Neural Networks

CMPT 419/726
Mo Chen
SFU Computing Science
Jan. 29, 2020

Bishop PRML Ch. 5

Neural Networks

- Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility
- We will focus on multi-layer perceptrons
  - Mathematical properties rather than plausibility

Applications of Neural Networks

- Many success stories for neural networks, old and new
  - Credit card fraud detection
  - Hand-written digit recognition
  - Face detection
  - Autonomous driving (CMU ALVINN)
  - Object recognition
  - Speech recognition

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

Feed-forward Networks

- We have looked at generalized linear models of the form:

$$y(\boldsymbol{x}, \boldsymbol{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\boldsymbol{x})\right)$$

  for fixed non-linear basis functions $\phi(\cdot)$
  - We now extend this model by allowing adaptive basis functions, and learning their parameters
- In feed-forward networks (a.k.a. multi-layer perceptrons) we let each basis function be another non-linear function of linear combination of the inputs:

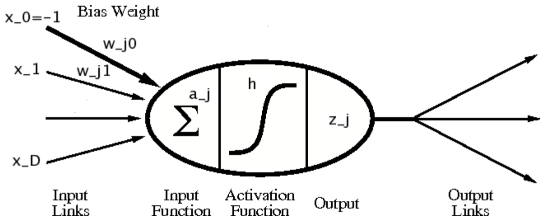
$$\phi_j(x) = f\left(\sum_{j=1}^{M} \cdots\right)$$

Feed-forward Networks

- Starting with input $\boldsymbol{x} = (x_1, \ldots, x_D)$, construct linear combinations:

$$a_j = \sum_{i=1}^{D} \left(w_{ji}^{(1)} x_i + x_{j0}^{(1)} \right)$$

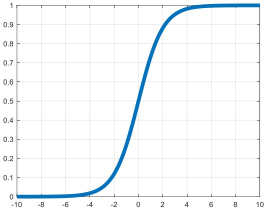
These a_j are known as activations
- Pass through an activation function $h(\cdot)$ to get output $z_j = h(a_j)$
  - Model of an individual neuron



from Russell and Norvig, AIMA2e

Activation Functions

- Can use a variety of activation functions
  - Sigmoidal (S-shaped)
    - Logistic sigmoid $1/(1+\exp(-a))$ (useful for binary classification)
    - Hyperbolic tangent $\tanh(\cdot)$
  - Radial basis function $z_j = \sum_i (x_i - w_{ji})^2$
  - Softmax
    - Useful for multi-class classification
  - Identity
    - Useful for regression
  - Threshold
  - ...
- Needs to be differentiable for gradient-based learning (later)
- Can use different activation functions in each unit

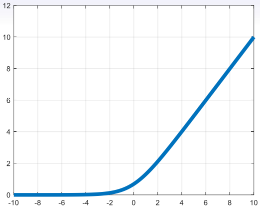


Activation Functions

Common choices of activation functions

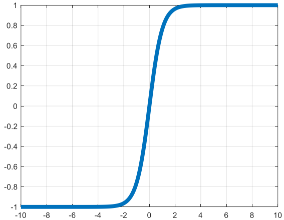
Softplus:

$$\log(1 + e^x)$$



Hyperbolic tangent:

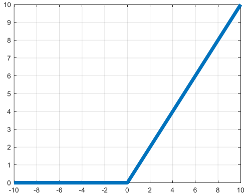
$$\tanh x$$



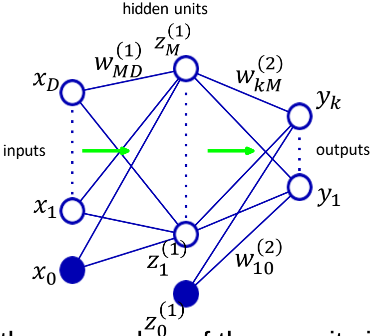
Rectified linear unit (ReLU):

$$\max(0, x)$$

Key feature: easy to differentiate



Feed-forward Networks



- Connect together a number of these units into a feed-forward network (DAG)
- Above shows a network with one layer of hidden units
- Implements function

$$y_k(x, w) = h^{(2)}\left(\sum_{j=1}^{M} w_{kj}^{(2)} h^{(1)}\left(\sum_{i=1}^{D} w_{ij}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

Network Training

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- For regression, training data are $(x_n, t_n), t_n \in \mathbb{R}$
  - Squared error naturally arises:

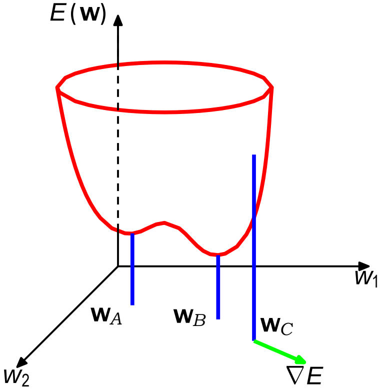
$$E(w) = \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2$$

- For binary classification, this is another discriminative model, ML:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}\{1 - y_n\}^{1-t_n}$$

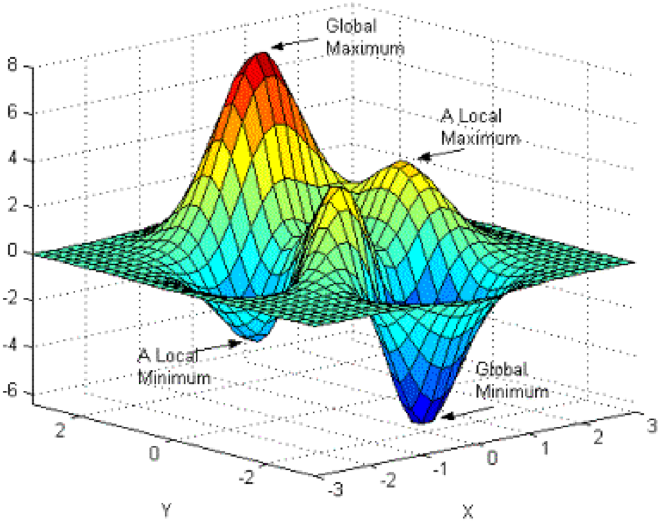
$$E(w) = -\sum_{n=1}^{N} \{t_n \ln(y_n) + (1 - t_n)\ln(1 - y_n)\}$$

