SDM5013:
Deep Learning and Reinforcement Learning

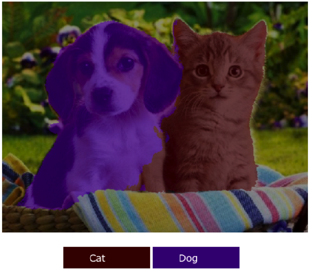
Zhiyun Lin

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

设计智造学院
School of System Design and Intelligent Manufacturing

- Multi-layer Perceptron
- Forward Propagation
- Backward Propagation

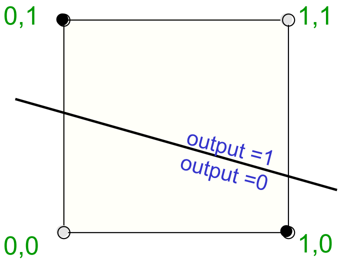
Motivating Examples



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 input
- 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 $\rightarrow$ 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

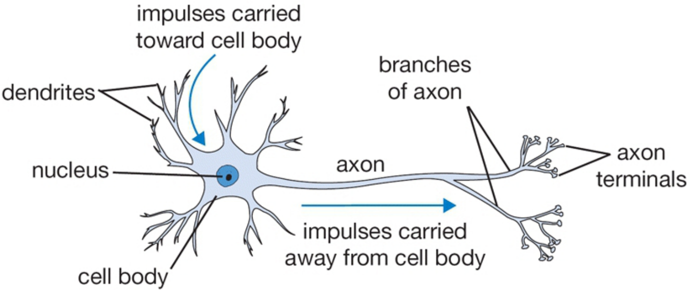


Figure : The basic computational unit of the brain: Neuron

Mathematical Model of a Neuron

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

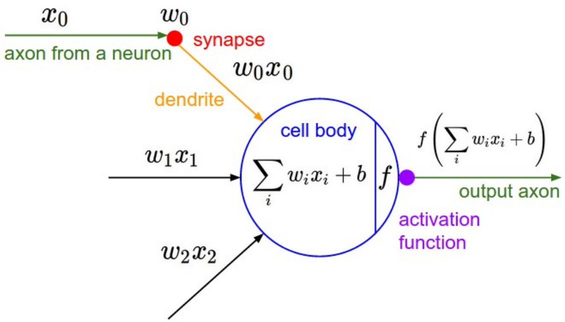
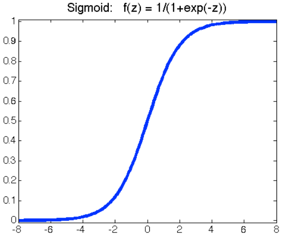


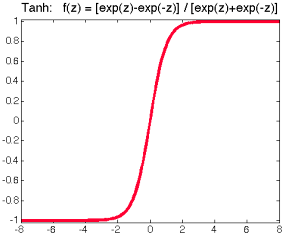
Figure : A mathematical model of the neuron in a neural network

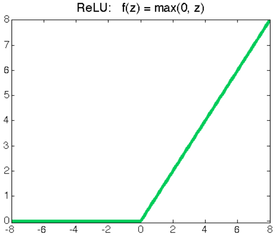
Activation Functions

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

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

Figure : Example code for computing the activation of a single neuron

Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:

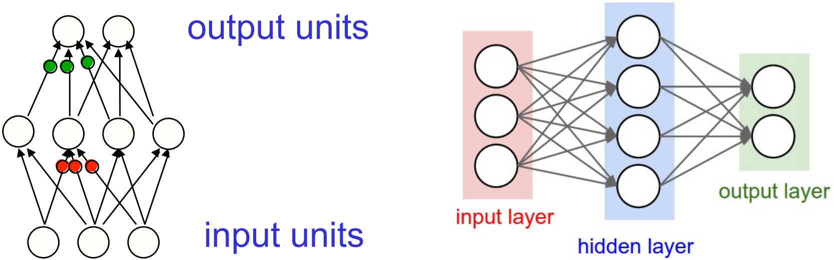


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

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

Neural Network Architecture (Multi-Layer Perceptron)

