Brown IgniteCS

An open **CS education club**!

Design curricula and teach at schools around Rhode Island!

Please join our emailing list if interested!

Also, contact us for more info or questions!

- jitpuwapat_mokkamakkul@brown.edu
- angel_arrazola@brown.edu
Recap: Forward Pass

Compute the prediction or evaluate the loss for a single input $\mathbf{x}$.

Goal of learning: Minimize the total loss for all $\mathbf{x}$ in training data.
Recap: Forward Pass

Compute the prediction or evaluate the loss for a single input $x$.

Goal of learning: Minimize the total loss for all $x$ in training data with respect to model parameters $W, b$. 

![Diagram of Forward Pass]

- $W, b$: Parameters
- FC: Linear layer
- Softmax
- $L$: Loss
Recap: Backpropagation (Backward Pass)

Gradient descent: $\Delta W = -\alpha \nabla \hat{L}(W)$ and $\Delta b = -\alpha \nabla \hat{L}(b)$

Backpropagation: Compute $\Delta W$ and $\Delta b$ via chain rule.

$$\frac{dL}{dw_{j,i}} = \frac{dl_j}{dw_{j,i}} \cdot \frac{dp_a}{dl_j} \cdot \frac{dL}{dp_a}$$
Recap: Computation graph

Parameter update:

\[
\frac{\partial e}{\partial w_k} = \frac{\partial e}{\partial h_k} \frac{\partial h_k}{\partial w_k}
\]

Local gradient

Local gradient

Upstream gradient:

\[
\frac{\partial e}{\partial h_k}
\]

Downstream gradient:

\[
\frac{\partial e}{\partial h_{k-1}} = \frac{\partial e}{\partial h_k} \frac{\partial h_k}{\partial h_{k-1}}
\]

Forward pass

Backward pass
Today’s goal – learn about deep learning frameworks

(1) Gradient Descent pseudocode

(2) Stochastic Gradient Descent (SGD)

(3) Automatic differentiation
Putting Everything Together: Gradient Descent

# $\delta W$ is 2-D matrix of 0’s in the shape of $W$

for each input and corresponding answer $a$:

```python
probabilities = run_network(input)
```

Forward pass

for $j$ in range(len(probabilities)):

```python
y_j = 1 if j == a else 0
```

Backward pass: Compute $\frac{\partial L}{\partial W_{ij}}$ for every $W_{ij}$

for $i$ in range(len(input)):

```python
delta_W[j][i] += alpha * (y_j - probabilities[j]) * input[i]
```

Over the entire dataset

$W += \delta W$
Putting Everything Together: Gradient Descent

# delta_W is 2-D matrix of 0’s in the shape of W

for each input and corresponding answer a:

```
probabilities = run_network(input)
```

Forward pass

```
for j in range(len(probabilities)):
    y_j = 1 if j == a else 0
    for i in range(len(input)):
        delta_W[j][i] += alpha * (y_j - probabilities[j]) * input[i]
```

Backward pass:
compute $\frac{\partial L}{\partial W_{ij}}$ for every $W_{ij}$

Over the entire dataset

```
W += delta_W
```

Gradient descent update
Gradient Descent: Limitation?

# $\delta_W$ is 2-D matrix of 0's in the shape of $W$

for each input and corresponding answer $a$:

probabilities = run_network(input)

for $j$ in range(len(probabilities)):

    $y_j = 1$ if $j == a$ else 0

    for $i$ in range(len(input)):

        $\delta_W[j][i] += \alpha * (y_j - probabilities[j]) * input[i]$

$W += \delta_W$

We iterate over the entire dataset...

...to update the weights only once
Stochastic Gradient Descent (SGD)

• Alternative is to train on batches: small subsets of the training data
• Why stochastic: Each batch is randomly sampled from the full training data
• We update the parameters after each batch
for each batch:

# delta_W is 2-D matrix of 0's in the shape of W

for each input and corresponding answer a in batch:

    probabilities = run_network(input)

    for j in range(len(probabilities)):
        y_j = 1 if j == a else 0

        for i in range(len(input)):
            delta_W[j][i] += alpha * (y_j - probabilities[j]) * input[i]

    W += delta_W
Stochastic Gradient Descent: Pseudocode

for each batch:

# delta_W is 2-D matrix of 0’s in the shape of W

for each input and corresponding answer a in batch:

probabilities = run_network(input)

for j in range(len(probabilities)):

y_j = 1 if j == a else 0

for i in range(len(input)):

\[ \text{delta}_W[j][i] += \alpha \times (y_j - \text{probabilities}[j]) \times \text{input}[i] \]

W += delta_W

Now we update weights after every batch
**Stochastic Gradient Descent (SGD)**

- Train on **batches**: small subsets of the training data
- We update the parameters after each batch
- This makes the training process **stochastic** or non-deterministic: -  
  * batches are a random subsample of the data  
  * **do not provide the gradient that the entire dataset** as a whole would provide at once
- Formally: the gradient of a randomly-sampled batch is an unbiased estimator of the gradient over the whole dataset
  - “Unbiased”: expected value == the true gradient, but may have large variance (i.e. the gradient may ‘jitter around’ a lot)
What size should the batch be?