Parameter Optimization



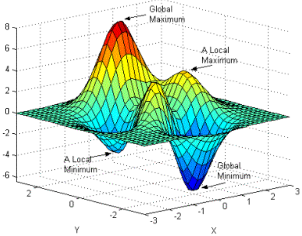
- For either of these problems, the error function $E(\boldsymbol{w})$ is nasty
  - Nasty = non-convex
  - Non-convex = has local minima

A Non-Convex function



Aside: Optimization Program

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && g_i(x) \leq 0, i = 1, \dots, n \\ & && h_j(x) = 0, j = 1, \dots, m \end{aligned}$$

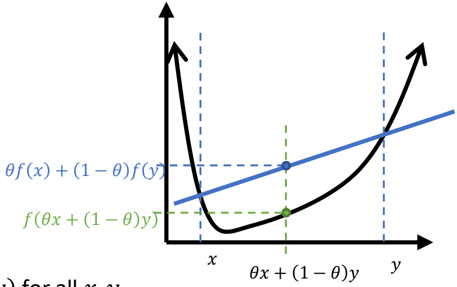


- Very difficult to solve in general
  - Trade-offs to consider: computation time, solution optimality

- Easy cases:
  - Find global optimum for **linear program**: f, g_i, h_j are linear
  - Find global optimum for **convex program**: f, g_i are convex, h_j is linear
  - Find local optimum for **nonlinear program**: f, g_i, h_j are differentiable

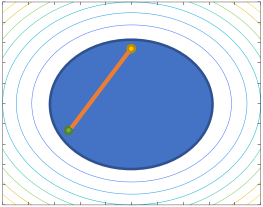
- Neural Networks: Nonlinear and unconstrained

Convex Functions

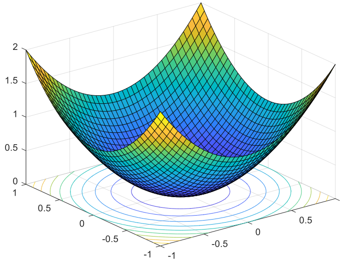


- **Convex function**
  $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y)$ for all $x, y \in \mathbb{R}^n$, for all $\theta \in [0,1]$

- Sublevel sets of convex functions, $\{x: f(x) \leq C\}$, are convex
  - **Convex shape** $\mathcal{C}$:
    $x_1, x_2 \in \mathcal{C}, \theta \in [0,1] \Rightarrow \theta x_1 + (1-\theta) x_2 \in \mathcal{C}$

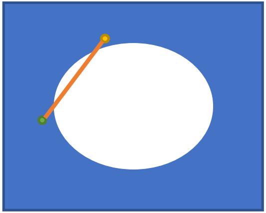


Convex Functions



- **Convex function**
  $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$ for all $x, y \in \mathbb{R}^n$, for all $\theta \in [0,1]$

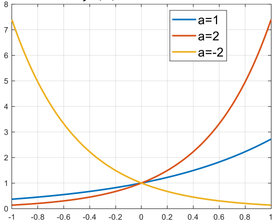
- Sublevel sets of convex functions, $\{x: f(x) \leq C\}$, are convex
  - **Convex shape** $\mathcal{C}$:
    $x_1, x_2 \in \mathcal{C}, \theta \in [0,1] \Rightarrow \theta x_1 + (1-\theta)x_2 \in \mathcal{C}$
  - *Super*level sets of convex functions are *not* convex!



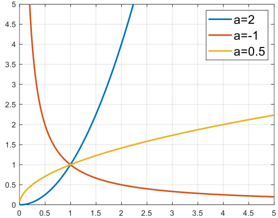
Common Convex Functions on $\mathbb{R}$

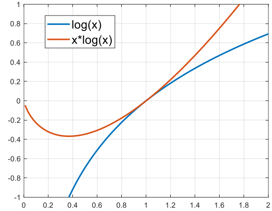
- $f(x) = e^{ax}$ is convex for all $x, a \in \mathbb{R}$
- $f(x) = x^a$ is convex on $x > 0$ if $a \geq 1$ or $a \leq 0$; concave if $0 < a < 1$
- $f(x) = \log x$ is concave
- $f(x) = x \log x$ is convex for $x > 0$ (or $x \geq 0$ if defined to be 0 when $x = 0$)

$f(x) = e^{ax}$



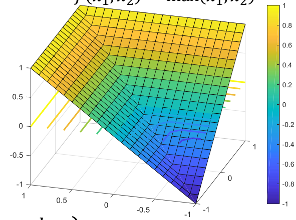
$f(x) = x^a$





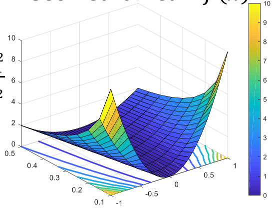
Common Convex Functions on $\mathbb{R}^n$

$f(x_1, x_2) = \max(x_1, x_2)$

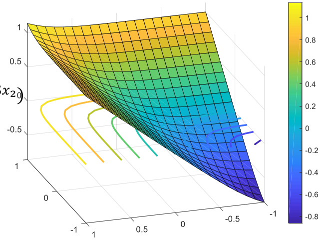


- $f(x) = Ax + b$ is convex for any A, b
- Every norm on $\mathbb{R}^n$ is convex
- $f(x) = \max(x_1, x_2, \ldots, x_n)$ is convex
- $f(x) = \frac{x_1^2}{x_2}$ (for $x_2 > 0$)
- Log-sum-exp softmax: $f(x) = \frac{1}{k}\log\left(e^{kx_1} + e^{kx_2} + \cdots + e^{kx_n}\right)$
- Geometric mean: $f(x) = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}},\ x_i > 0$