- Network with one layer of four hidden units:

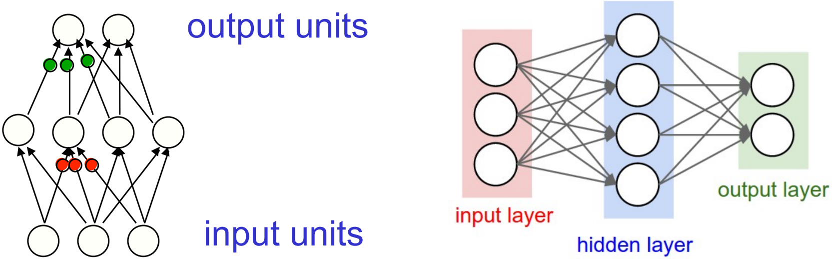


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

- 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 (Multi-Layer Perceptron)

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

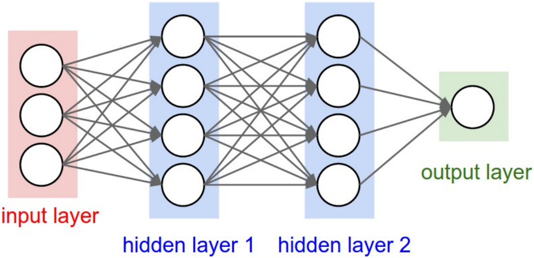
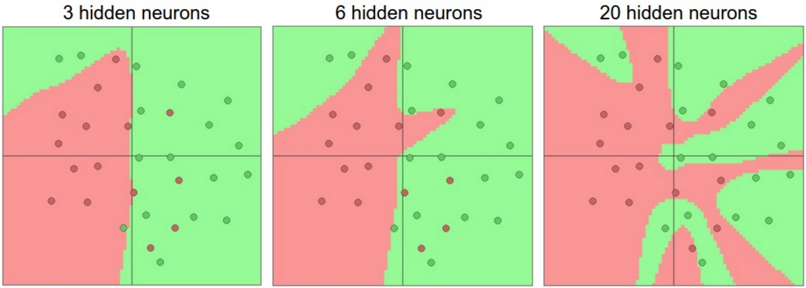


Figure : A 3-layer neural net with 3 input units, 4 hidden units in the first and second hidden layer and 1 output unit

- 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).
  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper

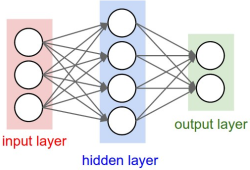


- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper? Read e.g.,: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana, Paper: paper]

Neural Networks

- 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:

$$
\begin{aligned}
h_j(\mathbf{x}) &= f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji}) \\
o_k(\mathbf{x}) &= g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})
\end{aligned}
$$

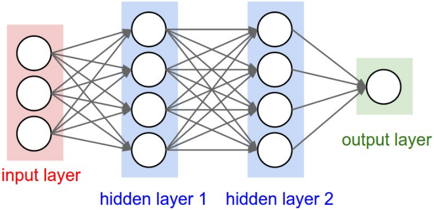
(j indexing hidden units, k indexing the output units, D number of inputs)

- Activation functions f, g: sigmoid/logistic, tanh, or rectified linear (ReLU)

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

Forward Pass in Python

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

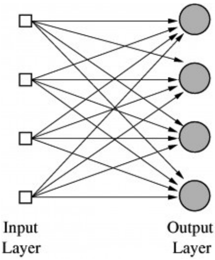



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate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

- Can be implemented efficiently using matrix operations
- Example above: W_1 is matrix of size 4×3, W_2 is 4×4. What about biases and W_3?

Special Case

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



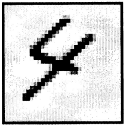
- Network:

$$
\begin{aligned}
o_k(\mathbf{x}) &= \frac{1}{1+\exp(-z_k)} \\
z_k &= w_{k0} + \sum_{j=1}^{J} x_j w_{kj}
\end{aligned}
$$

- Logistic regression!