- **Small batch size:** Fast, jittery updates
- **Large batch size:** Slower, stable updates

• Rule of thumb nowadays: Pick the largest batch size you can fit on your GPU!
Generalizing Backpropagation
Generalizing Backpropagation

• What if we want to add another layer to our model?
• Calculating derivatives by hand \textit{again} is a lot of work 😞

Can the computers do this for us? Yes 😊
Computer-based Derivatives

• **Numeric differentiation**
  
  - \( \frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \)
  
  - Pick a small step size \( \Delta x \)
  
  - Also called “finite differences”
Computer-based Derivatives

• **Numeric differentiation**
  
  - \( \frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x} \)
  
  - Pick a small step size \( \Delta x \)
  
  - Also called “finite differences”
  
  - Easy to implement
  
  - Arbitrarily inaccurate/unstable

\[
y = \frac{1}{x} \quad \Delta x = 0.5
\]
Computer-based Derivatives

• Numeric differentiation

• **Symbolic differentiation**
  • Computer “does algebra” and simplifies expressions
  • What Wolfram Alpha does https://www.wolframalpha.com/

\[
\frac{d}{dx} (2x + 3x^2 + x (6 - 2))
\]

\[\frac{d}{dx} (6x + 3x^2)\]

\[\frac{d}{dx} (2x + 3x^2 + x (6 - 2)) = 6 (x + 1)\]
Computer-based Derivatives

• Numeric differentiation

• **Symbolic differentiation**
  • Computer “does algebra” and simplifies expressions
  • What Wolfram Alpha does
  • Exact (no approximation error)
  • Complex to implement
  • Only handles static expressions (what about e.g. loops?)

• Example:

```python
while abs(x) > 5:
    x = x / 2
```

• This loop could run once or 100 times, it’s impossible to know
Computer-based Derivatives

- Numeric differentiation
- Symbolic differentiation
- **Automatic differentiation**
  - Use the chain rule at runtime
Computer-based Derivatives

• Numeric differentiation
• Symbolic differentiation

• **Automatic differentiation**
  • Use the chain rule at runtime
  • Gives exact results
  • Handles dynamics (loops, etc.)
  • Easier to implement
  • Can’t simplify expressions

• \( \sin^2 x + \cos^2 x \Rightarrow 1 \)

• Automatic differentiation doesn’t know this identity, will end up evaluating the entire expression on the left hand side
Computer-based Derivatives

• Numeric differentiation
• Symbolic differentiation
• **Automatic differentiation**
  • Use the chain rule at runtime
  • Gives exact results
  • Handles dynamics (loops, etc.)
  • Easier to implement
  • Can’t simplify expressions
  • What Tensorflow and PyTorch use

• $\sin^2 x + \cos^2 x \Rightarrow 1$
• Automatic differentiation doesn’t know this identity, will end up evaluating the entire expression on the left hand side
Two Main “Flavors” of Autodiff

• **Forward Mode Autodiff**
  • Compute derivatives alongside the program as it is running

• **Reverse Mode Autodiff**
  • Run the program, then compute derivatives (in reverse order)
Two Main “Flavors” of Autodiff

- **Forward Mode Autodiff**
  - Compute derivatives alongside the program as it is running

- **Reverse Mode Autodiff**
  - Run the program, then compute derivatives (in reverse order)
Forward Mode Autodiff

• Given $f(x, y) = x^2 + \log y$

Function inputs

$x$

$y$
Forward Mode Autodiff

- Given $f(x, y) = x^2 + \log y$

![Diagram]

- $c = x^2$
- $b = \log y$

$x \rightarrow c$

$y \rightarrow b$
Forward Mode Autodiff

• Given \( f(x, y) = x^2 + \log y \)
What is the chain rule for $\frac{de}{dx}$ and $\frac{de}{dy}$?

e = c + b = x^2 + \log y

\begin{align*}
c &= x^2 \\
b &= \log y
\end{align*}
Forward Mode Autodiff

• Idea: Augment each node...

\[
c = x^2 \\
\]
\[
e = c + b = x^2 + \log y \\
\]
\[
b = \log y \\
\]
\[
x \rightarrow c \\
\]
\[
y \rightarrow b \\
\]
Forward Mode Autodiff

