Introduction to Artificial Neural Networks

STAT5241 Section 2

Statistical Machine Learning

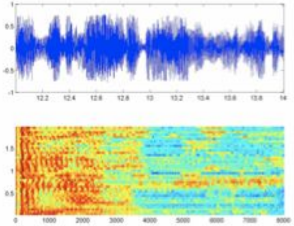
Xiaofei Shi

Columbia University

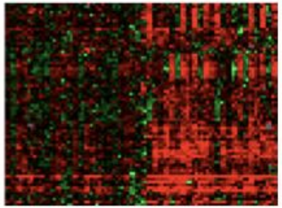
Images & Video

Text & Language

Speech & Audio

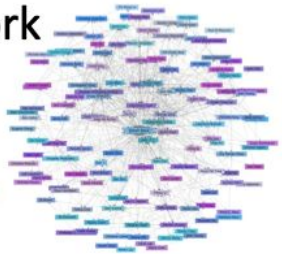


Gene Expression

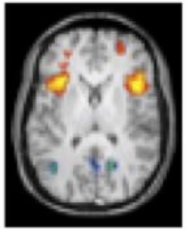


Product
Recommendation

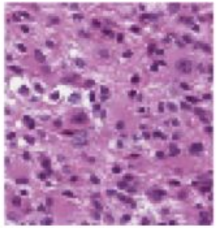
Relational Data/
Social Network



fMRI



Tumor region



Columbia University

Overview

- A neural network is a supervised learning method. It can be applied to both regression and classification problems.
- The main idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features.
- The nonlinear transformation contributes to the model flexibility.
- Today, we will focus on the most widely used ``vanilla'' neural net, also called the single hidden layer feedforward neural networks.

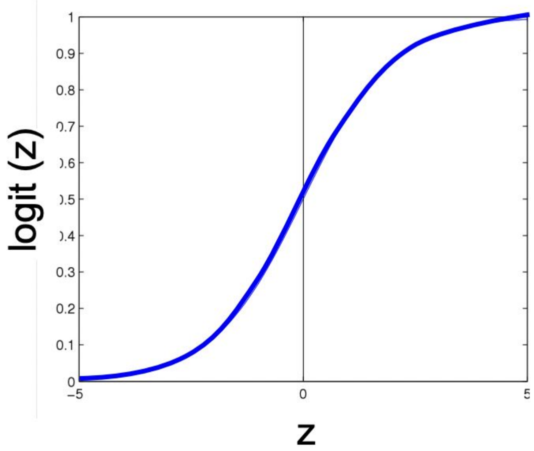
COLUMBIA UNIVERSITY

Recall Logistic Regression

$$P(Y=1|X) = \frac{1}{1+\exp(-(w_0 + \sum_i w_i X_i))}$$

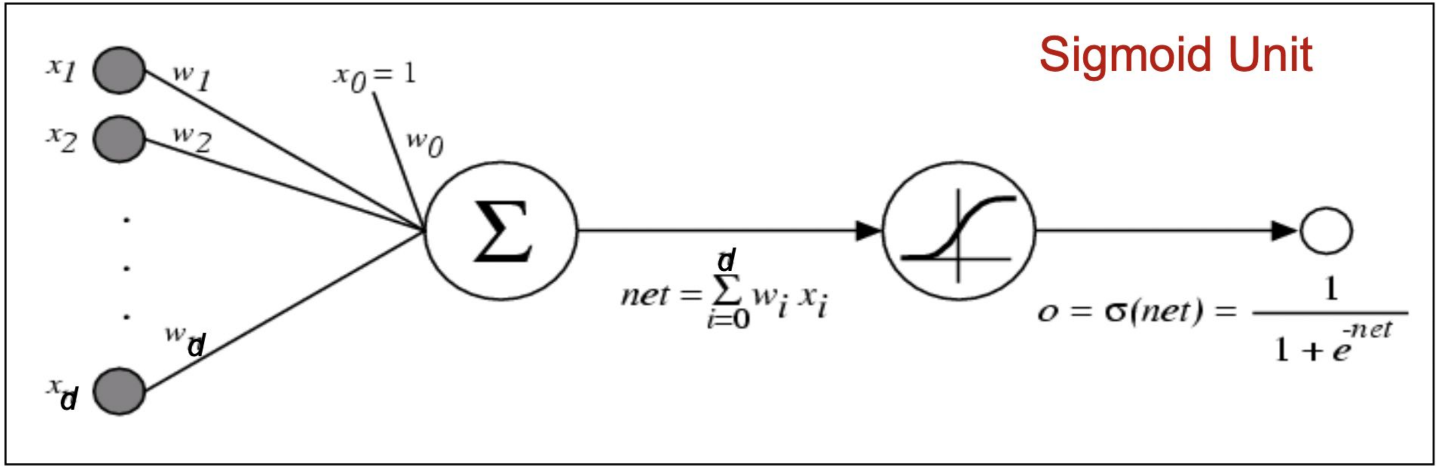
Logistic function applied to a linear function of the data

