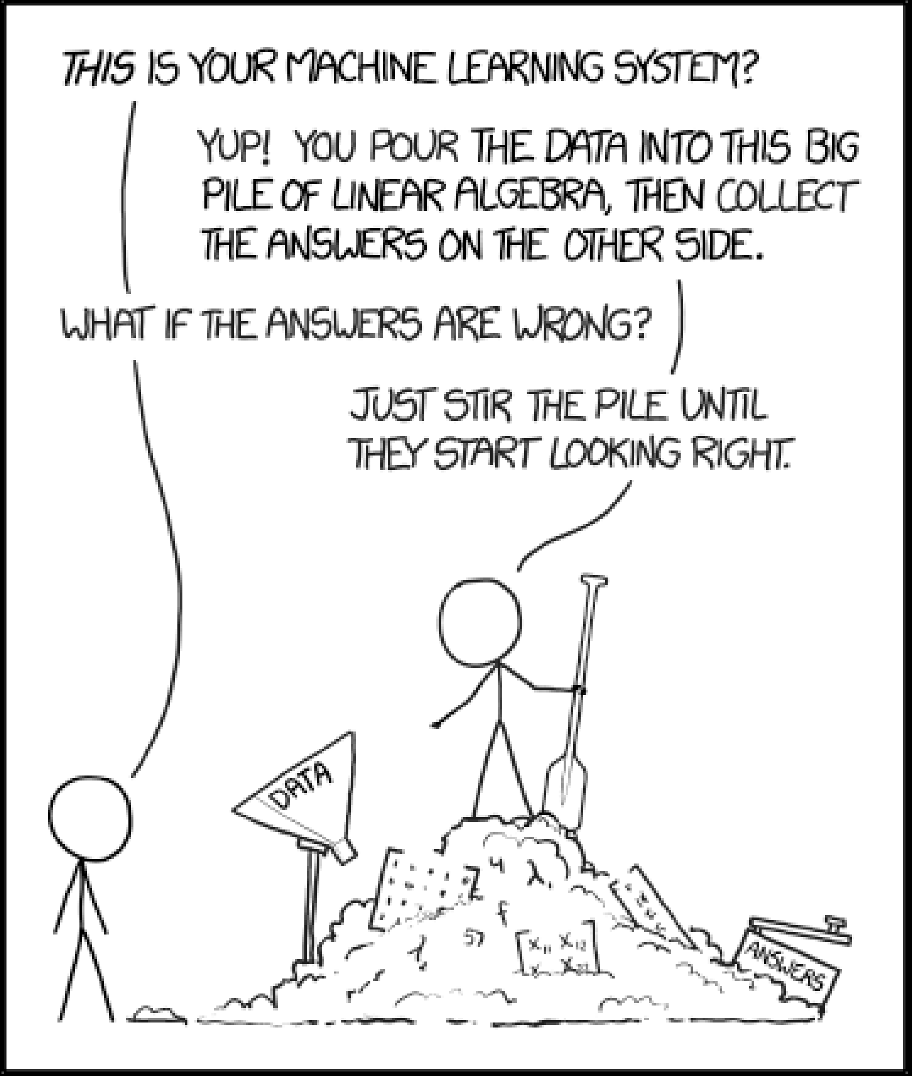
INFO 251: Applied Machine Learning




# Neural Networks, part 2

# Announcements

- PS4 due next Monday
- PS5 (Trees, forests, basic neural nets) posted next week

# Key concepts (last class)

- Mimicking basic neural processes
- The perceptron
- Perceptron limitations
- Rosenblatt's algorithm
- Perceptron training vs. gradient descent
- Multilayer networks
- Universal approximation theorem

# Outline

- **Learning multilayer weights: Intuition**
- Generalizing logistic regression
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- Backpropagation: Video
- Summary