• ...with functions that compute derivatives

\[ c = x^2 \]
\[ b = \log y \]
\[ e = c + b = x^2 + \log y \]

- \( \frac{\partial c}{\partial x} = 2x \)
- \( \frac{\partial b}{\partial y} = \frac{1}{y} \)
- \( \frac{\partial e}{\partial c} = 1 \)
- \( \frac{\partial e}{\partial b} = 1 \)
Forward Mode Autodiff

• Then, keep track of derivatives as you compute:

\[ \frac{dx}{dx} = 1 \]
\[ \frac{dc}{dx} = 2x \]
\[ e = c + b = x^2 + \log y \]
\[ \frac{de}{dc} = 1 \]
\[ \frac{de}{db} = 1 \]
Forward Mode Autodiff

• Then, keep track of derivatives as you compute:

\[ b = \log y \]
\[ c = x^2 \]
\[ e = c + b = x^2 + \log y \]

\[ \frac{dx}{dx} = 1 \]
\[ \frac{dc}{dx} = 2x \]
\[ \frac{de}{dc} = 1 \]
\[ \frac{de}{db} = 1 \]

\[ \frac{db}{dy} = \frac{1}{y} \]
Forward Mode Autodiff

• Then, keep track of derivatives as you compute:

\[
\begin{align*}
\frac{dx}{dx} &= 1 \\
\frac{dc}{dx} &= 2x \\
\frac{de}{dc} &= 1 \\
\frac{de}{db} &= 1
\end{align*}
\]

\[
e = c + b = x^2 + \log y
\]
Forward Mode Autodiff

• Can do the same thing starting from the second input:

\[
c = x^2
\]
\[
\frac{\partial c}{\partial x} = 2x
\]
\[
b = \log y
\]
\[
\frac{\partial b}{\partial y} = \frac{1}{y}
\]
\[
es = c + b = x^2 + \log y
\]
\[
\frac{\partial e}{\partial c} = 1
\]
\[
\frac{\partial e}{\partial b} = 1
\]
Forward Mode Autodiff

• Can do the same thing starting from the second input:

\[ c = x^2 \]
\[ \frac{\partial c}{\partial x} = 2x \]
\[ e = c + b = x^2 + \log y \]

\[ b = \log y \]
\[ \frac{\partial b}{\partial y} = \frac{1}{y} \]
\[ \frac{\partial b}{\partial c} = 1 \]
\[ \frac{\partial e}{\partial c} = 1 \]
\[ \frac{\partial e}{\partial b} = 1 \]
Forward Mode Autodiff

• Can do the same thing starting from the second input:

\[ c = x^2 \]
\[ \frac{\partial c}{\partial x} = 2x \]

\[ b = \log y \]
\[ \frac{\partial b}{\partial y} = 1 \]

\[ e = c + b = x^2 + \log y \]
\[ \frac{\partial e}{\partial c} = 1 \]
\[ \frac{\partial e}{\partial b} = 1 \]
Forward Mode Autodiff

• We can think of each node...

\[
c = x^2
\]

\[
\frac{\partial c}{\partial x} = 2x
\]

\[
b = \log y
\]

\[
\frac{\partial b}{\partial y} = \frac{1}{y}
\]

\[
e = c + b = x^2 + \log y
\]

\[
\frac{\partial e}{\partial c} = 1
\]

\[
\frac{\partial e}{\partial b} = 1
\]
Forward Mode Autodiff

• ...as operating on a (value, derivative) tuple:

\[
(x, \frac{dx}{dx}) = (x, 1) \\
(c, \frac{dc}{dx}) = (x^2, 2x) \\
(y, \frac{dy}{dy}) = (y, 1) \\
(b, \frac{db}{dy}) = (\log y, \frac{1}{y}) \\
(e, \frac{de}{dc}) = (c + b, 1) \\
(e, \frac{de}{db}) = (c + b, 1)
\]

These tuples are called **dual numbers**

Any questions?
Problems w/ Forward Mode for our use case

- For $f: \mathbb{R} \rightarrow \mathbb{R}^n$ (1 input to n outputs) we can differentiate in one pass
- For $f: \mathbb{R}^n \rightarrow \mathbb{R}$ (n inputs to 1 output) we need $n$ passes

$N =$ number of input features to the network, $K =$ number of nodes in the graph

Can you calculate the time and memory complexity?

