Lecture 9: Optimization
• Section this week: PyTorch tutorial (for PS5 and later)
• Reading:
  - Szeliski 5.3.5
  - Lecture notes by Roger Grosse (basis of this lecture)
  - Goodfellow Deep Feedforward Network
Recall: Learning parameters

Squared loss with single-variable network:

$$L = \frac{1}{2}(f(x) - y)^2$$

$$L = \frac{1}{2}(\sigma(wx + b) - y)^2$$

Example source: Roger Grosse
Computing derivatives with the chain rule

\[
\frac{\partial L}{\partial w} = \frac{\partial}{\partial w} \left[ \frac{1}{2} (y - \sigma(wx + b))^2 \right] \\
= (y - \sigma(wx + b)) \frac{\partial}{\partial w}(y - \sigma(wx + b)) \\
= (y - \sigma(wx + b)) \sigma'(wx + b) \frac{\partial}{\partial w}(wx + b) \\
= (y - \sigma(wx + b)) \sigma'(wx + b)x
\]

\[
\frac{\partial L}{\partial b} = \frac{\partial}{\partial b} \left[ \frac{1}{2} (y - \sigma(wx + b))^2 \right] \\
= (y - \sigma(wx + b)) \frac{\partial}{\partial b}(y - \sigma(wx + b)) \\
= (y - \sigma(wx + b)) \sigma'(wx + b) \frac{\partial}{\partial b}(wx + b) \\
= (y - \sigma(wx + b)) \sigma'(wx + b)
\]
Learning parameters

Writing out the layers:

\[ z = wx + b \]

\[ t = \sigma(z) \]

\[ L = \frac{1}{2}(y - t)^2 \]

Another way to write derivatives:

\[ \frac{\partial L}{\partial t} = y - t \]

\[ \frac{\partial L}{\partial z} = \frac{\partial L}{\partial t} \frac{\partial t}{\partial z} = \frac{\partial L}{\partial t} \sigma'(z) \]

\[ \frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} x \]

\[ \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \]

Each step computable, and can reuse computation.

Example source: Roger Grosse
Computation graph: loss

Adapted from: Roger Grosse
Computation graph: derivatives

Adapted from: Roger Grosse
One subtlety: nodes with multiple outputs

\[ L = (t - y)^2 + \|w\|^2 \]
Chain rule

Single-variable chain rule:
\[
\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}
\]

Multivariable chain rule:
\[
\frac{\partial}{\partial x} f(g(x), h(x)) = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} + \frac{\partial f}{\partial h} \frac{\partial h}{\partial x}
\]
Multivariable chain rule: example

\[ f(g(x), h(x)) = g(x) + g(x)h(x) \]
\[ g(x) = x^2 \]
\[ h(x) = \exp(x) \]

\[ \frac{\partial f}{\partial x} f(g(x), h(x)) = \frac{\partial}{\partial x} [g(x) + g(x)h(x)] \]
\[ = [1 + h(x)]g'(x) + g(x)h'(x) \]
\[ = (1 + \exp(x))(2x) + x^2 \exp(x) \]
\[ = 2x + 2x \exp(x) + x^2 \exp(x) \]
Backpropagation

Step #1: Compute loss and every value in computation graph.

Forward pass:
for $i = 0 \ldots N$:
compute each node value $v_i$ using values from previous nodes

where $N$ nodes $v_0, v_1, \ldots v_n$ are ordered topographically (i.e. inputs always come before outputs).
Step #2: Compute derivatives.

Starting from the loss, work your way to the inputs. Compute the derivative of the loss w.r.t. each value.

Summary: compute derivatives efficiently using the chain rule.
What about vector-valued functions?

Can use the same algorithm, but messy…

Adapted from: Roger Grosse
Handling vectors

Vector-valued functions:

\[
\begin{bmatrix}
    z_1 \\
    z_2
\end{bmatrix}
= \begin{bmatrix}
    2w_1 + 3w_2 \\
    5w_1 - 2w_2
\end{bmatrix}
\]

Multivariable chain rule:

\[
\frac{\partial L}{\partial w_i} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_i}
\]

Computation graph:

\[
\begin{array}{c}
    w_1 \\
    w_2 \\
\end{array} \rightarrow \begin{array}{c}
    z_1 \\
    z_2 \\
\end{array}
\]

