CS 486/686
Neural Networks II

Yuntian Deng

Lecture 20

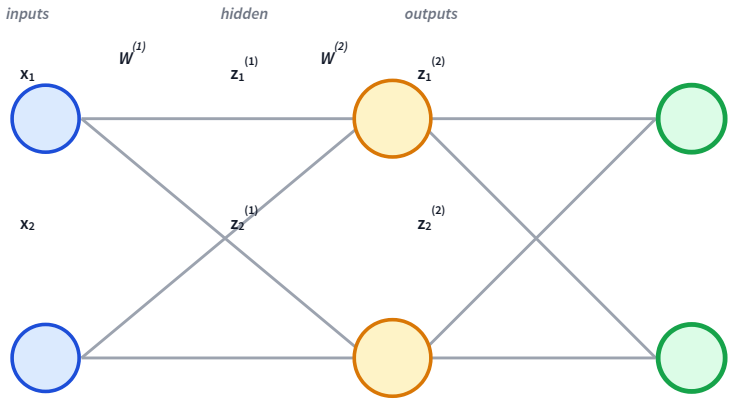
RN 19.6.2, 21.1, 21.2 · PM 7.5 · GBC 4.3, 6.5

Learning goals

- Explain the steps of **gradient descent**.
- Modify GD to speed up learning and ensure convergence.
- Describe the **back-propagation** algorithm (forward + backward passes).
- Compute the gradient for a weight in a multi-layer NN.
- Decide when to use a **neural network** vs a **decision tree**.

The training problem

A 2-layer feedforward network for spam classification — predict $[a_1^{(2)}, a_2^{(2)}]$ from two features (email length, sender trust).



Two outputs encode "spam" vs "ham":

- Spam → target $[1, 0]$
- Ham → target $[0, 1]$
- Ambiguous → $[0.5, 0.5]$

We have paired data (x_1, x_2, y). **How do we set the weights $W^{(1)}, W^{(2)}$?**

Loss function and gradient descent

Squared loss on a dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ with network output $\hat{y}^{(i)} = f(x^{(i)}; W)$:

$$L(W) = \tfrac{1}{2} \sum_{i=1}^{N} \left\| \hat{y}^{(i)} - y^{(i)} \right\|^2$$

Gradient descent: repeat for many iterations:

1. Compute the gradient $\nabla_W L(W) = \left[\partial L / \partial w \right]_{w \in W}$.
2. Update every weight: $w \leftarrow w - \eta \, \dfrac{\partial L}{\partial w}$.

$\eta > 0$ is the *learning rate* (step size).

Why this works

Direction: steepest descent

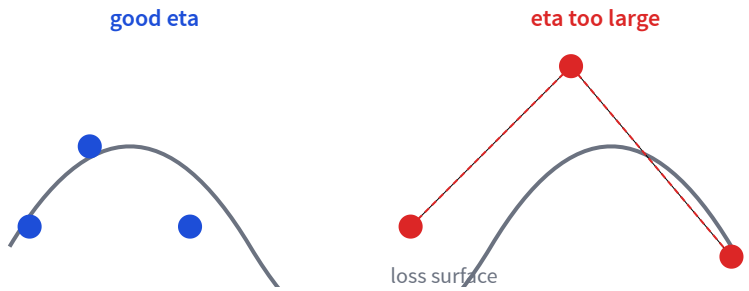
$\nabla_W L$ points in the direction of *fastest increase* of L.

Moving in $-\nabla_W L$ decreases L the fastest.

Step size: η trade-off

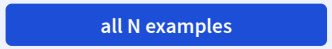
Too small $\Rightarrow$ very slow learning.

Too large $\Rightarrow$ overshoots the minimum, may diverge.



Three flavors of gradient descent

Batch GD



Gradient uses **all** N examples per step.

Stable, accurate gradient.

Slow for large N.

Stochastic GD

One example per step.

Cheap; noise helps escape local minima.

Noisy updates.

Mini-batch GD

A small batch (e.g. 32, 64) per step.

Best of both worlds; GPU-friendly.

Standard choice today.

Plus tricks like momentum, Adam, learning-rate schedules — covered in CS480/680.

Back-propagation: efficient gradients

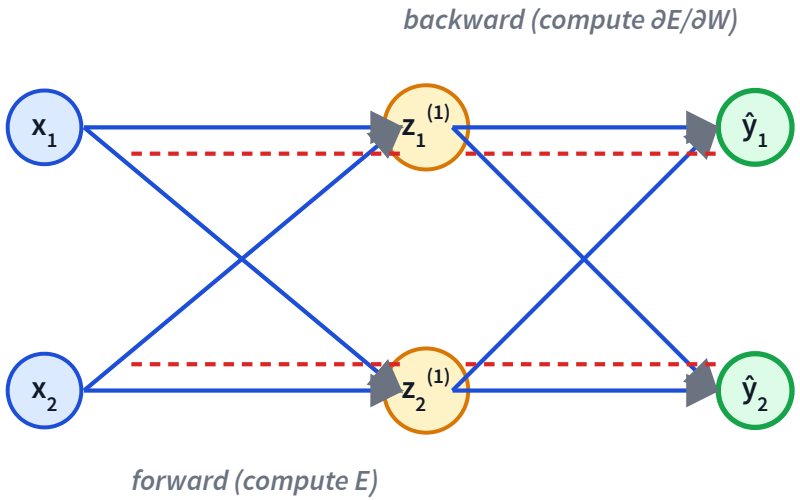
Two passes per training example (x, y):

→ Forward pass

Push inputs through layers, compute every $a^{(\ell)}$, $z^{(\ell)}$, and the loss E.