# Multi-layer networks are great!

- Multi-layer networks have great properties, e.g. can solve the XOR problem – this was recognized by Minsky and Papert (1969)
- However, people didn't know how to fit these multi-layer networks. What weights?
  - The perceptron training rule we discussed doesn't work with multiple layers
  - Still, no one has figured out how to generalize Rosenblatt's algorithm to multi-layer networks

# Intuition check 1

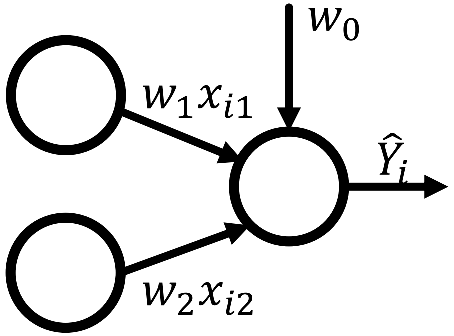
- Can off-the-shelf gradient descent, such as what you're implementing in PS4, be used to learn the weights in multi-layer networks?
  - No! The "activation" step function of a perceptron creates non-convex loss. Not differentiable

$$\hat{Y}_i = \begin{cases} 1 & \text{if } w_0 + w_1 x_{i1} + \cdots + w_n x_{in} > 0 \\ -1 & \text{otherwise} \end{cases}$$

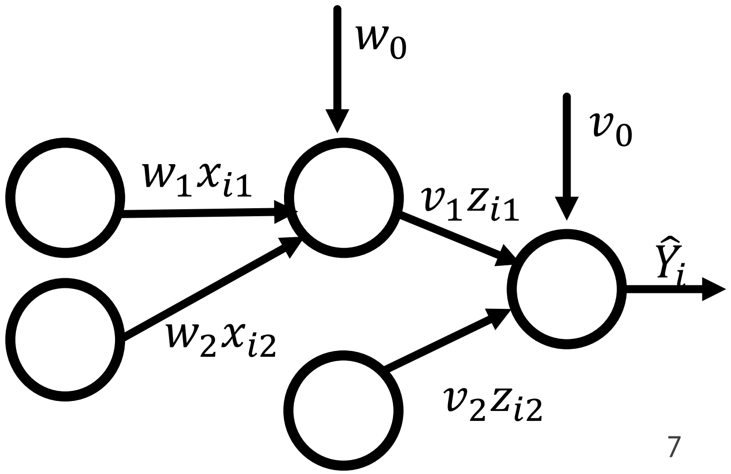
# Intuition check 2

- Well, if the issue is with the step function, can we just omit the step function entirely?

- No!
  - <u>Without activation</u>, each individual unit is linear
  - $\hat{Y}_i = w_0 + w_1 x_{i1} + \cdots + w_n x_{in}$




- Combination of units is more complex…
  - $\hat{Y}_i = v_0 + v_1(w_0 + \; w_1 x_{i1} + \cdots + w_m x_{im}) + \cdots + v_n x_{in}$
  - But still linear!

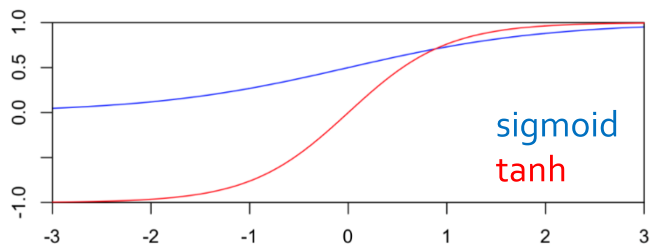


# So, what do we do?

- We need something like a step function to capture non-linearities

$$\hat{Y}_i = \begin{cases} 1 & \text{if } w_0 + w_1 x_{i1} + \cdots + w_n x_{in} > 0 \\ -1 & \text{otherwise} \end{cases}$$