Example Application

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



- How would you build your network?
- For example, use one hidden layer and the sigmoid activation function:

$$o_k(\mathbf{x}) = \frac{1}{1 + \exp(-z_k)}$$

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

- How can we train the network, that is, adjust all the parameters $\mathbf{w}$?

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \operatorname{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

  where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$

- Gradient descent:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

  where η is the learning rate (and E is error/loss)

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	$\mathrm{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

Training Neural Networks: Back-propagation

- 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)

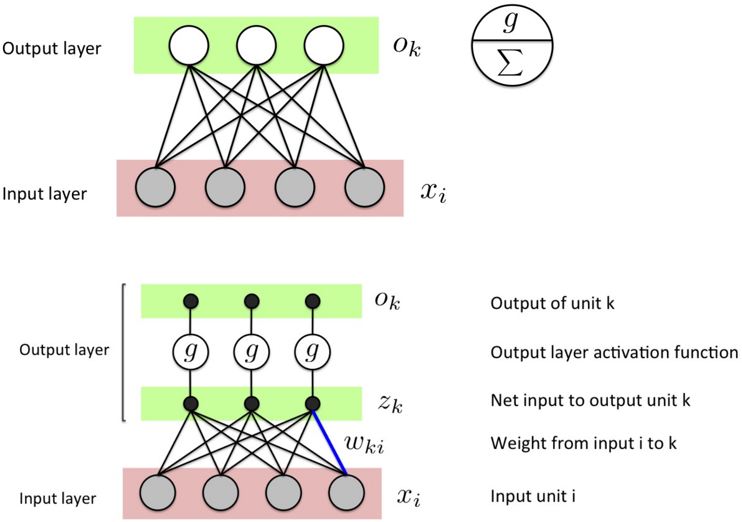
- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

Key Idea behind Backpropagation

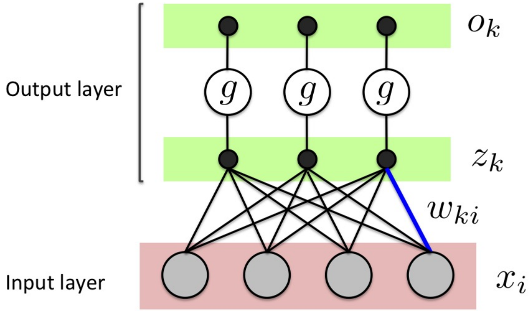
- We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore 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, its 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

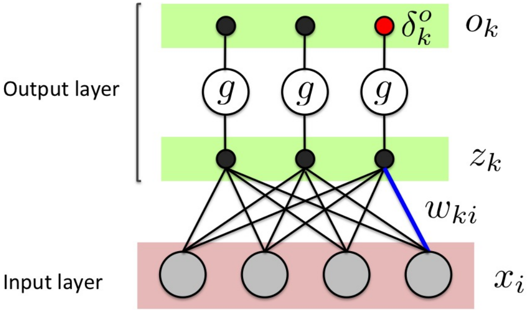


- Error gradients for single layer network:

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

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

Computing Gradients: Single Layer Network

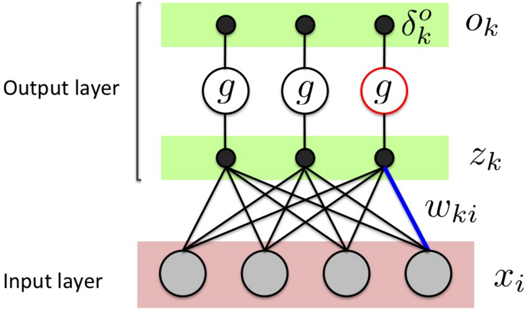


$$\delta_k^o = \frac{\partial E}{\partial o_k}$$

- Error gradients for single layer network:

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

Computing Gradients: Single Layer Network

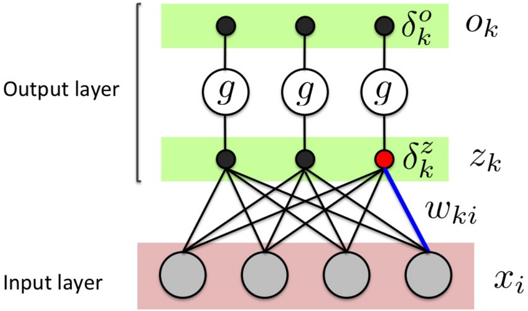


$$\frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

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

Computing Gradients: Single Layer Network

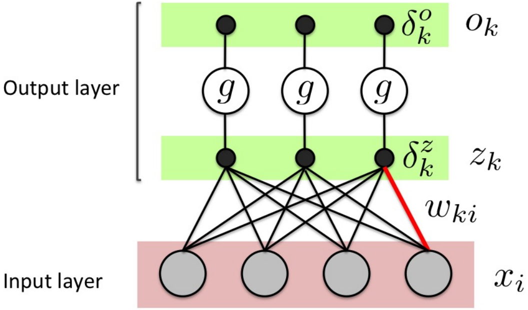


$$\delta_k^z = \delta_k^o \cdot \frac{\partial o_k}{\partial z_k}$$

- Error gradients for single layer network:

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

Computing Gradients: Single Layer Network



$$\frac{\partial z_k}{\partial w_{ki}}$$

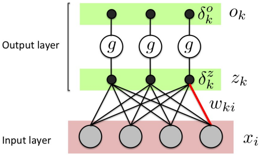
- Error gradients for single layer network:

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

Gradient Descent for Single Layer Network

- Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)} := \delta_k^o$$



Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1 + \exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1 - o_k^{(n)}) \end{aligned}$$

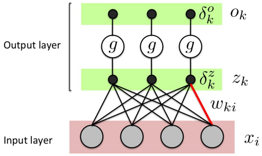
- The error gradient is then:

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

Gradient Descent for Single Layer Network

- Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)} := \delta_k^o$$



Using logistic activation functions:

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1 + \exp(-z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1 - o_k^{(n)}) \end{aligned}$$

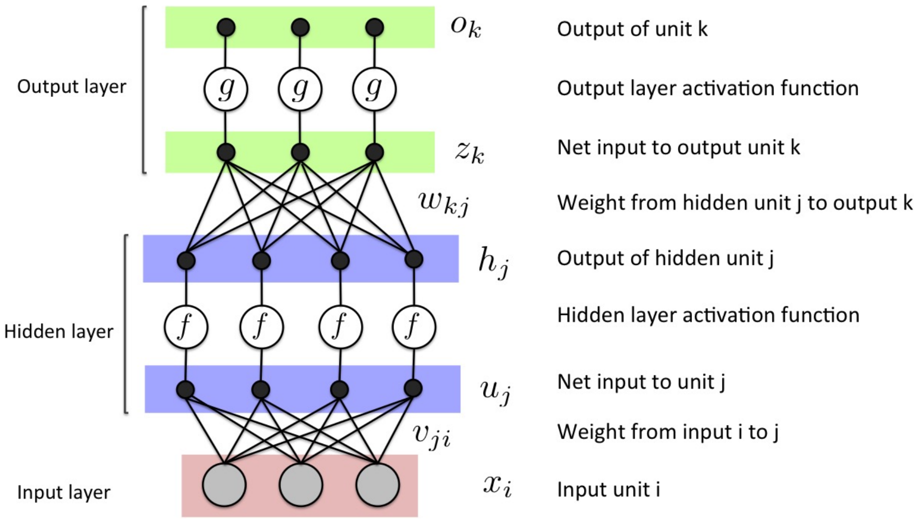
- The error gradient is then:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{ki}} = \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

- The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

Multi-layer Neural Network

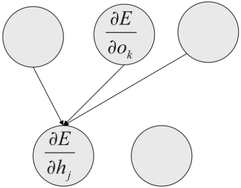


Back-propagation: Sketch on One Training Case

- Convert discrepancy between each output and its target value into an error derivative

