# CS 179: LECTURE 14

## NEURAL NETWORKS AND BACKPROPAGATION

# LAST TIME

- Intro to machine learning
- Linear regression
- Gradient descent
- Linear classification = minimize cross-entropy

# TODAY

- Derivation of gradient descent for linear classifier
- Using linear classifiers to build up neural networks
- Gradient descent for neural networks (backpropagation)

# REFRESHER ON THE TASK

- We are given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})\}$ as training data
- We want to classify each input $x$ into one of $m$ classes
- Each $x^{(n)}$ is a $d$-dimensional column vector $\left(x_1^{(n)}, \dots, x_d^{(n)}\right)^T$
- Each $y^{(n)}$ is a $m$-dimensional column vector $\left(y_1^{(n)}, \dots, y_m^{(n)}\right)^T$
- $y_k^{(n)} = 1$ iff $\mathrm{class}\left(x^{(n)}\right) = k$; otherwise, $y_k^{(n)} = 0$

# REFRESHER ON THE TASK

- Our model is parametrized by a matrix $\mathbf{W} \in \mathbb{R}^{(d+1)\times m}$
- Given a $d$-dimensional input vector $x = (x_1, \dots, x_d)^T$ and denoting $x' = (1, x_1, \dots, x_d)^T$, we compute an $m$-dimensional output vector $z = \mathbf{W}^T x'$
- We then classify $x$ as the class corresponding to the index of $z$ with the largest value

# LINEAR CLASSIFIER GRADIENT

- We will be going through some extra steps to derive the gradient of the linear classifier
- The reason will become clear when we start talking about neural networks

# LINEAR CLASSIFIER GRADIENT

- Define intermediate variables

$$z = \mathbf{W}^T x'$$

$$p_k = \frac{\exp(z_k)}{\sum_{j=1}^{m} \exp\left(z_j\right)};\ p = (p_1, \dots, p_m)^T$$

$$J = -\sum_{k=1}^{m} y_k \ln(p_k)$$

# LINEAR CLASSIFIER GRADIENT

- Simplify derivatives using the multivariate chain rule and the fact that $z_j = \sum_{i=0}^{d} \mathbf{W}_{ij}\, x_i$ (with $x_0 = 1$)

$$\frac{\partial J}{\partial \mathbf{W}_{ij}} = \sum_{k=1}^{m} \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \mathbf{W}_{ij}} = x_i \frac{\partial J}{\partial z_j}$$

$$\frac{\partial J}{\partial z_j} = -\sum_{i=1}^{m} \frac{y_i}{p_i} \frac{\partial p_i}{\partial z_j}$$

# LINEAR CLASSIFIER GRADIENT

- Compute the gradient of the softmax function

$$\frac{\partial p_j}{\partial z_i} = \begin{cases} p_i(1 - p_j) & i = j \\ -p_i \cdot p_j & \text{otherwise} \end{cases}$$

- Substituting this into the previous gradient, we can show

$$\boxed{\frac{\partial J}{\partial z_i} = p_i - y_i = \begin{cases} p_i - 1 & \text{class}(x) = i \\ p_i & \text{otherwise} \end{cases}}$$

# LINEAR CLASSIFIER GRADIENT

- Then, the gradient of the linear classifier's loss function wrt its parameters is

$$\boxed{\begin{gathered}\frac{\partial J}{\partial \mathbf{W}_{ij}} = \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial \mathbf{W}_{ij}} = x_i(p_j - y_j) \\ \nabla_{\mathbf{W}}[J] = x'(p-y)^T\end{gathered}}$$

