Topics:
• Backpropagation
• Computation Graph and Automatic Differentiation
Recap: Multiclass SVM loss

Given an example \((x_i, y_i)\) where \(x_i\) is the image and where \(y_i\) is the (integer) label,

and using the shorthand for the scores vector: \(s = f(x_i, W)\)

the SVM loss has the form:

\[
L_i = \sum_{j \neq y_i} \begin{cases} 
0 & \text{if } s_{y_i} \geq s_j + 1 \\
(s_j - s_{y_i}) + 1 & \text{otherwise}
\end{cases} 
\]

\[
= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)
\]

“Hinge Loss”

\(L_i\) scores for other classes

score for correct class

margin

score

\(s_{y_i}\)

\(s_j\)

\(L_i\)

1

Loss = 0:
Recap: Regularization

Q: How do we pick between W and 2W?
A: Opt for simpler functions to avoid overfit

How? Regularization!

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

\[ \lambda \text{ = regularization strength (hyperparameter)} \]

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing too well on training data
Recap: Softmax Classifier and Cross Entropy Loss

Want to interpret raw classifier scores as probabilities

\[
p_\theta(Y = y_i | X = x_i) = \frac{e^{s_yi}}{\sum_j e^{s_j}}
\]

Softmax Function

<table>
<thead>
<tr>
<th>Cat</th>
<th>3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>5.1</td>
</tr>
<tr>
<td>Frog</td>
<td>-1.7</td>
</tr>
</tbody>
</table>

How do we optimize the classifier? We maximize the probability of \( p_\theta(y_i | x_i) \)

1. Maximum Likelihood Estimation (MLE):
Choose weights to maximize the likelihood of observed data. In this case, the loss function is the Negative Log-Likelihood (NLL).

Finding a set of weights \( \theta \) that maximizes the probability of correct prediction: \( \arg\max_\theta \prod p_\theta(y_i | x_i) \)

This is equivalent to:

\[
\arg\max_\theta \sum \ln p_\theta(y_i | x_i)
\]

\[
L_i = -\ln p_\theta(y_i | x_i) = -\ln \left( \frac{e^{s_yi}}{\sum_j e^{s_j}} \right)
\]

2. Information theory view:
Derive NLL from the cross-entropy measurement. Also known as the cross-entropy loss

Cross Entropy:

\[
H(p, q) = - \sum p(x) \ln q(x)
\]

Cross Entropy Loss -> NLL

\[
H_i(p, p_\theta) = - \sum_{y \in Y} p(y | x_i) \ln p_\theta(y | x_i)
\]

\[
= -\ln p_\theta(y_i | x_i)
\]

\[
L = \sum H_i(p, p_\theta) = - \sum \ln p_\theta(y_i | x_i) \equiv \text{NLL}
\]
Why softmax?

Consider the following three basis for NLL:

1. Squash and clip value to (0, 1]
2. Logistic function
3. Logistic function but no log (just negative likelihood)

Use logistic function as example. Same as softmax but for binary classification

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

Consider the following three basis for NLL:

1. Squash and clip value to (0, 1]
2. Logistic function
3. Logistic function but no log (just negative likelihood)

Why this?

$$p_\theta(Y = y_i | X = x_i) = \frac{e^{s y_i}}{\sum_j e^{s_j}}$$

2. NLL w/ logistic: Strong guidance when classifier is wrong

Only saturate at convergence, e.g. $\sigma(3) \approx 0.95$
Recap: gradient-based optimization

As weights change, the gradients change as well

- This is often somewhat-smooth locally, so small changes in weights produce small changes in the loss

We can therefore think about iterative algorithms that take current values of weights and modify them a bit
Recap: The gradient descent algorithm

1. Choose a model: $f(x, W) = Wx$

2. Choose loss function: $L_i = |y - Wx_i|^2$

3. Calculate partial derivative for each parameter: $\frac{\partial L}{\partial w_i}$

4. Update the parameters: $w_i = w_i - \frac{\partial L}{\partial w_i}$

5. Add learning rate to prevent too big of a step: $w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$

Repeat 3-5
We can find the steepest descent direction by computing the **derivative**:

\[
\frac{\partial f}{\partial w} = \lim_{h \to 0} \frac{f(w + h) - f(w)}{h}
\]

