \begin{array}{c} 1 \times |h^{\ell-1}| \quad 1 \times |h^\ell| \quad |h^\ell| \times |h^{\ell-1}| \end{array} \right)
\end{align*}
Fully Connected (FC) Layer

Define:
\( h_i = w_i^T h^{\ell-1} \)
\( \frac{\partial h_i}{\partial w_i} = h^{(\ell-1),T} \)

\( h^\ell = W h^{\ell-1} \)

\( \frac{\partial h^\ell}{\partial h^{\ell-1}} = W \)

Note doing this on full \( W \) matrix would result in Jacobian tensor!

But it is sparse – each output only affected by corresponding weight row

\[
\begin{array}{c|c|c}
\frac{\partial L}{\partial w_i} & \frac{\partial L}{\partial h^\ell} & \frac{\partial h^\ell}{\partial w_i} \\
\hline
0 & \frac{\partial L}{\partial h^\ell} & \frac{\partial h^\ell}{\partial w_i} \\
0 & 0 & \frac{\partial h^\ell}{\partial w_i} \\
1 \times |h^{\ell-1}| & 1 \times |h^\ell| & |h^\ell| \times |h^{\ell-1}| \\
\end{array}
\]
Full Jacobian of ReLU layer is **large** (output dim x input dim)

- But again it is **sparse**
- Only **diagonal values non-zero** because it is element-wise
- An output value affected only by corresponding input value

Max function **funnels gradients through selected max**

- Gradient will be **zero** if input $\leq 0$

**Jacobian of ReLU**

$$ h^\ell = \max(0, h^{\ell-1}) $$

**Forward:**

$$ \frac{\partial L}{\partial h^\ell} = \frac{\partial L}{\partial h^{\ell-1}} \frac{\partial h^\ell}{\partial h^{\ell-1}} $$

**Backward:**

$$ \frac{\partial L}{\partial h^{\ell-1}} = \begin{cases} 0 & \text{if } h^{\ell-1} \leq 0 \\ 1 & \text{otherwise} \end{cases} $$
Backpropagation and Automatic Differentiation
Backpropagation does not really spell out how to efficiently carry out the necessary computations.

But the idea can be applied to any directed acyclic graph (DAG).

- Graph represents an ordering constraining which paths must be calculated first.

Given an ordering, we can then iterate from the last module backwards, applying the chain rule.

- We will store, for each node, its gradient outputs for efficient computation.

- We will do this automatically by computing backwards function for primitives and as you write code, express the function with them.

This is called reverse-mode automatic differentiation.
Computation = Graph
- Input = Data + Parameters
- Output = Loss
- Scheduling = Topological ordering

Auto-Diff
- A family of algorithms for implementing chain-rule on computation graphs

Deep Learning = Differentiable Programming
We want to find the partial derivative of output $f$ (output) with respect to all intermediate variables.

- Assign intermediate variables

  Simplify notation:
  
  Denote bar as: $\overline{a_3} = \frac{\partial f}{\partial a_3}$

- Start at end and move backward

Example

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_2) \]
The function is given by:

\[ f(x_1, x_2) = x_1x_2 + \sin(x_2) \]

The gradients from multiple paths summed are as follows:

1. **Path 1 (P1)**
   \[ a_1 = \frac{\partial f}{\partial a_1} = x_2 \]
   \[ a_3 = \frac{\partial f}{\partial a_3} = 1 \]
   \[ x_{P1}^2 = \frac{\partial f}{\partial x_1} = a_1 \cdot \cos(x_2) \]

2. **Path 2 (P2)**
   \[ a_2 = \frac{\partial f}{\partial a_2} = x_1 \]
   \[ x_{P2}^2 = \frac{\partial f}{\partial x_2} = a_2 \]

Gradients from multiple paths summed:

\[ a_{P1}^2 = \frac{\partial f}{\partial a_{P1}^2} = a_1 \]

\[ a_{P2}^2 = \frac{\partial f}{\partial a_{P2}^2} = a_2 \cdot x_2 \]

\[ a_{P1} = \frac{\partial f}{\partial a_{P1}} = a_1 \cdot x_2 \]
$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$

$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = \overline{a_3}$

$\overline{a_2} = \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_2} = \overline{a_3}$

Addition operation distributes gradients along all paths!
\[ f(x_1, x_2) = x_1 x_2 + \sin(x_2) \]

Multiplication operation is a gradient switcher (multiplies it by the values of the other term)

\[
\begin{align*}
\overline{x_2} & = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1 x_2)}{\partial x_2} = a_2 x_1 \\
\overline{x_1} & = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = a_2 x_2
\end{align*}
\]
Several other patterns as well, e.g.:

Max operation selects which path to push the gradients through:

- Gradient flows along the path that was “selected” to be max
- This information must be recorded in the forward pass

The flow of gradients is one of the most important aspects in deep neural networks:

- If gradients do not flow backwards properly, learning slows or stops!
Key idea is to explicitly store computation graph in memory and corresponding gradient functions.

Nodes broken down to basic primitive computations (addition, multiplication, log, etc.) for which corresponding derivative is known.

Computational Implementation

\[
x_2 = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = a_1 \cos(x_2)
\]
Note that we can also do **forward mode** automatic differentiation.

Start from **inputs** and propagate gradients forward.

Complexity is proportional to input size.