- More linear algebra! Again, GPU's are great for this stuff ☺

# STOCHASTIC GRADIENT DESCENT

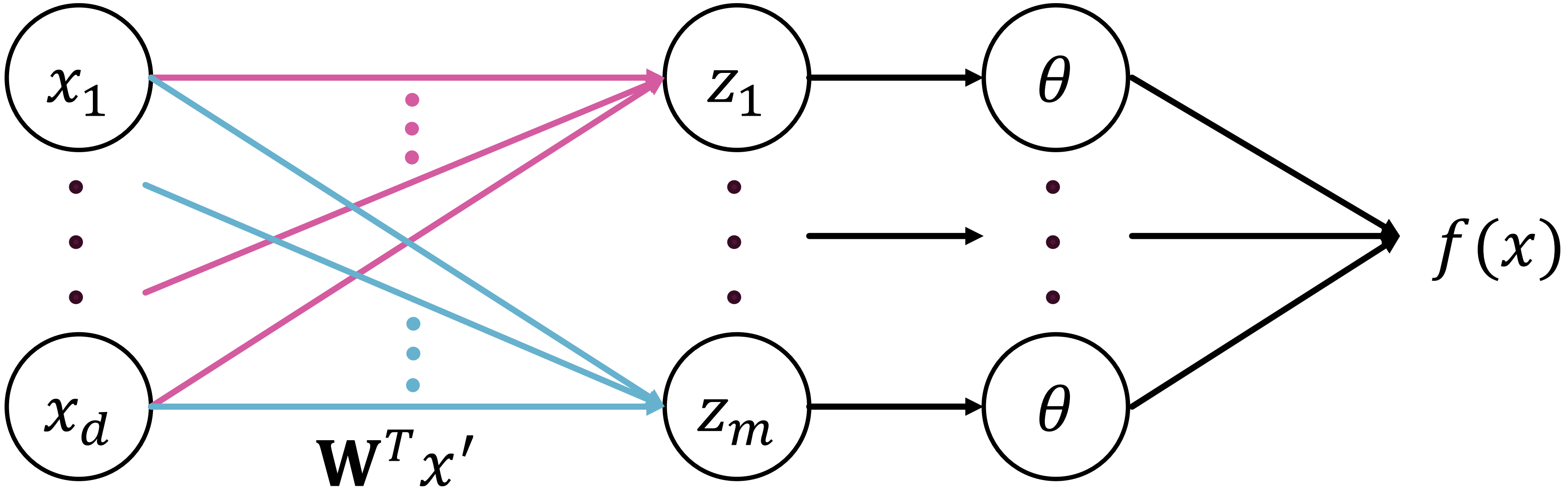
- While $\mathbf{W}$ has not converged
  - For each data point $(x, y)$ in the data set
    - Compute $z = \mathbf{W}^T x'$
    - Compute $p = \frac{\exp(z)}{\sum_{k=1}^{m} \exp(z_k)}$
    - Update $\mathbf{W} \leftarrow \mathbf{W} - \eta\, x' (p - y)^T$
- Alternatively, update per mini-batch instead of per data point

# LIMITATIONS OF LINEAR MODELS

- Most real-world data is not separable by a linear decision boundary
  - Simplest example: XOR gate
- What if we could combine the results of multiple linear classifiers?
  - Combine two OR gates with an AND gate to get a XOR gate

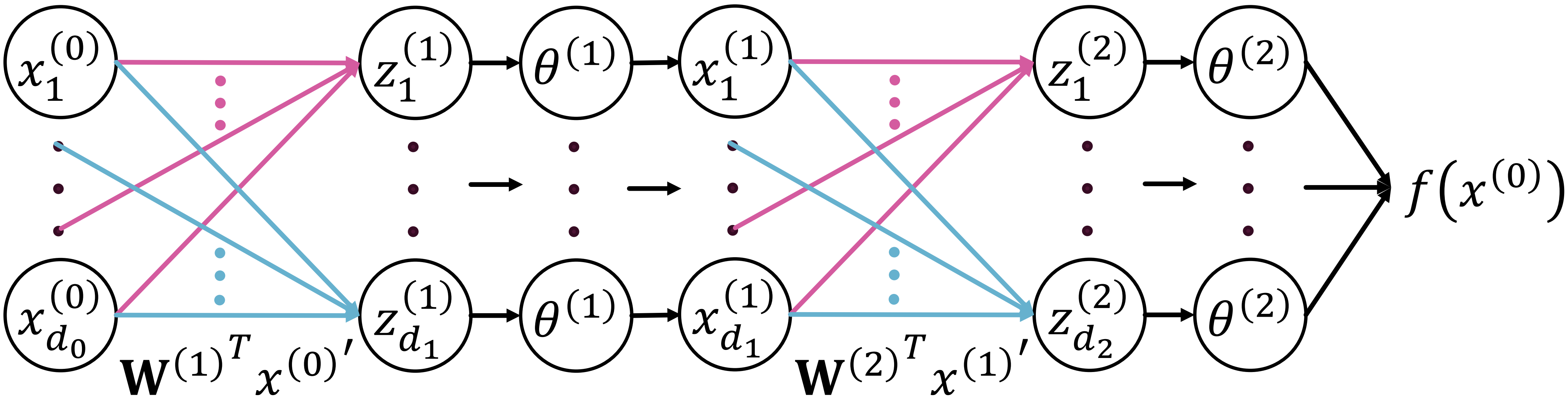
# ANOTHER VIEW OF LINEAR MODELS

- Combine all the components $x_i$ of our input $x$ in different ways in order to get different outputs $z_j$
- Push $z$ through some nonlinear function $\theta$ (e.g. softmax)



# NEURAL NETWORKS

- What if we used each $\theta(z_j)$ as the input to another classifier?
- This lets us compose multiple linear decision boundaries!