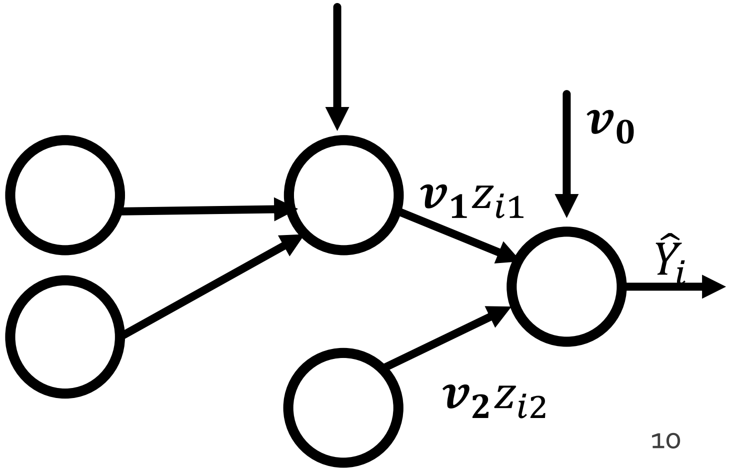
- But the step function itself creates issues for learning weights
  - It's nonlinear (good!), but not differential (bad!)
  - Gradient descent needs a differentiable function
- In other words, we need a nonlinear, differentiable function
  - Examples: sigmoid function, tanh()



# Outline

- Learning multilayer weights: Intuition
- **Generalizing logistic regression**
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- Backpropagation: Video
- Summary

# Generalizing logistic regression




- Recall the loss functions from before
  - Linear Regression
    - $J(\alpha, \beta) = \frac{1}{2N} \sum_{i=1}^{N} (Y_i - \alpha - \beta X_i)^2 \; [+\lambda \sum_{j=1}^{k} \theta_j^2]$
  - Logistic Regression (omitting regularization)
    - $J(\theta) = -\frac{1}{N} \sum_{i=1}^{N} Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \log\left(1 - \hat{Y}_i\right)$
  - Note: Something like this could work on the **last** (output) layer of a multi-layer network, to determine the weights on that layer
    - Such as the bold $\boldsymbol{v_i}$'s in the diagram

# Generalizing logistic regression

- However, neural networks have multiple layers/outputs (generalization of logistic function)

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} Y_{ik} \cdot \log \hat{Y}_{ik} + (1 - Y_{ik}) \log\left(1 - \hat{Y}_{ik}\right)$$

  - (This is just the cost function for a network with *k* outputs)
  - Here, each $\hat{Y}_{ik}$ is like a nested set of logistic regressions

- BUT: we don't have the target value (the $Y_{ik}$) for hidden layers!

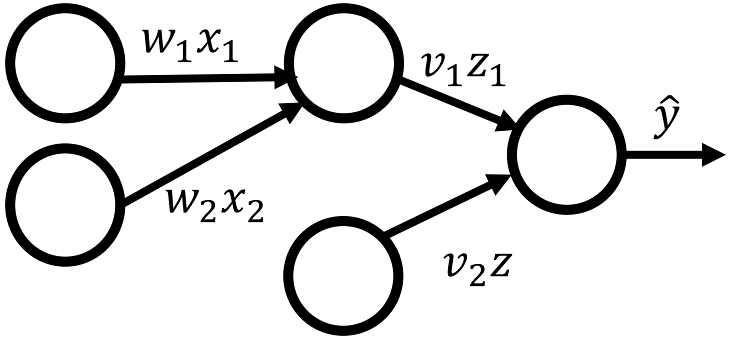
# Outline

- Learning multilayer weights: Intuition
- Generalizing logistic regression
- **Learning multilayer weights: simple case**
- Backpropagation: Intuition
- Backpropagation: Video
- Summary