Jacobian matrix:

\[
\begin{bmatrix}
    \frac{\partial z_1}{\partial w_1} & \frac{\partial z_1}{\partial w_2} \\
    \frac{\partial z_2}{\partial w_1} & \frac{\partial z_2}{\partial w_2}
\end{bmatrix}
\]

In matrix form:

\[
\begin{bmatrix}
    \frac{\partial L}{\partial w_1} \\
    \frac{\partial L}{\partial w_2}
\end{bmatrix} = \begin{bmatrix}
    \frac{\partial z}{\partial w}
\end{bmatrix}^T \begin{bmatrix}
    \frac{\partial L}{\partial z_1} \\
    \frac{\partial L}{\partial z_2}
\end{bmatrix}
\]

Adapted from: Roger Grosse
Backward step (with vectors)

Recall step #2: Computing derivatives.

<table>
<thead>
<tr>
<th>Backward pass:</th>
</tr>
</thead>
</table>

\[
\mathbf{v}_N = 1
\]

for \( i = N-1 \ldots 0 \):

\[
\frac{\partial L}{\partial \mathbf{v}_i} = \sum_{j \in \text{Outputs}(\mathbf{v}_i)} \mathbf{v}_j^T \frac{\partial L}{\partial \mathbf{v}_j}
\]

Forward pass is the same as it was before.

Adapted from: Roger Grosse
Example: multi-layer regression

\[ \mathcal{L} = \frac{1}{2} \| W_1 \tanh(W_0 x_0) - y \|^2 \]

Linear transformation \hspace{1cm} Elementwise operation \hspace{1cm} Squared \( L_2 \) norm
Rewrite as layers

We need to compute all these terms so that we can find the gradients at the bottom.

Source: Torralba, Isola, Freeman
Pointwise function

\[
x_{out_i} = h(x_{in_i})
\]

\[
\frac{\partial L}{\partial x_{out_i}} = \frac{\partial L}{\partial x_{in_i}} \cdot h'(x_{in_i})
\]

Example #1:
\[
x_{out_i} = \tanh(x_{in_i})
\]
\[
\frac{\partial L}{\partial x_{in_i}} = \frac{\partial L}{\partial x_{out_i}} (1 - \tanh^2(x_{in_i}))
\]

Example #2:
\[
x_{out_i} = \max(0, x_{in_i})
\]

What's the backward pass?

Source: Torralba, Isola, Freeman
Implementation

\[
x_{out_i} = h(x_{in_i})
\]

\[
\frac{\partial L}{\partial x_{out_i}} = \frac{\partial L}{\partial x_{in_i}} \cdot h'(x_{in_i})
\]

```python
class ReLULayer:
    def forward(self, x):
        # save for backward pass
        self.x = x
        return np.maximum(x, 0)
    def backward(self, grad_output):
        grad_input = grad_output.clone()
        grad_input[self.x < 0] = 0
        return grad_input
```

See https://pytorch.org/tutorials/beginner/pytorch_with_examples.html for functioning code.
Squared cost layer

\[ L = \frac{1}{2} \| x_{in} - y \|^2 \]

\[ \frac{\partial L}{\partial x_{in}} = x_{in} - y \]

Source: Torralba, Isola, Freeman
If we look at the j component of output $x_{out}$, with respect to the i component of the input, $x_{in}$:

$$\frac{\partial x_{out,i}}{\partial x_{in,j}} = W_{ij} \rightarrow \frac{\partial f(x_{in}, W)}{\partial x_{in}} = W$$

Therefore:

$$\frac{\partial L}{\partial x_{in}} = \frac{\partial L}{\partial x_{out}} \cdot W$$

Source: Torralba, Isola, Freeman
Linear layer

- Forward propagation: $\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \mathbf{W}) = \mathbf{W} \mathbf{x}_{\text{in}}$

- Backprop to input:
  $$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \mathbf{W}$$

Source: Torralba, Isola, Freeman
Linear layer

- Forward propagation: $x_{out} = f(x_{in}, W) = Wx_{in}$
- Backprop to weights:
  \[
  \frac{\partial L}{\partial W} = \frac{\partial L}{\partial x_{out}} \cdot \frac{\partial f(x_{in}, W)}{\partial W} = \frac{\partial L}{\partial x_{out}} \cdot \frac{\partial x_{out}}{\partial W}
  \]

