AI and Machine Learning

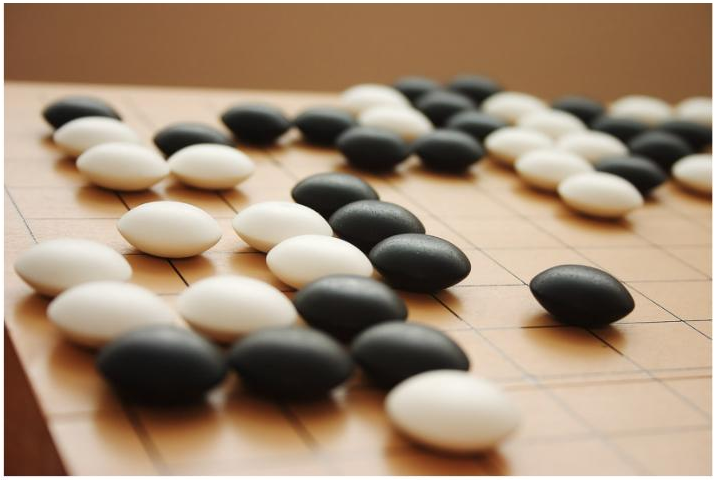
Zhiyun Lin

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Neural networks

- Multi-layer Perceptron
- Forward Propagation
- Backward Propagation

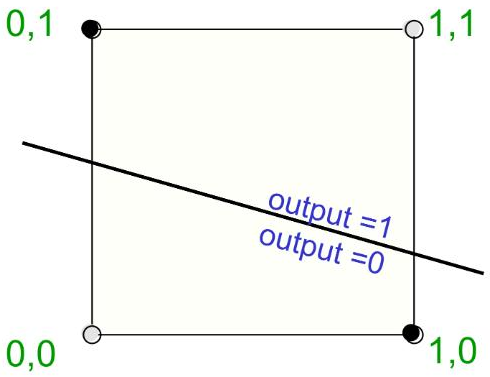
Motivating examples



Are you excited about deep learning?

Limitations of linear classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features x_i
- Many decisions involve non-linear functions of the inputs
- Canonical example: Do 2 input elements have the same value?



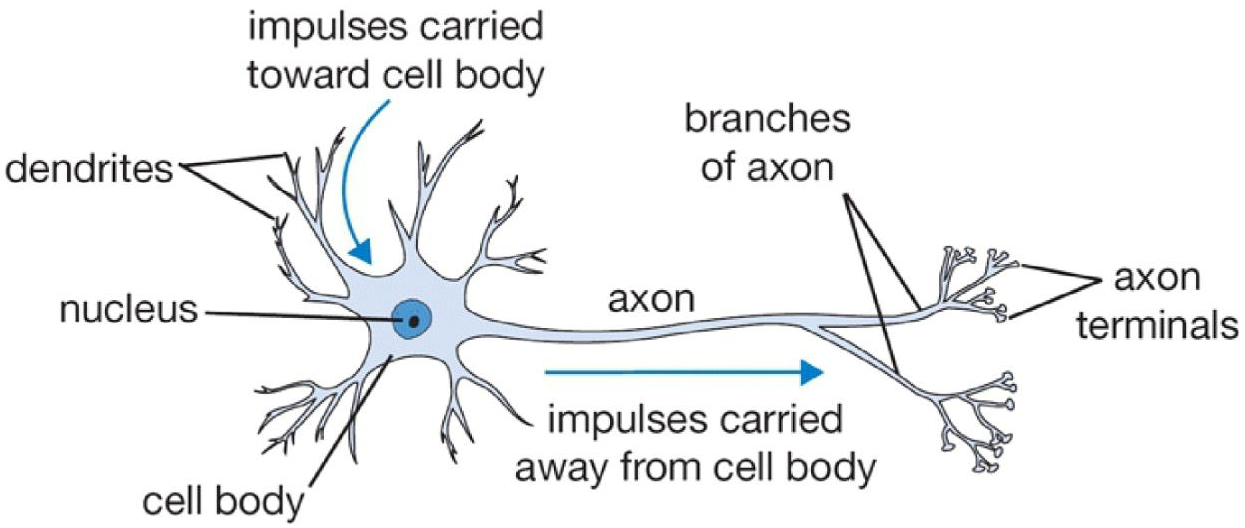
- The positive and negative cases cannot be separated by a plane
- **What can we do?**

How to construct nonlinear classifiers?

- We would like to construct **non-linear discriminative classifiers** that utilize functions of input variables
- Use a large number of simpler functions
  - If these functions are **fixed** (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs
  - Or we can make these functions **depend on additional parameters** → need an efficient method of training extra parameters

Inspiration: The brain

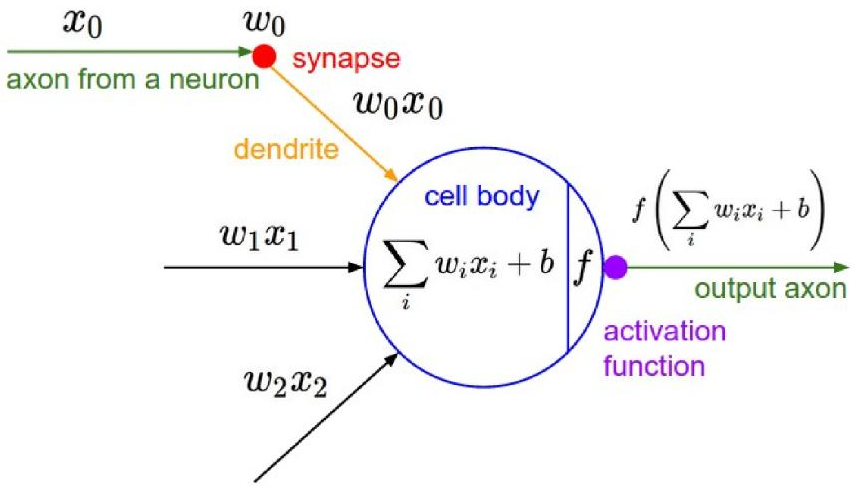
- Many machine learning methods inspired by biology.e.g., the (human) brain
- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons



> The basic computational unit of the brain: Neuron

Mathematical model of a neuron

- Neural networks define functions of the inputs (**hidden features**). computed by neurons
- Artificial neurons are called



> A mathematical model of the neuron in a neural network

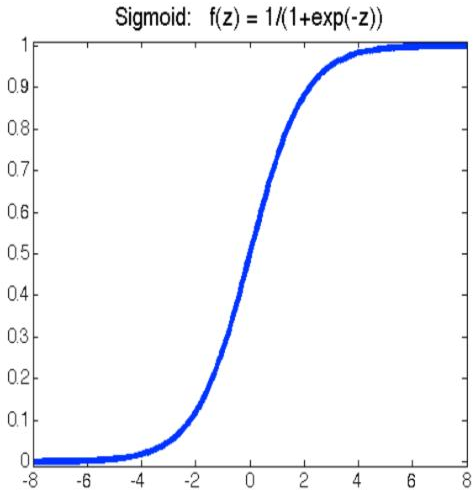
Activation functions

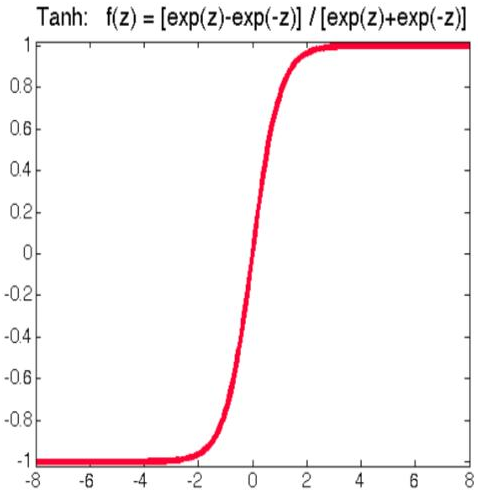
- Most commonly used activation functions:

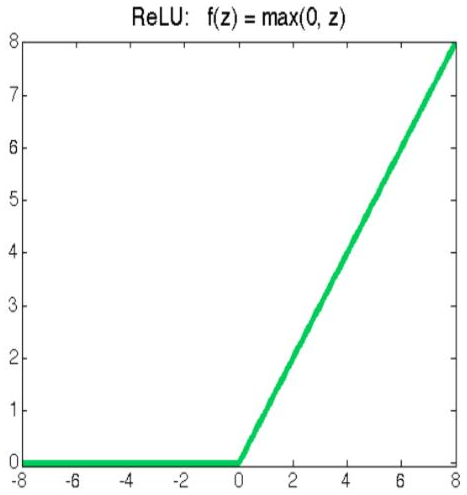
Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$

Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$

ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$







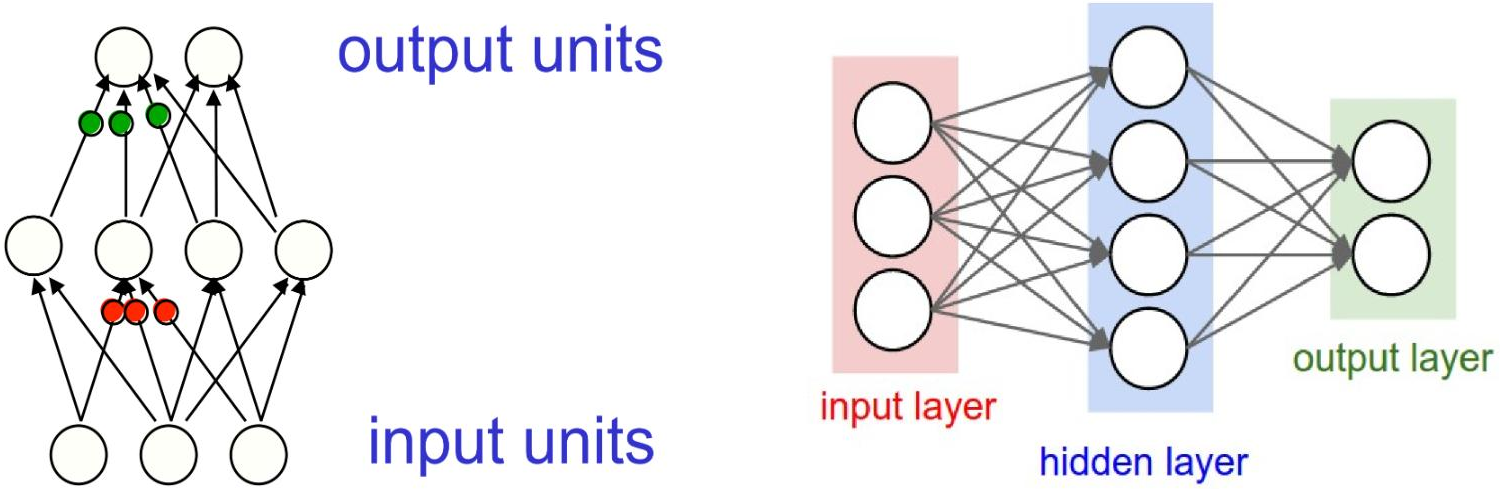
Neuron in Python

- Example in Python of a neuron with a sigmoid activation function

```
class Neuron:
    # ......
    def forward(self, inputs):
        # assume inputs and weights are 1-D numpy arrays and bias is a number
        cel1_body_sum = np.sum(inputs * self.weights) + self.bias
        # sigmoid activation function return output
        output = 1.0 / (1.0 + math.exp (- cell_body_sum))
        return output
```

Neural network architecture

- Network with one layer of four hidden units:

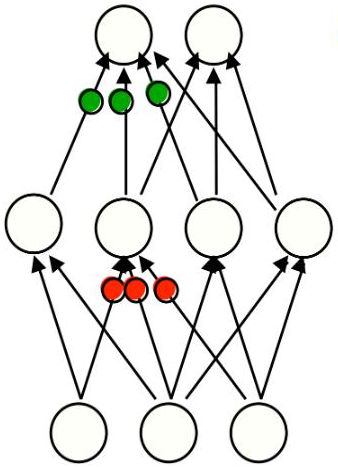


> Two different visualizations of a 2-layer neural network.
> In this example: 3 input units, 4 hidden units and 2 output units.

- Each unit computes its value based on linear combination of values of units that point into it, and an activation function

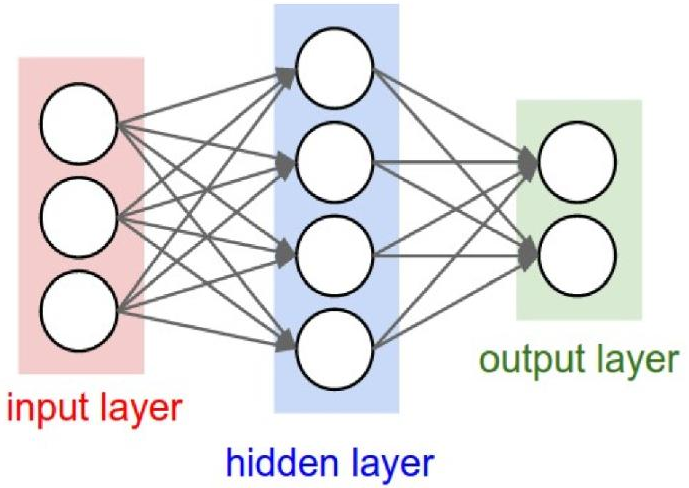
Neural network architecture

- Network with one layer of four hidden units:



output units

input units

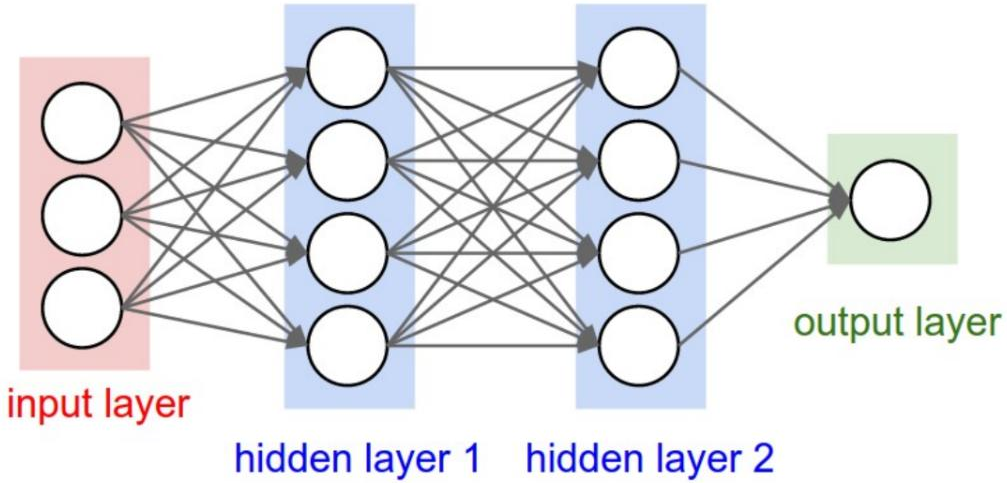


> Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- Naming conventions; **a 2-layer neural network**:
  - One layer of hidden units
  - One output layer (we do not count the inputs as a layer)

Neural network architecture

- Going deeper: a 3-layer neural network with two layers of hidden units

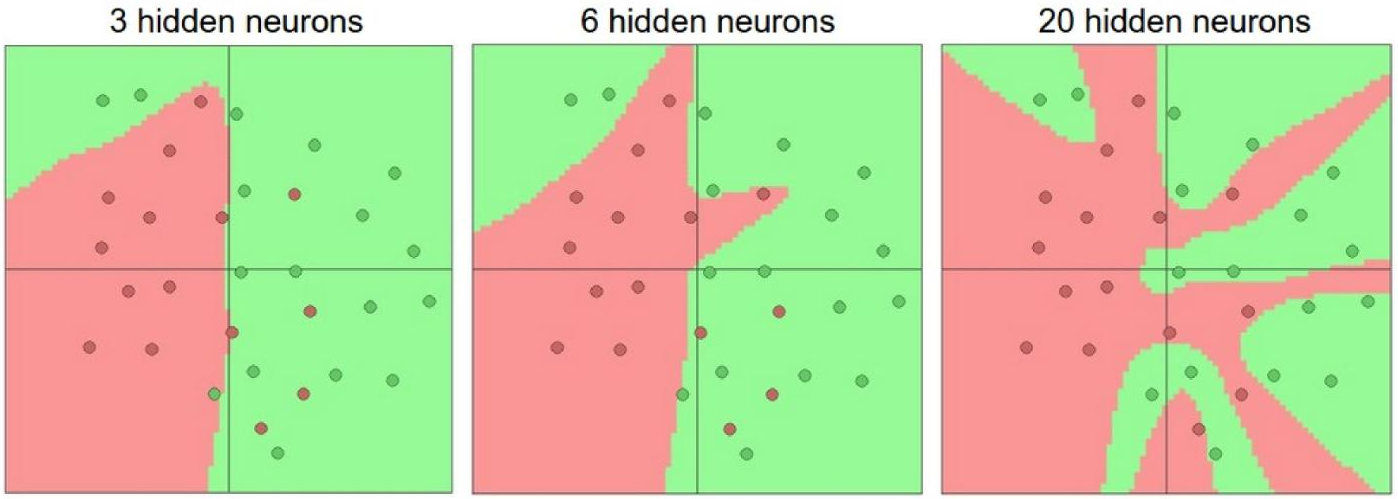


> A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- Naming conventions; **a N-layer neural network**:
  - $N-1$ layers of hidden units
  - One output layer

Representational power

- Neural network with **at least one hidden layer** is a universal approximator (can represent any function).



- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper?
  - Paper: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana

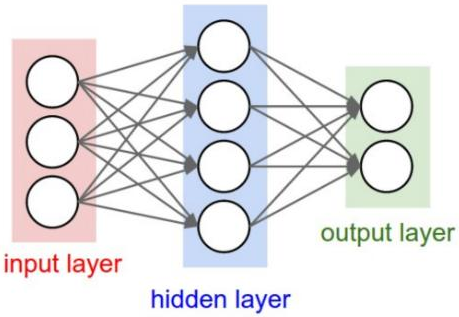
Neural networks

- The first neural network we learn is **Multi-Layer Perceptron (MLP)**

- We only need to know two algorithms
  - **Forward pass:** performs inference
  - **Backward pass:** performs learning

Forward pass

- What does the network compute?



