Topics:

- Jacobians
- Optimization

CS 4803-DL / 7643-A
ZSOLT KIRA
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

Layer 1 → Layer 2 → Layer 3

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
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

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
Neural Network Training

**Step 1:** Compute Loss on Mini-Batch: **Forward Pass**

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

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**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Neural Network Training
Neural Network Training

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: \( \{ \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \} \)

We will use the chain rule to do this:

\[
\text{Chain Rule: } \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}
\]
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
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>
</tbody>
</table>
Scalar Case

\[ x \in \mathbb{R} \quad \rightarrow \quad z \in \mathbb{R} \quad \rightarrow \quad y \in \mathbb{R} \quad \rightarrow \quad \text{Loss} \]

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

\[ \frac{dy}{2x} = \frac{2y}{2t} \cdot \frac{ct}{2x} \]

Scalar mult
For $x \in \mathbb{R}^d$, $g_1(x) \rightarrow \mathbb{R}^m$, $g_2(x) \rightarrow \mathbb{R}^c$:

\[ x \in \mathbb{R}^d \rightarrow g_1(x) \rightarrow \mathbb{R}^m \rightarrow g_2(x) \rightarrow \mathbb{R}^c \]

\[
\left[ \frac{\partial y}{\partial x} \right] = \left[ \frac{\partial y}{\partial x} \right] \cdot \left[ \frac{\partial z}{\partial x} \right]
\]

\[ J_{g_2 \circ g_1} = J_{g_2} \cdot J_{g_1} \]

**Vector Case**
Jacobian View of Chain Rule
Graphical View of Chain Rule
Chain Rule: Cascaded

\[
\frac{\partial h'}{\partial h} = \frac{\partial h'}{\partial h_{n-1}} \frac{\partial h_{n-1}}{\partial h_{n-2}} \cdots \frac{\partial h_1}{\partial h_0}
\]

\[
\begin{bmatrix}
\vdots \\
\end{bmatrix}
\leftarrow \begin{bmatrix}
\vdots \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\vdots \\
\end{bmatrix} 
\]

\[
\begin{bmatrix}
\vdots \\
\end{bmatrix} 
\]

\[
\begin{bmatrix}
\vdots \\
\end{bmatrix} 
\]
- 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 -> model -> loss function) is also a computation graph.

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

\[ w^T x \rightarrow \frac{1}{1 + e^{-u}} \rightarrow -\log \left( \frac{1}{1 + e^{-w^T x}} \right) \rightarrow L \]
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

\[
\begin{align*}
\bar{L} &= 1 \\
\bar{p} &= \frac{\partial L}{\partial p} = -\frac{1}{p} \\
\text{where } p &= \sigma(w^T x) \text{ and } \sigma(x) = \frac{1}{1+e^{-x}}
\end{align*}
\]

\[
\begin{align*}
\bar{u} &= \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \bar{p} \sigma(1 - \sigma)
\end{align*}
\]

\[
\begin{align*}
\bar{w} &= \frac{\partial L}{\partial w} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w} = \bar{u} x^T
\end{align*}
\]

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

\[
\begin{align*}
\bar{w} &= \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)} \sigma(w^T x)(1 - \sigma(w^T x)) x^T \\
&= -\left(1 - \sigma(w^T x)\right) x^T
\end{align*}
\]

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)

\[
\bar{w} = -\frac{1}{\sigma(w^T x)} \sigma(w^T x)(1 - \sigma(w^T x))x^T
\]
Computation Graph / Global View of Chain Rule

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

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

\[ \frac{\partial}{\partial u} = \frac{\partial}{\partial w} \]
\[ \frac{\partial}{\partial p} = \frac{\partial}{\partial w} \sigma \]

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

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

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

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

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

Computational / Tensor View

Graph View

We want to to compute:

\[ \left\{ \frac{\partial L}{\partial h_{i-1}}, \frac{\partial L}{\partial W} \right\} \]

Backpropagation View

(Recursive Algorithm)

Different Views of Equivalent Ideas
Define:
\[ h_i = w_i^T h_{\ell-1} \]

\[ h_\ell = W h_{\ell-1} \]
Define:
\[ h_i = w_i^T h^{\ell-1} \]
\[ \frac{\partial h^\ell_i}{\partial w_i} = h^{(\ell-1),T} \]
\[ 1 \times |h^{\ell-1}| \quad 1 \times |h^\ell| \quad |h^\ell| \times |h^{\ell-1}| \]
\[ h^\ell = Wh^{\ell-1} \]

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

Define:
\[ h_i = w_i^T h^{\ell-1} \]

\[ \frac{\partial h^\ell_i}{\partial w_i} = h^{(\ell-1),T} \]

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

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

\[ \frac{\partial L}{\partial h^{\ell}} = \frac{\partial L}{\partial w_i} \frac{\partial h^\ell_i}{\partial w_i} \]

\[ \begin{bmatrix} \frac{\partial L}{\partial w_i} \\ \frac{\partial h^\ell_i}{\partial w_i} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{\partial h^\ell_i}{\partial w_i} \end{bmatrix} \]

\[ 1 \times |h^{\ell-1}| \quad 1 \times |h^\ell| \quad |h^\ell| \times |h^{\ell-1}| \]
We can employ any differentiable (or piecewise differentiable) function.

