CSE 559A: Computer Vision

Fall 2018: T-R: 11:30-1pm @ Lopata 101

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Course Staff: Zhihao Xia, Charlie Wu, Han Liu

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

November 1, 2018
• Look at Proposal Feedback

• **Important:** This Friday, Office Hours will be shorter.
  - Only from 10:30AM - 11 AM (Lopata 103)
  - Recitation Next Friday

• Colloquium of Potential Interest
  - "Visualizing Scalar Data with Computational Topology and Machine Learning" - Josh Levine from UA
  - 11 AM - Noon, Friday (Lopata 101)

• Advertisement: New Course being offered next semester
  - CSE 659A: Advances in Computer Vision
MACHINE LEARNING

\[ w = \arg \min_w \frac{1}{T} \sum_t C_t(w) \]

\[ C_t(w) = y_t \log[1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log[1 + \exp(w^T \tilde{x}_t)] \]

- Defined linear classifier on augmented vector \( \tilde{x} \)
- Used gradient descent to learn \( w \).
  - Looked at behavior of gradients.
  - Simplified computation with stochasticity.
- At test time, sign of \( w^T \tilde{x} \) gives us our label.

This is for binary classification. What about the multi-class case? \( y \in \{1, 2, 3, \ldots C\} \)
Multi-Class Classification

- Want to map an input $x$ to a class label $y \in \{1, 2, 3, \ldots, C\}$
- Binary case: $f$ outputs a single number between 0,1 that represents $P(y = 1)$.
- Multi-class case: $f$ outputs a $C$ dimensional vector that represents a probability distribution over $C$ classes.

$$f(x; W) = \text{SoftMax}(W^T \tilde{x}) = [p_1, p_2, p_3, \ldots p_C]^T$$

- Here our learnable parameter is now the $N \times C$ matrix $W$ ($N$ is length of feature vector $\tilde{x}$).
- $p_i$ represents the probability of class $i$
- Each $p_i > 0$, and $\sum_i p_i = 1$
- SoftMax is a generalization of Sigmoid

$$[p_1, p_2, \ldots]^T = \text{SoftMax}([l_1, l_2, \ldots]^T) \rightarrow p_i = \frac{\exp(l_i)}{\sum_{i'} \exp(l_{i'})}$$

- At Test Time: $y = \arg \max_i p_i$
- $y = \arg \max_i l_i$
Multi-Class Classification

\[ f(x; W) = \text{SoftMax}(W^T \tilde{x}) = [p_1, p_2, p_3, \ldots p_C]^T \]
\[ [p_1, p_2, \ldots]^T = \text{SoftMax}([l_1, l_2, \ldots]^T) \rightarrow p_i = \frac{\exp(l_i)}{\sum_{i'} \exp(l_{i'})} \]

What about the Loss?

Multi-Class Cross Entropy Loss

\[ L(y, f(x)) = L(y, [p_1, p_2, \ldots]^T) = -\log p_y \]

- Another way to write it:
  - \( y^1 = [\delta_1, \delta_2, \ldots], \) where \( \delta_i = 1 \) if \( y = i \) and 0 otherwise.
  - Called a 1-Hot encoding of the class
  - \( y^1 \) also represents a "probability distribution", where the right class has probability 1.
  - In some cases, if you have uncertainty in your training data, \( y^1 \) could be a distribution too.

\[ L(y^1 = [\delta_1, \delta_2, \ldots], [p_1, p_2, \ldots]^T) = -\sum_i \delta_i \log p_i \]
Multi-Class Classification

\[ [l_1, l_2, \ldots]^T = W^T \tilde{x} \]
\[ p_i = \frac{\exp(l_i)}{\sum_{i'} \exp(l_{i'})} \]
\[ L([\delta_1, \delta_2, \ldots], [p_1, p_2, \ldots]^T) = -\sum_i \delta_i \log p_i \]

- We're going to use gradient descent to learn \( W \). What is \( \nabla_W L \)?
- First, what is \( \frac{\partial L}{\partial l_i} \)? Take 5 mins.
- Derivative is \( p_i - \delta_i \)
  - This means that you'll get gradients for all classes (not just the true class)
  - Negative gradient wants you to increase probability for right class, and decrease for other classes
- What is \( \nabla_W L \)? Take a few minutes!

\[ \nabla_W L = \tilde{x} \ [p_1 - \delta_1, p_2 - \delta_2, \ldots] \]

This is a matrix multiply or outer-product of an \( N \times 1 \) vector with an \( 1 \times C \) vector.
For regression and both binary and multi-class classification:

- Defined linear classifier on augmented vector $\tilde{x}$
- Run optimization to learn parameters

The problem is:

- The definition of augmented vector $\tilde{x}$ is hand-crafted
- We have manually engineered our features.
- The only thing we're learning is a linear classifier on top.

Want to learn the features themselves!

Given that SGD works, what's stopping us from learning a function $g$ such that $g(x) = \tilde{x}$?
CLASSIFICATION

\[ x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn } w} w^T \tilde{x} \xrightarrow{\text{Classify}} \]

- > 0 True
- < 0 False

\[ \tilde{x} ? \]

Cat or not Cat?

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

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

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

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 $\theta$. 
CLASSIFICATION

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

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

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

What about $\nabla_\theta C_t$?

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

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

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

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

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 $\nabla_\theta C_t$?

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

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!
CLASSIFICATION

The Multi-Layer Perceptron

\[ x \overset{\theta x}{\longrightarrow} h \overset{\kappa(h)}{\longrightarrow} \tilde{h} \overset{w^T \tilde{h}}{\longrightarrow} y \overset{\sigma(y)}{\longrightarrow} p \]

- \( \kappa \) 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
The Multi-Layer Perceptron

\[
\begin{align*}
x & \rightarrow \ h = \theta x \\
\tilde{h}^j & = \kappa(h^j) \\
\tilde{h} & \rightarrow \ y = w^T \tilde{h} \\
\rightarrow \ p & = \sigma(y)
\end{align*}
\]

- This is a neural network:
  - A complex function formed by *composition* of "simple" linear and non-linear functions.
- 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:

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

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 its 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. $y$. 


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
def 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.

```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)
```

- 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.

```
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)
```

- 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.