← Backward pass

Propagate ∂E back through the layers, compute every $\partial E / \partial W^{(\ell)}$.



Forward pass

Compute pre-activations $a^{(\ell)}$ and activations $z^{(\ell)} = g(a^{(\ell)})$ layer by layer.

$$a_j^{(1)} = \sum_i x_i \, W_{i,j}^{(1)}, \qquad z_j^{(1)} = g(a_j^{(1)})$$

$$a_k^{(2)} = \sum_j z_j^{(1)} \, W_{j,k}^{(2)}, \qquad z_k^{(2)} = g(a_k^{(2)})$$

$$E = E\left(z^{(2)}, y\right)$$

Cache every $a^{(\ell)}$ and $z^{(\ell)}$ — we'll need them on the way back.

Backward pass

Walk back through the network, computing local errors δ via the chain rule.

Layer 2 (output side):

$$\frac{\partial E}{\partial W^{(2)}_{j,k}} = \delta^{(2)}_k \, z^{(1)}_j, \qquad \delta^{(2)}_k = \frac{\partial E}{\partial z^{(2)}_k} \, g'\big(a^{(2)}_k\big)$$

Layer 1 (propagate back):

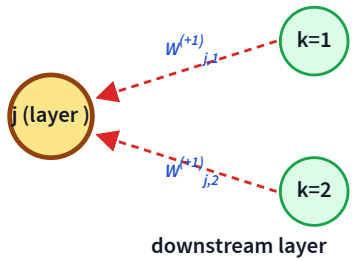
$$\frac{\partial E}{\partial W^{(1)}_{i,j}} = \delta^{(1)}_j \, x_i, \qquad \delta^{(1)}_j = \left(\sum_k \delta^{(2)}_k \, W^{(2)}_{j,k}\right) g'\big(a^{(1)}_j\big)$$

Pattern: gradient = local error $\times$ input to that weight.

The recursive structure of δ

For unit j in layer ℓ, define $\delta_j^{(\ell)} = \dfrac{\partial E}{\partial a_j^{(\ell)}}$. Then:

$$\delta_j^{(\ell)} = \begin{cases} \dfrac{\partial E}{\partial z_j^{(\ell)}}\, g'(a_j^{(\ell)}), & \text{output unit (base)} \\ \left(\displaystyle\sum_k \delta_k^{(\ell+1)}\, W_{j,k}^{(\ell+1)} \right) g'(a_j^{(\ell)}), & \text{hidden unit (recursive)} \end{cases}$$



δ for a hidden unit = weighted sum of downstream δs, modulated by g'.

Backprop in matrix form

Stack layer activations into vectors. Let $\delta_\ell = \partial E / \partial z^{(\ell)}$ and W_ℓ be the weight matrix.

Algorithm.

1. Initialize weights W_ℓ for every layer.
2. **Forward:** push x through, cache $z^{(1)}, z^{(2)}, \ldots$
3. **Output δ:** set $\delta_n = \partial E / \partial z^{(n)}$.
4. **Backward sweep,** for $\ell = n, n-1, \ldots, 1$:

$$\delta_{\ell-1} = \delta_\ell \cdot \frac{\partial g(x^{(\ell)})}{\partial x^{(\ell)}} \cdot W_\ell$$

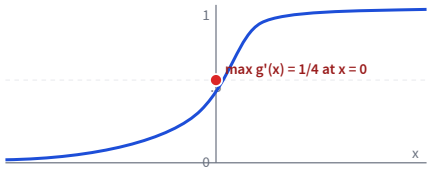
$$\frac{\partial E}{\partial W_\ell} = \delta_\ell \cdot \frac{\partial g(x^{(\ell)})}{\partial x^{(\ell)}} \cdot z^{(\ell-1)}$$

5. Plug $\partial E / \partial W_\ell$ into gradient descent.

The sigmoid derivative trick

For $g(x) = \dfrac{1}{1+e^{-x}}$:

$$g'(x) = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = g(x)\left(1-g(x)\right)$$



Why it matters: during the forward pass we already compute $g(a)$. Reuse it — no need to redo the exponential during the backward pass.

Similar shortcuts exist for tanh ($1 - g^2$) and ReLU (0 or 1).

When to use (or not use) a neural network

Reach for an NN when

- High-dimensional, real-valued, or noisy inputs.
- Target function form is unknown (no good hand-crafted model).
- Interpretability is not a priority.
- Plenty of training data is available.

Avoid an NN when

- Architecture is hard to choose (layers, units, activations).
- Weights need to be inspected and explained.
- Data is tabular and small — overfitting risk is high.

Neural network vs decision tree

	Neural network	Decision tree
Data type	Images, audio, text	Tabular data
Data size	Needs a lot; easy to overfit small data	Works with very little data
Target function	Arbitrary functions	Nested if/else rules
Architecture	Layers, units, activations, init, η all critical	A few hyperparameters (depth, pruning)
Interpretability	Black box	Easy to explain to humans
Train / inference	Slow to train and run	Fast

Revisiting learning goals

- Explain the steps of **gradient descent**.
- Modify GD to speed up learning and ensure convergence.
- Describe the **back-propagation** algorithm (forward + backward passes).
- Compute the gradient for a weight in a multi-layer NN.
- Decide when to use a **neural network** vs a **decision tree**.

Next: Neural Networks III

- Convolutional networks — weight sharing for images.
- Word embeddings and sequence models.
- A peek at **transformers** and large language models.

See you on Thursday!