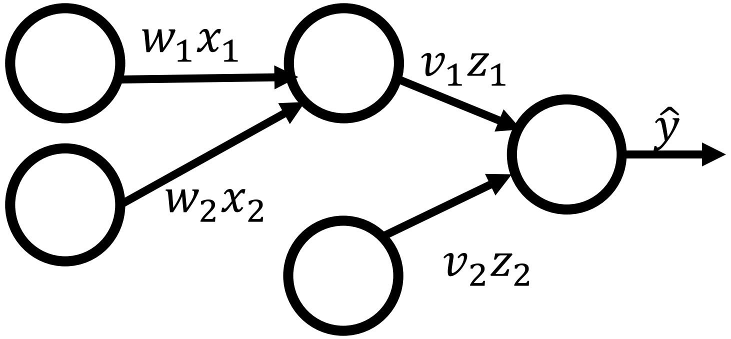
# Two-layer backpropagation

- Backprop = gradient descent + chain rule
  - Solution and algorithm for two layers not too complex (see Daume)
- Objective: $$\min_{\mathbf{W},v} \sum_n \frac{1}{2}\Big(y_n - \underbrace{\sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)}_{\text{This is the prediction } \hat{y}_n}\Big)^2$$
  - $n$ indexes observations
  - $i$ indexes hidden units
  - $v$ is second layer weights
  - $f$ is the sigmoid function
  - $\boldsymbol{w}_i$ is the vector of weights feeding into node $i$
  - $\mathbf{W}$ is first layer weights

$z_1 = f(w_1x_1 + w_2x_2)$
$z_2 = f(w_3x_1 + w_4x_2)$



# Two-layer backpropagation




- Objective: $$\min_{\mathbf{W},v} \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(w_i \cdot x_n)\right)^2$$

- Apply chain rule: $$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(w_i \cdot x)\right)^2$$
$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial w_i}$$
$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(w_i \cdot x)\right) v_i = -ev_i$$
$$\frac{\partial f_i}{\partial w_i} = f'(w_i \cdot x)x$$

prediction error $(y - \hat{y})$

- Solution: $$\nabla_{w_i} = -ev_i f'(w_i \cdot x)x$$

# Learning multilayer weights

- Solution with two layers

$$\nabla_{w_i} = -e v_i f'(w_i \cdot x) x$$

- Does this make sense?
  - If predictive error ($e$) is small, take small steps
  - If $v_i$ is small, hidden unit $i$ has little influence on output, gradient should be small
  - If $e$ or $v_i$ changes sign, gradient should also flip sign

# Outline

- Learning multilayer weights: Intuition
- Generalizing logistic regression
- Learning multilayer weights: simple case
- **Backpropagation: Intuition**
- Backpropagation: Video
- Summary