If we look at how the parameter $W_{ij}$ changes the cost, only the $i$ component of the output will change, therefore:

\[
\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial x_{out_i}} \cdot \frac{\partial x_{out_i}}{\partial W_{ij}} = \frac{\partial L}{\partial x_{out_i}} \cdot x_{in_j} \\
\frac{\partial x_{out_i}}{\partial W_{ij}} = x_{in_j}
\]

Source: Torralba, Isola, Freeman
Linear layer

\[ x_{\text{out}} = W x_{\text{in}} \]

\[ \frac{\partial L}{\partial x_{\text{out}}} \]

\[ \frac{\partial L}{\partial x_{\text{in}}} = \frac{\partial L}{\partial x_{\text{out}}} \cdot W \]

\[ \frac{\partial L}{\partial W} = x_{\text{in}} \cdot \frac{\partial L}{\partial x_{\text{out}}} \]

Source: Torralba, Isola, Freeman
Recall: multilayer model

We need to compute all these terms so that we can find the gradients at the bottom.

Source: Torralba, Isola, Freeman
First we compute the derivative of the loss with respect to the output:

$$\frac{\partial L}{\partial x_3} = x_3 - y$$

$$W_0 = \begin{pmatrix} 1 \\ 0.2 \\ 1 \end{pmatrix} \quad W_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

By the chain rule, we can derive equations, working backwards, for each remaining term we need:

$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial x_3} \frac{\partial x_3}{\partial x_2} = \frac{\partial L}{\partial x_3} W_1$$

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial x_2} \frac{\partial x_2}{\partial x_1} = \frac{\partial L}{\partial x_2} \frac{\partial \tanh(x_1)}{\partial x_1} = \frac{\partial L}{\partial x_2} (1 - \tanh^2(x_1))$$

ending up with our two gradients needed for the weight update:

$$\frac{\partial L}{\partial W_0} = \frac{\partial L}{\partial x_1} \frac{\partial x_1}{\partial W_0} = x_0 \frac{\partial L}{\partial x_1}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial x_3} \frac{\partial x_3}{\partial W_1} = x_2 \frac{\partial L}{\partial x_3}$$

Source: Torralba, Isola, Freeman
The values for input vector $x_0$ and target $y$ are also given by the first slide:

\[
x_0 = \begin{pmatrix} 1.0 \\ 0.1 \end{pmatrix} \quad y = 0.5
\]

Finally, we simply plug these values into our equations and compute the numerical updates:

**Forward pass:**

\[
x_1 = W_0 x_0 = \begin{pmatrix} 1 & -3 \\ 0.2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.3 \end{pmatrix}
\]

\[
x_2 = \tanh(x_1) = \begin{pmatrix} 0.604 \\ 0.291 \end{pmatrix}
\]

\[
x_3 = W_1 x_2 = \begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} 0.604 \\ 0.291 \end{pmatrix} = 0.313
\]

\[
\mathcal{L} = \frac{1}{2} (x_3 - y)^2 = 0.017
\]

Source: Torralba, Isola, Freeman
Backward pass:

\[
\frac{\partial L}{\partial x_3} = x_3 - y = -0.1869
\]

\[
\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial x_3} W_1 = -0.1869 (1 \quad -1) = (-0.1869 \quad 0.1869)
\]

\[
\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial x_2} (1 - \tanh^2(x_1)) = (-0.1869 \quad 0.1869) \begin{pmatrix} 1 - \tanh^2(0.7) & 0 \\ 0 & 1 - \tanh^2(0.3) \end{pmatrix} = (-0.1186 \quad 0.171)
\]

\[
\frac{\partial L}{\partial W_0} = x_0 \frac{\partial L}{\partial x_1} = \begin{pmatrix} 1.0 \\ 0.1 \end{pmatrix} (-0.1186 \quad 0.171) = \begin{pmatrix} -0.1186 \quad 0.171 \\ -0.01186 \quad 0.0171 \end{pmatrix}
\]

\[
\frac{\partial L}{\partial W_1} = x_2 \frac{\partial L}{\partial x_3} = \begin{pmatrix} 0.604 \\ 0.291 \end{pmatrix} (-0.1186) = \begin{pmatrix} -0.113 \\ -0.054 \end{pmatrix}
\]