Logistic function (or Sigmoid): $\frac{1}{1+exp(-z)}$



Logistic function as a graph

$$\text{Output, } o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$

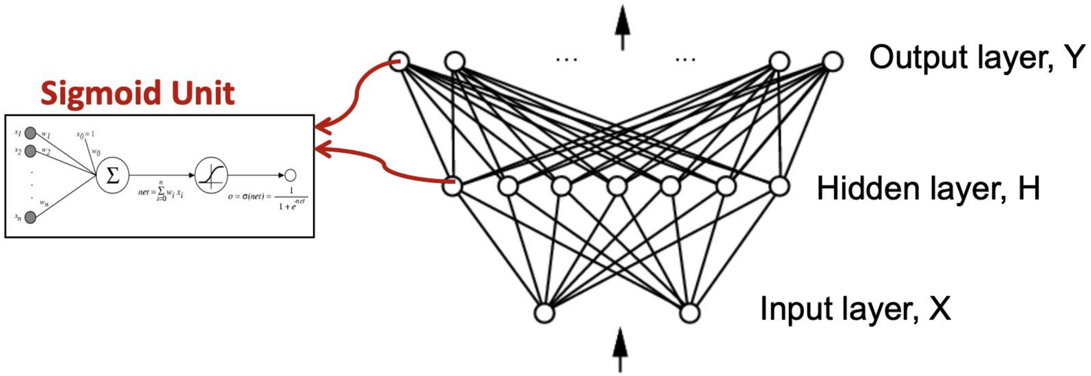


COLUMBIA UNIVERSITY

Use neural networks to learn f: X -> Y

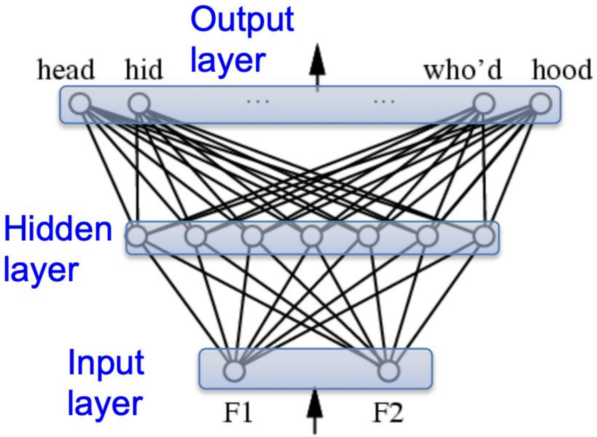
- f can be a **non-linear** function
- X (vector of) continuous and/or discrete variables
- Y (**vector** of) continuous and/or discrete variables

- Neural networks - Represent f by ***network*** of logistic/sigmoid units:

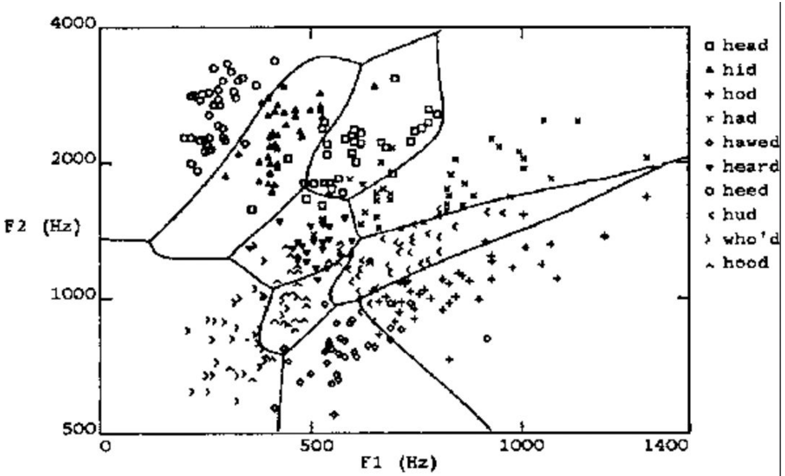


COLUMBIA UNIVERSITY

Use neural networks to learn f: X -> Y



Two layers of logistic units



Highly non-linear decision surface

COLUMBIA UNIVERSITY

Consider humans:

- Neuron switching time ~ .001 second
- Number of neurons ~ 10^{10}
- Connections per neuron ~ 10^{4-5}
- Scene recognition time ~ .1 second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

Properties of artificial neural nets (ANN's):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process

Columbia University

Overview

- Derived features Z_m are obtained by applying the *activation function* σ to linear combinations of the inputs:

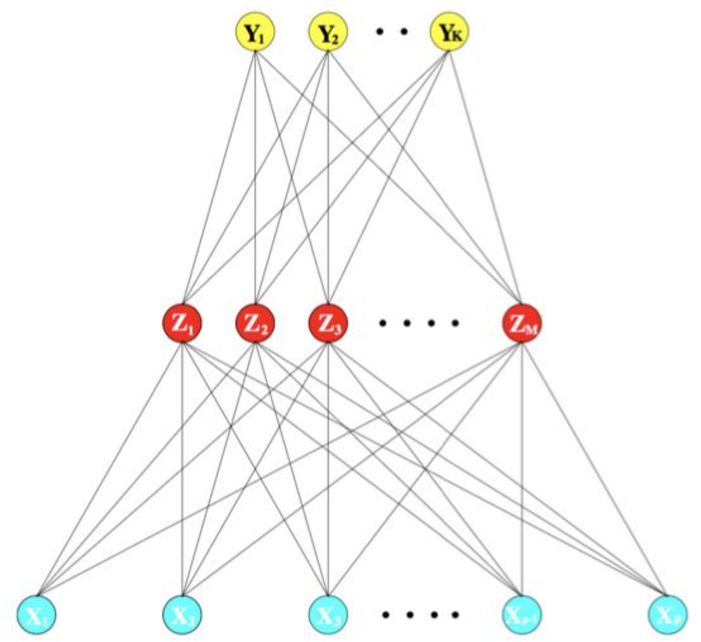
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M.$$

- The target Y_k (or T_k in the figure) is modeled as a function of linear combinations of the Z_m:

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K.$$

- The output function $g_k(T)$ allows a final transformation of the vector of outputs T:

$$f_k(X) = g_k(T), \quad k = 1, \ldots, K.$$

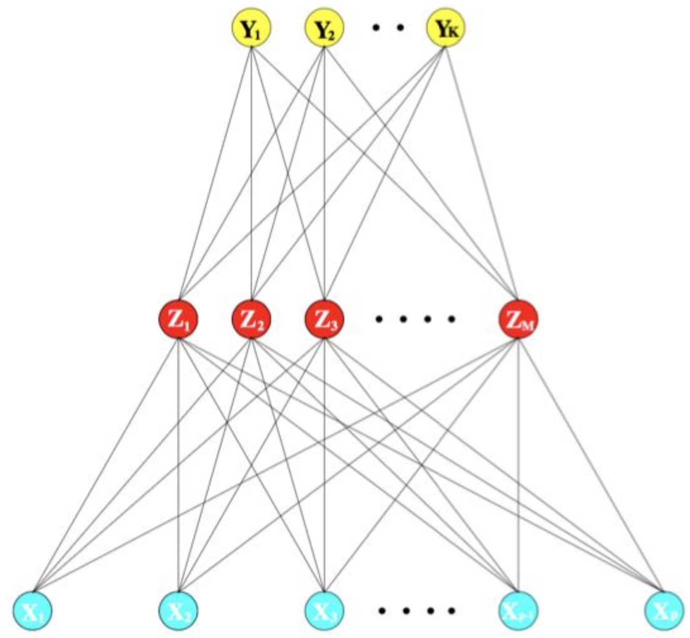


Schematic of a single hidden layer, feed-forward neural network

Columbia University

Artificial neurons

- Each artificial neuron has inputs and produces a single output which can be sent to multiple other neurons. The inputs can be the feature values of a sample of external data, such as images or documents, or they can be the outputs of other neurons.
- The outputs of the final output neurons of the neural net accomplish the task.



Schematic of a single hidden layer, feed-forward neural network

COLUMBIA UNIVERSITY

Artificial neurons

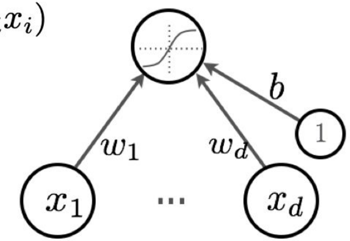
- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron output activation:

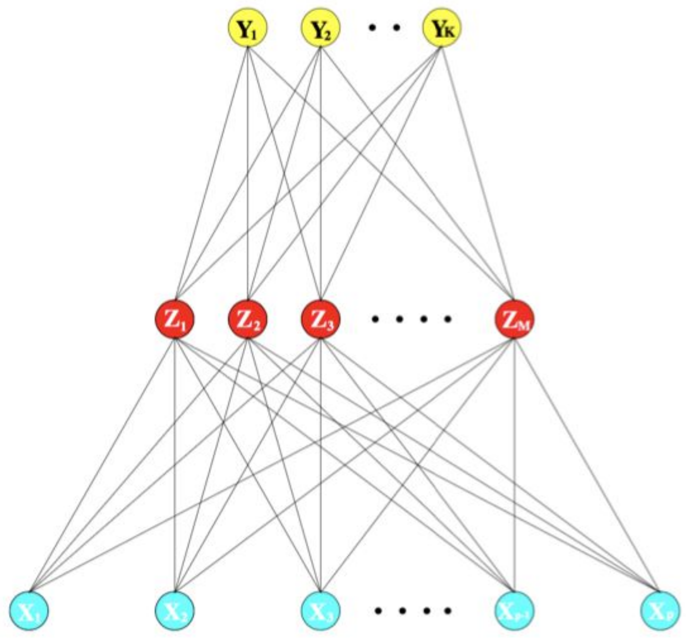
$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$

where
$\mathbf{w}$ are the weights (parameters)
b is the bias term
$g(\cdot)$ is called the activation function



Activation functions

- An activation function of a node defines the output of that node given an input or set of inputs.
- Usually nonlinear.