$f(x) = \frac{x_1^2}{x_2}$



$f(x) = \frac{1}{5}\log(e^{5x_1} + e^{5x_2})$



Descent Methods

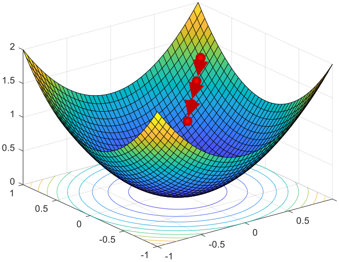
- The typical strategy for optimization problems of this sort is a descent method:

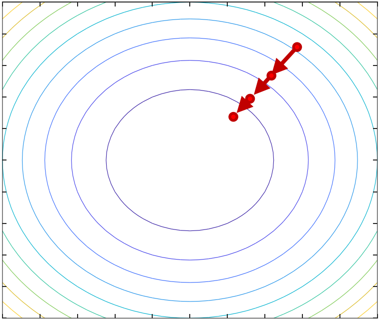
$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \Delta\boldsymbol{w}^{(\tau)}$$

- As we've seen before, these come in many flavours
  - Gradient descent $\nabla E\left(w^{(\tau)}\right)$
  - Stochastic gradient descent $\nabla E_n\left(\boldsymbol{w}^{(\tau)}\right)$
  - Newton-Raphson (second order)
- All of these can be used here, stochastic gradient descent is particularly effective
  - Redundancy in training data, escaping local minima

Numerical Solution: Gradient Methods

- Start from x^0 and construct a sequence x^k such that $x^k \rightarrow x^*$
  - Calculate x^{k+1} from x^k by "going down the gradient"
  - Unconstrained case: $x^{k+1} = x^k - \alpha^k \nabla f(x),\ \alpha^k > 0$





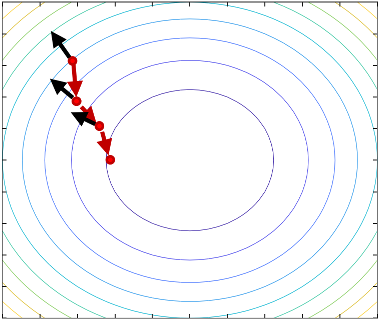
Numerical Solution: Gradient Methods

- Start from x^0 and construct a sequence x^k such that $x^k \rightarrow x^*$
  - Calculate x^{k+1} from x^k by "going down the gradient"
  - Unconstrained case: $x^{k+1} = x^k - \alpha^k \nabla f(x),\ \alpha^k > 0$

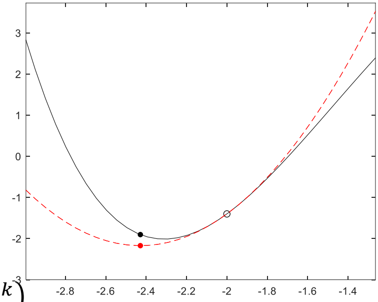
- More generally, $x^{k+1} = x^k + \alpha^k d^k$ for some d such that

$$\nabla f\left(x^k\right) \cdot d^k < 0$$

- Tuning parameters: descent direction d^k, and step size α^k



Descent Direction



- Steepest descent: $d^k = -\nabla f\left(x^k\right)$
  - $x^{k+1} = x^k - \alpha^k \nabla f(x)$
  - Simple but sometimes leads to slow convergence
- Newton's method: $d^k = \left(\nabla^2 f\left(x^k\right)\right)^{-1} \nabla f\left(x^k\right)$
  - Minimize the quadratic approximation:

$$f^k(x) = f\left(x^k\right) + \nabla f\left(x^k\right)^{\mathsf{T}}\left(x - x^k\right) + \frac{1}{2}\left(x - x^k\right)^{\mathsf{T}} \mathrm{H}f\left(x^k\right)\left(x - x^k\right)$$

  - Set gradient to zero to obtain next iterate

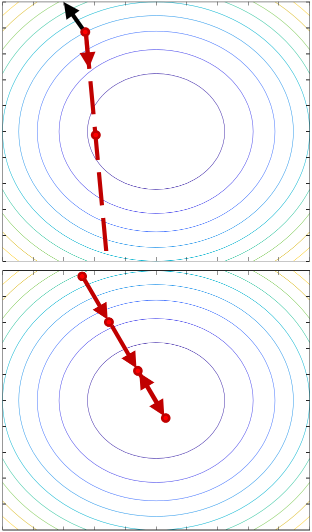
$$\nabla f^k(x) = \nabla f\left(x^k\right) + \mathrm{H}f\left(x^k\right)\left(x - x^k\right) = 0$$

$$\Rightarrow x^{k+1} = x^k - \left(\mathrm{H}f\left(x^k\right)\right)^{-1} \nabla f\left(x^k\right)$$

  - Fast convergence, but matrix inverse required
  - Alternatively, use an algorithm to minimize a quadratic function

Step Size

- Recall $x^{k+1} = x^k + \alpha^k d^k$, with $\nabla f\left(x^k\right)^\top d^k < 0$
- Line search: choose $\alpha^k = \min_{\alpha \geq 0} f\left(x^k + \alpha^k d^k\right)$
  - Requires minimization
- Constant step size: $\alpha^k = \alpha$
  - May not converge
- Diminishing step size: $\alpha^k \to 0$
  - Still need to explore all regions $\sum \alpha^k = \infty$
  - For example: $\alpha^k = \frac{\alpha^0}{k}$



Numerical Solution: Second Order Methods

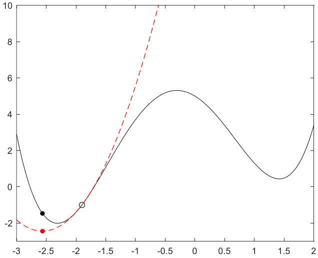
$$\text{minimize} \quad f(x) \quad \longrightarrow \quad \underset{d_x}{\text{minimize}} \quad \left(\boldsymbol{r^k}\right)^\top d_x + \frac{1}{2} d_x^\top \boldsymbol{B_k} d_x$$

