Training Neural Nets

COMPSCI 371D — Machine Learning
Outline

1. The Softmax Simplex
2. Loss and Risk
3. Back-Propagation
4. Stochastic Gradient Descent
The Softmax Simplex

• Neural-net classifier: $\hat{y} = h(x) : X \subseteq \mathbb{R}^d \rightarrow Y$

• The last layer of a neural net used for classification is a soft-max layer

$$p = \sigma(z) = \frac{\exp(z)}{1^T \exp(z)}$$

• The net is $p = f(x, w) : X \rightarrow P$

• The classifier is $\hat{y} = h(x) = \arg \max p = \arg \max f(x, w)$

• $P$ is the set of all nonnegative real-valued vectors $p \in \mathbb{R}^e$ whose entries add up to 1 (with $e = |Y|$):

$$P \overset{\text{def}}{=} \{ p \in \mathbb{R}^e : p \geq 0 \text{ and } \sum_{c=1}^{e} p_c = 1 \}.$$
The Softmax Simplex

\[ P \overset{\text{def}}{=} \{ \mathbf{p} \in \mathbb{R}^e : \mathbf{p} \geq 0 \text{ and } \sum_{i=1}^e p_i = 1 \} \]

- Decision regions are polyhedral:
  \[ P_c = \{ p_c \geq p_j \text{ for } j \neq c \} \text{ for } c = 1, \ldots, e \]
- A network transforms images into points in \( P \)
Loss and Risk (Déjà Vu)

- Ideal loss would be 0-1 loss on network output $\hat{y}$
- 0-1 loss is constant where it is differentiable!
- Not useful for computing a gradient
- Use cross-entropy loss on the softmax output $p$ as a proxy loss
  \[ \ell(y, p) = - \log p_y \]
- Risk, as usual:
  \[ L_T(w) = \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) \quad \text{where} \quad \ell_n(w) = \ell(y_n, f(x_n, w)) \]
- We need $\nabla L_T(w)$ and therefore $\nabla \ell_n(w)$
Back-Propagation

- We need $\nabla L_T(w)$ and therefore $\nabla \ell_n(w) = \frac{\partial \ell_n}{\partial w}$
- Computations from $x$ to $\ell_n$ form a \textbf{chain}
- Apply the \textbf{chain} rule
- Every derivative of $\ell_n$ w.r.t. layers before $k$ goes through $x^{(k)}$
  \[
  \frac{\partial \ell_n}{\partial w^{(k)}} = \frac{\partial \ell_n}{\partial x^{(k)}} \frac{\partial x^{(k)}}{\partial w^{(k)}}
  \]
  \[
  \frac{\partial \ell_n}{\partial x^{(k-1)}} = \frac{\partial \ell_n}{\partial x^{(k)}} \frac{\partial x^{(k)}}{\partial x^{(k-1)}} \quad \text{(recursion!)}
  \]
- Start: $\frac{\partial \ell_n}{\partial x^{(K)}} = \frac{\partial \ell}{\partial p}$
Local Jacobians

- Local computations at layer $k$: \[ \frac{\partial x^{(k)}}{\partial w(k)} \text{ and } \frac{\partial x^{(k)}}{\partial x^{(k-1)}} \]
- Partial derivatives of $f^{(k)}$ with respect to layer weights and input to the layer
- Local Jacobian matrices, can compute by knowing what the layer does
- The start of the process can be computed from knowing the loss function, \[ \frac{\partial \ell_n}{\partial x^{(k)}} = \frac{\partial \ell}{\partial p} \]
- Another local Jacobian
- The rest is going recursively from output to input, one layer at a time, accumulating \[ \frac{\partial \ell_n}{\partial w(k)} \] into a vector \( \frac{\partial \ell_n}{\partial w} \)
The Forward Pass

- All local Jacobians, \( \frac{\partial x^{(k)}}{\partial w^{(k)}} \) and \( \frac{\partial x^{(k)}}{\partial x^{(k-1)}} \), are computed numerically for the current values of weights \( w^{(k)} \) and layer inputs \( x^{(k-1)} \).
- Therefore, we need to know \( x^{(k-1)} \) for training sample \( n \) and for all \( k \).
- This is achieved by a forward pass through the network: Run the network on input \( x_n \) and store \( x^{(0)} = x_n, x^{(1)}, \ldots \).
Back-Propagation Spelled Out for $K = 3$

\[
\begin{align*}
\mathbf{x}_n &= \mathbf{x}^{(0)} \\
\mathbf{f}^{(1)}(\mathbf{x}^{(1)}) &= \mathbf{w}^{(1)} \\
\mathbf{f}^{(2)}(\mathbf{x}^{(2)}) &= \mathbf{w}^{(2)} \\
\mathbf{f}^{(3)}(\mathbf{x}^{(3)}) &= \mathbf{w}^{(3)} \\
\mathbf{p} &= \mathbf{y}_n \\
\ell &= \ell_n
\end{align*}
\]

