Deep Learning (BEV033DLE)
Lecture 4. Backpropagation

Alexander Shekhovtsov
Czech Technical University in Prague

✦ What should it do
  • Geometric understanding

✦ How to compute
  • Forward / backward propagation
  • Implementation
  • General DAG, total derivatives
  • Pitfalls
What is Backpropagation?
What is Backpropagation?

A) Method to learn neural networks
What is Backpropagation?

A) Method to learn neural networks
B) Method to optimize training loss
What is Backpropagation?

A) Method to learn neural networks
B) Method to optimize training loss
C) Defines the step direction for gradient descent
What is Backpropagation?

A) Method to learn neural networks
B) Method to optimize training loss
C) Defines the step direction for gradient descent
D) Rules for computing gradient of a composite function
What is Backpropagation?

A) Method to learn neural networks
B) Method to optimize training loss
C) Defines the step direction for gradient descent
D) Rules for computing gradient of a composite function
E) Computationally efficient automatic differentiation for scalar-valued composite functions
Linear Approximation to a Function

- **Function** $f: \mathbb{R}^m \to \mathbb{R}^n$
- **Local linear approximation**: $f(x + \Delta x) = f(x) + J(x)\Delta x + o(\|\Delta x\|)$

$$f(x + \Delta x) \approx f(x) + \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_m \end{pmatrix}$$

Jacobian (matrix)

- Linear approximation is sufficient for finding descent directions
  - (Steepest) Gradient Descent, Mirror-Descent
- For a sum of functions their linear approximations add up
  - SGD, Stochastic MD
- For a composition of functions their linear approximations compose

Remark: partial derivatives should be continuous in a neighborhood. If not (e.g. with ReLU) usually it is not a problem, but we will see a pitfall later.
Compositions

- **Linear function**: \( f(x) = Ax \),
  \[
  J_f = A
  \]

- **Composition of linear functions**: \( f(x) = (A \circ B)x = ABx \),
  \[
  J_f = AB
  \]

- **Composition of non-linear functions**: \( f = g \circ h \),
  \[
  J_f = J_g J_h
  \]

- **Chain Rule**: approximate every function in the composition locally around its argument and compose approximations

**Example** \( f = \sqrt{\log(x^2)} \):

Composition: \( \sqrt{} \circ \log \circ \text{pow}_2 \)

\[
J_f = \left( \frac{\partial \sqrt{z}}{\partial z} \right)_{z=\log(x^2)} \left( \frac{\partial \log y}{\partial y} \right)_{y=x^2} \left( \frac{\partial x^2}{\partial x} \right)_{x} = J_f = (z^{-1/2})(y^{-1})(2x)
\]
Consider composition of functions: \( F = f \circ g \circ h \)

Compose linear approximations: \( J_F = J_f J_g J_h \)

(Notice the order is the same)

Let \( f \) be a scalar loss: \( f : \mathbb{R}^n \rightarrow \mathbb{R} \), then

\[
J_F = \left( \frac{\partial f}{\partial g_1}, \frac{\partial f}{\partial g_2}, \ldots, \frac{\partial f}{\partial g_n} \right) \begin{pmatrix} J_g \\ J_h \end{pmatrix}
\]

- Matrix product is associative, we can choose how to group multiplications
- Going left-to-right is cheaper: \( O(Ln^2) \) vs. \( O((L - 1)n^3 + n^2) \)
- Where is backward pass?
Gradient

- Consider scalar-valued function: \( f : \mathbb{R}^n \rightarrow \mathbb{R} \)

\[
f(x + \Delta x) \approx f(x) + J\Delta x
\]

- Jacobian \( J \) is a row vector \( \left( \cdots \frac{\partial f}{\partial x_i} \cdots \right) \)
- What is the steepest descent direction for the linear approximation?

\[
\min_{\|\Delta x\|=1} \left( f(x) + J\Delta x \right) \Rightarrow \Delta x = -\frac{J^T}{\|J\|}
\]