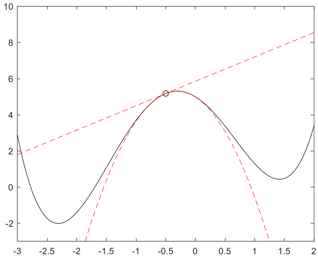
where $d_x \coloneqq x - x^k$,

- Quadratize $f(x)$:

$$r^k = \nabla f(x_k)$$
$$B_k = \mathrm{H} f(x_k)$$

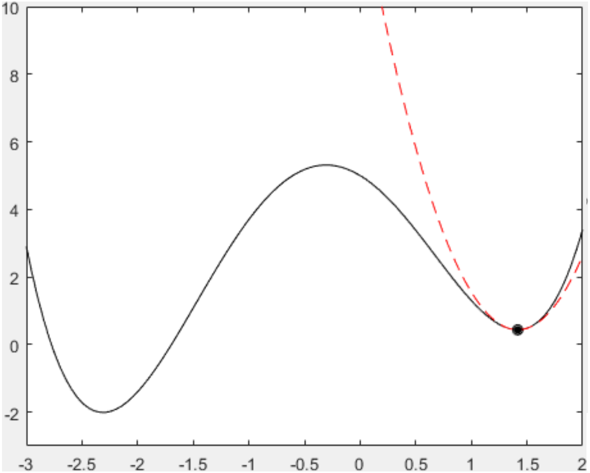
  - Convexify if needed, eg. by removing negative eigenvalues





Example

$$\text{minimize } 0.5x^4 + 0.8x^3 - 3x^2 - 2x + 5$$
$$\text{subject to } -3 \leq x \leq 2$$



Computing Gradients

- The function $y(\boldsymbol{x}_n, \boldsymbol{w})$ implemented by a network is complicated
  - It isn't obvious how to compute error function derivatives with respect to weights
- Numerical method for calculating error derivatives, use finite differences:

$$\frac{\partial E_n}{\partial w_{ji}} \approx \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon}$$

- How much computation would this take with W weights in the network?
  - $O(|W|)$ per partial derivative (evaluation of E_n)
  - $O(|W|^2)$ total per gradient descent step (there are $|W|$ partial derivatives)

Outline

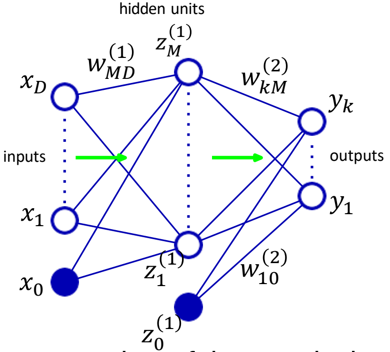
Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

Feed-forward Networks



- Connect together a number of these units into a feed-forward network (DAG)
- Above shows a network with one layer of hidden units
- Implements function:

$$y_{(n),k}(x_n, w) = h^{(2)}\left(\underbrace{\sum_{j=1}^{M} w_{kj}^{(2)} h^{(1)}\left(\overbrace{\sum_{i=1}^{D} w_{ji}^{(1)} x_{(n),i} + w_{j0}^{(1)}}^{a_{(n),j}}\right)}_{z_{(n),j}} + w_{k0}^{(2)}\right)$$

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}^{(m)}}$
    - $O(W)$ to compute derivatives wrt all weights
- First, feed training example x_n forward through the network, storing all activations a_j
- Calculating derivatives for weights connected to output nodes is easy
    - e.g. For linear output nodes $y_k = \sum_i w_{ki}^{(L)} z_{(n),i}^{(L-1)}$:

$$\frac{\partial E_n}{\partial w_{ki}^{(L)}} = \frac{\partial}{\partial w_{ki}^{(L)}} \frac{1}{2} \left(y_{(n),k} - t_{(n),k}\right)^2 = \left(y_{(n),k} - t_{(n),k}\right) z_{(n),i}^{(L-1)}$$

- For hidden layers, propagate error backwards from the output nodes

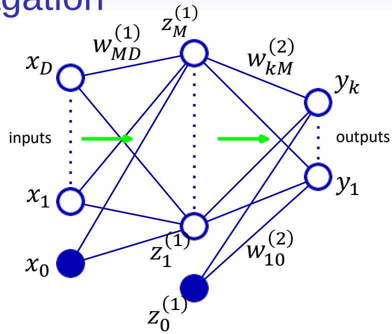
Opportunity to participate in robotics research

The SFU Rosie and MARS Labs are running an experiment to better understand human navigational intent – that is, predicting where a human may move to in the next several seconds.

*Experiment takes 30 Min.
*Each participant will receive a $10 Starbucks gift card.
*Spaces are limited to the first 40 students.

For more information:
http://tiny.cc/kdbjjz

Error Backpropagation



$y_{(n),k}, E_n$:

- n: data point
- k: component

$w_{ji}^{(m)}$:

- m: layer
- j: index matching output
- i: index matching input

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k}\left(y_{(n),k} - t_{(n),k}\right)^2, \qquad y_{(n),k} = \sum_i w_{ki}^{(L)} z_{(n),i}^{(L-1)}$$

$$E_n(w) = \frac{1}{2}\sum_{k}\left(y_{(n),k} - t_{(n),k}\right)^2$$

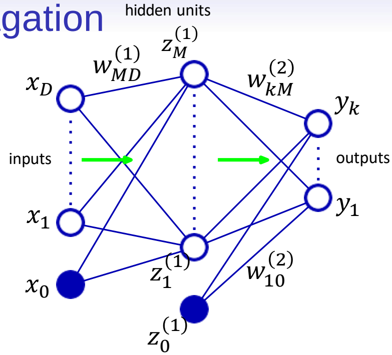
$$\frac{\partial E_n}{\partial w_{ki}^{(L)}} = \frac{\partial}{\partial w_{ki}^{(L)}}\frac{1}{2}\sum_{k'}\left(y_{(n),k'} - t_{(n),k'}\right)^2 = \left(y_{(n),k} - t_{(n),k}\right) z_{(n),i}^{(L-1)} \qquad (*)$$

Chain Rule for Partial Derivatives

- A "reminder"
- For $f(x, y)$, with f differentiable wrt x and y, and x and y differentiable wrt u:

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$$

Error Backpropagation



- We can write

$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \frac{\partial}{\partial w_{ji}^{(m)}} E_n\left(a_{(n),1}^{(m)}, a_{(n),2}^{(m)}, \ldots, a_{(n),D}^{(m)}\right)$$