- **Gradient** is multi-dimensional derivatives
- Notation: \( \frac{\partial f}{\partial w} \) is the gradient of \( f \) (e.g., a loss function) with respect to variable \( w \) (e.g., a weight vector).
- \( \frac{\partial f}{\partial w} \) is of the **same shape** as \( w \)
- **Intuitively**: Measures how the function changes as the variable \( w \) changes by a small step size
- Steepest descent direction is the **negative gradient**
- **Gradient descent**: Minimize loss by changing parameters

Composing simple functions creates complex analytical gradients

\[
\begin{align*}
\sin(x) & \\
\log(x) & \\
\cos(x) & \\
x^3 & \\
\exp(x) & \\
\end{align*}
\]

Compose into a complex function

\[-\log\left(\frac{1}{1 + e^{-w \cdot x}}\right)\]

Adapted from slides by: Marc'Aurelio Ranzato, Yann LeCun
Decomposing a Function

This time: Chain rule and Backpropagation!

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

Adapted from slides by: Marc'Aurelio Ranzato, Yann LeCun
Functions can be made **arbitrarily complex** (subject to memory and computational limits), e.g.:

\[ f(x, W) = \sigma(W_5 \sigma(W_4 \sigma(W_3 \sigma(W_2 \sigma(W_1 x)))) \]

We can use **any type of differentiable function (layer)** we want!
We are learning **complex models** with significant amount of parameters (millions or billions)

How do we compute the gradients of the **loss** (at the end) with respect to **internal** parameters?

Intuitively, want to understand how **small changes** in weight are **propagated** to affect the **loss function** at the end

\[
\frac{\partial L}{\partial w_i}
\]
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.

The **backpropagation algorithm** will then process this graph, **one module at a time**.

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
This is a computation graph!

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

Backpropagation (roughly):
1. Calculate local gradients for each node (e.g., \(\frac{\partial u}{\partial w}\))
2. Trace the computation graph (backward) to calculate the global gradients for each node w.r.t. to the loss function.
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \(x = -2, y = 5, z = -4\)

Want: \(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\)
Backpropagation: a simple example

Let's consider the function:

$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

We have:

$$q = x + y$$

$$\frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1$$

1. Calculate local gradients

Want:

$$\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z}$$
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\[ e.g. \ x = -2, \ y = 5, \ z = -4 \]

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \[ \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \]

1. Calculate local gradients
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\[ \text{e.g. } x = -2, \ y = 5, \ z = -4 \]

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

\[
\begin{aligned}
\frac{\partial f}{\partial x}, \quad &\frac{\partial f}{\partial y}, \quad &\frac{\partial f}{\partial z}
\end{aligned}
\]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} \]

Upstream gradient Local gradient

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} \]

Upstream gradient

Local gradient

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} \]

Upstream gradient

Local gradient

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} \]

Upstream gradient

Local gradient

Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Patterns in backward flow

How does a local gradient modify the upstream gradient? \( f = 2(xy + \max(z, w)) \)
Patterns in backward flow

How does a local gradient modify the upstream gradient? \( f = 2(xy + \max(z,w)) \)

**Q:** What is an **add** gate?
Patterns in backward flow

How does a local gradient modify the upstream gradient? \( f = 2(xy + \max(z, w)) \)

**add gate: gradient replicator**

\[
\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b} = 1
\]
Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(x y + \max(z, w))$

**add** gate: gradient replicator

**Q:** What is a **max** gate?

![Diagram showing the calculation and gradient flow with nodes and edges representing variables and operations.](image)
Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(xy + \max(z, w))$

- **add** gate: gradient replicator
- **max** gate: gradient router

only the path selected by the max operator gets the upstream gradient
Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(xy + \max(z, w))$

- **add** gate: gradient replicator
- **max** gate: gradient router

**Q:** What is a **mul** gate?
Patterns in backward flow

How does a local gradient modify the upstream gradient? \( f = 2(xy + \max(z, w)) \)

- **add** gate: gradient replicator
- **max** gate: gradient router
- **mul** gate: gradient switcher

\[
\begin{align*}
    f &= a \cdot b \\
    \frac{\partial f}{\partial a} &= b \\
    \frac{\partial f}{\partial b} &= a
\end{align*}
\]
Upstream gradients add at fork branches

... as long as the branches join at some point in the graph
Upstream gradients add at fork branches