- Output of the network can be written as:

$$h_j(x) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(x) = g(w_{k0} + \sum_{j=1}^{J} h_j(x) w_{kj})$$

> j indexing hidden units, k indexing the output units, D number of inputs, J number of hidden units

Forward pass

- Output of the network can be written as:

$$h_j(x) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

$$o_k(x) = g(w_{k0} + \sum_{j=1}^{J} h_j(x) w_{kj})$$

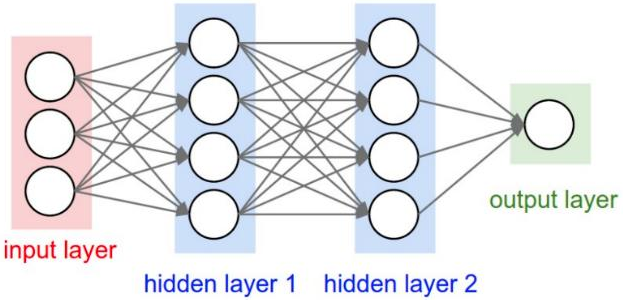
> j indexing hidden units, k indexing the output units, D number of inputs, J number of hidden units

- **Activation functions** f, g: sigmoid/logistic, tanh, or ReLU

$$\sigma(z) = \frac{1}{1+\exp(-z)}, \quad \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}, \quad \text{ReLU}(z) = \max(0, z)$$

Forward pass: An example

- 3-layer network

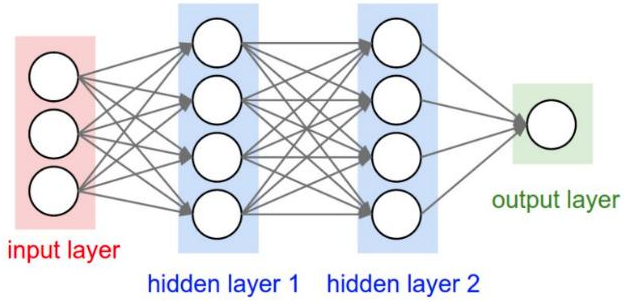


- Can be implemented efficiently using matrix operations

$$
\begin{aligned}
h^1 &= f(W_1 x + b_1) \\
h^2 &= f(W_2 h^1 + b_2) \\
o &= g(W_3 h^2 + b_3)
\end{aligned}
$$

- Example above: W_1 is matrix of size 4×3, W_2 is 4×4.
  - What about biases and W_3?
  - What does the (i, j)-th entry of W_1 refer to? e.g, $W_1(2, 3)$?

