GENERAL

- Proposal feedback out.
  - Do a git pull on your proposal repository.
  - Look at it NOW!
- Problem set 3 keys also out (do a pull to look at them).

GRADIENT DESCENT

- Last Time: We talked about gradient descent as a way of optimizing cost functions that weren't simple linear least squares.
- Specifically, for the case of logistic regression (sigmoid-based $f$ and a cross-entropy loss).

$$
\nabla_w C(w) = \begin{bmatrix}
\frac{\partial}{\partial w_1} C(w) \\
\frac{\partial}{\partial w_2} C(w) \\
\vdots
\end{bmatrix}
$$

If $C(w) = \frac{1}{T} \sum_t C_t(w)$, then

$$
\nabla_w C(w) = \frac{1}{T} \sum_t \nabla_w C_t(w)
$$

Logistic Regression

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

What is $\nabla_w C_t(w)$, the gradient of the loss from a single training example?
\[ C_t(w) = y_i \log[1 + \exp(-w^T \hat{x}_i)] + (1 - y_i) \log[1 + \exp(w^T \hat{x}_i)] \]

Ok, what is the derivative of
\[ C_t(p) = y_i \log[1 + \exp(-p)] + (1 - y_i) \log[1 + \exp(p)] \]
with respect to \( p \) (where \( p \) is a scalar).

Take 5 mins!

\[ \frac{\partial}{\partial p} C_t(p) = \frac{\exp(p)}{1 + \exp(p)} - y_i \]

\[ \frac{\partial}{\partial p} C_t(p) = y_i \frac{-\exp(-p)}{1 + \exp(-p)} + (1 - y_i) \frac{\exp(p)}{1 + \exp(p)} \]

\[ = \frac{\exp(p)}{1 + \exp(p)} - y_i \left[ \frac{\exp(-p)}{1 + \exp(-p)} + \frac{\exp(p)}{1 + \exp(p)} \right] \]

\[ = \frac{\exp(p)}{1 + \exp(p)} - y_i \left[ \frac{1}{1 + \exp(-p)} + \frac{\exp(p)}{1 + \exp(p)} \right] \]

Also, changing \( p \) makes a much bigger difference in the corresponding probability, when \( p \) is near 0 / probability near 0.5.
GRADIENT DESCENT

\[ C_t(w) = y_i \log [1 + \exp(-w^T \hat{x}_i)] + (1 - y_i) \log [1 + \exp(w^T \hat{x}_i)] \]

\[ C_t(p) = y_i \log [1 + \exp(-p)] + (1 - y_i) \log [1 + \exp(p)] \]

\[ \frac{\partial}{\partial p} C_t(p) = \frac{\exp(p)}{1 + \exp(p)} - y_i \]

But this is still derivative with respect to \( p \). We want gradient with respect to \( w \).

\[ \frac{\partial}{\partial w} C_t(w) =? \times \left[ \frac{\exp(w^T \hat{x}_i)}{1 + \exp(w^T \hat{x}_i)} - y_i \right] \]

GRADIENT DESCENT

\[ C_t(w) = y_i \log [1 + \exp(-w^T \hat{x}_i)] + (1 - y_i) \log [1 + \exp(w^T \hat{x}_i)] \]

\[ C_t(p) = y_i \log [1 + \exp(-p)] + (1 - y_i) \log [1 + \exp(p)] \]

\[ \frac{\partial}{\partial p} C_t(p) = \frac{\exp(p)}{1 + \exp(p)} - y_i \]

But this is still derivative with respect to \( p \). We want gradient with respect to \( w \).

\[ \frac{\partial}{\partial w} C_t(w) = \hat{y}_i \times \left[ \frac{\exp(w^T \hat{x}_i)}{1 + \exp(w^T \hat{x}_i)} - y_i \right] \]

\[ \nabla_w C_t(w) =? \]

GRADIENT DESCENT

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log [1 + \exp(-w^T \hat{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \hat{x}_t)] \]

Putting it together:

- At each iteration \( i \),
  - Based on current \( w \), compute \( f(x_t, w) = \hat{y}_t \)
  - Compute derivative of the "output" as \( \hat{y}_t - y_t \)
  - Multiply by \( x_t \) to get \( \nabla_w \)
  - Change \( w \) by subtracting some \( \gamma \) times this gradient.
**GRADIENT DESCENT**

\[ w = \arg \min_w \frac{1}{T} \sum_{t=1}^{T} y_t \log [1 + \exp(-w^T \hat{x}_t)] + (1 - y_t) \log [1 + \exp(w^T \hat{x}_t)] \]

Putting it together:

- At each iteration \( i \),
  - Based on current \( w \), compute \( f(x_i, w) = \hat{y}_i \) for every training sample
  - Compute derivative of the “output” as \( \hat{y}_i - y_i \) for every training sample
  - Multiply by \( x_i \) and average all training samples to get \( \nabla w \)
  - Change \( w \) by subtracting some \( \gamma \) times this gradient.

\[ C(w) = \frac{1}{T} \sum_t C_t(w) \Rightarrow \nabla_w C = \frac{1}{T} \sum_t \nabla_w C_t \]

Expensive when we have a LOT of training data.