Join at menti.com | use code 42316933

these derivatives are being calculated multiple times
Problems w/ Forward Mode for our use case

- For \( f: \mathbb{R} \to \mathbb{R}^n \) (1 input to n outputs) we can differentiate in one pass
- For \( f: \mathbb{R}^n \to \mathbb{R} \) (n inputs to 1 output) we need n passes

\( N = \) number of input features to the network, \( K = \) number of nodes in the graph

Forward mode: \( O(N \times K) \) time, \( O(1) \) memory
Two Main “Flavors” of Autodiff

• Forward Mode Autodiff
  • Compute derivatives alongside the program as it is running

• Reverse Mode Autodiff
  • Run the program, then compute derivatives (in reverse order)
Reverse Mode Autodiff

• Idea: first, run the function forward to produce the graph
• $f(x, y) = x^2 + \log y$

\[ c = x^2 \]
\[ b = \log y \]
\[ e = c + b = x^2 + \log y \]
Reverse Mode Autodiff

- Then, compute derivatives *backward* from the final node toward the inputs

\[ c = x^2 \]

\[ e = c + b = x^2 + \log y \]
Reverse Mode Autodiff

- Then, compute derivatives *backward* from the final node toward the inputs

\[ e = c + b = x^2 + \log y \]
Reverse Mode Autodiff

- Then, compute derivatives \textit{backward} from the final node toward the inputs

\[ c = x^2 \]
\[ b = \log y \]
\[ e = c + b = x^2 + \log y \]
Reverse Mode Autodiff

• Then, compute derivatives **backward** from the final node toward the inputs

\[
\begin{align*}
\frac{dc}{dx} & \cdot \frac{de}{dc} \cdot \frac{de}{de} = \frac{d(x^2)}{dx} \cdot 1 = 2x \\
\frac{de}{dc} & \cdot \frac{de}{de} = \frac{d(c + b)}{dc} \cdot 1 = 1 \\
\frac{de}{de} & = 1
\end{align*}
\]

\[
e = c + b = x^2 + \log y
\]
Reverse Mode Autodiff

• Then, compute derivatives \textit{backward} from the final node toward the inputs

\[ c = x^2 \]
\[ e = c + b = x^2 + \log y \]
\[ \frac{dc}{dx} \cdot \frac{de}{dc} \cdot \frac{de}{de} = \frac{d(x^2)}{dx} \cdot 1 = 2x \]
\[ \frac{de}{dc} \cdot \frac{de}{de} = \frac{d(c + b)}{dc} \cdot 1 = 1 \]
\[ \frac{de}{db} \cdot \frac{de}{de} = \frac{d(c + b)}{db} \cdot 1 = 1 \]
Reverse Mode Autodiff

- Then, compute derivatives \textit{backward} from the final node toward the inputs.

\[
\begin{align*}
\frac{dc}{dx} &= \frac{de}{dc} = \frac{d(x^2)}{dx} \cdot 1 = 2x \\
\frac{db}{dy} &= \frac{de}{db} = \frac{d(\log y)}{dy} \cdot 1 = \frac{1}{y} \\
\frac{de}{dc} &= \frac{d(c + b)}{dc} \cdot 1 = 1 \\
\frac{de}{db} &= \frac{d(c + b)}{db} \cdot 1 = 1
\end{align*}
\]

\[c = x^2, \quad b = \log y, \quad e = c + b = x^2 + \log y\]
Reverse Mode Autodiff

• Then, compute derivatives **backward** from the final node toward the inputs

\[
\frac{d}{dx} c \cdot \frac{d}{dc} e = \frac{d}{dx} (x^2) \cdot 1 = 2x
\]

\[
\frac{de}{dc} \cdot \frac{de}{de} = \frac{d}{dc} (c + b) \cdot 1 = 1
\]

\[
\frac{d}{dy} b \cdot \frac{db}{de} = \frac{d}{dy} (\log y) \cdot 1 = \frac{1}{y}
\]

\[
\frac{de}{db} \cdot \frac{de}{de} = \frac{d}{db} (c + b) \cdot 1 = 1
\]

\[c = x^2\]

\[b = \log y\]

\[e = c + b = x^2 + \log y\]

\[O(K)\] time, \(O(K)\) memory

\(N = \text{number of input features to the network}, K = \text{number of nodes in the graph}\)
Reverse Mode Autodiff is Time Efficient

• Forward mode: $O(N \times K)$ time, $O(1)$ memory
  • $N =$ number of inputs features to the network,
  • $K =$ number of nodes in the graph

• Reverse mode: $O(K)$ time, $O(K)$ memory

• The memory cost comes from having to keep the entire graph from the forward pass in order to then differentiate backwards
Recap

Gradient Descent pseudocode

Stochastic Gradient Descent

Batching

Computer based derivatives

Deep Learning Frameworks

Numeric differentiation

Symbolic differentiation

Automatic differentiation (Autodiff)
(1) Forward mode
(2) Reverse mode

Chain Rule

\[ \frac{d}{dx} f(g(h(x))) = \frac{df}{dg} \frac{dg}{dh} \frac{dh}{dx} \]