Forward pass in Python



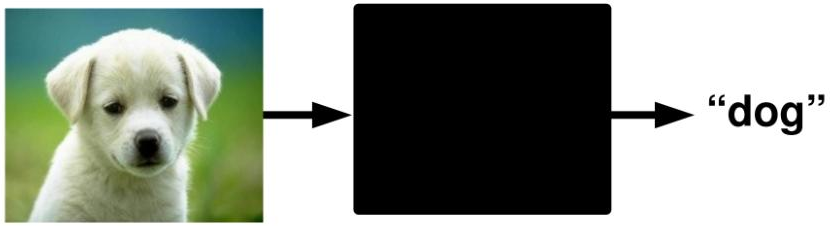
- Example code for a forward pass for a 3-layer network in Python:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0 / (1.0 + np.exp(-x))      # activation function (use sigmoid)
x = np.random.randn(3, 1)                   # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1)                  # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2)                 # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3                   # output neuron (1x1)
```

Forward propagation: Prediction

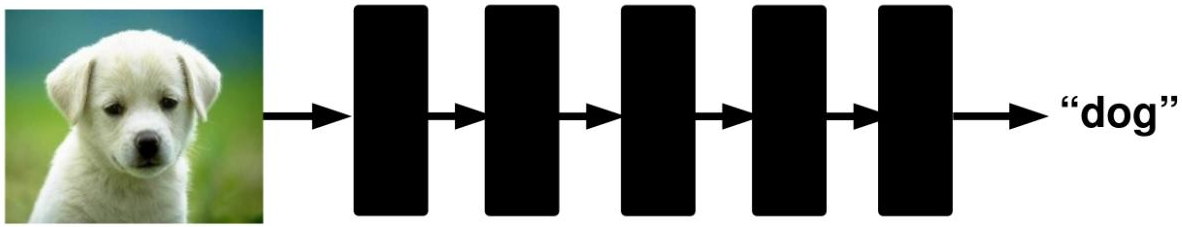
Supervised Learning: Examples

Classification



Supervised Deep Learning

Classification

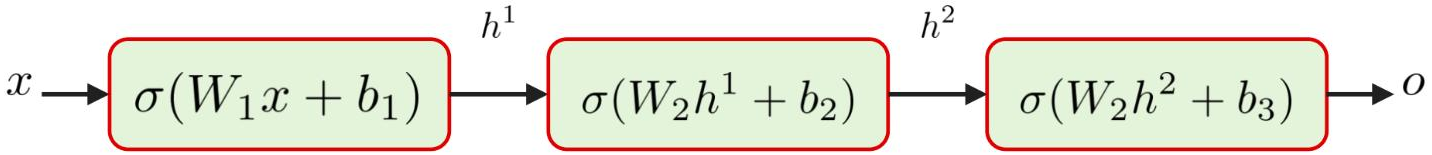


Summary of forward propagation

- Deep learning uses **composite of simple functions** (e.g., ReLU, sigmoid, tanh) to create complex non-linear functions

> Note: a composite of linear functions is linear!

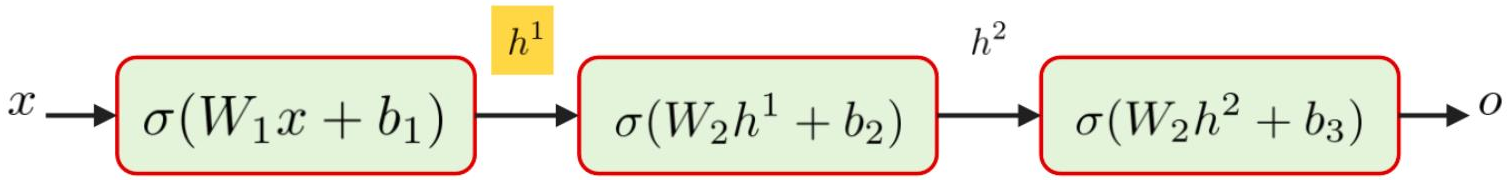
- Example: 2 hidden layer NN (now matrix and vector form!)



- x is the input
- o is the output (what we want to predict)
- h^i is the i-th hidden layer
- W_i and b_i are the parameters of the i-th layer

Summary of forward propagation

- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input

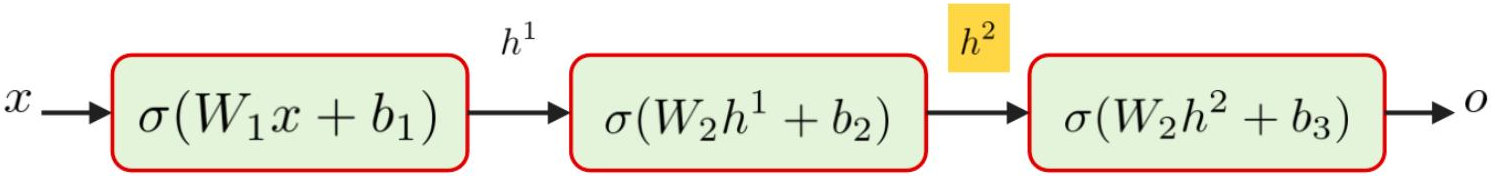


- Do it in a compositional way,

$$h^1 = \sigma(W_1x + b_1)$$

Summary of forward propagation

- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input

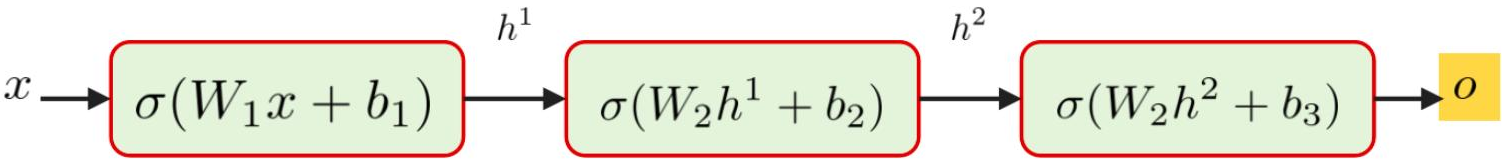


- Do it in a compositional way

$$
\begin{aligned}
h^1 &= \sigma(W_1x + b_1) \\
h^2 &= \sigma(W_2h^1 + b_2)
\end{aligned}
$$

Summary of forward propagation

- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input



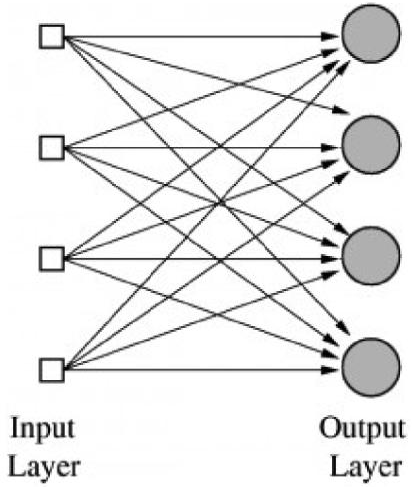
- Do it in a compositional way

$$
\begin{aligned}
h^1 &= \sigma(W_1x + b_1) \\
h^2 &= \sigma(W_2h^1 + b_2) \\
o &= \sigma(W_3h^2 + b_3)
\end{aligned}
$$

Now we come to Backward Propagation

Special case

- What is a single layer (no hiddens) network with a sigmoid act. function?



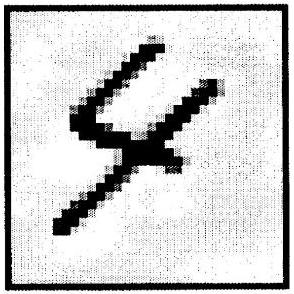
- Network:

$$z_k = w_{k0} + \sum_{i=1}^{D} w_{ki} x_i, \qquad o_k = \frac{1}{1 + \exp(-z_k)}$$

- Logistic regression!

Example application

- Classify image of handwritten digit (32x32 pixels): 4 vs non-4



- How would you build your network?
- E.g., use one unit at the output layer and the sigmoid activation function:

$$z = w_0 + \sum_{i=1}^{D} w_i x_i, \qquad o = \frac{1}{1 + \exp(-z)}$$

- What is the input feature and what is D for this example?

Training neural networks

- How can we **train** the network, that is, adjust all the parameters $w = [w_0, w_1, \cdots, w_D]$?
- Define a loss function, e,g, **cross-entropy**

$$E(w) = -\sum_{n=1}^{N} \left[t^{(n)} \log o^{(n)} + \left(1 - t^{(n)}\right) \log \left(1 - o^{(n)}\right) \right]$$

- Gradient descent:

$$w(t+1) = w(t) - \eta \nabla E(w)$$