Claim: \( \frac{\partial L}{\partial x} = \frac{\partial L}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial L}{\partial f_2} \frac{\partial f_2}{\partial x} \)

\[ = 1 \cdot e^x + 1 \cdot 2x = e^x + 2x \]

Derivation: \( L = e^x + x^2 \)

\[ \frac{\partial L}{\partial x} = e^x + 2x \]
Upstream gradients add at fork branches

Claim: \[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial L}{\partial f_2} \frac{\partial f_2}{\partial x}
\]
\[
= x^2 \cdot e^x + e^x \cdot 2x
\]

Derivation: \[
L = e^x \cdot x^2
\]
\[
\frac{\partial L}{\partial x} = e^x \cdot 2x + e^x \cdot x^2
\]
Duality in F(orward)prop and B(ack)prop

(C) Dhruv Batra
Given this computation graph, the training algorithm will:

- Calculate the current model’s outputs (called the **forward pass**)
- Calculate the gradients for each module (called the **backward pass**)

**Overview of Training**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters.
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters.

- Assume that we have the gradient of the loss with respect to the **module’s outputs** (given to us by upstream module).
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters.

- Assume that we have the gradient of the loss with respect to the **module’s outputs** (given to us by upstream module).
- We can calculate the gradient of the loss with respect to the module’s weights.

![Diagram of the backward pass](attachment:backward_pass_diagram.png)

**Overview of Training**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun.

\[
\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^l} \frac{\partial h^l}{\partial W}
\]
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters:

- Assume that we have the gradient of the loss with respect to the **module’s outputs** (given to us by upstream module)
- We can calculate the gradient of the loss with respect to the module’s weights
- We will also pass the gradient of the loss with respect to the **module’s inputs**
  - This is not required for update the module’s weights, but passes the gradients back to the previous module

![Diagram of the backpropagation algorithm](image)
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters:

- Assume that we have the gradient of the loss with respect to the **module’s outputs** (given to us by upstream module).
- We can calculate the gradient of the loss with respect to the module’s weights.
- We will also pass the gradient of the loss with respect to the **module’s inputs**.
  - This is not required for update the module’s weights, but passes the gradients back to the previous module.
- Becomes the **upstream gradient** for the previous module.

Using this computation graph, the training algorithm will:

- Calculate the current model’s outputs (called the **forward pass**).
- Calculate the gradients for each module (called the **backward pass**).

Backward pass is a recursive algorithm that:

- **Starts at** the loss function where we know how to calculate the gradients.
- **Progresses back through the modules**.
- **Ends in the input layer** where we do not need gradients (no parameters).

This algorithm is called **backpropagation**.
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters

- Assume that we have the gradient of the loss with respect to the **module’s outputs** (given to us by upstream module)
- We can calculate the gradient of the loss with respect to the module’s weights
- We will also pass the gradient of the loss with respect to the **module’s inputs**
  - This is not required for update the module’s weights, but passes the gradients back to the previous module
  - Becomes the **upstream gradient** for the previous module
- **Gradient descent**: update weight with gradient with respect to loss
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 tracing the entire graph, aggregate and assign gradients at each function / parameters, from output to input.

This is called reverse-mode **automatic differentiation**

---

A General Framework
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
Deep Learning Framework = Differentiable Programming Engine

• Computation = Graph
  – Input = Data + Parameters
  – Output = Loss
  – Scheduling = Topological ordering

• What do we need to do?
  – Generic code for representing the graph of modules
  – Specify modules (both forward and backward functions)
Modularized implementation: forward / backward API

Graph (or Net) object  (rough psuedo code)

```python
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Modularized implementation: forward / backward API

(x, y, z are scalars)
Modularized implementation: forward / backward API

```python
class MultiplyGate(object):
    def forward(self, x, y):
        z = x * y
        self.x = x  # must keep these around!
        self.y = y
        return z
    def backward(self, dz):
        dx = self.y * dz  # [dz/dx * dL/dz]
        dy = self.x * dz  # [dz/dy * dL/dz]
        return [dx, dy]
```

(x, y, z are scalars)
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))
Neural Turing Machine

input image

loss

Figure reproduced with permission from a Twitter post by Andrej Karpathy.

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
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

Adapted from figure by Andrej Karpathy
Autodiff from scratch: micrograd repo, video tutorial
Next time:

• More on backprop but for (shallow) neural nets!
• Jacobians
• Activation functions