- Using the chain rule:

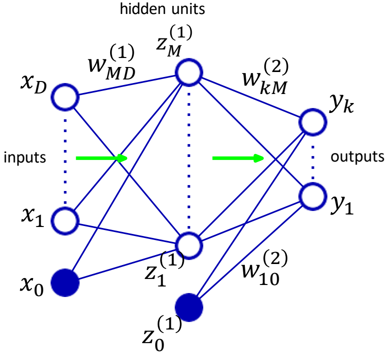
$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \frac{\partial E_n}{\partial a_{(n),j}^{(m)}} \frac{\partial a_{(n),j}^{(m)}}{\partial w_{ji}^{(m)}} + \sum_{k \neq j} \frac{\partial E_n}{\partial a_{(n),k}^{(m)}} \frac{\partial a_{(n),k}^{(m)}}{\partial w_{ji}^{(m)}}$$

where $\sum_k(\cdots)$ runs over all other nodes k in the same layer (m)

- Since $a_{(n),k}^{(m)}$ does not depend on $w_{ji}^{(m)}$, all terms in the summation go to 0:

$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \frac{\partial E_n}{\partial a_{(n),j}^{(m)}} \frac{\partial a_{(n),j}^{(m)}}{\partial w_{ji}^{(m)}}$$

Error Backpropagation cont.



- Introduce error $\delta_{(n),j}^{(m)} := \frac{\partial E_n}{\partial a_{(n),j}^{(m)}}$

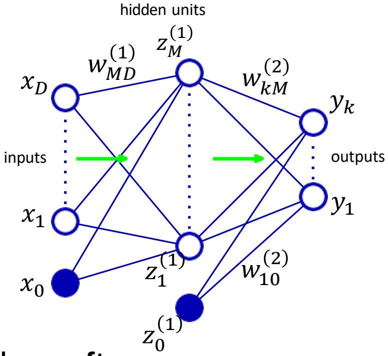
$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \delta_{(n),j}^{(m)} \frac{\partial a_{(n),j}^{(m)}}{\partial w_{ji}^{(m)}}$$

- Other factor is

$$\frac{\partial a_{(n),j}^{(m)}}{\partial w_{ji}^{(m)}} = \frac{\partial}{\partial w_{ji}^{(m)}} \sum_k w_{jk}^{(m)} z_k^{(m-1)} = z_i^{(m-1)}$$

$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \delta_{(n),j}^{(m)} z_i^{(m-1)}$$

Error Backpropagation cont.



- Error $\delta_{(n),j}^{(m)}$ can also be computed using chain rule:

$$\delta_{(n),j}^{(m)} := \frac{\partial E_n}{\partial a_{(n),j}^{(m)}} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_{(n),k}^{(m+1)}}}_{\delta_k^{(m+1)}} \frac{\partial a_{(n),k}^{(m+1)}}{\partial a_{(n),j}^{(m)}}$$

where $\sum_k(\cdots)$ runs over all nodes k in the layer **after**.

$$a_{(n),k}^{(m+1)} = \sum_i w_{ki}^{(m+1)} z_{(n),i}^{(m)} = \sum_i w_{ki}^{(m+1)} h^{(m)}\left(a_{(n),i}^{(m)}\right)$$

$$\frac{\partial a_{(n),k}^{(m+1)}}{\partial a_{(n),j}^{(m)}} = w_{kj}^{(m+1)} \left(h^{(m)}\right)' \left(a_{(n),j}^{(m)}\right)$$

$$\delta_{(n),j}^{(m)} = \sum_k \delta_{(n),k}^{(m+1)} w_{kj}^{(m+1)} \left(h^{(m)}\right)' \left(a_{(n),j}^{(m)}\right) = \left(h^{(m)}\right)' \left(a_{(n),j}^{(m)}\right) \sum_k \delta_{(n),k}^{(m+1)} w_{kj}^{(m+1)}$$

Error Backpropagation cont.

- Error $\delta_{(n),j}^{(m)}$ can also be computed using chain rule:

$$\delta_{(n),j}^{(m)} := \frac{\partial E_n}{\partial a_{(n),j}^{(m)}} = \sum_k \underbrace{\frac{\partial E_n}{\partial a_{(n),k}^{(m+1)}}}_{\delta_k} \frac{\partial a_{(n),k}^{(m+1)}}{\partial a_{(n),j}^{(m)}}$$

where $\sum_k(\cdots)$ runs over all nodes k in the layer **after**.

- Eventually:

$$\delta_{(n),j}^{(m)} = \left(h^{(m)}\right)' \left(a_{(n),j}^{(m)}\right) \sum_k \delta_{(n),k}^{(m+1)} \, w_{kj}^{(m+1)}$$

  - A weighted sum of the later error “caused” by this weight

Error Backpropagation cont.

- Eventually:

$$\delta_{(n),j}^{(m)} = \left(h^{(m)}\right)' \left(a_{(n),j}^{(m)}\right) \sum_k \delta_{(n),k}^{(m+1)} \, w_{jk}^{(m+1)}$$

where $\sum_k (\cdots)$ runs over all nodes k in the layer **after**.