> where η is the learning rate

Useful derivatives

name	function	derivative
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

> where $\cosh(z) = \frac{\exp(z)+\exp(-z)}{2}$

- You should try to derive them!

Training neural networks

- **Back-Propagation:** An efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

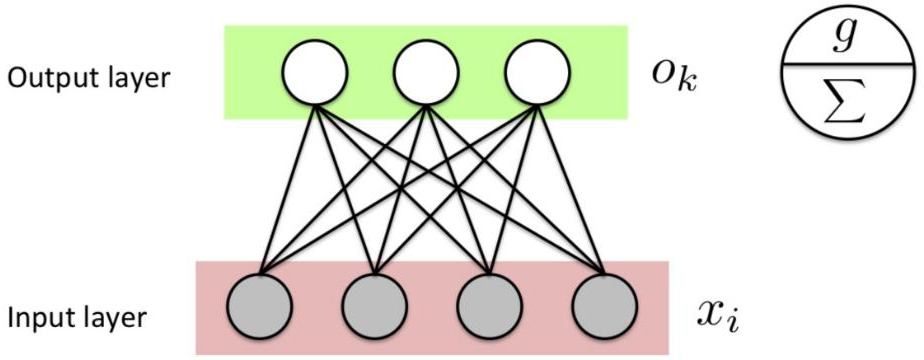
- Given any loss function $E()$, activation functions $g()$ and $f()$, just need to derive gradients

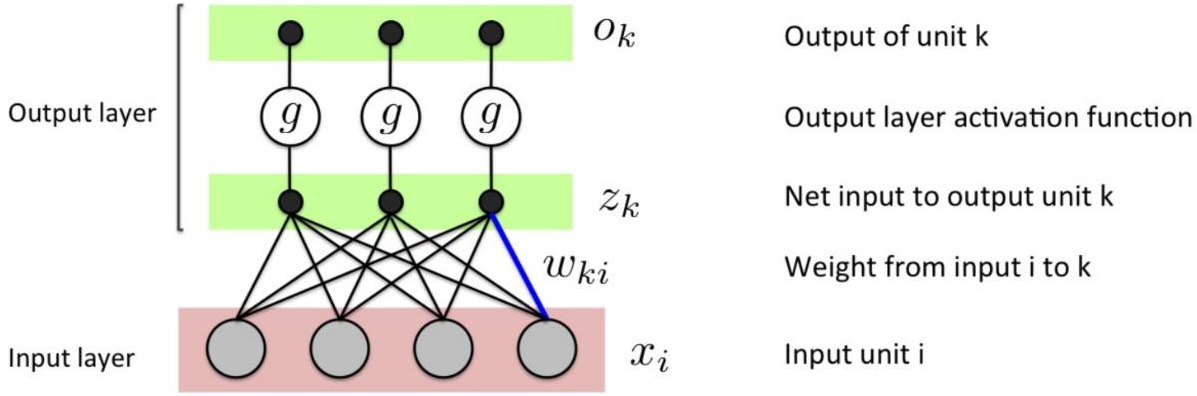
Key idea behind backpropagation

- We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use **error derivatives w.r.t. hidden activities**
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently Once we have the error derivatives for the hidden activities, it's easy to get the error derivatives for the weights going into a hidden unit
- This is just the **chain rule**!

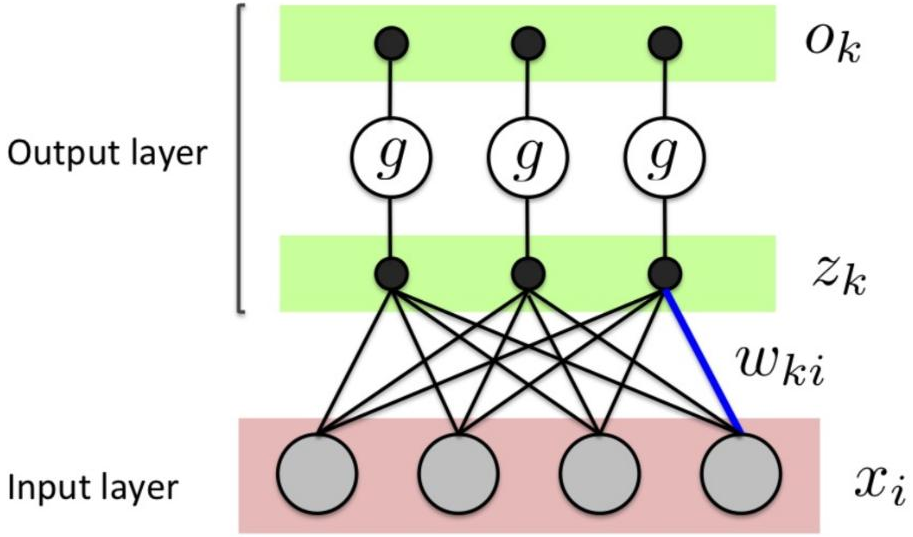
Computing gradients: Single layer network

- Let's take a single layer network and draw it a bit differently





Computing gradients: Single layer network

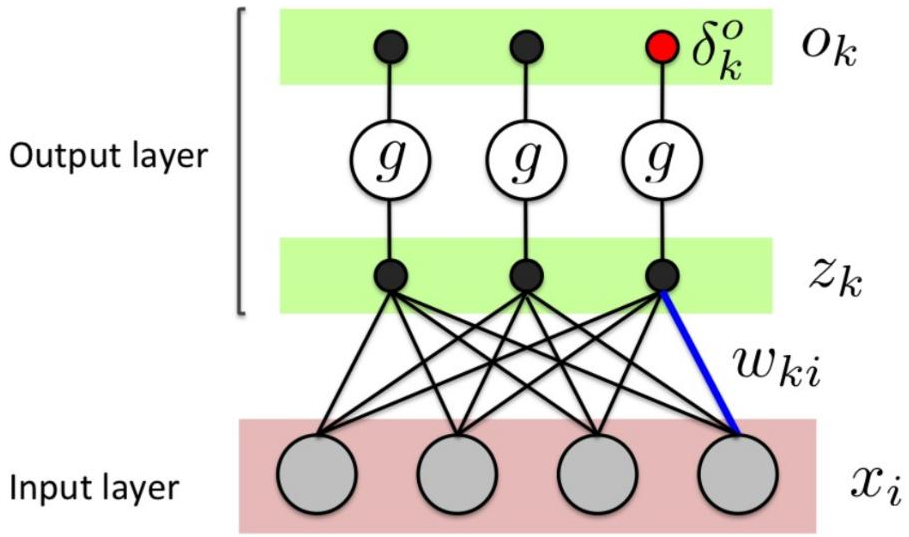


- Error gradients for single layer network:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

- Error gradient is computable for any continuous activation function g(). and any continuous error function

Computing gradients: Single layer network

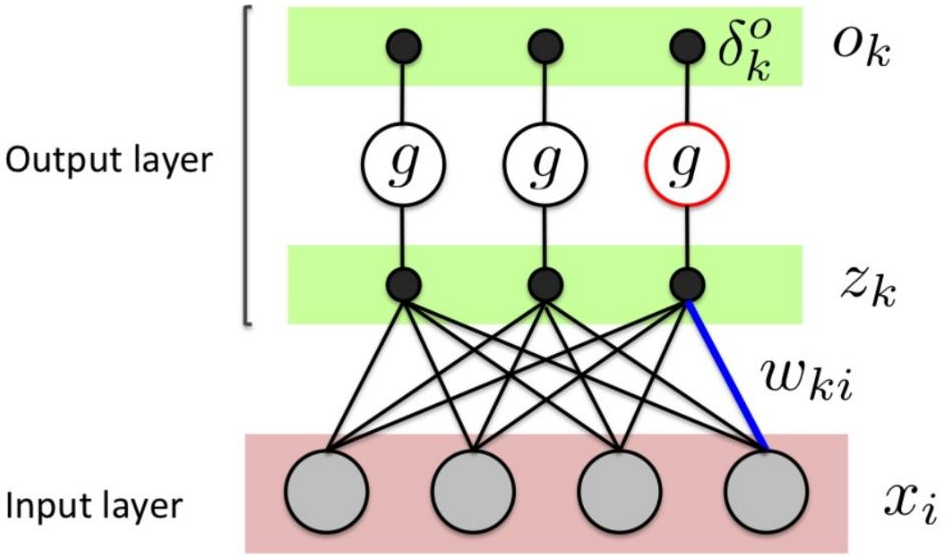


$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

- Define the **gradient of the loss wrt the output**:

$$\frac{\partial E}{\partial w_{ki}} = \underbrace{\frac{\partial E}{\partial o_k}}_{\delta_k^o} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

Computing gradients: Single layer network

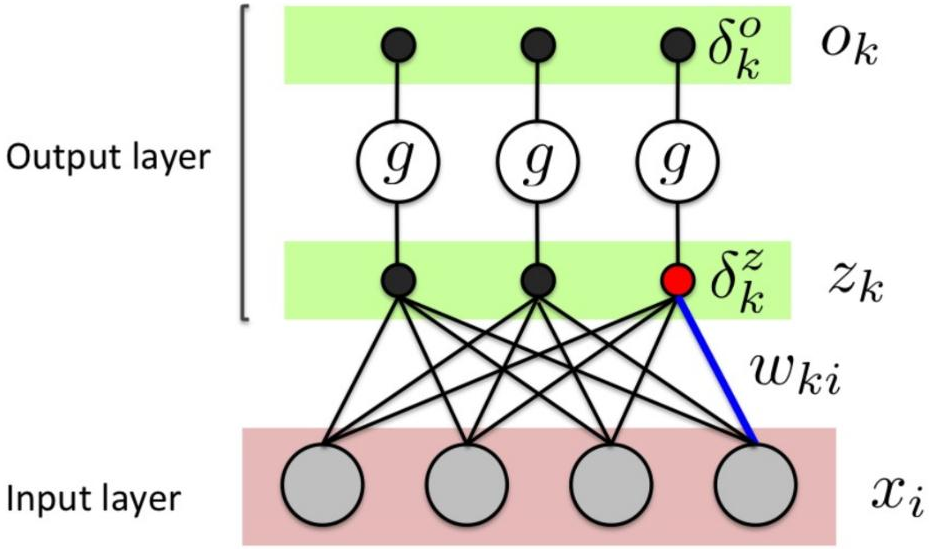


$$\frac{\partial o_k}{\partial z_k}$$

- Define the **gradient of the output wrt the weighted sum**:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^o \frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}}$$

Computing gradients: Single layer network

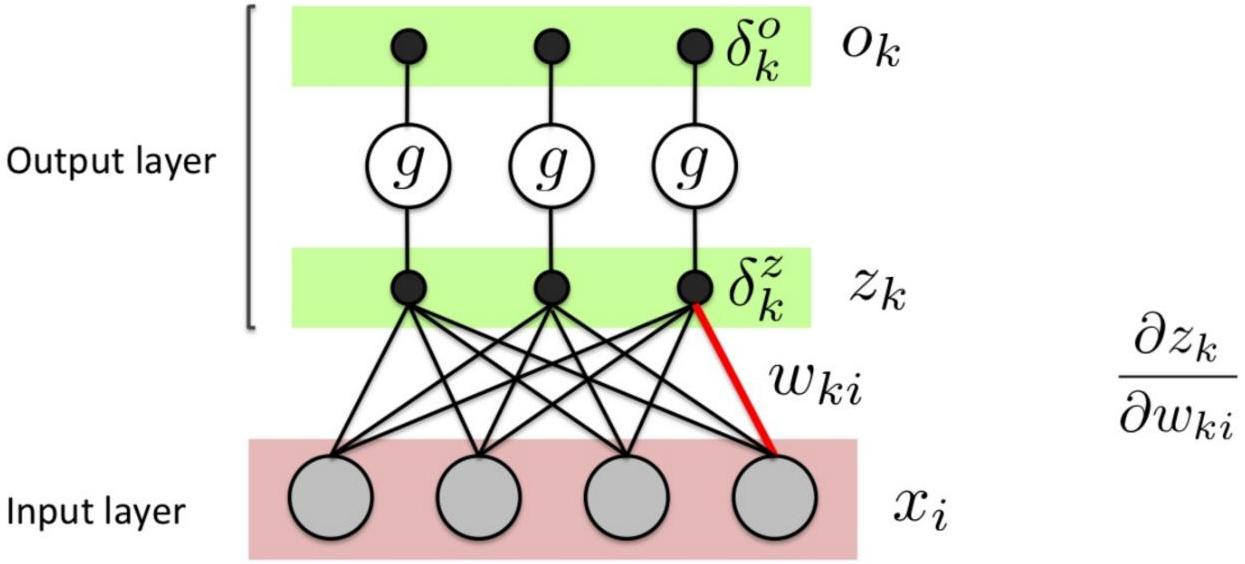


$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

- Backward pass for the **gradient of the loss wrt the weighted sum**:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \underbrace{\delta_k^o \cdot \frac{\partial o_k}{\partial z_k}}_{\delta_k^z}\frac{\partial z_k}{\partial w_{ki}}$$

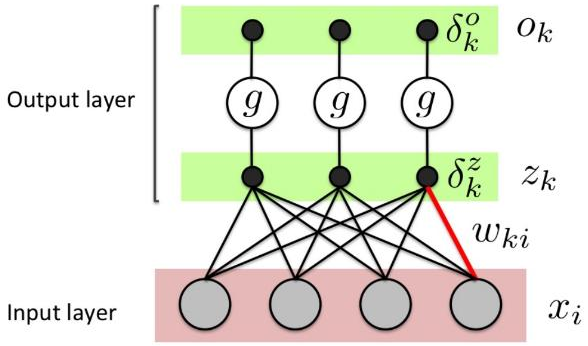
Computing gradients: Single layer network



- Define the **gradient of the weighted sum wrt the weights**:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \frac{\partial z_k}{\partial w_{ki}} = \delta_k^z \cdot x_i$$

Computing gradients: An example



- Assuming the loss function is MSE on a single example n, namely,

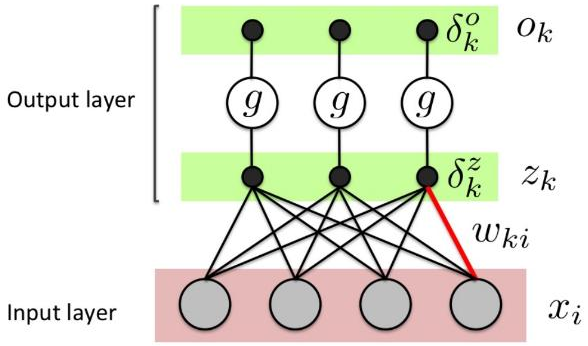
$$E = \sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$$

- And consider the logistic activation function:

$$g(z) = \frac{1}{1 + \exp(-z)}$$

