Training Neural Nets
Outline

1. The Softmax Simplex
2. Loss and Risk
3. Back-Propagation
4. Training by 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

\[
p = \sigma(z) = \frac{\exp(z)}{\mathbf{1}^T \exp(z)}
\]

• 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

\[ x_n = x^{(0)} \]
\[ f^{(1)} \]
\[ w^{(1)} \]
\[ x^{(1)} \]
\[ f^{(2)} \]
\[ w^{(2)} \]
\[ x^{(2)} \]
\[ f^{(3)} \]
\[ w^{(3)} \]
\[ x^{(3)} = p \]
\[ \ell \]
\[ y_n \]

- 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 chain
- Apply the 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 $k$: \( \frac{\partial x^{(k)}}{\partial w^{(k)}} \) 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 $x^{(0)} = x_n$, $x^{(1)}$, ...
Back-Propagation Spelled Out for $K = 3$

(again forward pass)

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

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

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

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

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

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

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

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

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

- 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:
  \[ \mathbf{V} = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \rightarrow \]
  \[ \mathbf{w} = [v_{11}, v_{12}, v_{13}, v_{21}, v_{22}, v_{23}, b_1, b_2]^T \]
- \[ \frac{\partial \mathbf{z}}{\partial \mathbf{w}} \] is a \( 2 \times 8 \) matrix
- With \( e \) outputs and \( d \) inputs, an \( e \times e(d + 1) \) matrix
Jacobian 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:

\[
Z_1 = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_7
\]

\[
Z_2 = w_4 x_1 + w_5 x_2 + w_6 x_3 + w_8
\]

\[
\frac{\partial z}{\partial w} = \begin{bmatrix}
    \frac{\partial z_1}{\partial w_1} & \frac{\partial z_1}{\partial w_2} & \frac{\partial z_1}{\partial w_3} & \frac{\partial z_1}{\partial w_4} & \frac{\partial z_1}{\partial w_5} & \frac{\partial z_1}{\partial w_6} & \frac{\partial z_1}{\partial w_7} & \frac{\partial z_1}{\partial w_8} \\
    \frac{\partial z_2}{\partial w_1} & \frac{\partial z_2}{\partial w_2} & \frac{\partial z_2}{\partial w_3} & \frac{\partial z_2}{\partial w_4} & \frac{\partial z_2}{\partial w_5} & \frac{\partial z_2}{\partial w_6} & \frac{\partial z_2}{\partial w_7} & \frac{\partial z_2}{\partial w_8}
\end{bmatrix}
\]

\[
\frac{\partial z}{\partial w} = \begin{bmatrix}
    x_1 & x_2 & x_3 & 0 & 0 & 0 & 1 & 0 \\
    0 & 0 & 0 & x_1 & x_2 & x_3 & 0 & 1
\end{bmatrix}
\]

- Obvious pattern: Repeat \(x^T\), staggered, \(e\) times
- Then append the \(e \times e\) identity at the end
Training by Stochastic Gradient Descent

- (This slide is a review)
- Compute $\nabla \ell_n(w) = \nabla \ell(y_n, h(x_n; w))$
- Gradient descent would loop over $T$ to compute $\nabla L_T(w) = \frac{1}{N} \sum_{n=1}^{N} \nabla \ell_n(w)$ to find $\hat{w} = \arg \min L_T(w)$
- $L_T(w)$ is (very) non-convex, so we look for local minima
- $w \in \mathbb{R}^m$ with $m$ very large: No Hessians
- Even gradient descent is too expensive:
  - $N$ training samples means $N$ loss gradients to compute
  - Each loss gradient requires back-propagation to compute $m$ derivatives $\nabla \ell_n(w)$
  - Each iteration of GD takes $\Omega(mN)$ computations
- Use stochastic gradient descent so the loop over $T$ is replaced by a loop over a mini-batch $B$
Training is a Deep Loop Indeed!

For each epoch:
  For each mini-batch:
    For each training sample:
      For each layer (forward):
        For each weight:
          Multiply weight and input
          Accumulate into layer output
          Add bias
      For each layer (backwards):
        For each weight:
          Compute a derivative
          Accumulate to compute minibatch gradient
      Move by a mini-step