- Memory savings (all forward pass, no need to store activations)

- However, in most cases our **inputs** (images) are large and **outputs** (loss) are small.
Assume given
\[
\frac{\partial h^{\ell-1}}{\partial x}
\]

\[
\frac{\partial h^{\ell}}{\partial x} = \frac{\partial h^{\ell}}{\partial h^{\ell-1}} \frac{\partial h^{\ell-1}}{\partial x}
\]

See https://www.cc.gatech.edu/classes/AY2020/cs7643_spring/slides/autodiff_forward_reverse.pdf
A graph is created on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```

(Note above)
Computation Graphs in PyTorch

Back-propagation uses the dynamically built graph

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

From pytorch.org
Convolutional network (AlexNet)
Neural Turing Machine

input image

loss

Figure reproduced with permission from a Twitter post by Andrej Karpathy.
Computation graphs are not limited to mathematical functions!

Can have control flows (if statements, loops) and backpropagate through algorithms!

Can be done dynamically so that gradients are computed, then nodes are added, repeat

Differentiable programming

Program Space

Software 1.0

Software 2.0

(optimization)

Adapted from figure by Andrej Karpathy

Power of Automatic Differentiation
Optimization of Deep Neural Networks Overview
Backpropagation, and automatic differentiation, allows us to optimize any function composed of differentiable blocks.

- **No need to modify** the learning algorithm!
- The complexity of the function is only limited by **computation and memory**
A network with two or more hidden layers is often considered a **deep** model.

**Depth is important:**

- Structure the model to represent an inherently compositional world.
- Theoretical evidence that it leads to parameter efficiency.
- Gentle dimensionality reduction (if done right).
There are still many design decisions that must be made:

- **Architecture**
- **Data Considerations**
- **Training and Optimization**
- **Machine Learning Considerations**

Local Minima
We must design the **neural network architecture**:

- What **modules (layers)** should we use?
- How should they **be connected together**?
- Can we use our **domain knowledge** to add architectural biases?
Example Architectures

Fully Connected Neural Network
Example Architectures

- Fully Connected Neural Network
- Convolutional Neural Networks
Different architectures are suitable for different applications or types of input.
As in traditional machine learning, **data** is key:

- Should we **pre-process** the data?
- Should we **normalize** it?
- Can we **augment** our data by adding noise or other perturbations?
Even given a good neural network architecture, we need a **good optimization algorithm to find good weights**

- What **optimizer** should we use?
- Different optimizers make **different weight updates** depending on the gradients
- How should we **initialize** the weights?
- What **regularizers** should we use?
- What **loss function** is appropriate?
The practice of machine learning is complex: For your particular application you have to trade off all of the considerations together.

- Trade-off between model capacity (e.g. measured by # of parameters) and amount of data
- Adding appropriate biases based on knowledge of the domain
Architectural Considerations
Determining what modules to use, and how to connect them is part of the **architectural design**

- Guided by the **type of data used** and its **characteristics**
  - Understanding your data is always the first step!
- **Lots of data types (modalities)** already have good architectures
  - Start with what others have discovered!
- **The flow of gradients** is one of the key principles to use when analyzing layers
- **Combination** of linear and non-linear layers
- Combination of **only** linear layers has same representational power as one linear layer
- **Non-linear layers** are crucial
- Composition of non-linear layers enables complex transformations of the data

\[ w_1^T(w_2^T(w_3^Tx)) = w_4^Tx \]
Several aspects that we can **analyze**:

- **Min/Max**
- **Correspondence between input & output statistics**
- **Gradients**
  - At initialization (e.g. small values)
  - At extremes
- **Computational complexity**
**Min:** 0, **Max:** 1

- Output **always positive**
- Saturates at **both ends**
- **Gradients**
  - Vanishes at both end
  - Always positive
- **Computation:** Exponential term

\[ h^\ell = \sigma(h^{\ell-1}) \]

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\[ \frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial W} \]
- Min: -1, Max: 1
- Centered
- Saturates at **both ends**
- **Gradients**
  - Vanishes at both end
  - Always positive
- Still somewhat computationally heavy

\[ h^\ell = \tanh(h^{\ell-1}) \]
- **Min:** 0, **Max:** Infinity
- Output always **positive**
- No **saturation** on positive end!
- **Gradients**
  - 0 if $x \leq 0$ (dead ReLU)
  - Constant otherwise (does not vanish)
- **Cheap to compute (max)**

$h^\ell = \max(0, h^{\ell-1})$
- **Min:** -Infinity, **Max:** Infinity
- Learnable parameter!
- No saturation
- Gradients
  - No dead neuron
- Still cheap to compute

Leaky ReLU

\[ h^\ell = \max(\alpha h^{\ell-1}, h^{\ell-1}) \]
Selecting a Non-Linearity

Which **non-linearity** should you select?

- Unfortunately, **no one activation function is best** for all applications
- **ReLU** is most common starting point
  - Sometimes leaky ReLU can make a big difference
- **Sigmoid** is typically avoided unless clamping to values from [0,1] is needed
• **Backpropagation**: Recursive, modular algorithm for chain rule + gradient descent

• **When we move to vectors and matrices:**
  • Computation graph (composition of functions) => Multiplication of Jacobians along path

• **Automatic differentiation:**
  • Reduction of modules to simple operations we know (simple multiplication, etc.)
  • Automatically build computation graph in background as write code
  • Automatically compute gradients via backward pass

• **We now have a generic algorithm! Considerations:**
  • Architecture
  • Data Considerations
  • Training and Optimization
  • Machine Learning Considerations