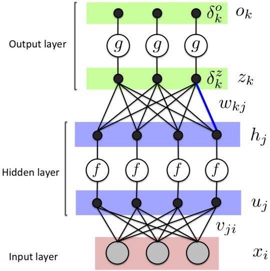
$$E = \frac{1}{2}\sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at k to each unit j according to its influence on k (depends on w_{kj})]



- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

Gradient Descent for Multi-layer Network



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

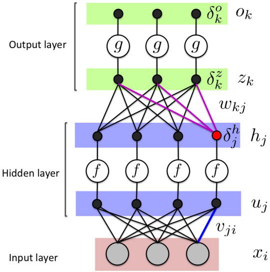
$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

$$\frac{\partial E}{\partial h_j^{(n)}} =$$

Gradient Descent for Multi-layer Network



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

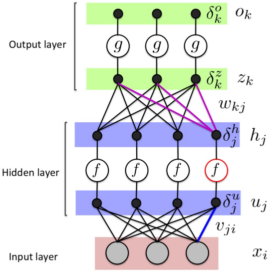
$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

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

Gradient Descent for Multi-layer Network



$\frac{\partial h_j}{\partial u_j}$

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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

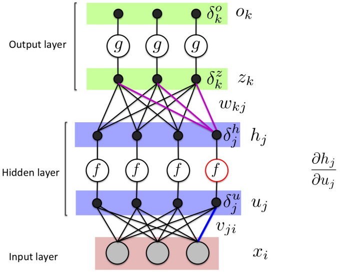
where δ_k is the error w.r.t. the net input for unit k

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

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

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} =$$

Gradient Descent for Multi-layer Network



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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

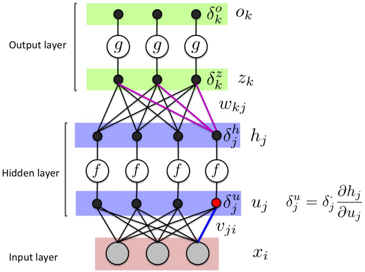
where δ_k is the error w.r.t. the net input for unit k

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

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

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} =$$

Gradient Descent for Multi-layer Network



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

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{z,(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

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

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

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{u,(n)} x_i^{(n)}$$

Choosing Activation and Loss Functions

- When using a neural network for regression, sigmoid activation and MSE as the loss function work well
- For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression)

$$E = -\sum_{n=1}^{N} t^{(n)} \log o^{(n)} + (1 - t^{(n)}) \log(1 - o^{(n)})$$

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

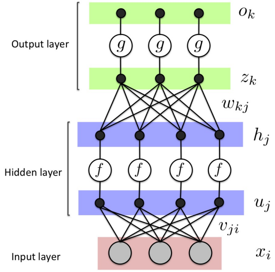
- We can then compute via the chain rule

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

$$\frac{\partial o}{\partial z} = o(1 - o)$$

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z} = (o - t)$$

Multi-class Classification



- For multi-class classification problems, use cross-entropy as loss and the softmax activation function

$$E = -\sum_n \sum_k t_k^{(n)} \log o_k^{(n)}$$

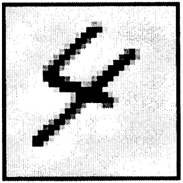
$$o_k^{(n)} = \frac{\exp(z_k^{(n)})}{\sum_j \exp(z_j^{(n)})}$$

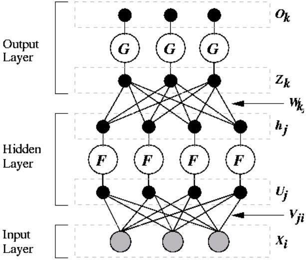
- And the derivatives become

$$\frac{\partial o_k}{\partial z_k} = o_k(1 - o_k)$$

$$\frac{\partial E}{\partial z_k} = \sum_j \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial z_k} = \boxed{(o_k - t_k) o_k (1 - o_k)}$$

Example Application





- Now trying to classify image of handwritten digit: 32x32 pixels
- 10 output units, 1 per digit
- Use the softmax function:

$$
\begin{aligned}
o_k &= \frac{\exp(z_k)}{\sum_j \exp(z_j)} \\
z_k &= w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}
\end{aligned}
$$

- What is J ?