- **What are the error gradients**: δ_k^o, δ_k^z, and $\frac{\partial E}{\partial w_{ki}}$?

Computing gradients: An example



- Forward propagation:

$$\text{FP-1:} \quad z_k = w_{k0} + \sum_{i=1}^{D} w_{ki} x_i$$

$$\text{FP-2:} \quad o_k = g(z_k)$$

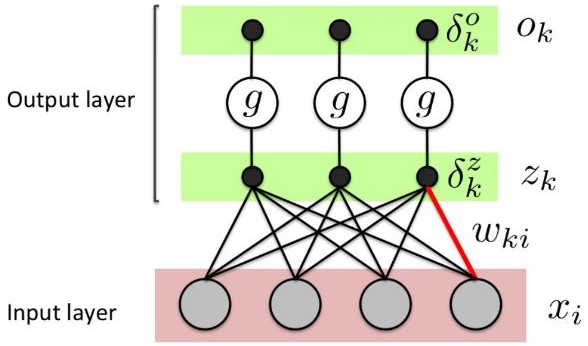
- Backward propagation:

$$\text{BP-1:} \quad \delta_k^o = \frac{\partial E}{\partial o_k} = o_k - t_k$$

$$\text{BP-2:} \quad \delta_k^z = \delta_k^o \cdot g'(z) = \delta_k^o \cdot g(z)(1 - g(z)) = \delta_k^o \cdot o_k(1 - o_k)$$

$$\text{BP-3:} \quad \frac{\partial E}{\partial w_{ki}} = \delta_k^z \cdot x_i$$

Computing gradients: A matrix form



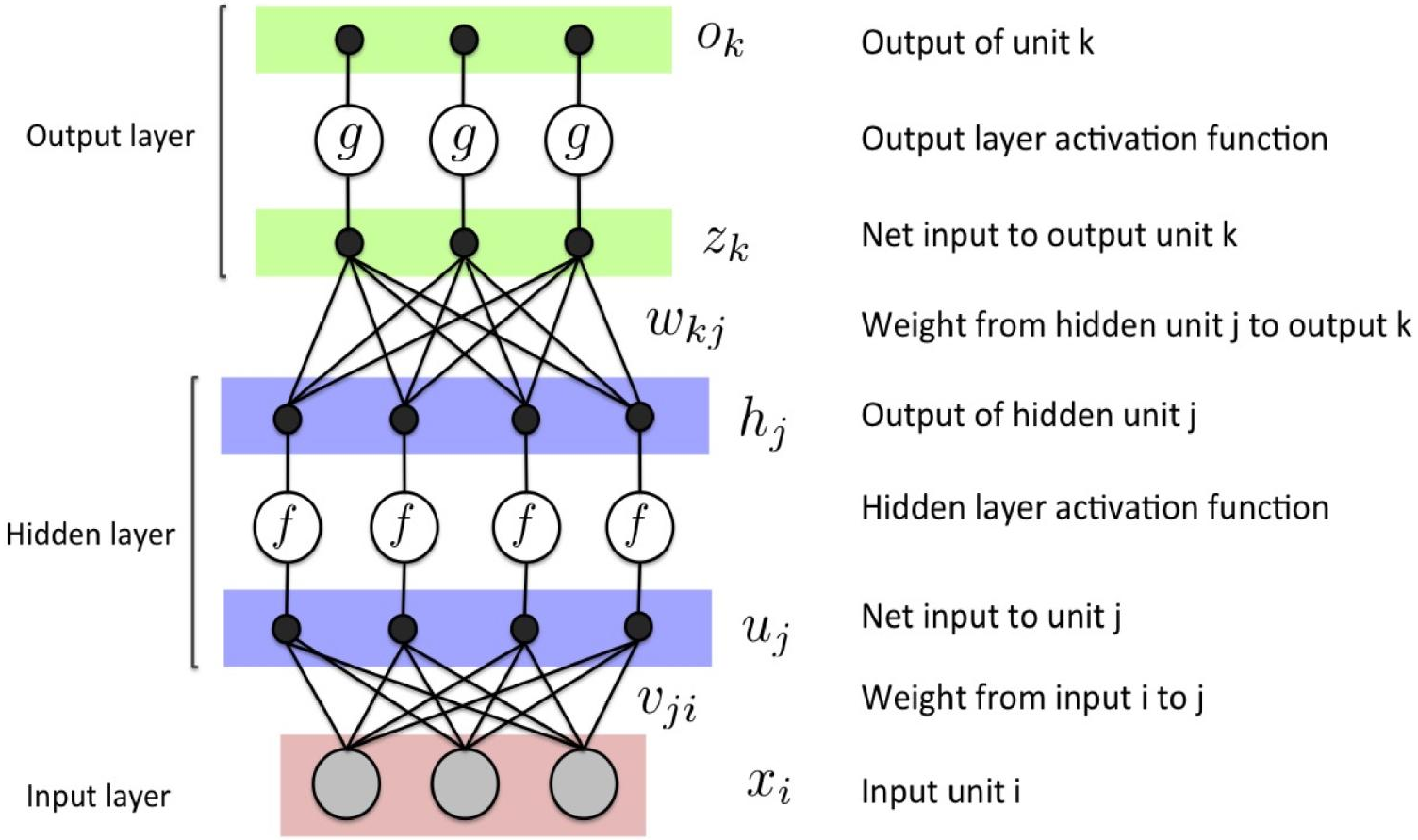
- The gradient of the loss wrt the weights:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}, \qquad \frac{\partial E}{\partial W} = \begin{bmatrix} \delta_1^z \\ \delta_2^z \\ \delta_3^z \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} = \begin{bmatrix} \cdots & \vdots & \cdots \\ \cdots & \delta_k^z x_i & \cdots \\ \cdots & \vdots & \cdots \end{bmatrix}$$

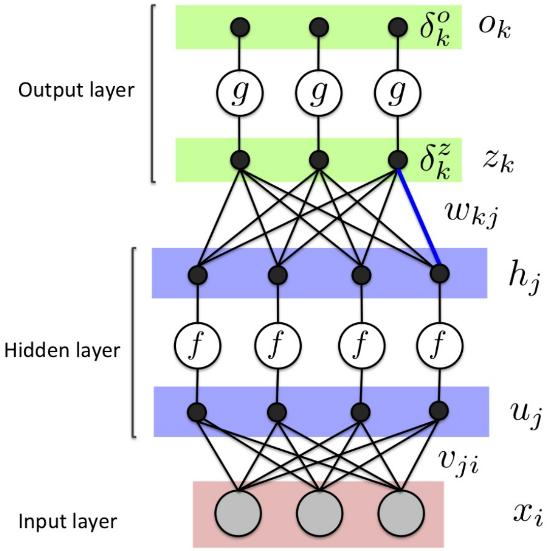
- The gradient of the loss wrt the biases:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}, \qquad \frac{\partial E}{\partial b} = \begin{bmatrix} \delta_1^z \\ \delta_2^z \\ \delta_3^z \end{bmatrix}$$

Multi-layer neural network



Gradient descent for multi-layer network

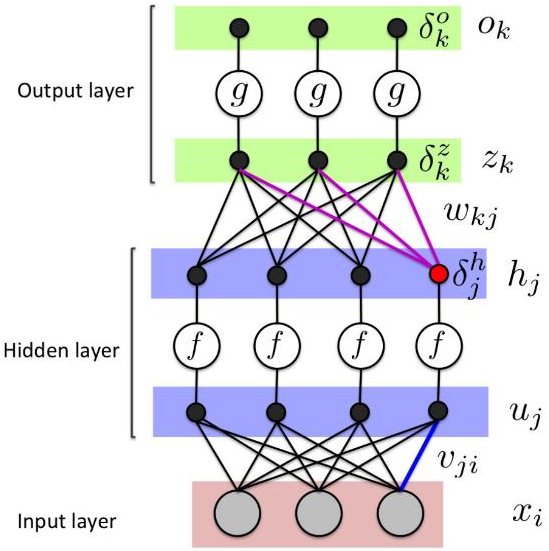


- The **output-layer weight gradients** for MLP are the same as for a single layer network

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \delta_k^{z,(n)} h_j^{(n)}$$

> where $\delta_k^{z,(n)}$ is the error w.r.t. the net input for unit k

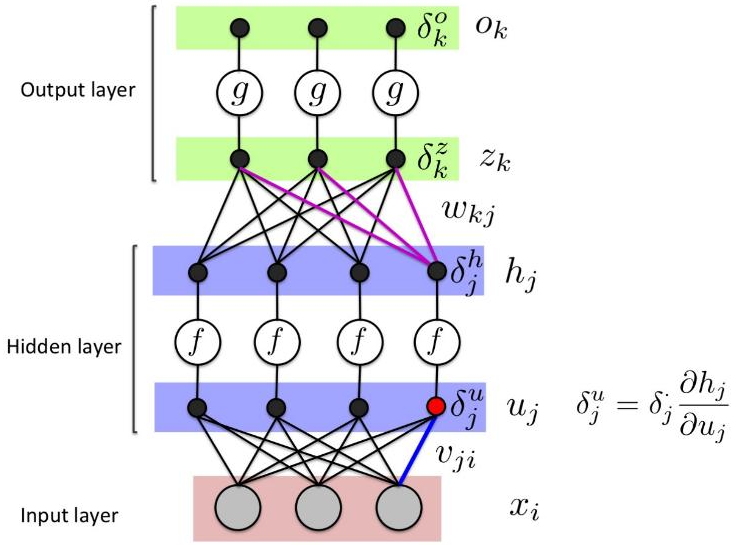
Gradient descent for multi-layer network



- **Hidden weight gradients** are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

Gradient descent for multi-layer network

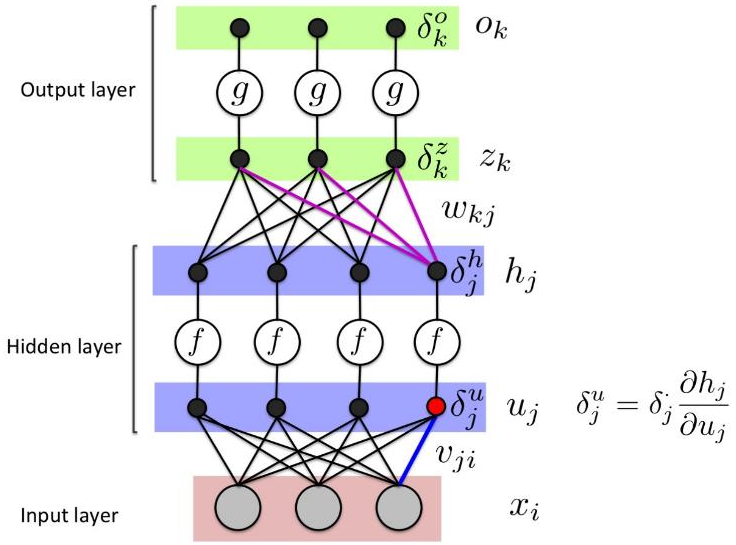


- **Hidden weight gradients** are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial u_j^{(n)}} = \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} = \delta_j^{h,(n)} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} = \delta_j^{u,(n)}$$

Gradient descent for multi-layer network

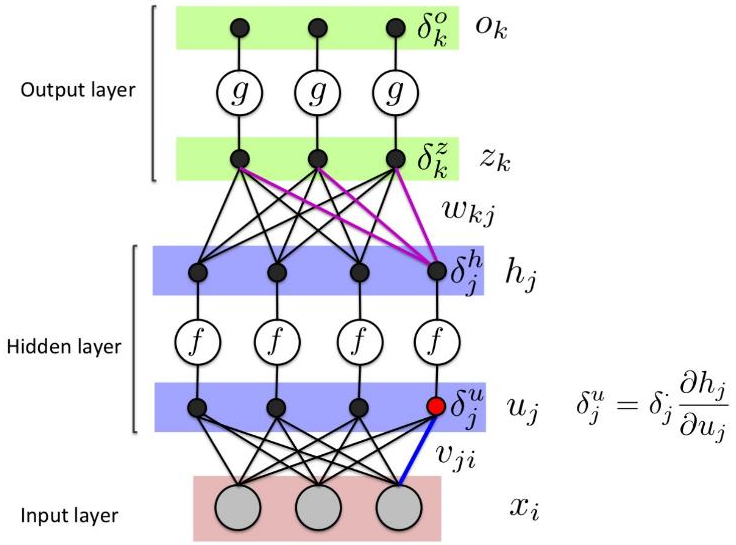


- **Hidden weight gradients** are then computed via back-prop:

$$\frac{\partial E}{\partial u_j^{(n)}} = \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} = \delta_j^{h,(n)} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} = \delta_j^{u,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \frac{\partial E}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \delta_j^{u,(n)} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \delta_j^{u,(n)} x_i^{(n)}$$

Computing gradients: An example



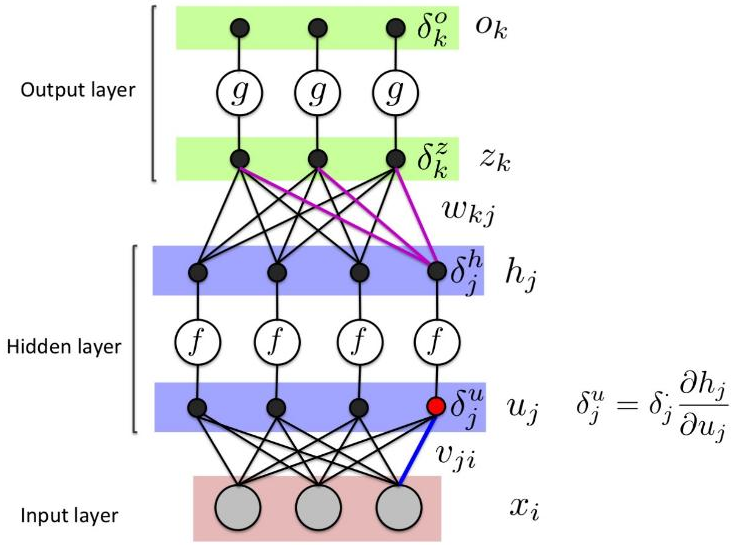
- For binary classification, then cross-entropy error function often does better