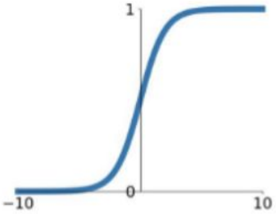


Schematic of a single hidden layer, feed-forward neural network

Activation functions

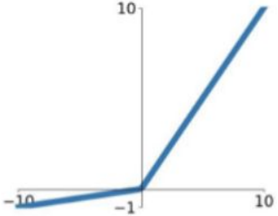
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



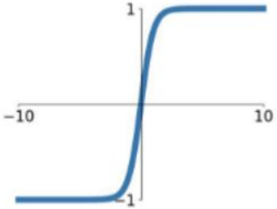
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

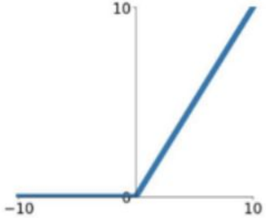


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

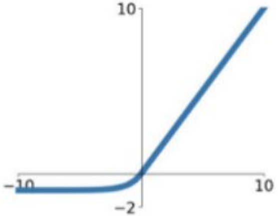
ReLU

$\max(0, x)$



ELU

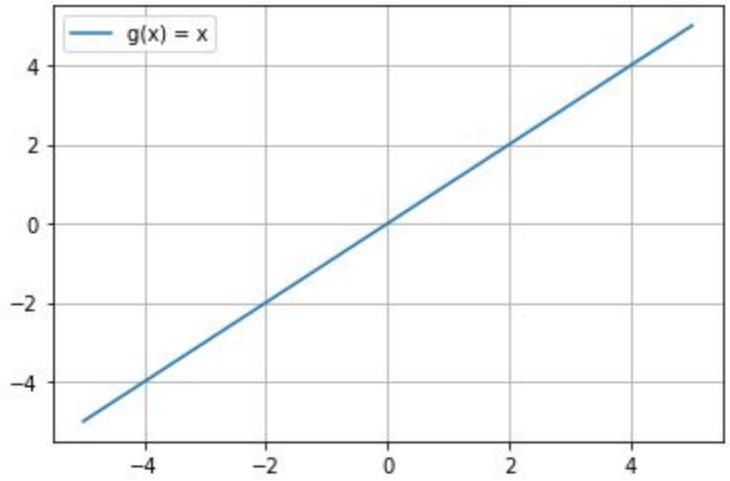
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Columbia University

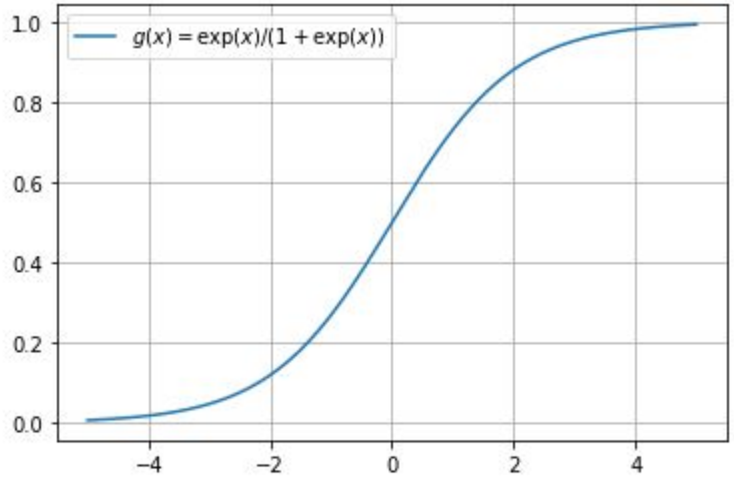
Activation function: linear

- No nonlinear transformation
- No input squashing



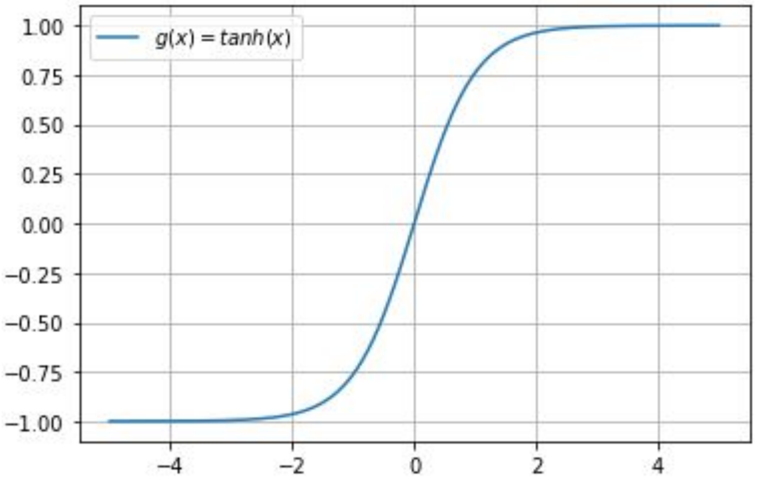
Activation function: sigmoid

- Always positive.
- Squashing the neuron's output between 0 and 1.
- Strictly increasing.