(after forward pass)

\[
\frac{\partial \ell_n}{\partial \mathbf{x}^{(3)}} = \frac{\partial \ell}{\partial \mathbf{p}}
\]

\[
\frac{\partial \ell_n}{\partial \mathbf{w}^{(3)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(3)}} \frac{\partial \mathbf{x}^{(3)}}{\partial \mathbf{w}^{(3)}}
\]

\[
\frac{\partial \ell_n}{\partial \mathbf{x}^{(2)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(3)}} \frac{\partial \mathbf{x}^{(3)}}{\partial \mathbf{x}^{(2)}}
\]

\[
\frac{\partial \ell_n}{\partial \mathbf{w}^{(2)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(2)}} \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{w}^{(2)}}
\]

\[
\frac{\partial \ell_n}{\partial \mathbf{x}^{(1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(2)}} \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(1)}}
\]

\[
\frac{\partial \ell_n}{\partial \mathbf{w}^{(1)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(1)}} \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{w}^{(1)}}
\]

\[
\left( \frac{\partial \ell_n}{\partial \mathbf{x}^{(0)}} = \frac{\partial \ell_n}{\partial \mathbf{x}^{(1)}} \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{x}^{(0)}} \right)
\]

\[
\frac{\partial \ell_n}{\partial \mathbf{w}} = \begin{bmatrix}
\frac{\partial \ell_n}{\partial \mathbf{w}^{(1)}} \\
\frac{\partial \ell_n}{\partial \mathbf{w}^{(2)}} \\
\frac{\partial \ell_n}{\partial \mathbf{w}^{(3)}} 
\end{bmatrix}
\]

(Jacobians in blue are local, those in red are what we want eventually)
Computing Local Jacobians

\[
\begin{align*}
\frac{\partial x^{(k)}}{\partial w^{(k)}} \quad \text{and} \quad \frac{\partial x^{(k)}}{\partial x^{(k-1)}}
\end{align*}
\]

- Easier to make a “layer” as simple as possible
- \( z = Vx + b \) is one layer (Fully Connected (FC), affine part)
- \( z = \rho(x) \) (ReLU) is another layer
- Softmax, max-pooling, convolutional,...
Local Jacobians for a FC Layer

\[ \mathbf{z} = \mathbf{Vx} + \mathbf{b} \]

- \[ \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{V} \] (easy!)
- \[ \frac{\partial \mathbf{z}}{\partial \mathbf{w}} \]: What is \[ \frac{\partial \mathbf{z}}{\partial \mathbf{V}} \]? Three subscripts: \[ \frac{\partial z_i}{\partial v_{jk}} \]. A 3D tensor?
- For a general package, tensors are the way to go
- Conceptually, it may be easier to vectorize everything:

\[ \mathbf{V} = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \rightarrow \quad \mathbf{w} = [v_{11}, v_{12}, v_{13}, v_{21}, v_{22}, v_{23}, b_1, b_2]^T \]

- \[ \frac{\partial \mathbf{z}}{\partial \mathbf{w}} \] is a \( 2 \times 8 \) matrix
- With \( e \) outputs and \( d \) inputs, an \( e \times e(d + 1) \) matrix
Jacobian $w$ for a FC Layer

$$
\begin{bmatrix}
  z_1 \\
  z_2
\end{bmatrix}
= 
\begin{bmatrix}
  w_1 & w_2 & w_3 \\
  w_4 & w_5 & w_6
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3
\end{bmatrix}
+ 
\begin{bmatrix}
  w_7 \\
  w_8
\end{bmatrix}
$$

- Don’t be afraid to spell things out:
  
  $$
  z_1 = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_7 \\
  z_2 = w_4 x_1 + w_5 x_2 + w_6 x_3 + w_8 
  $$

- Obvious pattern: Repeat $x^T$, staggered, $e$ times
- Then append the $e \times e$ identity at the end
Training

- Compute $\nabla \ell_n(w) = \nabla \ell(y_n, h(x_n); w)$
- Loop over $T$ to compute $\nabla L_T(w) = \frac{1}{N} \sum_{n=1}^{N} \nabla \ell_n(w)$
- $\hat{w} = \arg\min L_T(w)$
- $L_T(w)$ is (very) non-convex, so we look for local minima
- $w \in \mathbb{R}^m$ with $m$ very large: No Hessians
- Gradient descent
- Even so, every step calls back-propagation $N$ times
- Back-propagation computes $m$ derivatives $\nabla \ell_n(w)$
- Computational complexity is $\Omega(mN)$ per step
- Even gradient descent is way too expensive!
No Line Search