- Above recursion relation needs last set of errors: $\delta_j^{(L)}$

$$\frac{\partial E_n}{\partial w_{ji}^{(m)}} = \delta_{(n),j}^{(m)} z_i^{(m-1)}$$ (by definition)

$$\frac{\partial E_n}{\partial w_{ji}^{(L)}} = \delta_{(n),j}^{(L)} z_{(n),i}^{(L-1)} = \left(y_{(n),j} - t_{(n),j}\right) z_{(n),i}^{(L-1)}$$ (from before $(*)$)

$$\delta_{(n),j}^{(L)} = y_{(n),j} - t_{(n),j}$$ (by comparison)

Summary

$O(W)$

Output Definition / forward propagation

$$y_{(n),k}(x_n, w) = h^{(m+1)}\left(\sum_{j=1}^{M} w_{jk}^{(m+1)} h^{(m)}\left(\sum_{i=1}^{D} w_{ij}^{(m)} z_{(n),i}^{(m-1)} + w_{0j}^{(m)}\right) + w_{k0}^{(m+1)}\right)$$

- Save $\boldsymbol{z}, \boldsymbol{a}$

Gradient computation / backpropagation

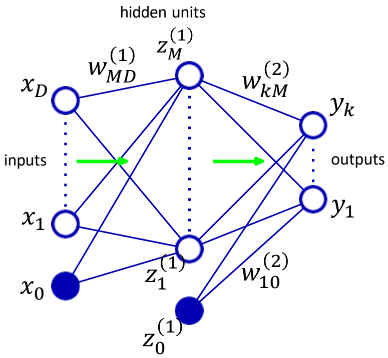
- Last layer: $\dfrac{\partial E_n}{\partial w_{ik}^{(L)}} = \left(y_{(n),k} - t_{(n),k}\right) z_{(n),i}^{(L-1)}$
- Previous layers: Define $\delta_{(n),j}^{(m)} := \dfrac{\partial E_n}{\partial a_{(n),j}^{(m)}}$

  Starting from last layer,

$$\delta_{(n),j}^{(L)} = y_{(n),j} - t_{(n),j}$$

Recursion: $\dfrac{\partial E_n}{\partial w_{ij}^{(m)}} = \delta_{(n),j}^{(m)} z_{(n),i}^{(m-1)}$,

where $\delta_{(n),j}^{(m)} = \left(h^{(m)}\right)' \left(a_{(n),j}^{(m)}\right) \sum_k \delta_k^{(m+1)} w_{jk}^{(m+1)}$



Summary

Output Definition / forward propagation

$O(W)$

$$y_{(n),k}(x_n, w) = h^{(m+1)}\left(\sum_{j=1}^{M} w_{jk}^{(m+1)} h^{(m)}\left(\sum_{i=1}^{D} w_{ij}^{(m)} z_{(n),i}^{(m-1)} + w_{0j}^{(m)}\right) + w_{k0}^{(m+1)}\right)$$

- Save $\boldsymbol{z}, \boldsymbol{a}$

Gradient computation / backpropagation

Goes through one layer of weights

- Last layer: $\frac{\partial E_n}{\partial w_{ik}^{(L)}} = \left(y_{(n),k} - t_{(n),k}\right) z_{(n),i}^{(L-1)}$
- Previous layers: Define $\delta_{(n),j}^{(m)} := \frac{\partial E_n}{\partial a_{(n),j}^{(m)}}$

$O(W)$

Starting from last layer,

$$\delta_{(n),j}^{(L)} = y_{(n),j} - t_{(n),j}$$

Recursion: $\frac{\partial E_n}{\partial w_{ij}^{(m)}} = \delta_{(n),j}^{(m)} z_{(n),i}^{(m-1)}$,

Goes through one layer of weights

where $\delta_{(n),j}^{(m)} = \left(h^{(m)}\right)'\left(a_{(n),j}^{(m)}\right) \sum_k \delta_k^{(m+1)} w_{jk}^{(m+1)}$

Descent Methods

- Error function:

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k}\left(y_{(n),k} - t_{(n),k}\right)^2, \quad E_n(w) = \frac{1}{2}\sum_{k}\left(y_{(n),k} - t_{(n),k}\right)^2$$

  - $y(x, w)$ is a neural network, very complex
  - Cannot solve $\arg\min_w E(w)$ explicitly (like in linear regression)
- Gradient Descent:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta^{(\tau)} \nabla E\left(w^{(\tau)}\right)$$

- Stochastic Gradient Descent:
  - n chosen randomly

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta^{(\tau)} \nabla E_n\left(w^{(\tau)}\right)$$

  - A batch $\mathcal{N}$ chosen randomly

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta^{(\tau)} \sum_{n\in\mathcal{N}} \nabla E_n\left(w^{(\tau)}\right)$$

Tensorflow Playground

- https://playground.tensorflow.org

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Deep Learning

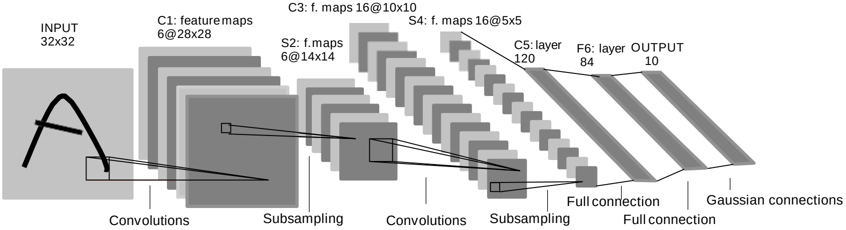
Deep Learning

- Collection of important techniques to improve performance:
  - Multi-layer networks
  - Convolutional networks, parameter tying
  - Hinge activation functions (ReLU) for steeper gradients
  - Momentum
  - Drop-out regularization
  - Sparsity
  - Auto-encoders for unsupervised feature learning
  - ...
- Scalability is key, can use lots of data since stochastic gradient descent is memory-efficient, can be parallelized

Hand-written Digit Recognition

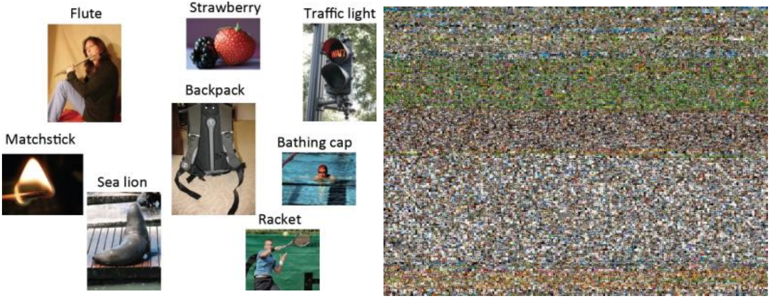
- MNIST - standard dataset for hand-written digit recognition
  - 60000 training, 10000 test images

LeNet-5, circa 1998



- LeNet developed by Yann LeCun et al.
  - Convolutional neural network
    - Local receptive fields (5x5 connectivity)
    - Subsampling (2x2)
    - Shared weights (reuse same 5x5 “filter”)
    - Breaking symmetry