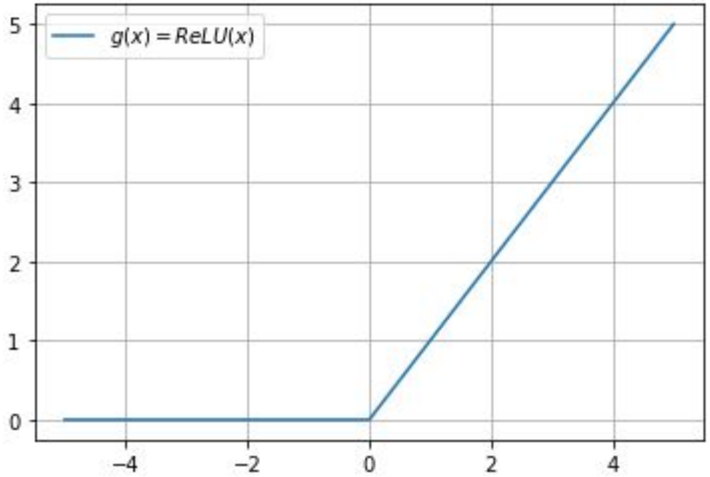
Activation function: tanh

- Can be positive and negative.
- Squashing the neuron's output between -1 and 1.
- Strictly increasing.



Activation function: sigmoid

- Always positive.
- Pushing the neuron's output above 0.
- Strictly increasing.



COLUMBIA UNIVERSITY

Prediction using neural networks

Prediction – Given neural network (hidden units and weights), use it to predict the label of a test point

Forward Propagation –

Start from input layer

For each subsequent layer, compute output of sigmoid unit

Sigmoid unit:

$$o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i x_i)$$

1-Hidden layer,
1 output NN:

$$o(\mathbf{x}) = \sigma\left(w_0 + \sum_h w_h \underbrace{\sigma(w_0^h + \sum_i w_i^h x_i)}_{\text{Oh}}\right)$$

Fitting neural networks: regression tasks

Recall our model is:

$$
\begin{aligned}
Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \ldots, M. \\
T_k &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K. \\
f_k(X) &= g_k(T), \quad k = 1, \ldots, K.
\end{aligned}
$$

The unknow parameters of the model are often called *weights*. We denote the complete set of weights by θ, which consists of

$$
\begin{aligned}
\{\alpha_{0m}, \alpha_m;\ m = 1, 2, \ldots, M\} &\quad M(p+1) \text{ weights}, \\
\{\beta_{0k}, \beta_k;\ k = 1, 2, \ldots, K\} &\quad K(M+1) \text{ weights}.
\end{aligned}
$$

For regression, we use the squared error loss

$$
R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{n} (y_{ik} - f_k(x_i))^2.
$$

Fitting neural networks: classification tasks

Recall our model is:

$$
\begin{aligned}
Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \ldots, M. \\
T_k &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K. \\
f_k(X) &= g_k(T), \quad k = 1, \ldots, K.
\end{aligned}
$$

The unknow parameters of the model are often called *weights*. We denote the complete set of weights by θ, which consists of

$$
\begin{aligned}
\{\alpha_{0m}, \alpha_m;\ m = 1, 2, \ldots, M\} &\quad M(p+1) \text{ weights}, \\
\{\beta_{0k}, \beta_k;\ k = 1, 2, \ldots, K\} &\quad K(M+1) \text{ weights}.
\end{aligned}
$$

For classification we use either squared error or corss-entropy

$$
R(\theta) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log f_k(x_i),
$$

and the correponding classifier is $G(x) = \text{argmax}_k f_k(x)$.

COLUMBIA UNIVERSITY

Connection to gradient descent

Assume we use squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and let $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then we have

$$R(\theta) \equiv \sum_{i=1}^{n} R_i = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2,$$

where

$$f_k(x_i) = g_k(\beta_{0k} + \beta_k^T z_i) = g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km}\sigma(\alpha_{0m} + \alpha_m^T x_i)\right).$$

The derivatives are

$$\begin{aligned}
\frac{\partial R_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi}, \\
\frac{\partial R_i}{\partial \alpha_{ml}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il},
\end{aligned}$$

Columbia University

Connection to gradient descent

Assume we use squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and let $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then we have

$$R(\theta) \equiv \sum_{i=1}^{n} R_i = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2,$$

where

$$f_k(x_i) = g_k(\beta_{0k} + \beta_k^T z_i) = g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km}\sigma(\alpha_{0m} + \alpha_m^T x_i)\right).$$

