GENERAL

- 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

**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? \( y \in \{1, 2, 3, \ldots, C\} \)

\[
\begin{align*}
    w &= \arg \min_w \frac{1}{T} \sum_z C_t(w) \\
    C_t(w) &= y_t \log [1 + \exp(-w^T \tilde{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \tilde{x}_t)]
\end{align*}
\]

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

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

- 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.

**Multi-Class Classification**

Want to map an input \( x \) to a class label \( y \in \{1, 2, 3, \ldots, C\} \)

- Binary case: \( f(x; W) = \text{Sigmoid}(W^T \tilde{x}) \)
- Multi-class case: \( f(x; W) \) 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 \( \sum p_i = 1 \)

**SoftMax is a generalization of Sigmoid**

\[
[p_1, p_2, \ldots]^T = \text{SoftMax}(l_1, l_2, \ldots)^T \rightarrow p_i = \frac{\exp(l_i)}{\sum \exp(l_i)}
\]

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

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

\[ [p_1, p_2, \ldots]^T = \text{SoftMax}([l_1, l_2, \ldots]^T) \rightarrow p_i = \frac{\exp(l_i)}{\sum_i \exp(l_i)} \]

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

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

For regression and both binary and multi-class classification:

- Defined linear classifier on augmented vector \( \hat{x} \)
- Run optimization to learn parameters

The problem is:

- The definition of augmented vector \( \hat{x} \) 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 \( g(x) = \hat{x} \)?
Learn $\tilde{x} = g(x; \theta)$

$$w = \arg \min \frac{1}{T} \sum \gamma_i \log [1 + \exp(-w^T \tilde{x}_i)] + (1 - \gamma_i) \log [1 + \exp(w^T \tilde{x}_i)]$$

$$\theta, w = \arg \min \frac{1}{T} \sum \gamma_i \log [1 + \exp(-w^T g(x_i; \theta))] + (1 - \gamma_i) \log [1 + \exp(w^T g(x_i; \theta))]$$

- 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 $C_i$ is the cost of one sample)

$$\nabla_{\theta} C_i = \tilde{x}_i \left[ \frac{\exp(w^T \tilde{x}_i)}{1 + \exp(w^T \tilde{x}_i)} - y_i \right]$$

What about now?

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

---

The Multi-Layer Perceptron

$$x \xrightarrow{\theta} h \xrightarrow{\sigma} \tilde{h} \xrightarrow{\sigma} y \xrightarrow{\sigma \circ} p$$

- $x$ is an "element-wise" non-linearity.
  - For example $\sigma(x) = \sigma(x)$. 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 $x$ 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

- 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 / BACK-PROPAGATION

- 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 its a matrix, just think of each element as a separate parameter)

Express $\frac{dL}{d\theta}$ in terms of $\frac{dL}{dx}$ and $\frac{dL}{d\theta}^j$ - 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 $j$

$$\frac{dL}{d\theta}^j = \sum_i \frac{dL}{dx} \cdot \frac{dL}{d\theta}^i$$

We can similarly compute gradients for the "other" input to $g$, i.e. $y$.

### AUTOGRAD / BACK-PROPAGATION

Our very own autograd system:

- Build a directed computation graph with a (python) list of nodes $G = [n1,n2,n3 ...]$.
- 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.

Somewhere in the training loop, where the values of parameters have been set before.

```
x.set(...)  
edf.Forward()  
print(y.top)
```

- 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.

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.