- Gradient \( \nabla_x f \) is the column vector of partial derivatives \( \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} = J^T \)
Backpropagation on a Higher Level

✦ Summary so far:

• Composition of functions — **forward**

\[
\begin{align*}
\text{Variables } x & \rightarrow f_1 & \rightarrow f_2 & \rightarrow f_3 & \rightarrow L = f_3(f_2(f_1(x))) \in \mathbb{R}
\end{align*}
\]

• Linear approximation \( L(x + \Delta x) \approx L(x) + J_L \Delta x \)

• Compose linear approximations: \( J_L = J_3 \circ J_2 \circ J_1 \)

• Transposed for the gradient: \( \nabla_x L = J_L^T = J_1^T \circ J_2^T \circ J_3^T \)

• Go in the **backward** order multiplying one matrix-vector at a time
  (reverse mode automatic differentiation)
Exercise

Let \( f : \mathbb{R}^n \to \mathbb{R}^m \). Match concepts on the left and explanations on the right.

a) Gradient of \( f \)  
   1) A linear mapping approximating \( f \) locally around a point.

b) Derivative of \( f \)  
   2) Expression of the derivative in coordinates as a matrix.

b) Jacobian of \( f \)  
   3) Column vector of partial (or total) derivatives in case \( f \) is scalar-valued, i.e. \( m = 1 \).
Examples

◆ General procedure for back-propagating one layer $y = y(x)$
  - $L = L(y)$ – the loss function of the layer’s output (may be composite)
  - Assume already computed gradient $\nabla_y L$
  - We compute $\nabla_x L := (J_y)^\top (\nabla_y L)$, in components: $\nabla_x L = \sum_j (\frac{\partial y_j}{\partial x_i}) (\nabla y_j L)$

◆ $y = W x$
  - $\nabla_x := W^\top \nabla_y$

◆ $y = x + z$
  - $\nabla_x := \nabla_y$, $\nabla_z := \nabla_y$

◆ $y_j = \sum_k w_k x_{j+k} + b_j$
  - $\nabla_b := \nabla_y$
  - $\nabla w_k := \sum_j \frac{\partial y_j}{\partial w_k} \nabla y_j = \sum_j x_{j+k} \nabla y_j$
  - $\nabla x_i := \sum_j \frac{\partial y_j}{\partial x_i} \nabla y_j = \sum_j w_{i-j} \nabla y_j$

Various special cases of linear dependencies can be handled in $O(n)$ instead of $O(n^2)$
A detailed complete example will follow
Computation Graph, Forward Propagation

❖ Approach 1:

• Declare

```python
import torch
import torch.nn as nn

net = nn.Sequential(
    nn.Linear(748, 200),
    nn.ReLU(),
    nn.Linear(200, 10),
    nn.Softmax(),
)
```

Nothing is computed yet

• Execute it with some input (forward propagation)

```python
x = torch.randn(748)
y = net.forward(x)
```

Software already knows the graph (here sequence), what inputs and parameters each operation has and how to apply it, saves the output of each operation, may optimize the computation.
Computation Graph, Forward Propagation

✦ Approach 2:
  - Compute what we need

Declare and initialize variables

```python
from torch.nn import Parameter
import torch.nn.functional as F

W = Parameter(torch.randn(10, 748))
b = Parameter(torch.randn(10))
```

Perform some operations

```python
a = W.matmul(x) + b
y = F.softmax(a)
loss = -(t * y.log()).sum()
```