The derivatives are

$$\begin{aligned}
\frac{\partial R_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi}, \\
\frac{\partial R_i}{\partial \alpha_{ml}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il},
\end{aligned}$$

Columbia University

Updating rule

Assume we use squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ and let $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$. Then we have

$$R(\theta) \equiv \sum_{i=1}^{n} R_i = \sum_{i=1}^{n} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2,$$

where

$$f_k(x_i) = g_k(\beta_{0k} + \beta_k^T z_i) = g_k\left(\beta_{0k} + \sum_{m=1}^{M} \beta_{km}\sigma(\alpha_{0m} + \alpha_m^T x_i)\right).$$

A gradient update at the $(r+1)$st iteration has the form

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}, \\ \alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}}. \end{aligned}$$

Updating rule

If we write the gradients as

$$
\begin{aligned}
\frac{\partial R_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi}, \\
\frac{\partial R_i}{\partial \alpha_{ml}} &= -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il}.
\end{aligned}
$$

Back propagation

If we write the gradients as

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi},$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il}.$$

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi},$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il}.$$

Back propagation

If we write the gradients as

$$\begin{aligned}\frac{\partial R_i}{\partial \beta_{km}} &= \delta_{ki} z_{mi}, \\ \frac{\partial R_i}{\partial \alpha_{ml}} &= s_{mi} x_{il}.\end{aligned}$$

In some sense, δ_{ki} and s_{mi} are "errors" at the output and hidden layer units. The errors satisfy

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}.$$

They are called the *back-propagation equations*. The updates can be implemented with a two-pass algorithm:

- *forward pass*: fix weights, compute the predicted values $\hat{f}_k(x_i)$.
- *backward pass*: errors δ_{ki} are computed, and back-propagated to give the errors s_{mi}. Then use both sets of errors to compute the gradients.

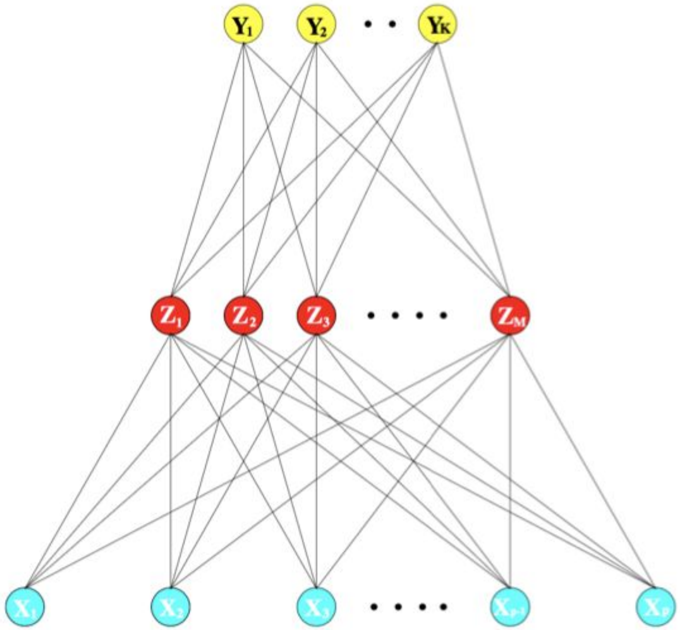
Columbia University

Back propagation

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast



Schematic of a single hidden layer, feed-forward neural network

Columbia University

Starting values

- If the weights are near zero, then the operative part of the sigmoid is roughly zero.
- Usually starting values for weights are chosen to be random values near zero.
- Hence the model starts out nearly linear, and becomes nonlinear as the weights increases.

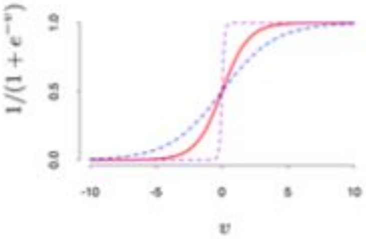


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1+\exp(-v))$ *(red curve), commonly used in the hidden layer of a neural network. Included are* $\sigma(sv)$ *for* $s = \frac{1}{2}$ *(blue curve) and* $s = 10$ *(purple curve). The scale parameter* s *controls the activation rate, and we can see that large* s *amounts to a hard activation at* $v = 0$. *Note that* $\sigma(s(v - v_0))$ *shifts the activation threshold from* 0 *to* v_0.

Multiple minima

The error function $R(\theta)$ is nonconvex, possessing many local minima.

The solution we obtained from back-propagation is a local minimum.

Usually, we try a number of random starting configuration, and choose the solution giving lowest error, or use the average predictions over the collection of networks as the final prediction.