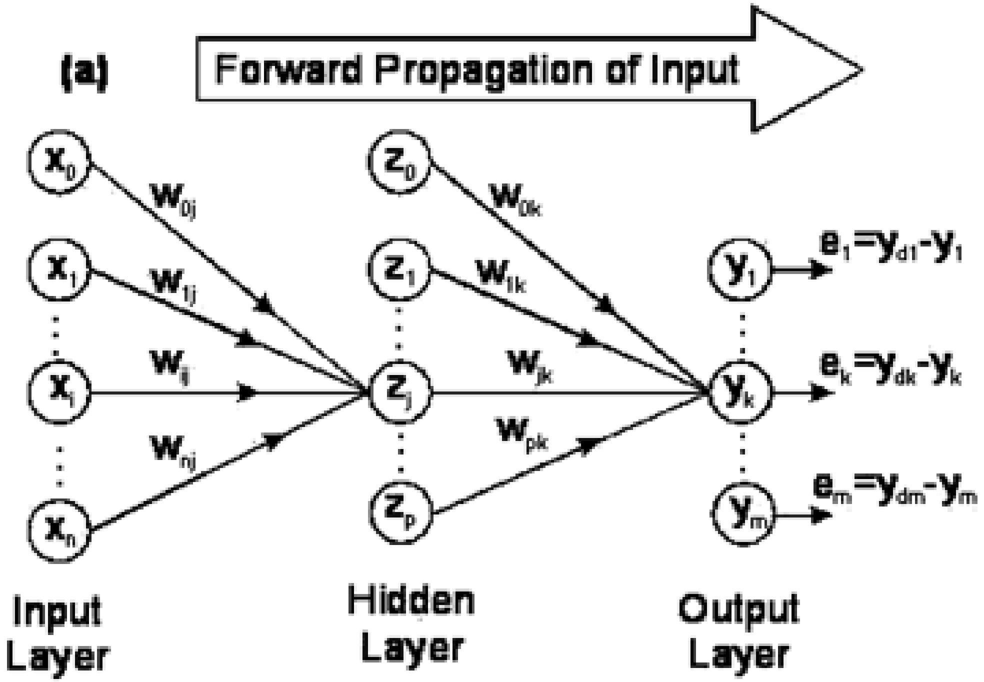
# Backpropagation

- How to generalize from two layers?
- Sketch of procedure

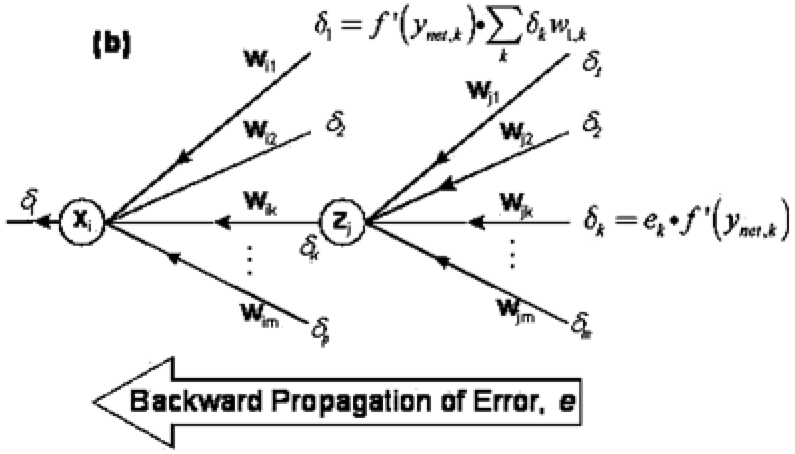
1. Forward Propagation -> Outputs
2. Backward Propagation -> Generate "deltas"
3. Weight Update -> same as in gradient descent

# Intuition

- **Forward Propagation**
  - Given a training example ($X_1$ , …, $X_n$) and output $Y_i$:
  - Propagate inputs/activations forward, applying sigmoid function on dot products