ImageNet



- ImageNet - standard dataset for object recognition in images (Russakovsky et al.)
  - 1000 image categories, ≈1.2 million training images (ILSVRC 2013)

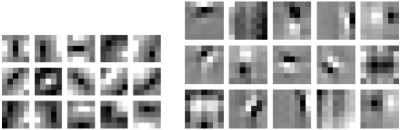
GoogLeNet, circa 2014

- GoogLeNet developed by Szegedy et al., CVPR 2015
- Modern deep network
- ImageNet top-5 error rate of 6.67% (later versions even better)
- Comparable to human performance (especially for fine-grained categories)

ResNet, circa 2015

- ResNet developed by He et al., ICCV 2015
- 152 layers
- ImageNet top-5 error rate of 3.57%
- Better than human performance (especially for fine-grained categories)

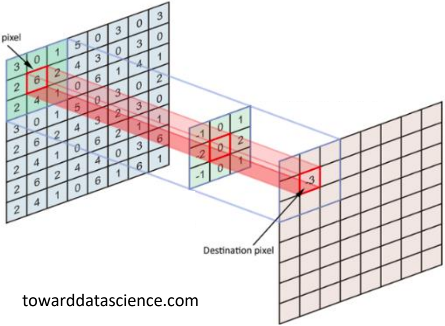
Key Component 1: Convolutional Filters



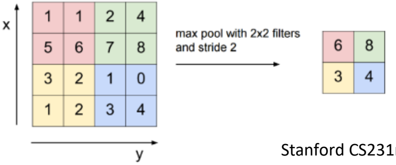
- Share parameters across network
- Reduce total number of parameters
- Provide translation invariance, useful for visual recognition

Common Operations

- Fully connected (dot product)
- Convolution
  - Translationally invariant
  - Controls overfitting
- Pooling (fixed function)
  - Down-sampling
  - Controls overfitting
- Nonlinearity layer (fixed function)
  - Activation functions, e.g. ReLU

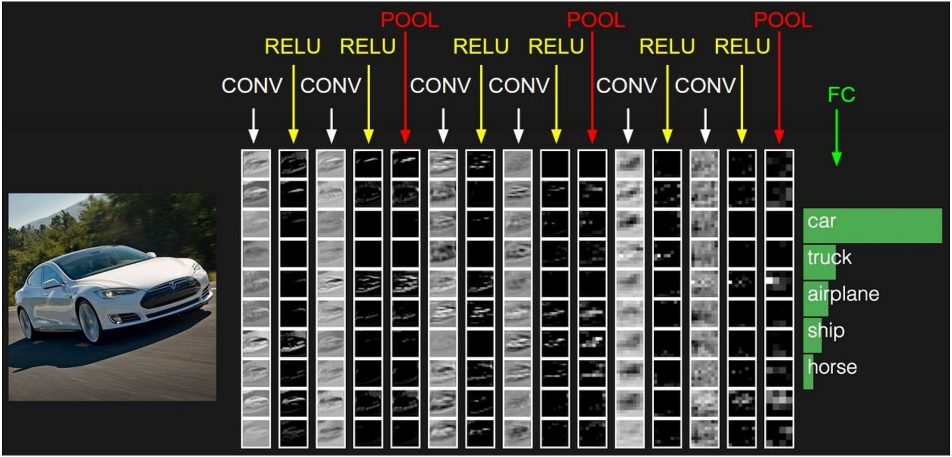


towarddatascience.com

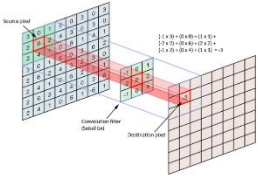


Stanford CS231n

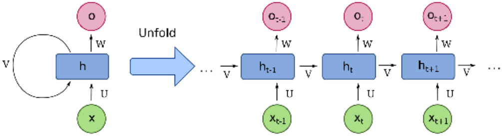
Example: Small VGG Net From Stanford CS231n



Neural Network Architectures



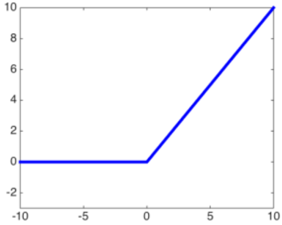
- Convolutional neural network (CNN)
  - Has translational invariance properties from convolution
  - Common used for computer vision
- Recurrent neural network RNN
  - Has feedback loops to capture temporal or sequential information
  - Useful for handwriting recognition, speech recognition, reinforcement learning
  - Long short-term memory (LSTM): special type of RNN with advantages in numerical properties
- Others
  - General feedforward networks, variational autoencoders (VAEs), conditional VAEs, generative adversarial networks



Training Neural Networks

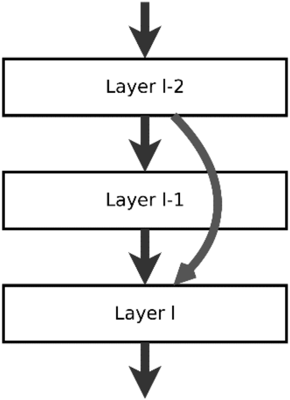
- Data preprocessing
  - Removing bad data
  - Transform input data (e.g. rotating, stretching, adding noise)
- Training process (optimization algorithm)
  - Choice of loss function (eg. L1 and L2 regularization)
  - Dropout: randomly set neurons to zero in each training iteration
  - **Learning rate** (step size) and other hyperparameter tuning
- Software packages: efficient gradient computation
  - Caffe, Torch, Theano, TensorFlow

Key Component 2: Rectified Linear Units (ReLUs)



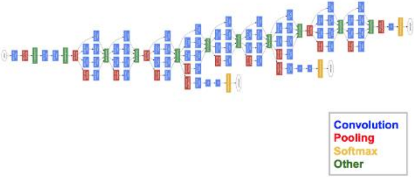
- Vanishing gradient problem
  - If derivatives very small, no/little progress via stochastic gradient descent
  - Occurs with sigmoid function when activation is large in absolute value
- ReLU: $h(a_j) = \max(0, a_j)$
- Non-saturating, linear gradients (as long as non-negative activation on some training data)
- Sparsity inducing