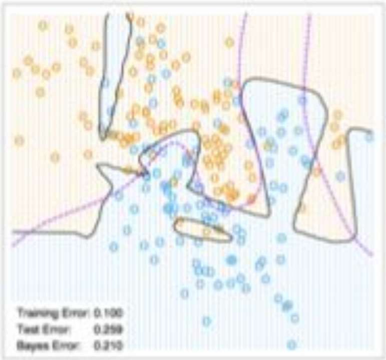
Multiple minima

- Often neural networks have too many weights and will overfit the data at the global minimum of R.
- A regularization method is *weight decay*. We add a penalty to the error function $R(\theta) + \lambda J(\theta)$, where

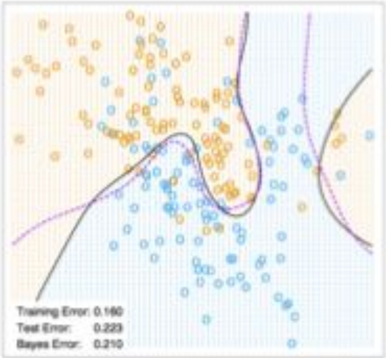
$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,l} \alpha_{ml}^2.$$

- $\lambda \geq 0$ is a tuning parameter, can be chosen by cross-validation.

Neural Network - 10 Units, No Weight Decay

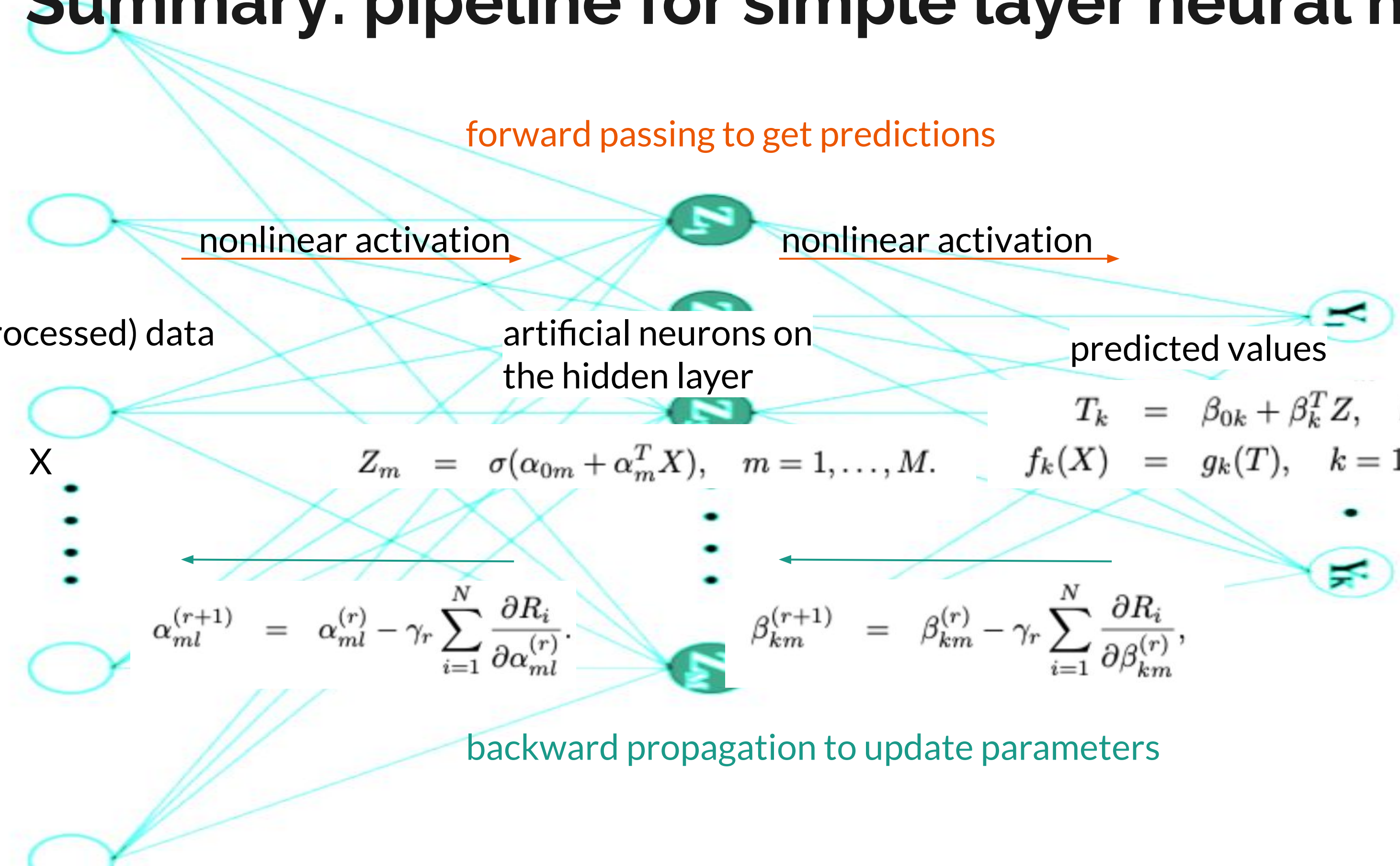


Neural Network - 10 Units, Weight Decay=0.02



Columbia University

Summary: pipeline for simple layer neural net