$$E(w) = -\left[t^{(n)} \log o^{(n)} + \left(1 - t^{(n)}\right) \log \left(1 - o^{(n)}\right)\right]$$

- It can be derived that

$$\frac{\partial E}{\partial o^{(n)}} = (o^{(n)} - t^{(n)})/(o^{(n)}(1 - o^{(n)}))$$

Computing gradients: An example

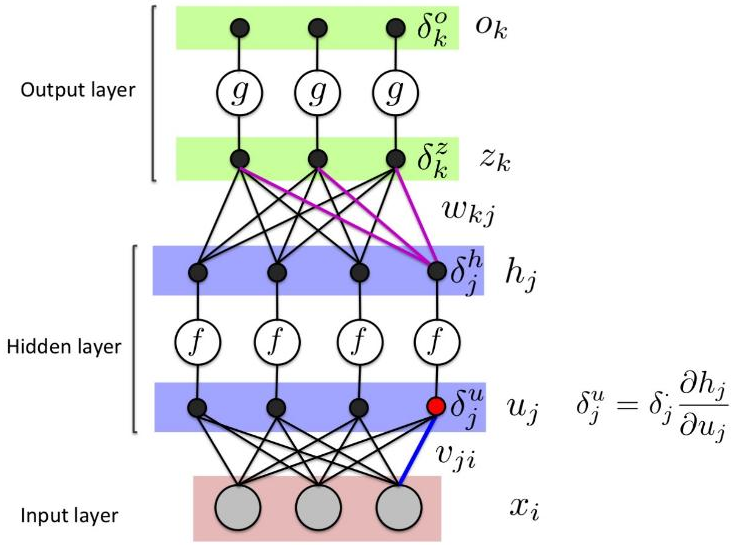


- For binary classification, then cross-entropy loss function often does better

$$E(w) = -\left[t^{(n)} \log o^{(n)} + \left(1 - t^{(n)}\right) \log \left(1 - o^{(n)}\right)\right]$$

- Consider logistic activation functions.
- What are **the error gradients**: δ_k^o, δ_k^z, $\frac{\partial E}{\partial w_{kj}}$, δ_j^h, δ_j^u, and $\frac{\partial E}{\partial v_{ji}}$?

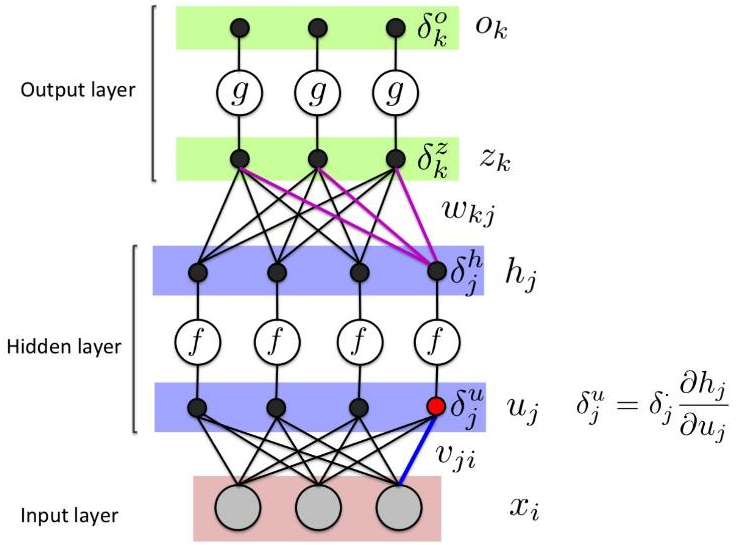
Computing gradients: A matrix form



- The gradient of the loss wrt the output weights and biases:

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \delta_1^z \\ \delta_2^z \\ \delta_3^z \end{bmatrix} \begin{bmatrix} h_1 & h_2 & h_3 & h_4 \end{bmatrix} = \begin{bmatrix} \cdots & \vdots & \cdots \\ \cdots & \delta_k^z h_j & \cdots \\ \cdots & \vdots & \cdots \end{bmatrix}, \qquad \frac{\partial E}{\partial b_w} = \begin{bmatrix} \delta_1^z \\ \delta_2^z \\ \delta_3^z \end{bmatrix}$$

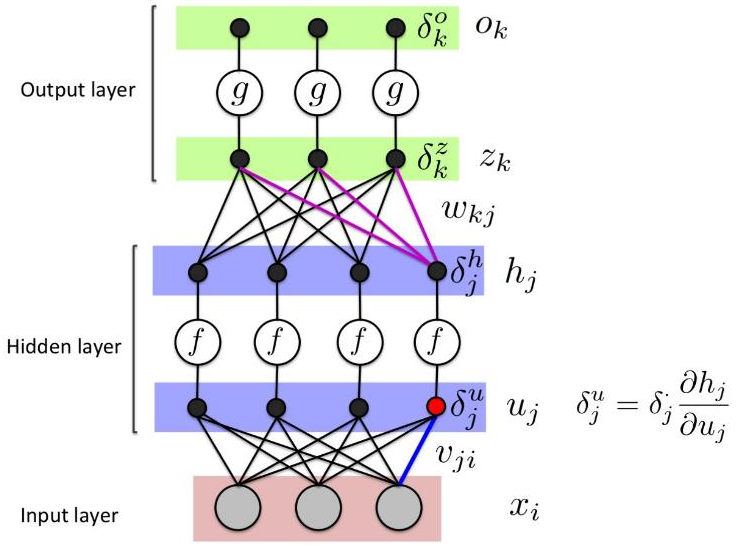
Computing gradients: A matrix form



- The gradient of the loss wrt the hidden weights and biases:

$$\frac{\partial E}{\partial V} = \begin{bmatrix} \delta_1^u \\ \delta_2^u \\ \delta_3^u \\ \delta_4^u \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} \cdots & \vdots & \cdots \\ \cdots & \delta_j^u x_i & \cdots \\ \cdots & \vdots & \cdots \end{bmatrix}, \qquad \frac{\partial E}{\partial b_v} = \begin{bmatrix} \delta_1^u \\ \delta_2^u \\ \delta_3^u \\ \delta_4^u \end{bmatrix}$$

Computing gradients: A matrix form



- The error gradients: Back propagation

$$\delta^z = \begin{bmatrix} \delta_1^z \\ \delta_2^z \\ \delta_3^z \end{bmatrix} = g'(z)\frac{\partial E}{\partial o}, \qquad \delta^u = \begin{bmatrix} \delta_1^u \\ \delta_2^u \\ \delta_3^u \\ \delta_4^u \end{bmatrix} = f'(z)W^T\delta^z$$

Ways to use weight derivatives

- Update in the scalar form

> after a full sweep through the training data (batch gradient descent)

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(o^{(n)}, t^{(n)})}{\partial w_{ki}}$$

> after each training case (stochastic gradient descent)

> after a mini-batch of training cases

- Update in the matrix form (batch update)

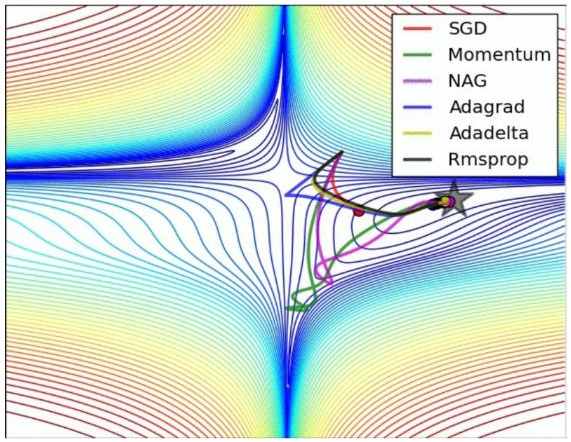
$$W \leftarrow W - \eta \frac{\partial E}{\partial W} = W - \eta \sum_{n=1}^{N} \frac{\partial E(o^{(n)}, t^{(n)})}{\partial W}$$

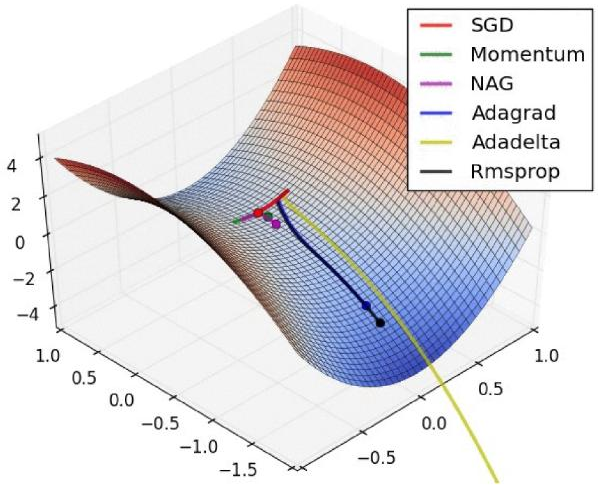
Ways to use weight derivatives

- How much to update
  - Use a fixed learning rate
  - Add momentum

$$w_{ki} \leftarrow w_{ki} - v$$

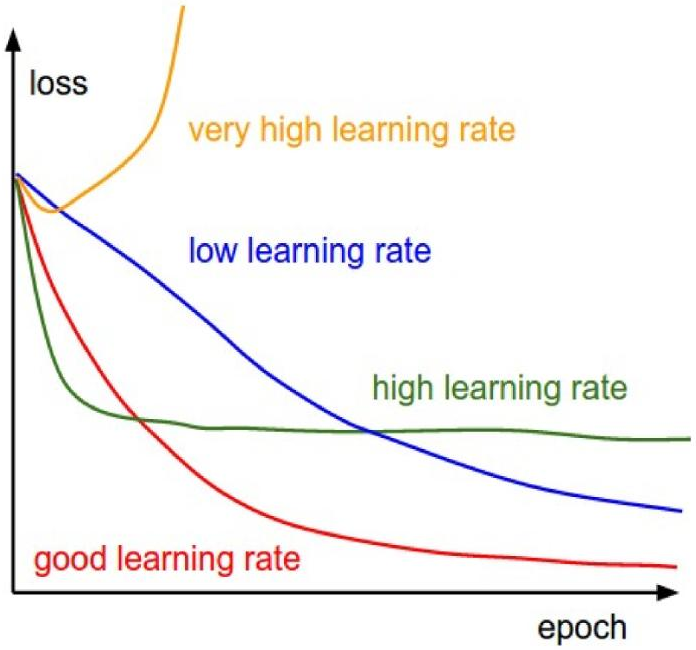
$$v \leftarrow \gamma v + \eta \frac{\partial E}{\partial w_{ki}}$$

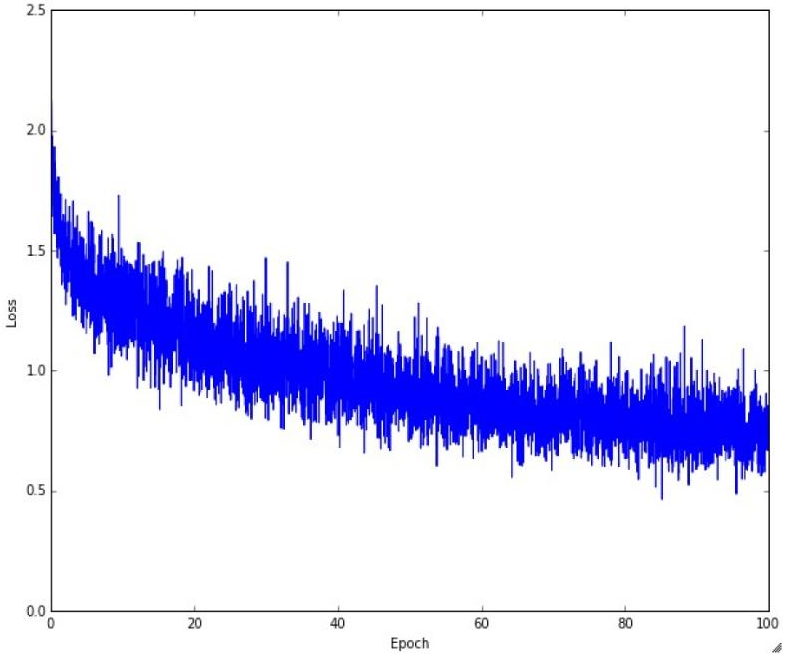