Ways to Use Weight Derivatives

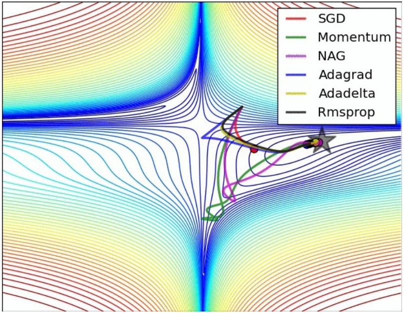
- How often to update
  - after a full sweep through the training data (batch gradient descent)

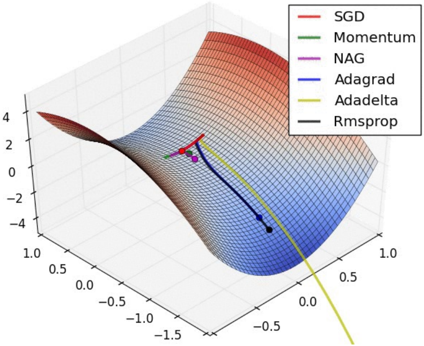
$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(\mathbf{o}^{(n)}, \mathbf{t}^{(n)}; \mathbf{w})}{\partial w_{ki}}$$

  - after each training case (stochastic gradient descent)
  - after a mini-batch of training cases
- How much to update
  - Use a fixed learning rate
  - Adapt the learning rate
  - Add momentum

$$\begin{aligned} w_{ki} &\leftarrow w_{ki} - v \\ v &\leftarrow \boxed{\gamma v} + \eta \frac{\partial E}{\partial w_{ki}} \end{aligned}$$

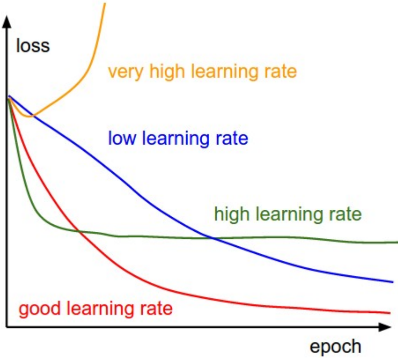
Comparing Optimization Methods





Monitor Loss During Training

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



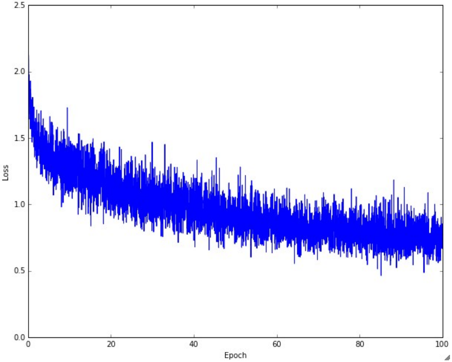
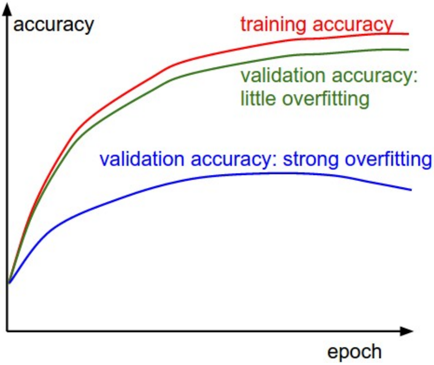


Figure : **Left:** Good vs bad parameter choices, **Right:** How a real loss might look like during training. What are the bumps caused by? How could we get a more smooth loss?

Monitor Accuracy on Train/Validation During Training

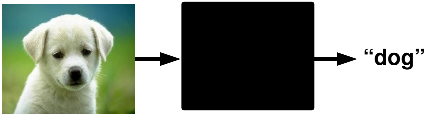
- Check how your desired performance metrics behaves during training



Why "Deep"?