**STOCHASTIC GRADIENT DESCENT**

\[ w = \arg \min_w \frac{1}{T} \sum_{t} C(x_t, y_t; w) \]

\[ \nabla_w = \frac{1}{T} \sum_{t} \nabla_w C(x_t, y_t; w) \]

Remember, summation over training samples meant to approximate an expectation over \( P_{XY} (x, y) \).

\[ \frac{1}{T} \sum_{t} C(x_t, y_t; w) \rightarrow E_{P_{XY}(x,y)} C(x,y;w) \]

\[ \frac{1}{T} \sum_{t} \nabla_w C(x_t, y_t; w) \rightarrow E_{P_{XY}(x,y)} \nabla_w C(x,y;w) \]

In other words, we are approximating the “true” gradient with gradients over samples.

What if we used a smaller number of samples in each iteration, but different samples in different iterations?

**STOCHASTIC GRADIENT DESCENT**

- Single sample
  \[ w_{i+1} \leftarrow w_i - \gamma \nabla_w C_t(x_t, y_t; w_i) \]

  At each iteration, choose a random \( t \in \{1, 2, \ldots, T\} \).

- “Mini”-batched SGD (sometimes GD is called Batched GD)
  \[ w_{i+1} \leftarrow w_i - \gamma \nabla_w C_t(x_t, y_t; w_i) \frac{1}{B} \sum_{t \in B} C_t(x_t, y_t; w_i) \]

  At each iteration, choose a random smaller batch \( B \) of size \( B \ll T \).

With replacement? Without replacement?

In practice:

- Shuffle order of training examples
- Choose a batch size
- Take consecutive groups of \( B \) samples as you loop through iterations
  - \([1,8]\) in iteration 1
  - \([B+1,2B]\) in iteration 2
  - …
- Once you reach the end of the training set (called one “epoch”), shuffle the order again.
STOCHASTIC GRADIENT DESCENT

\[ w_{i+1} \leftarrow w_i - \gamma \frac{1}{B} \sum_{t \in B} \nabla_w C_t(x_t, y_t; w_i) \]

General Notes

- The gradient over a mini-batch is an "approximation", or a "noisy" version of the gradient over the true training set.
  \[ \frac{1}{B} \sum_{t \in B} \nabla_w C_t(x_t, y_t; w_i) = \frac{1}{T} \sum_{t=1}^{T} \nabla_w C_t(x_t, y_t; w_i) + \epsilon \]

- Typically, if you decrease the batch-size, you will want to decrease your step size (because you are "less sure" about the gradient).

STOCHASTIC GRADIENT DESCENT

\[ w_{i+1} \leftarrow w_i - \gamma \frac{1}{B} \sum_{t \in B} \nabla_w C_t(x_t, y_t; w_i) \]

General Notes

- Say your cost function is convex, and you care only about decreasing this cost (not worried about overfitting)
  - Larger batch size will always give you "better" gradients.
  - But diminishing returns after a batch size.
  - Computational cost is number of examples per iteration \( \times \) number of iterations for convergence
    - Higher batch means more computation per iteration, but may mean fewer iterations required to converge.
    - Best combination of step size and batch size is an empirical question.
    - Another factor: parallelism.
      - Note that you can compute the gradient of all samples of your batch in parallel.
      - Ideally, you want to at least "satrate" all available parallel threads.

STOCHASTIC GRADIENT DESCENT

Momentum

Standard SGD:

\[ g_{i+1} = \frac{1}{B} \sum_{t \in B} \nabla_w C_t(x_t, y_t; w_i) \]

\[ w_{i+1} \leftarrow w_i - \gamma g_{i+1} \]

With Momentum:

For \( \beta < 1 \):

\[ g_{i+1} = \frac{1}{B} \sum_{t \in B} \nabla_w C_t(x_t, y_t; w_i) + \beta g_i \]

\[ w_{i+1} \leftarrow w_i - \gamma g_{i+1} \]

- Keep adding the gradient from a previous batch, again and again across iterations, with decaying weight.
- Remember: \( g_i \) was computed with respect to a different position in \( w \) space.
- People often use \( \beta \) as high as 0.9 or 0.99.
- Will need to revisit "best" value of \( \gamma \) when you change \( \beta \).
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.

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

Multi-Class Classification

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

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

At Test Time: \( y = \arg \max_i l_i \)

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.

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

Here our learnable parameter is now the \( N \times C \) matrix \( W \) (\( N \) is length of feature vector \( \hat{x} \)).

- \( p_i \) represents the probability of class \( i \)
- Each \( p_i > 0 \), and \( \sum p_i = 1 \)
- \( \text{SoftMax} \) is a generalization of \( \text{Sigmoid} \)

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

We’re going to use gradient descent to learn \( W \). What is \( \nabla_W L \)?

- First, what is \( \frac{\partial L}{\partial W} \)? 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!

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

This is a matrix multiply or outer-product of an \( N \times 1 \) vector with a \( 1 \times C \) vector.

Multi-Class Classification

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

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

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

This is for binary classification. What about the multi-class case? \( y \in \{1, 2, 3, \ldots C\} \)
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 $g(x) = \tilde{x}$?