Monitor loss during rraining

- Check how your loss behaves during training, to spot wrong hyper parameters, bugs, etc

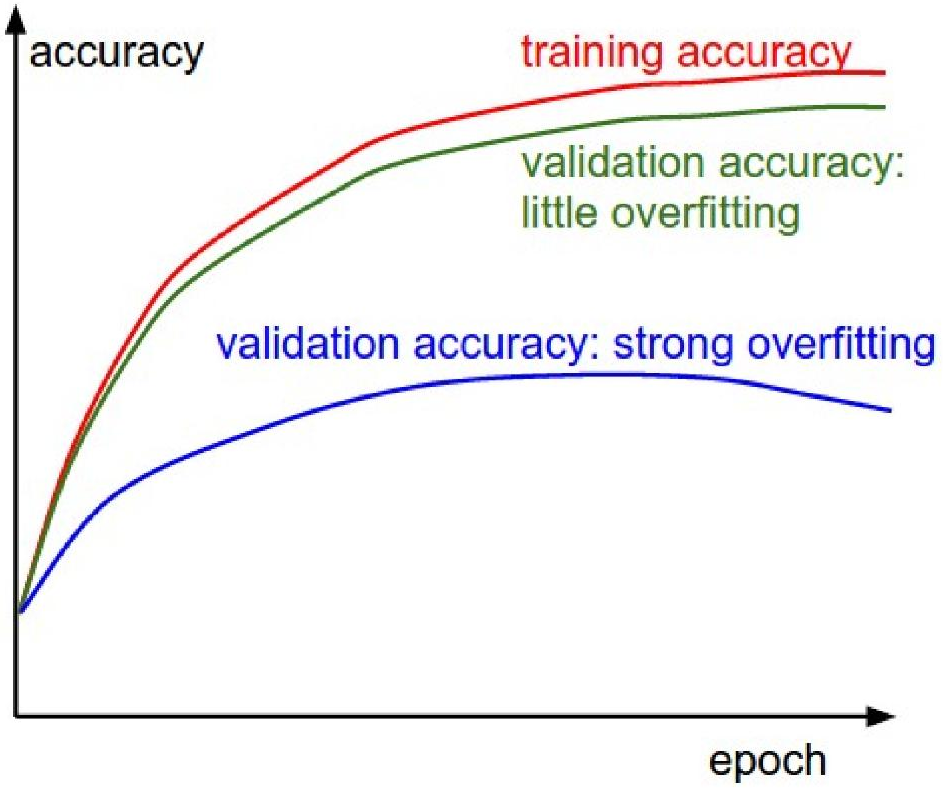




Left: Good vs bad parameter choices. **Right:** How a real loss might look like

Monitor accuracy on train/validation

- Check how your desired performance metrics behaves during training



Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains **noise**
  - The target values may be unreliable.
  - There is **sampling error**: There will be accidental regularities just because of the particular training cases that were chosen
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity
  - If the model is very flexible it can model the sampling error really well. This is **a disaster**.

Preventing overfitting

- Use a model that has the right capacity
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)
- Standard ways to limit the capacity of a neural net:
  - Limit the number of hidden units.
  - Limit the norm of the weights.
  - Stop the learning before it has time to overfit.

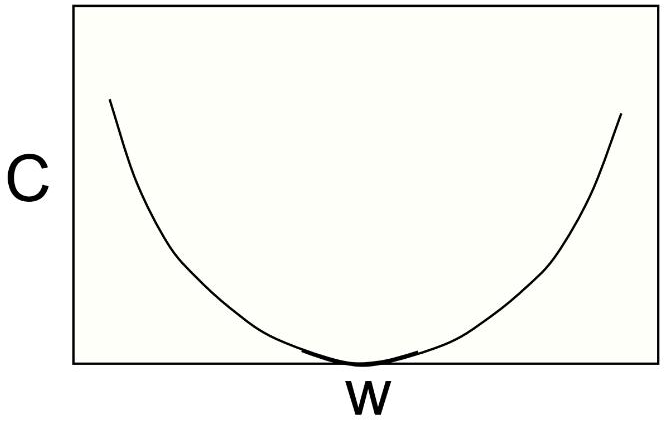
Limiting the size of the weights

- Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

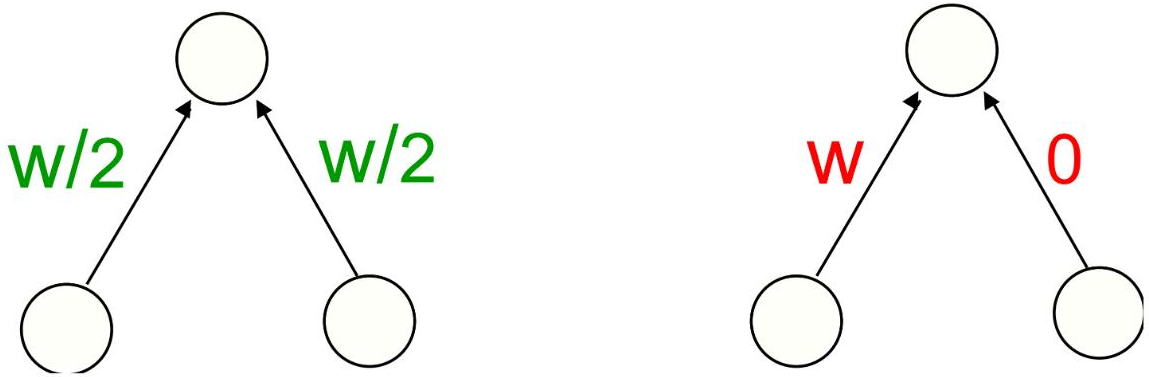
Keeps weights small unless they have big error derivatives.

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i. \qquad \text{When } \frac{\partial C}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda} \frac{\partial \ell}{\partial w_i}$$



The effect of weight-decay

- It prevents the network from using weights that it does not need
  - This can often improve **generalization** a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a **smoother** model in which the output changes more slowly as the input changes.
- But, if the network has two similar inputs it prefers to put half the weight on each rather than all the weight on one → other form of weight decay?



Deciding how Much to restrict the capacity

- How do we decide which regularizer to use and how strong to make it?
- So use a separate **validation set** to do model selection.

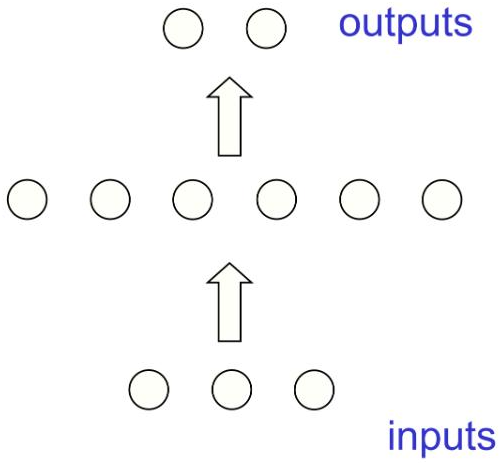
Using a validation set

- Divide the total dataset into three subsets
  - **Training data** is used for learning the parameters of the model.
  - **Validation data** is not used for learning but is used for deciding what type of model and what amount of regularization works best
  - **Test data** is used to get a final, unbiased estimate of how well the network works.We expect this estimate to be worse than on the validation data
- We could then re-divide the total dataset to get another unbiased estimate of the true error rate.

Preventing overfitting by early stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse
- The capacity of the model is limited because the weights have not had time to grow big.

Why early stopping works



- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.