Key Component 3: Many, Many Layers



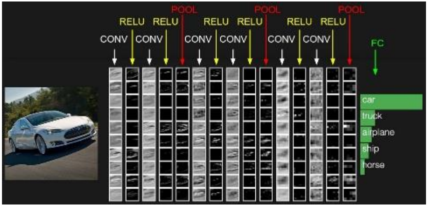
- **ResNet**: $\approx$152 layers ("shortcut connections")
- GoogLeNet: $\approx$27 layers ("Inception" modules)
- VGG Net: 16-19 layers (Simonyan and Zisserman, 2014)
- AlexNet: 8 layers (Krizhevsky et al., 2012)

Key Component 3: Many, Many Layers



- ResNet: ≈152 layers ("shortcut connections")
- **GoogLeNet**: ≈27 layers ("Inception" modules)
- VGG Net: 16-19 layers (Simonyan and Zisserman, 2014)
- AlexNet: 8 layers (Krizhevsky et al., 2012)

Key Component 3: Many, Many Layers



- ResNet: $\approx$152 layers ("shortcut connections")
- GoogLeNet: $\approx$27 layers ("Inception" modules)
- **VGG Net**: 16-19 layers (Simonyan and Zisserman, 2014)
- AlexNet: 8 layers (Krizhevsky et al., 2012)

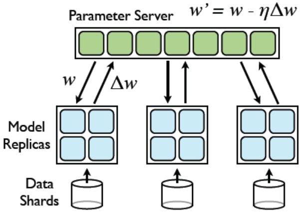
Key Component 4: Momentum

- Trick to escape plateaus / local minima
- Take exponential average of previous gradients

$$\overline{\frac{\partial E_n}{\partial w_{ji}}}^{\tau} = \overline{\frac{\partial E_n}{\partial w_{ji}}}^{\tau} + \alpha \overline{\frac{\partial E_n}{\partial w_{ji}}}^{\tau-1}$$

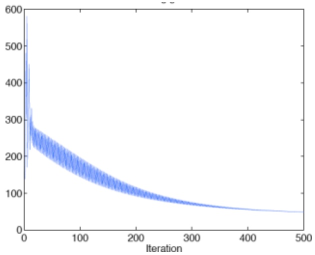
- Maintains progress in previous direction

Key Component 5: Asynchronous Stochastic Gradient Descent



- Big models won't fit in memory
- Want to use compute clusters (e.g. 1000s of machines) to run stochastic gradient descent
- How to parallelize computation?
- Ignore synchronization across machines
  - Just let each machine compute its own gradients and pass to a server storing current parameters
  - Ignore the fact that these updates are inconsistent
  - Seems to just work (e.g. Dean et al. NIPS 2012)

Key Component 6: Learning Rate Schedule



- How to set learning rate η?:

$$\boldsymbol{w}^{\tau} = \boldsymbol{w}^{\tau-1} + \eta \nabla \boldsymbol{w}$$

- Option 1: Run until validation error plateaus. Drop learning rate by x%
- Option 2: Adagrad, adaptive gradient. Per-element learning rate set based on local geometry (Duchi et al. 2010)

Key Component 7: Data Augmentation

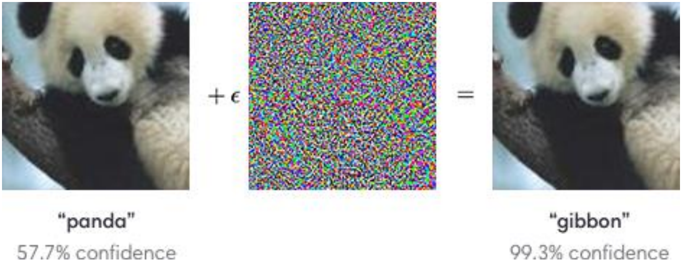
- Augment data with additional synthetic variants (10x amount of data)
- Or just use synthetic data, e.g. Sintel animated movie (Butler et al. 2012)

Key Component 8: Data and Compute

- Get lots of data (e.g. ImageNet)
- Get lots of compute (e.g. CPU cluster, GPUs)
- Cross-validate like crazy, train models for 2-3 weeks on a GPU
- Researcher gradient descent (RGD) or Graduate student descent (GSD): get 100s of researchers to each do this, trying different network structures

Challenges

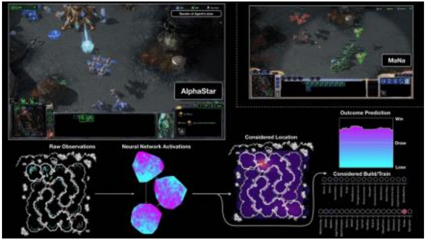
Interpretability:



Challenges

Data efficiency:

- ImageNet: 14 million images, 20000 categories
- AlphaStar: 200 years of gameplay



Challenges

- Problem formulation (what are you trying to predict?)
- Choice of model and optimization algorithm
- Data collection, post-processing
- Feature selection
- ...

More information

- https://sites.google.com/site/deeplearningsummerschool
- http://tutorial.caffe.berkeleyvision.org/
- ufldl.stanford.edu/eccv10-tutorial
- http://www.image-net.org/challenges/LSVRC/2012/supervision.pdf
- Courses: Deep Learning, Natural Language Processing, Computer Vision
- Project ideas
  - Long short-term memory (LSTM) models for temporal data
  - Learning embeddings (word2vec, FaceNet)
  - Structured output (multiple outputs from a network)
  - Zero-shot learning (learning to recognize new concepts without training data)
  - Transfer learning (use data from one domain/task, adapt to another)

Conclusion

- Readings: Ch. 5.1, 5.2, 5.3
- Feed-forward networks can be used for regression or classification
  - Similar to linear models, except with adaptive non-linear basis functions
  - These allow us to do more than e.g. linear decision boundaries
- Different error functions
- Learning is more difficult, error function not convex
  - Use stochastic gradient descent, obtain (good?) local minimum
- Backpropagation for efficient gradient computation