CSE 559A: Computer Vision

Fall 2017: T-R: 11:30-1pm @ Lopata 101

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Staff: Abby Stylianou (abby@wustl.edu), Jarett Gross (jarett@wustl.edu)

http://www.cse.wustl.edu/~ayan/courses/cse559a/

Nov 14, 2017
GENERAL

- PSET 4 Due today

- PSET 2 & 3 Grades posted
  - (in course website directory)

- PSET 5 will be posted today(ish)
  - Will include some stuff we talk about on Thursday
  - Due two weeks from Thursday

- Projects
  - Proposals look good in general
  - Try to make sure project's on track
    - We may follow up with a few of you over email
  - Presentation schedule will be posted next week
**CLASSIFICATION**

\[ x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn} \ w} w^T \tilde{x} \xrightarrow{\text{Classify}} \begin{cases} > 0 & \text{True} \\ < 0 & \text{False} \end{cases} \]

- Hand crafted
- Automatic
- Data-driven

\[ \tilde{x} \quad ? \]

- What is an encoding such that a 'linear' classifier on it will suffice?
- Just list of pixels / quadratic (now N2 dimensional vector)?
- Kernel methods help with dimensionality, but still hand-crafted.
CLASSIFICATION

- Learn $\tilde{x} = g(x; \theta)$

$$w = \arg \min_w \frac{1}{T} \sum_t y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)]$$

$$\theta, w = \arg \min_{\theta, w} \frac{1}{T} \sum_t y_t \log [1 + \exp(-w^T g(x_t; \theta))] + (1 - y_t) \log [1 + \exp(w^T g(x_t; \theta))]$$

- Again, use (stochastic) gradient descent.
  - But this time, the cost is no longer convex.
Learn $\tilde{x} = g(x; \theta)$

$$w = \arg \min_w \frac{1}{T} \sum_t y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)]$$

$$\theta, w = \arg \min_{\theta,w} \frac{1}{T} \sum_t y_t \log [1 + \exp(-w^T g(x_t; \theta))] + (1 - y_t) \log [1 + \exp(w^T g(x_t; \theta))]$$

Again, use (stochastic) gradient descent.
- But this time, the cost is no longer convex.
- Turns out .. doesn't matter (sort of).

Recall in the previous case: (where $C_t$ is the cost of one sample)

$$\nabla_w C_t = \tilde{x}_t \left[ \frac{\exp(w^T \tilde{x}_t)}{1 + \exp(w^T \tilde{x}_t)} - y_t \right]$$

What about now?

Exactly the same, with $\tilde{x} = g(x; \theta)$ for the current value of $\theta$. 
- Learn $\tilde{x} = g(x; \theta)$

$$\theta, w = \arg\min_{\theta, w} \frac{1}{T} \sum_{t} y_t \log [1 + \exp(-w^T g(x_t; \theta))] + (1 - y_t) \log [1 + \exp(w^T g(x_t; \theta))]$$

$$\nabla_w C_t = \tilde{x}_t \left[ \frac{\exp(w^T \tilde{x}_t)}{1 + \exp(w^T \tilde{x}_t)} - y_t \right]$$

What about $\nabla_{\theta} C_t$?

First, what is the $\nabla_{\tilde{x}_t} C_t$?

$$\nabla_{\tilde{x}_t} C_t = w \left[ \frac{\exp(w^T \tilde{x}_t)}{1 + \exp(w^T \tilde{x}_t)} - y_t \right]$$
CLASSIFICATION

- Learn $\tilde{x} = g(x; \theta)$

$$\theta, w = \arg \min_{\theta, w} \frac{1}{T} \sum_t y_t \log [1 + \exp(-w^T g(x_t; \theta))] + (1 - y_t) \log [1 + \exp(w^T g(x_t; \theta))]$$

$$\nabla_{x_i} C_t = w \left[ \frac{\exp(w^T \tilde{x}_t)}{1 + \exp(w^T \tilde{x}_t)} - y_t \right]$$

- Now, let’s say $\theta$ was an $M \times N$ matrix, and $g(x; \theta) = \theta x$.
  - $N$ is the length of the vector $x$
  - $M$ is the length of the encoded vector $\tilde{x}$

What is $\nabla_\theta C_t$?

$$\nabla_\theta C_t = (\nabla_{x_i} C_t) x_t^T$$

- This is actually a linear classifier on $x$
  - $w^T \theta x = (\theta^T w)^T x = \tilde{w}^T x$

- But because of our factorization, is no longer convex.
- If we want to increase the expressive power of our classifier, $g$ has to be non-linear!
The Multi-Layer Perceptron

\[
x \xrightarrow{\theta} h \xrightarrow{\kappa} \tilde{h} \xrightarrow{\theta} \tilde{h} \xrightarrow{\kappa} y \xrightarrow{w^T} \hat{y} \xrightarrow{\sigma} p
\]