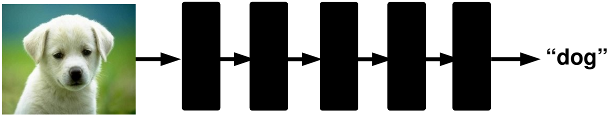
Supervised Learning: Examples

Classification



Supervised Deep Learning

Classification



Neural Networks

- Deep learning uses composite of simple functions (e.g., ReLU, sigmoid, tanh, max) to create complex non-linear functions
- Note: a composite of linear functions is linear!
- Example: 2 hidden layer NNet (now matrix and vector form!) with ReLU as nonlinearity

$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2 + b^3} \longrightarrow \mathbf{y}$$

  - $\mathbf{x}$ is the input
  - $\mathbf{y}$ is the output (what we want to predict)
  - $\mathbf{h}^i$ is the i-th hidden layer
  - W_i are the parameters of the i-th layer

Evaluating the Function

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

$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2 + b^3} \longrightarrow \mathbf{y}$$

- Do it in a compositional way,

$$\mathbf{h}^1 = \max(0, W_1^T\mathbf{x} + b^1)$$

Evaluating the Function

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

$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2 + b^3} \longrightarrow \mathbf{y}$$

- Do it in a compositional way

$$\begin{aligned}
\mathbf{h}^1 &= \max(0, W_1^T\mathbf{x} + b_1) \\
\mathbf{h}^2 &= \max(0, W_2^T\mathbf{h}^1 + b_2)
\end{aligned}$$

Evaluating the Function

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

$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T \mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T \mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T \mathbf{h}^2 + b^3} \longrightarrow \boxed{\mathbf{y}}$$

- Do it in a compositional way

$$\begin{aligned}
\mathbf{h}^1 &= \max(0, W_1^T \mathbf{x} + b_1) \\
\mathbf{h}^2 &= \max(0, W_2^T \mathbf{h}^1 + b_2) \\
\mathbf{y} &= W_3^T \mathbf{h}^2 + b_3
\end{aligned}$$

Learning

$$\mathbf{x} \rightarrow \boxed{\max(0, W_1^T \mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T \mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T \mathbf{h}^2 + b^3} \rightarrow \mathbf{y}$$

- We want to estimate the parameters, biases and hyper-parameters (e.g., number of layers, number of units) such that we do good predictions
- Collect a training set of input-output pairs $\{\mathbf{x}^{(n)}, \mathbf{t}^{(n)}\}$
- For classification: Encode the output with 1-K encoding $\mathbf{t} = [0, .., 1, .., 0]$
- Define a loss per training example and minimize the empirical risk

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_n \ell(\mathbf{w}, \mathbf{x}^{(n)}, \mathbf{t}^{(n)})$$

with N number of examples and $\mathbf{w}$ contains all parameters

Loss Function: Classification

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_n \ell(\mathbf{w}, \mathbf{x}^{(n)}, \mathbf{t}^{(n)})$$

- Probability of class k given input (softmax):

$$p(c_k = 1 | \mathbf{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)}$$

- Cross entropy is the most used loss function for classification

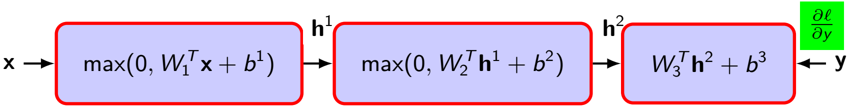
$$\ell(\mathbf{w}, \mathbf{x}^{(n)}, \mathbf{t}^{(n)}) = -\sum_k t_k^{(n)} \log p(c_k | \mathbf{x})$$

- Use gradient descent to train the network

$$\min_{\mathbf{w}} \frac{1}{N} \sum_n \ell(\mathbf{w}, \mathbf{x}^{(n)}, \mathbf{t}^{(n)})$$