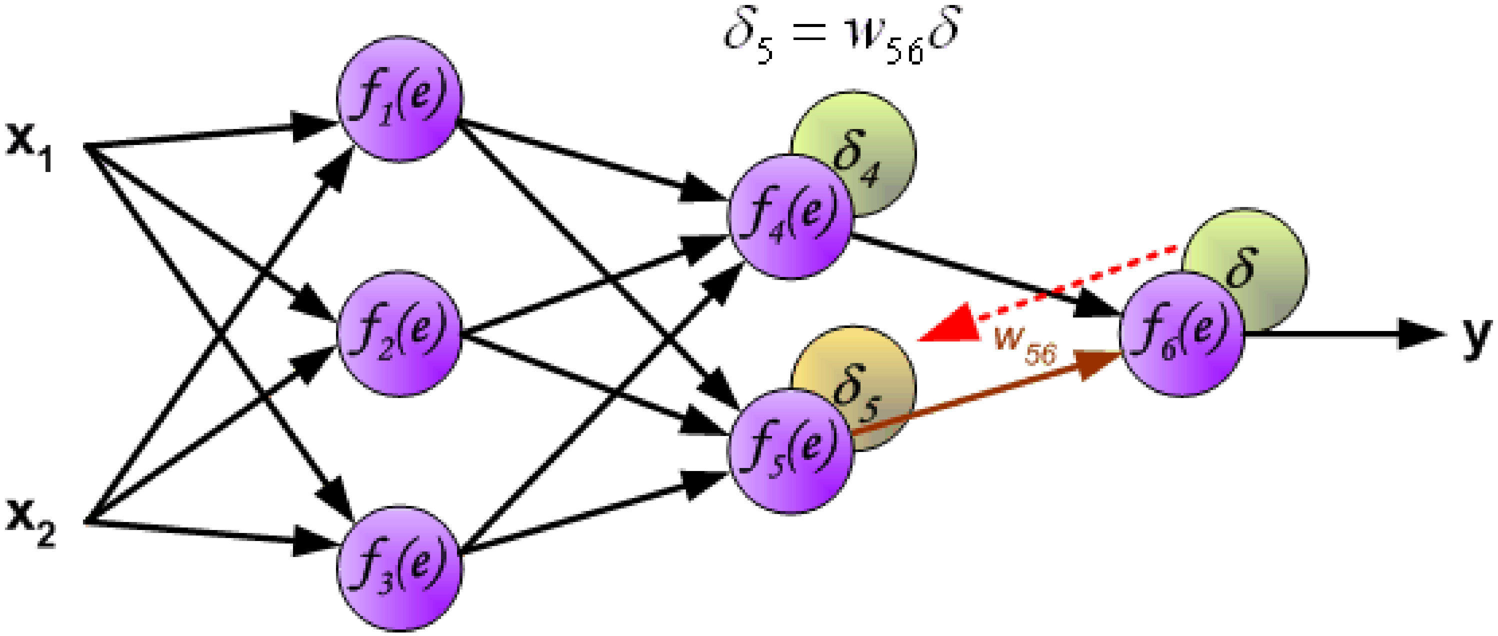
# Intuition




- **Backward Propagation**
  - For a single training example i:
    - $\text{Cost}(i) = Y \cdot \log \hat{Y}_i + (1 - Y_i) \log\left(1 - \hat{Y}_i\right)$
    - i.e., how close is output to actual value?
  - Idea is to propagate costs backwards to earlier nodes
    - Compute $\delta_{jK}$ = "error" of $j^{\text{th}}$ node in $K^{\text{th}}$ (output) layer
      - $\delta_{jK} = Y_{jK}\left(1 - Y_{jK}\right)\left(\hat{Y}_{jK} - Y_{jK}\right)$
    - Note: getting these partials is a bit complex, but mostly just chain rule + gradient descent (see Hastie ch. 11)