- \(\kappa\) is an "element-wise" non-linearity.
  - For example \(\kappa(x) = \sigma(x)\). More on this later.
  - Has no learnable parameters.
- \(\sigma\) is our sigmoid to convert log-odds to probability.
  \[
  \sigma(y) = \frac{\exp(y)}{1 + \exp(y)}
  \]
- Multiplication by \(\theta\) and action of \(\kappa\) is a "layer".
  - Called a "hidden" layer, because you're learning a "latent representation".
  - Don't have direct access to the true value of its outputs
  - Learning a representation that jointly with a learned classifier is optimal
CLASSIFICATION

The Multi-Layer Perceptron

\[ x \rightarrow h = \theta x \rightarrow \tilde{h}^j = \kappa(h^i) \rightarrow \tilde{h} \rightarrow y = \mathbf{w}^T \tilde{h} \rightarrow p = \sigma(y) \rightarrow p \]

- This network has learnable parameters \( \theta, w \).
- Train by gradient descent with respect to classification loss.
- What are the gradients?

Doing this manually is going to get old really fast.

Autograd

- Express complex function as a composition of simpler functions.
- Store this as nodes in a 'computation graph'
- Use chain rule to automatically back-propagate

Popular Autograd Systems: Tensorflow, Torch, Caffe, MXNet, Theano, ...

We'll write our own!
Say we want to minimize a loss $L$, that is a function of parameters and training data.

Let's say for a parameter $\theta$ we can write:

$$L = f(x); x = g(\theta, y)$$

where $y$ is independent of $\theta$, and $f$ does not use $\theta$ except through $x$.

Now, let's say I gave you the value of $y$ and the gradient of $L$ with respect to $x$.

- $x$ is an $N$-dimensional vector
- $\theta$ is an $M$-dimensional vector (if it's a matrix, just think of each element as a separate parameter)

Express $\frac{\partial L}{\partial \theta^j}$ in terms of $\frac{\partial L}{\partial x^i}$ and $\frac{\partial g(\theta, y)^i}{\partial \theta^j}$: which is the partial derivative of one of the dimensions of the outputs of $g$ with respect to one of the dimensions of its inputs.

For every $j$

$$\frac{\partial L}{\partial \theta^j} = \sum_i \frac{\partial L}{\partial x^i} \frac{\partial g(\theta, y)^i}{\partial \theta^j}$$

We can similarly compute gradients for the "other" input to $g$, i.e. $y$. 
$L = f(x, x'); x = g(\theta, y), x' = g'(\theta, y')$

Let's say a specific variable had two "paths" to the loss.

$$\frac{\partial L}{\partial \theta^j} = \sum_i \frac{\partial L}{\partial x^i} \frac{\partial g(\theta, y)^i}{\partial \theta^j} + \sum_i \frac{\partial L}{\partial x'^i} \frac{\partial g'(\theta, y')^i}{\partial \theta^j}$$
Our very own autograd system:

- Build a directed computation graph with a (python) list of nodes $G = [n_1, n_2, n_3 ...]$
- Each node is an "object" that is one of three kinds:
  - Input
  - Parameter
  - Operation ...

We will define the graph by calling functions that define functional relationships.

```python
import edf
x = edf.Input()
theta = edf.Parameter()
y = edf.matmul(theta, x)
y = edf.tanh(y)
w = edf.Parameter()
y = edf.matmul(w, y)
```
We will define the graph by calling functions that define functional relationships.

```
import edf
x = edf.Input()
theta = edf.Parameter()
y = edf.matmul(theta,x)
y = edf.tanh(y)
w = edf.Parameter()
y = edf.matmul(w,y)
```

- Each of these statements adds a node to the list of nodes.
- Operation nodes are added by matmul, tanh, etc., and are linked to previous nodes that appear before it in the list as input.
- Every node object is going to have a member element n.top which will be the value of its "output"
  - This can be an arbitrary shaped array.
- For input and parameter nodes, these top values will just be set (or updated by SGD).
- For operation nodes, the top values will be computed from the top values of their inputs.
  - Every operation node will be an object of a class that has a function called forward.
- A forward pass will begin with values of all inputs and parameters set.
- Then we will go through the list of nodes in order, and compute the value of all operation nodes.
A forward pass will begin with values of all inputs and parameters set.

Then we will go through the list of nodes in order, and compute the value of all operation nodes.

Because nodes were added in order, if we go through them in order, the tops of our inputs will be available.
Somewhere in the training loop, where the values of parameters have been set before.

And this will give us the value of the output.

But now, we want to compute "gradients".

For each "operation" class, we will also define a function `backward`.

All operation and parameter nodes will also have an element called `grad`.

We will have to then back-propagate gradients in order.