- Line search is out of the question
- Fix some step multiplier $\alpha$, called the *learning rate*
  \[ w_{t+1} = w_t - \alpha \nabla L_T(w_t) \]
- How to pick $\alpha$? Cross-validation is too expensive
- Tradeoffs:
  - $\alpha$ too small: Slow progress
  - $\alpha$ too big: Jump over minima
- Frequent practice:
  - Start with $\alpha$ relatively large, and monitor $L_T(w)$
  - When $L_T(w)$ levels off, decrease $\alpha$
- Alternative: Fixed decay schedule for $\alpha$
- Another (recent) option: Change $\alpha$ adaptively
  (Adam, 2015, later improvements)
Manual Adjustment of $\alpha$

- Start with $\alpha$ relatively large, and monitor $L_T(w_t)$
- When $L_T(w_t)$ levels off, decrease $\alpha$
- Typical plots of $L_T(w_t)$ versus iteration index $t$: 

![Graph showing risk versus epoch for PInNets and ResNets](image)
Batch Gradient Descent (Review)

- We have seen GD and SGD under function optimization
- We review these as they are crucial for neural networks
- $\nabla L_T(w) = \frac{1}{N} \sum_{n=1}^{N} \nabla \ell_n(w)$
- Taking a macro-step $-\alpha \nabla L_T(w_t)$ is the same as taking the $N$ micro-steps $-\frac{\alpha}{N} \nabla \ell_1(w_t), \ldots, -\frac{\alpha}{N} \nabla \ell_N(w_t)$
- First compute all the $N$ steps at $w_t$, then take all the steps
- Thus, standard gradient descent is a *batch* method: Compute the gradient at $w_t$ using the entire batch of data, then move
- Even with no line search, $N$ micro-steps are expensive
- Can we spend the same amount of effort more effectively?
Stochastic Gradient Descent (Review)

- Taking a macro-step $-\alpha \nabla L_T(w_t)$ is the same as taking the $N$ micro-steps $-\frac{\alpha}{N} \nabla \ell_1(w_t), \ldots, -\frac{\alpha}{N} \nabla \ell_N(w_t)$
- First compute all the $N$ steps at $w_t$, then take all the steps
- Can we spend the same amount of effort more effectively?
- Key observation: $-\nabla \ell_n(w)$ is a poor estimate of $-\nabla L_T(w)$, but an estimate all the same: Micro-steps are correct on average!
- After each micro-step, we are on average in a better place
- How about computing a new micro-gradient after every micro-step?
- Now each micro-step gradient is evaluated at a point that is on average better (lower risk) than in the batch method
Batch vs Stochastic GD (Review)

- $s_n(w) = -\frac{\alpha}{N} \nabla \ell_n(w)$

- Batch:
  - Compute $s_1(w_t), \ldots, s_N(w_t)$
  - Move by $s_1(w_t)$, then $s_2(w_t)$, \ldots then $s_N(w_t)$
    (or equivalently move once by $s_1(w_t) + \ldots + s_N(w_t)$)

- Stochastic (SGD):
  - Compute $s_1(w_t)$, then move by $s_1(w_t)$ from $w_t$ to $w_t^{(1)}$
  - Compute $s_2(w_t^{(1)})$, then move by $s_2(w_t^{(1)})$ from $w_t^{(1)}$ to $w_t^{(2)}$
    \vdots
  - Compute $s_N(w_t^{(N-1)})$, then move by $s_N(w_t^{(N-1)})$ from $w_t^{(N-1)}$
    to $w_t^{(N)} = w_{t+1}$

- In SGD, each micro-step is taken from a better (lower risk) place on average than in batch descent
Why “Stochastic?” (Review)

- Progress occurs only *on average*
- Many micro-steps are bad, but they are good on average
- Progress is a random walk

https://towardsdatascience.com/
Reducing Variance: Mini-Batches (Review)

- Each data sample is a poor estimate of $T$: High-variance micro-steps
- Each micro-step takes full advantage of the estimate, by moving right away: Lower-bias micro-steps than batch steps
- High variance *may* hurt more than low bias helps
- Can we lower variance at the expense of slightly increased bias?
- Average $B$ samples at a time: Take *mini-steps*
- With bigger $B$,
  - Higher bias
  - Lower variance
- The $B$ samples are a *mini-batch*
Mini-Batches (Review)

- Scramble $T$ at random ($T$ has $N$ samples)
- Divide $T$ into $J$ mini-batches $T_j$ of size $B$, so $N \approx JB$
- $w^{(0)} = w$
- For $j = 1, \ldots, J$:
  - Batch gradient:
    $$g_j = \nabla L_{T_j}(w^{(j-1)}) = \frac{1}{B} \sum_{n=(j-1)B+1}^{jB} \nabla \ell_n(w^{(j-1)})$$
  - Move: $w^{(j)} = w^{(j-1)} - \alpha g_j$
- This for loop amounts to one macro-step
- Each execution of the entire loop uses the training data once
- Each execution of the entire loop is an epoch
- Repeat over several epochs until a stopping criterion is met