# Intuition

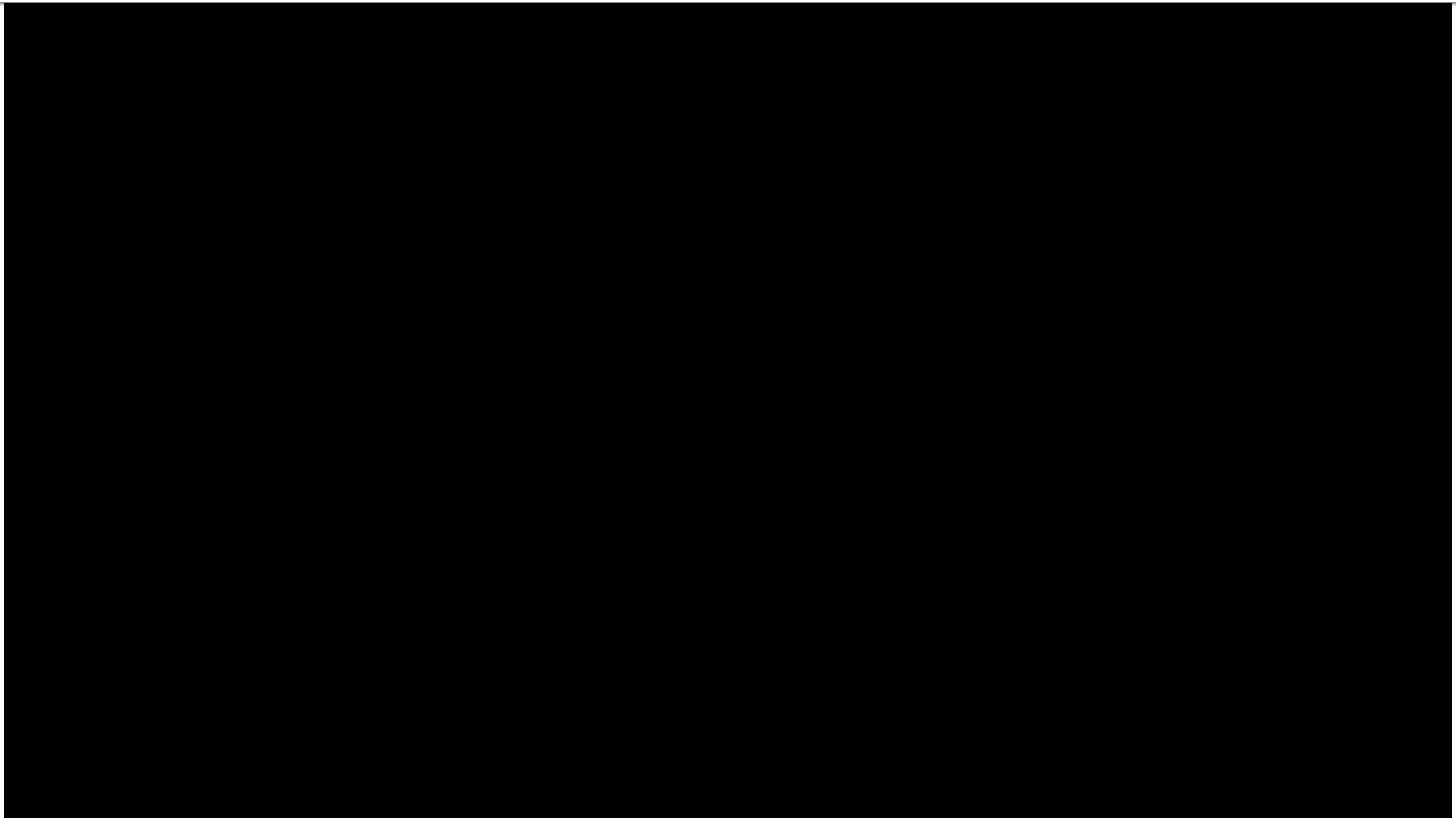
- **Update weights**
  - For each hidden unit $h$ in $k^{th}$ layer, update each weight as $+\eta\delta_{hk}x_i$