Wow! Any computation can be made a part of a neural network
For the purpose of example we will propagate these larger blocks
Backward Propagation

\[ L = -t^T \log(y) \]

\[ \nabla y_i = \frac{\partial L}{\partial y_i} = -\frac{\partial}{\partial y_i} \sum_j t_j \log(y_j) = -\frac{1}{y_i} t_i \]
Recall: \( y_j = \frac{e^{a_j}}{\sum_i e^{a_i}} \)

\[
\nabla a_i = \sum_j \frac{\partial y_j}{\partial a_i} \nabla y_j \\
= \sum_j (y_i[i=j] - y_i y_j) \nabla y_j = y_i (\nabla y_i - \sum_j y_j \nabla y_j)
\]

\[
\nabla a = (\text{Diag}(y) - yy^T) \nabla y = y \odot \nabla y - y(y^T \nabla y)
\]

(need to know either input \( a \) or directly the output \( y \))
Recall: \( a_j = \sum_i W_{ji} x_i + b \)
\( \nabla_b = \nabla_a \)

\[
\nabla x_i = \sum_j \frac{\partial a_j}{\partial x_i} \nabla a_j = \sum_j W_{ij} \nabla a_j \\
\nabla x = W^T \nabla a 
\]

\[
\nabla W_{ji} = \sum_j \frac{\partial a_j}{\partial W_{ji}} \nabla a_j = x_i \nabla a_j \\
\nabla W = (\nabla a)x^T - \text{outer (column-row) product} 
\]
Backward Propagation

Recall: \( y_j = \frac{e^{a_j}}{\sum_i e^{a_i}} \)

\[
\nabla_a = (\text{Diag}(y) - yy^T) \nabla_y = y \odot \nabla_y - y(y^T \nabla y)
\]

```python
class MySoftmax(torch.autograd.Function):
    @staticmethod
    def forward(ctx, a):
        y = a.exp()
        y /= y.sum()
        ctx.save_for_backward(y)
        return y

    @staticmethod
    def backward(ctx, dy):
        y = ctx.saved_tensors
        da = y * dy - y * (y * dy).sum()
        return da
```
Recall: \( y_j = \frac{e^{a_j}}{\sum_i e^{a_i}} \)

\[ \nabla_a = (\text{Diag}(y) - yy^T) \nabla_y = y \odot \nabla_y - y(y^T \nabla_y) \]

```python
class MySoftmax:
    def forward(self, a):
        y = a.exp()
        y /= y.sum()
        self.y = y
        return y

    def backward(self, dy):
        y = self.y
        da = y * dy - y * (y * dy).sum()
        return da

    def cleanup(self):
        del self.y
```
Consider the case when some of the inputs are used in several places.

The total derivative rule emerges:

\[
\frac{d}{db} f(b, y(b)) = \frac{\partial f}{\partial b} + \frac{\partial f}{\partial y} \frac{dy}{db}
\]

Follows from the composition:

\[
f(\bar{b}) = f(\bar{b}^1, y(\bar{b}^2))
\]

We can say then:

- Jacobian of composition = “total Jacobian”
- Gradient of a composition = “total gradient”
Need to find the order of processing

- a node may be processed when all its parents are ready
- some operations can be executed in parallel
- for the backward pass we reverse the edges
General DAG

- Any directed acyclic graph can be topologically ordered
  - Equivalent to a layered network with skip connections
  - Equivalent to a layered network with extended layer outputs
Any directed acyclic graph can be topologically ordered

- Equivalent to a layered network with skip connections
- Equivalent to a layered network with extended layer outputs

Can be made a total order, but here we do not see what can be executed in parallel
Pitfalls

✧ Discontinuous Gradients Example

A nice smooth function: \( y = \min(e^x, 1) + \max(x + 1, 1) \)
Suppose we initialized with \( x = 0 \)
Why the gradient is zero?

✧ Exponents e.g. in the softmax \( x = \frac{e^x}{\sum_i e^{x_i}} \) will overflow when \( x_i > 88.7 \)
  • may be cancelled in the numerator and denominator in advance
  (∗) logsoftmax is a more friendly function with bounded derivatives

Advanced Variants

✧ Derivatives of implicit functions
✧ Derivatives of optimization problems