GENERAL

- Look at your proposal feedback!
- Problem Set 4 due Today.
- Problem set 5 will be out by tonight.
- No class Thursday. No office hours on Friday.

CLASSIFICATION

\[ x \xrightarrow{\text{Encode}} \tilde{x} \]
CLASSIFICATION

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

CLASSIFICATION

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

CLASSIFICATION

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

CLASSIFICATION

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

\[ \text{Automatic} \]
\[ \text{Data-driven} \]

\[ \text{Hand crafted} \]
\[ \text{Automatic} \]
\[ \text{Data-driven} \]
CLASSIFICATION

$x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn } \mathbf{w}} w^T \tilde{x} \xrightarrow{\text{Classify}} > 0 \text{ True} < 0 \text{ False}$

Hand crafted
Automatic
Data-driven

Cat or not Cat?

CLASSIFICATION

$x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn } \mathbf{w}} w^T \tilde{x} \xrightarrow{\text{Classify}} > 0 \text{ True} < 0 \text{ False}$

Hand crafted
Automatic
Data-driven

Cat or not Cat?

CLASSIFICATION

$x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn } \mathbf{w}} w^T \tilde{x} \xrightarrow{\text{Classify}} > 0 \text{ True} < 0 \text{ False}$

Hand crafted
Automatic
Data-driven

Cat or not Cat?

CLASSIFICATION

$x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn } \mathbf{w}} w^T \tilde{x} \xrightarrow{\text{Classify}} > 0 \text{ True} < 0 \text{ False}$

Hand crafted
Automatic
Data-driven

$	ilde{x}?$

Cat or not Cat?

CLASSIFICATION

$x \xrightarrow{\text{Encode}} \tilde{x} \xrightarrow{\text{Learn } \mathbf{w}} w^T \tilde{x} \xrightarrow{\text{Classify}} > 0 \text{ True} < 0 \text{ False}$

Hand crafted
Automatic
Data-driven

$	ilde{x}?$

Cat or not Cat?

- What is an encoding such that a 'linear' classifier on it will suffice?
Learn an binary classification on its output.
Again, use (stochastic) gradient descent.
But this time, the cost is no longer convex.

\[
\begin{align*}
\tilde{x} &= \text{arg log } [1 + \exp(-g(x; \theta))] + (1 - y) \log[1 + \exp(w^T \tilde{x})] \\
\tilde{x} &= \text{arg } [1 \text{ + } \exp(w^T \tilde{x})] + (1 - y) \log[1 + \exp(w^T \tilde{x})]
\end{align*}
\]

Again, use (stochastic) gradient descent.
But this time, the cost is no longer convex.

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

What about now?
Exactly the same, with \( \tilde{x} = g(x; \theta) \) for the current value of \( \theta \).
Take 5 mins!

What about $\nabla_\theta C_t$?
First, what is the $\nabla_\theta C_t$?

What about $\nabla_\theta C_t$?
First, what is the $\nabla_\theta C_t$?

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 $\nabla_\theta C_t$?
Take 5 mins!
Learn 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 $\nabla_{\theta} C_t$?

$$\nabla_{\theta} C_t = \nabla_{\tilde{x}} C_t$$

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

$x \xrightarrow{h=\theta x} h$
The Multi-Layer Perceptron

\[ x \xrightarrow{h} h = \theta x \xrightarrow{\tilde{h}} \tilde{h} = \kappa(\tilde{h}) \xrightarrow{y} y = w^T \tilde{h} \xrightarrow{p} p \]

- \(\kappa\) is an "element-wise" non-linearity.
  - For example \(\kappa(x) = \sigma(x)\). More on this later.
  - Has no learnable parameters.

\[ \sigma(y) = \frac{\exp(y)}{1 + \exp(y)} \]

- 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

- 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 $\frac{\partial \theta^j y^i}{\partial \theta^j}$: 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. $y$.

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)
```
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.

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

```python
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.
Code from `pset5/mnist.py`

```python
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()

W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp, W1)
y = edf.add(y, B1)
y = edf.RELU(y)
y = edf.matmul(y, W2)
y = edf.add(y, B2) # This is our final prediction
```

**BRIEF DETOUR**

- What are RELUs?