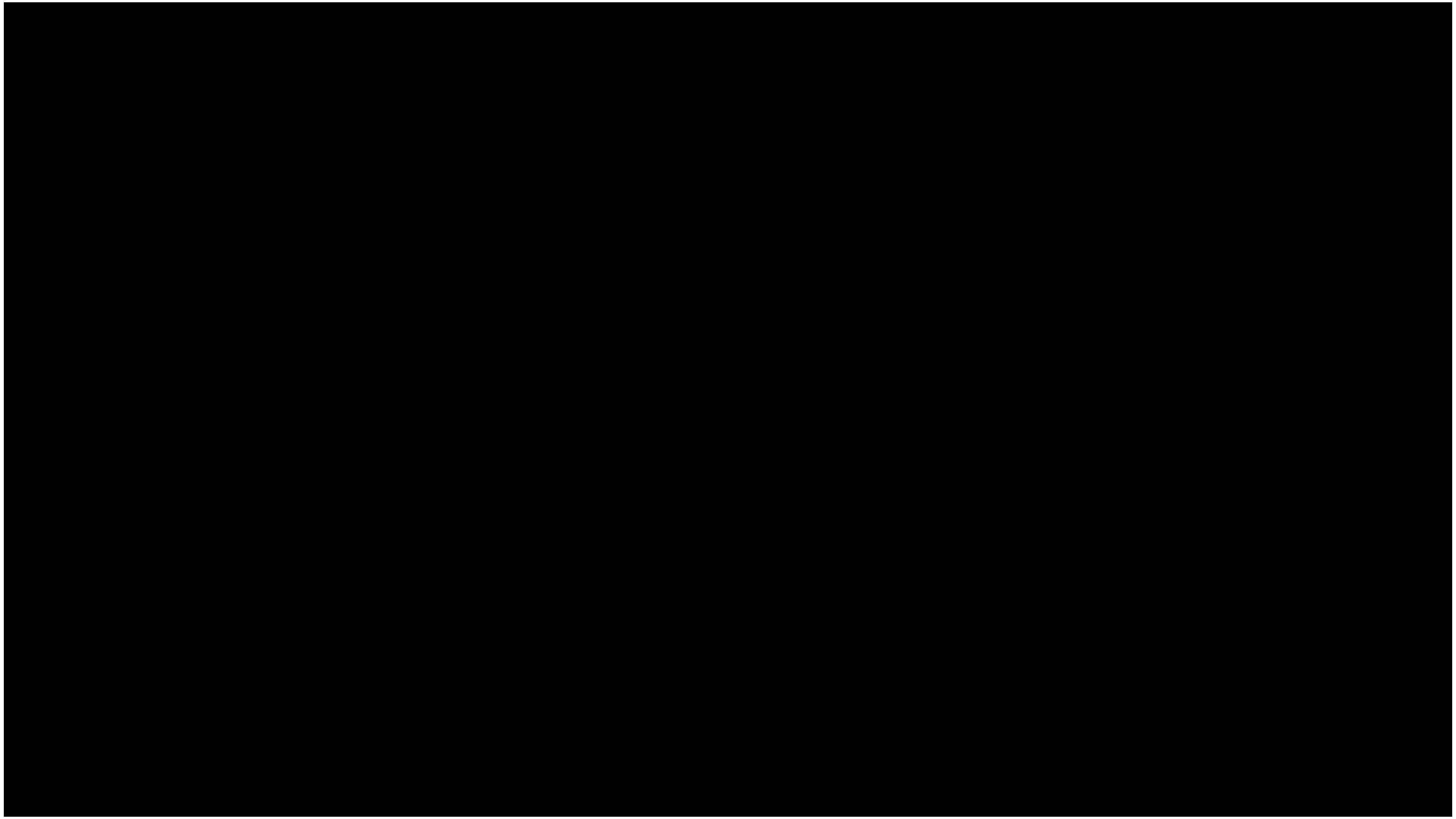
# Outline

- Learning multilayer weights
- Generalizing logistic regression
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- **Backpropagation: Video**
- Summary

# What is back-propagation?



# Video #2



# Outline

- Learning multilayer weights
- Generalizing logistic regression
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- Backpropagation: Video
- **Summary**

# Neural Networks: Issues

- Non-convex, sensitive to initialization
  - Common solutions: Randomize initialization (small random/uniform weights), train multiple networks
- Avoiding overfitting
  - Early stopping
  - Penalize large weights (explicit regularization)
  - Include fewer layers, weights per layer

# Tuning networks

- Many considerations
  - How many layers?
  - How many units per layer?
  - How to initialize?
  - What learning rate?
  - Weight regularization?
  - When to stop?
- Tuning deep networks can take time
  - Network architecture
  - Layer-wise initialization
  - Alternative optimization

# Neural Networks: Summary

- Very flexible, can model complex and non-linear relationships
- Compute-intensive
- Can be parallelized!
- Very hard to interpret, i.e. “black box”