A common choice is the Rectified Linear Unit:
- Provides non-linearity but better gradient flow than sigmoid.
- Performed element-wise.

How many parameters for this layer?

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

\[ \text{Jacobian of ReLU} \]

\[
\begin{bmatrix}
 h^\ell \\
 h^{\ell-1}
\end{bmatrix} = \text{max}(0, h^{\ell-1})
\]

**Forward:**

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

**Backward:**

\[
\frac{\partial L}{\partial h^{\ell-1}} = \begin{cases} 
 1 & \text{if } h^{\ell-1} > 0 \\
 0 & \text{otherwise}
\end{cases}
\]
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:

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

\[ a_3 = \frac{\partial f}{\partial a_3} = 1 \]

\[ a_1 = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = a_3 \]

\[ a_2 = \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_2} = a_3 \]

\[ x_2^{P1} = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = a_1 \cos(x_2) \]

\[ x_2^{P2} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1 x_2)}{\partial x_2} = a_2 x_1 \]

\[ x_1 = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = a_2 x_2 \]
\[ f(x_1, x_2) = x_1x_2 + \sin(x_2) \]

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

\[
\bar{x}_2 = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = \bar{a}_1 \cos(x_2)
\]
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
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)
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))
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

Power of Automatic Differentiation

Adapted from figure by Andrej Karpathy
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**

```
X → Model → \(-\log(p)\) → L
```

**The Power of Deep Learning**
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**
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

Input Data → Predictions
Example Architectures

- **Fully Connected Neural Network**: Input Data → Predictions
- **Convolutional Neural Networks**: Input Image → Predictions
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**

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

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

$h^\ell = \max(\alpha h^{\ell-1}, h^{\ell-1})$
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
Initialization
Initializing the Parameters

The parameters of our model must be initialized to something

- Initialization is **extremely important**!
- Determined how **statistics of outputs** (given inputs) behave
- Determines how well **gradients flow** in the beginning of training (important)
- Could **limit use of full capacity** of the model if done improperly
- Initialization that is **close to a good (local) minima** will converge faster and to a better solution
Initializing values to a constant value leads to **a degenerate solution!**

- What happens to the **weight updates**?

- Each node has the same input from previous layers so gradients **will be the same**

- As a result, **all weights will be updated** to the same exact values

\[ w_i = c \quad \forall i \]
Common approach is **small normally distributed random numbers**

- E.g. $N(\mu, \sigma)$ where $\mu = 0, \sigma = 0.01$
- **Small weights** are preferred since no feature/input has prior importance
- Keeps the model within the **linear region of most activation functions**
Deeper networks (with many layers) are more sensitive to initialization

- With a deep network, activations (outputs of nodes) get smaller
- Standard deviation reduces significantly
- Leads to small updates – smaller values multiplied by upstream gradients
- Larger initial values lead to saturation

Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

Ideally, we’d like to maintain the variance at the output to be similar to that of input!

- This condition leads to a simple initialization rule, sampling from uniform distribution:
  \[ \text{Uniform} \left( -\frac{\sqrt{6}}{n_j + n_{j+1}}, + \frac{\sqrt{6}}{n_j + n_{j+1}} \right) \]

- Where \( n_j \) is fan-in (number of input nodes) and \( n_{j+1} \) is fan-out (number of output nodes)

Distribution of activation values of a network with tanh non-linearities, for increasingly deep layers

*From "Understanding the difficulty of training deep feedforward neural networks." AISTATS, 2010.*
In practice, **simpler versions** perform empirically well:

\[
N(0, 1) \ast \sqrt{\frac{1}{n_j}}
\]

- This analysis holds for **tanh or similar activations**.
- Similar analysis for **ReLU activations** leads to:

\[
N(0, 1) \ast \sqrt{\frac{1}{n_j/2}}
\]

Summary

Key takeaway: **Initialization matters!**

- Determines the **activation** (output) statistics, and therefore **gradient statistics**
- If gradients are **small**, no learning will occur and no improvement is possible!
- Important to reason about **output/gradient statistics** and analyze them for new layers and architectures