  RELU(x) = \text{max}(0, x)

  What is \frac{\partial \text{RELU}(x)}{\partial x}?

  - 0 if \(x < 0\), 1 otherwise.

  - So your gradients are passed un-changed if the input is positive.
  - But completely attenuated if the input is negative.
  - So there's still the possibility of your optimization getting stuck, if you reach a point where all inputs to the \text{RELU} are negative.
  - Will talk about initialization, etc. later.

- Previous activations would be sigmoid-like: \(\sigma(x) = \frac{\exp(x)}{1 + \exp(x)}\)
  - Great when you want a probability, bad when you want to learn by gradient descent.
  - For both high and low-values of \(x\), \(\sigma(x)/\sigma = 0\)
  - So if you weren't careful, you would end up with high-magnitude activations, and the network stops learning.

  Gradient descent is very fragile!
Code from `pset5/edf.py`

```python
ops = []; params = []; values = []
...
class Param:
    def __init__(self):
        params.append(self)
    ...
class Value:
    def __init__(self):
        values.append(self)
    ...
class matmul:
    def __init__(self, x, y):
        ops.append(self)
        self.x = x
        self.y = y
```

When you construct an object, it just does book-keeping!

```python
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()
W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp, W1)
y = edf.relu(y)
y = edf.matmul(y, W2)
y = edf.relu(y)
y = edf.matmul(y, W2)
y = edf.add(y, B2) # This is our final prediction
```

```python
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()
W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp, W1)
y = edf.relu(y)
y = edf.matmul(y, W2)
y = edf.relu(y)
y = edf.matmul(y, W2)
y = edf.add(y, B2) # This is our final prediction
```
# Inputs and parameters
inp = edf.Value()
lab = edf.Value()
W1 = edf.Param()
B1 = edf.Param()
W2 = edf.Param()
B2 = edf.Param()

# Model
y = edf.matmul(inp, W1)
y = edf.add(y, B1)
y = edf.RELU(y)
y = edf.matmul(y, W2)
y = edf.add(y, B2) # This is our final prediction
At this point, we've just defined the graph; no actual computation has happened!
Now let's train this thing!

Beginning of training:

```
    nHidden = 1024; K = 10
    W1.set(xavier((28*28,nHidden)))
    B1.set(np.zeros((nHidden)))
    W2.set(xavier((nHidden,K)))
    B2.set(np.zeros((K)))
```

Initialize weights randomly.

In each iteration of training:

```
    for iters in range(...):
        ... 
        inp.set(train_im[idx[b:b+BSZ],:])
        lab.set(train_lb[idx[b:b+BSZ]])
        ... 
```

Load data into the 'values' or inputs.

What is this set function anyway?

```
set is the only function that the classes param and value have.
```

```
class Value:
    def __init__(self):
        values.append(self)
    def set(self,value):
        self.top = np.float32(value).copy()

class Param:
    def __init__(self):
        params.append(self)
    def set(self,value):
        self.top = np.float32(value).copy()
```

Sets a member called “top” to be an array that holds these values.

• mean will be across a batch.
• accuracy is actual accuracy of hard predictions (not differentiable).

Note that we are loading our input data in batches as matrices.
• inp is BSZ x N
• Then we’re doing a matmul with W1 which is N x nHidden.
• Output will be BSZ x nHidden
• Essentially, we’re replacing vector matrix multiply for a single sample with matrix-matrix multiply for a batch of samples.
And this will work. It will print the loss and accuracy values for the set inputs, given the current value of the parameters.

What is this magical function forward?

From edf.py

```python
# Global forward
def Forward():
    for c in ops: c.forward()
```

But the operation classes have their own forward function.

```python
class matmul:
    def __init__(self, x, y):
        ops.append(self)
        self.x = x
        self.y = y
    def forward(self):
        self.top = np.matmul(self.x.top, self.y.top)
    . . .
```
So the forward pass computes the loss. But we want to learn the parameters. The SGD function is pretty simple.

```
for iters in range(...):
    ...
    inp.set(train_im[idx[b:b+BSZ],:])
    lab.set(train_lb[idx[b:b+BSZ]])
    edf.Forward()
    print(loss.top, acc.top)
    edf.Backward(loss)
    edf.SGD(lr)
```

The SGD function requires gradients with respect to loss to be present.

That's what backward does!