# NEURAL NETWORKS

- Why the nonlinearity $\theta$?
  - $\mathbf{W}^{(2)^T}\left(\mathbf{W}^{(1)^T}x'\right)$ is still a linear function $x$
  - $\mathbf{W}^{(2)^T}\theta\left(\mathbf{W}^{(1)^T}x'\right)$ is no longer a linear function in $x$
  - $\theta$ makes the model more expressive
- The nonlinearity $\theta$ is also known as an **activation function**

# EXAMPLES OF ACTIVATIONS

- $\theta(z) = \max(0, z)$ (**ReLU activation**) is most common
- $\theta(z) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (**tanh function**) is occasionally used as well

# UNIVERSAL APPROXIMATOR THM

- It is possible to show that if your neural network is big enough, it can approximate any continuous function arbitrarily well! (Hornik 1991)
- This is why neural nets are important

# NEURAL NETWORKS

- But why stop at just 2 layers of linear function/nonlinearity?
- We can have arbitrarily many $L$ layers!
  - $x^{(\ell-1)}$ is the input to layer $\ell$ ($x^{(0)}$ is the data given)
  - $x^{(\ell)} = \theta^{(\ell)}\left(\mathbf{W}^{(\ell)^T} x^{(\ell-1)'}\right)$ is the output of layer $\ell$
  - The loss function is applied to $x^{(L)} = \theta^{(L)}\left(z^{(L)}\right)$ (the final output), though it is sometimes easier to apply it to $z^{(L)}$ directly (e.g. softmax cross-entropy loss w/ linear classifier)

# BACKPROPAGATION

- So how do we take the gradient of a neural network with respect to every parameter matrix $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$?
- Define $z^{(\ell)} = \mathbf{W}^{(\ell)^T} x^{(l-1)'}$ and $\delta^{(\ell)} = \nabla_{z^{(\ell)}}[J]$. By chain rule,

$$\frac{\partial J}{\partial \mathbf{W}_{ij}^{(\ell)}} = \sum_{k=1}^{d_\ell} \frac{\partial J}{\partial z_k^{(\ell)}} \frac{\partial z_k^{(\ell)}}{\partial \mathbf{W}_{ij}^{(\ell)}} = x_i^{(\ell-1)} \frac{\partial J}{\partial z_j^{(\ell)}} = x_i^{(\ell-1)} \delta_j^{(\ell)}$$

$$\boxed{\nabla_{\mathbf{W}^{(\ell)}}[J] = x^{(\ell-1)'} \delta^{(\ell)^T}}$$

# BACKPROPAGATION

- To find $\delta^{(\ell)} = \nabla_{z^{(\ell)}}[J]$, apply the chain rule again:

$$\frac{\partial J}{\partial z_i^{(\ell-1)}} = \frac{\partial J}{\partial x_i^{(\ell-1)}} \frac{\partial x_i^{(\ell-1)}}{\partial z_i^{(\ell-1)}} = \frac{\partial J}{\partial x_i^{(\ell-1)}} \theta^{(\ell-1)'}\left(z_i^{(\ell-1)}\right)$$

$$\frac{\partial J}{\partial x_i^{(\ell-1)}} = \sum_{j=0}^{d_\ell} \frac{\partial J}{\partial z_j^{(\ell)}} \frac{\partial z_j^{(\ell)}}{\partial x_i^{(\ell-1)}} = \sum_{j=0}^{d_\ell} \delta_j^{(\ell)} \mathbf{W}_{ij}^{(\ell)} = \left(\mathbf{W}^{(\ell)} \delta^{(\ell)}\right)_i$$

$$\boxed{\delta_i^{(\ell-1)} = \theta^{(\ell-1)'}\left(z_i^{(\ell-1)}\right) \left(\mathbf{W}^{(\ell)} \delta^{(\ell)}\right)_i}$$

# BACKPROPAGATION

- We know $x^{(0)}$ and the current values of $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$
- If we do a forward pass through the neural network, we will compute every $x^{(1)}, \ldots, x^{(L)}$ and $z^{(1)}, \ldots, z^{(L)}$
- From the linear classifier, we know that $\delta^{(L)} = x^{(L)} - y$
- $\theta^{(\ell-1)'}\left(z_i^{(\ell-1)}\right)$ is easy to compute
- We have all we need to do stochastic gradient descent!