Source: Torralba, Isola, Freeman
Gradient updates:

\[
W_{0}^{k+1} = W_{0}^{k} + \eta \left( \frac{\partial L}{\partial W_{0}} \right)^{T}
\]

\[
= \begin{pmatrix} 1 & -3 \\ 0.2 & 1 \end{pmatrix} - 0.2 \begin{pmatrix} -0.1186 & 0.171 \\ -0.01186 & 0.0171 \end{pmatrix}
\]

\[
= \begin{pmatrix} 1.02 & -3.0 \\ 0.17 & 1.0 \end{pmatrix}
\]

\[
W_{1}^{k+1} = W_{1}^{k} + \eta \left( \frac{\partial L}{\partial W_{1}} \right)^{T}
\]

\[
= \begin{pmatrix} 1 & -1 \end{pmatrix} - 0.2 \begin{pmatrix} -0.113 & -0.054 \end{pmatrix}
\]

\[
= \begin{pmatrix} 1.02 & -0.989 \end{pmatrix}
\]

Source: Torralba, Isola, Freeman
Automatic differentiation

- **Backpropagation**: algorithm for computing derivatives
- Also known as reverse-mode autodifferentiation
- Usually implement backprop using autodifferentiation ("autodiff") software package.
  - Build your program out of primitive operations, similar to *numpy* operations
  - Behind the scenes, the autodiff package builds the computation graph for you!
  - It’s *not* finite difference. Computes exact gradients using backprop!
Automatic differentiation

```python
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))

# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01
print "Trained loss:", training_loss(weights)
```

Source: Roger Grosse
Optimization

Loss:
\[ J = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(x_i, y_i) \]
Sum over N training pairs

For example, linear layer:
\[ x_{out} = W x_{in} \]
\[ \frac{\partial L}{\partial x_{out}} = \frac{\partial L}{\partial x_{in}} \cdot W \]

Gradient descent:
\[ W^{k+1} \leftarrow W^k + \eta \left( \frac{\partial J}{\partial W} \right) \]

Average over examples:
\[ \frac{\partial J}{\partial \mathbf{W}} = \frac{1}{N} \sum_{i=1}^{N} \left. \mathbf{x}_{in} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{out}} \right|_{x_i, y_i} \]
\[ \frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{x}_{in} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{out}} \]
Momentum

- Also known as the **heavy ball** method
- Remember old gradient values and “build up speed”
- Reduces oscillations
- Almost always helpful for training
- $0 \leq \beta < 1$

$$
\begin{align*}
    p^k &\leftarrow \frac{\partial J}{\partial W} + \beta p^{k-1} \\
    W^{k+1} &\leftarrow W^k - \eta p^k
\end{align*}
$$

- Adaptive methods, like Adam, “adapt” learning rate for each parameter. Can sometimes be helpful for poor scaling, but won’t cover in this class.
\[ \beta = 0 \quad \text{i.e. disable momentum} \]

Source: https://distill.pub/2017/momentum
Momentum

\[ \beta = 0.99 \]

Source: https://distill.pub/2017/momentum
Unit visualization via backprop

How much the “chameleon” score is increased or decreased by changing the image pixels.

Source: Torralba, Isola, Freeman
Unit visualization via backprop

$$\arg \max_{x^{(0)}} x_j^{(L)}$$

$$\mathbf{x}^{(0)^{k+1}} \leftarrow \mathbf{x}^{(0)^k} + \eta \frac{\partial x_j^{(L)}}{\partial \mathbf{x}^{(0)}}$$

Source: Torralba, Isola, Freeman
Unit visualization via backprop

arg max \( x_j^{(L)} + \lambda R(x^{(0)}) \)

\[ x^{(0)}_{k+1} \leftarrow x^{(0)}_k + \eta \frac{\partial (x_j^{(L)} + \lambda R(x^{(0)}))}{\partial x^{(0)}} \]

Make an image that maximizes the “cat” output neuron:

[https://distill.pub/2017/feature-visualization/]

Source: Torralba, Isola, Freeman
Unit visualization via backprop

Make an image that maximizes the value of a random neuron in the middle of the network:

$\arg \max_{\mathbf{x}^{(0)}} x_j^{(L)} + \lambda R(\mathbf{x}^{(0)})$

$\mathbf{x}^{(0)k+1} \leftarrow \mathbf{x}^{(0)k} + \eta \frac{\partial (x_j^{(L)} + \lambda R(\mathbf{x}^{(0)}))}{\partial \mathbf{x}^{(0)}}$

[https://distill.pub/2017/feature-visualization/]

Source: Torralba, Isola, Freeman
“Deep dream” [https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html]

Source: Torralba, Isola, Freeman
Next week: convolutional neural networks