Backpropagation

- Efficient computation of the gradients by applying the chain rule



$$\begin{aligned} p(c_k = 1|\mathbf{x}) &= \frac{\exp(y_k)}{\sum_{j=1}^{C}\exp(y_j)} \\ \ell(\mathbf{x}^{(n)}, \mathbf{t}^{(n)}, \mathbf{w}) &= -\sum_k t_k^{(n)} \log p(c_k|\mathbf{x}) \end{aligned}$$

- Compute the derivative of loss w.r.t. the output

$$\frac{\partial \ell}{\partial y} = p(c|\mathbf{x}) - t$$

- Note that the forward pass is necessary to compute $\frac{\partial \ell}{\partial y}$

Backpropagation

- Efficient computation of the gradients by applying the chain rule

$$\mathbf{x} \rightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xleftarrow{\frac{\partial \ell}{\partial \mathbf{h}^2}} \boxed{W_3^T\mathbf{h}^2 + b^3} \xleftarrow{\frac{\partial \ell}{\partial y}} \mathbf{y}$$

- We have computed the derivative of loss w.r.t the output

$$\frac{\partial \ell}{\partial y} = p(c|\mathbf{x}) - t$$

- Given $\frac{\partial \ell}{\partial y}$ if we can compute the Jacobian of each module

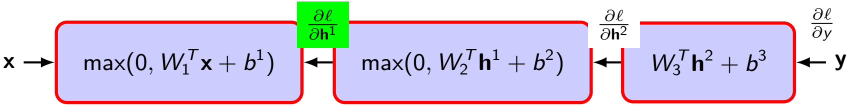
$$\frac{\partial \ell}{\partial W_3} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial W_3} = (p(c|\mathbf{x}) - t)(\mathbf{h}^2)^T$$

$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial \mathbf{h}^2} = (W_3)^T(p(c|\mathbf{x}) - t)$$

- Need to compute gradient w.r.t. inputs and parameters in each layer

Backpropagation

- Efficient computation of the gradients by applying the chain rule



$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial \mathbf{h}^2} = (W_3)^T(p(c|\mathbf{x}) - t)$$

- Given $\frac{\partial \ell}{\partial \mathbf{h}^2}$ if we can compute the Jacobian of each module

$$\frac{\partial \ell}{\partial W_2} = \frac{\partial \ell}{\partial \mathbf{h}^2}\frac{\partial \mathbf{h}^2}{\partial W_2}$$

$$\frac{\partial \ell}{\partial \mathbf{h}^1} = \frac{\partial \ell}{\partial \mathbf{h}^2}\frac{\partial \mathbf{h}^2}{\partial \mathbf{h}^1}$$

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.

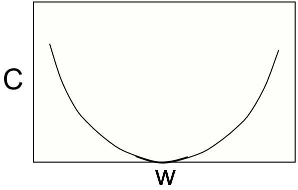
Limiting the size of the Weights

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

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

- Keeps weights small unless they have big error derivatives.

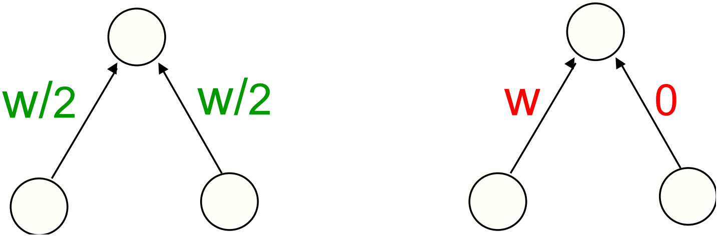
$$\frac{\partial C}{\partial w_i} = \frac{\partial \ell}{\partial w_i} + \lambda w_i$$



when $\dfrac{\partial C}{\partial w_i} = 0, \quad w_i = -\dfrac{1}{\lambda}\dfrac{\partial \ell}{\partial w_i}$

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 very 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.

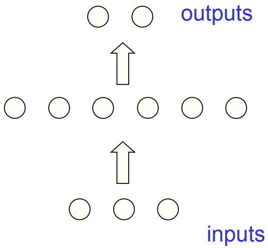
Using a Validation Set

- 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.