# BACKPROPAGATION

- Fix a learning rate $\eta$ and initialize $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(L)}$ randomly
- For each data point $(x^{(0)}, y)$ in the data set
  - Compute each $z^{(\ell)} = {\mathbf{W}^{(\ell)}}^T {x^{(\ell-1)}}'$ and $x^{(\ell)} = \theta^{(\ell)}\left(z^{(\ell)}\right)$
  - Initialize $\delta^{(L)} = x^{(L)} - y$
  - For each $\ell$ counting down from $L$ to 1
    - Calculate $\alpha^{(\ell)} = \nabla_{x^{(\ell-1)}}[J] = \mathbf{W}^{(\ell)} \delta^{(\ell)}$
    - Set $\delta_i^{(\ell-1)} = \alpha_i^{(\ell)} {\theta^{(\ell-1)}}'\left(z_i^{(\ell-1)}\right)$ for each $i = 1, \dots, d_{\ell-1}$
    - Update $\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta\left(x^{(\ell-1)} {\delta^{(\ell)}}^T\right)$

# BACKPROPAGATION

- Forward pass
  - We are given $x^{(0)}$
  - $x^{(\ell+1)}$ depends on $z^{(\ell+1)}$, which depends on $x^{(\ell)}$
- Backward pass
  - We have $\delta^{(L)}$ from the forward pass
  - $\delta^{(\ell-1)}$ depends on $\delta^{(\ell)}$
  - We need $\delta^{(\ell)}$ because $\nabla_{\mathbf{W}^{(\ell)}}[J]$ depends on $\delta^{(\ell)}$

# BACKPROPAGATION

- This is stochastic gradient descent for a neural network!
- In Homework #5, you will:
  - Implement a linear classifier
  - Extend it to a 2-layer neural network
- Before discussing implementation details, let's talk about parallelizing the backpropagation algorithm

# PARALLELIZATION

- By its nature, the backpropagation algorithm seems fundamentally sequential
- However, each sequential step is a linear algebra operation
  - Parallelize with cuBLAS
- Minibatch stochastic gradient descent
  - Compute the gradient for each data point in the minibatch
  - Use a parallel reduction to take the average at the end

# USING MINIBATCHES

- Consider a minibatch size of $k$
  - Construct a $d_\ell \times k$ matrix $\mathbf{X}^{(\ell)}$ where column $i$ is the $x^{(\ell)}$ corresponding to data point $i$ in the mini-batch
  - Construct a $d_\ell \times k$ matrix $\Delta^{(\ell)}$ where column $i$ is the $\delta^{(\ell)}$ corresponding to data point $i$ in the mini-batch
  - Define $\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)^T}\mathbf{X}^{(\ell-1)'}$
- After fixing a learning rate $\eta$ and initializing $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(\ell)}$ randomly, we have the following algorithm:

# USING MINIBATCHES

- For each minibatch $(\mathbf{X}^{(0)}, \mathbf{Y})$ of size $k$ in the data set
  - Compute each $\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)^T}\mathbf{X}^{(\ell-1)}$ and $\mathbf{X}^{(\ell)} = \theta^{(\ell)}\left(\mathbf{Z}^{(\ell)}\right)$
  - Initialize $\Delta^{(L)} = \mathbf{X}^{(L)} - \mathbf{Y}$
  - For each $\ell$ counting down from $L$ to 1
    - Calculate $\mathbf{A}^{(\ell)} = \nabla_{\mathbf{X}^{(\ell-1)}}[J] = \mathbf{W}^{(\ell)}\Delta^{(\ell)}$
    - Set $\Delta_{ij}^{(\ell-1)} = \mathbf{A}_{ij}^{(\ell)}\theta^{(\ell-1)'}\left(\mathbf{Z}_{ij}^{(\ell-1)}\right)$ for all $i = 1, \dots, d_{\ell-1}$ and $j = 1, \dots, k$
    - Update $\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \frac{1}{k}\eta\left(\mathbf{X}^{(\ell-1)'}\Delta^{(\ell)^T}\right)$

# IMPLEMENTATION

- You can do all the matrix multiplications using cuBLAS
- The only new computation is $\Delta_{ij}^{(\ell-1)} = \mathbf{A}_{ij}^{(\ell)} \theta^{(\ell-1)'}\left(\mathbf{Z}_{ij}^{(\ell-1)}\right)$
- This differentiation and pointwise multiplication step (and much more) is done for you for free by another CUDA package called cuDNN (Deep Neural Nets)
- Next time, you will learn the basics of cuDNN