$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \ldots, M.$$

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K.$$

$$f_k(X) = g_k(T), \quad k = 1, \ldots, K.$$

$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}}.$$

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}},$$

COLUMBIA UNIVERSITY

Expressive capability of ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

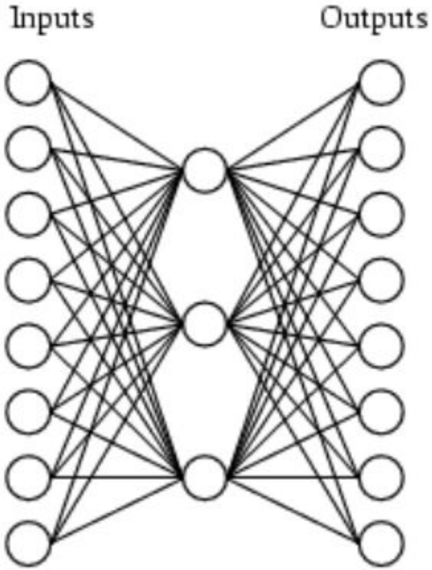
Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Example

target function:

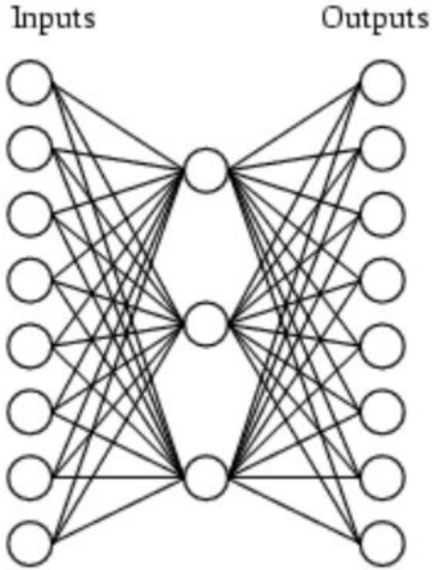
Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001



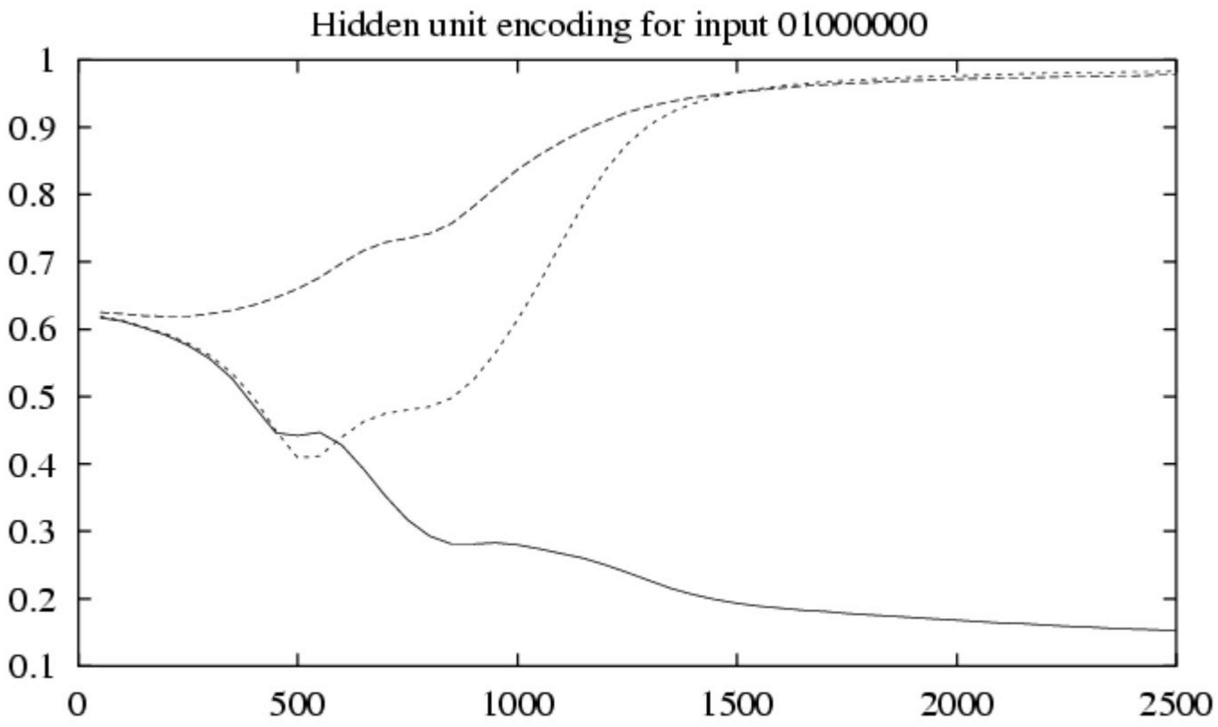
Example

Learned hidden layer representation:

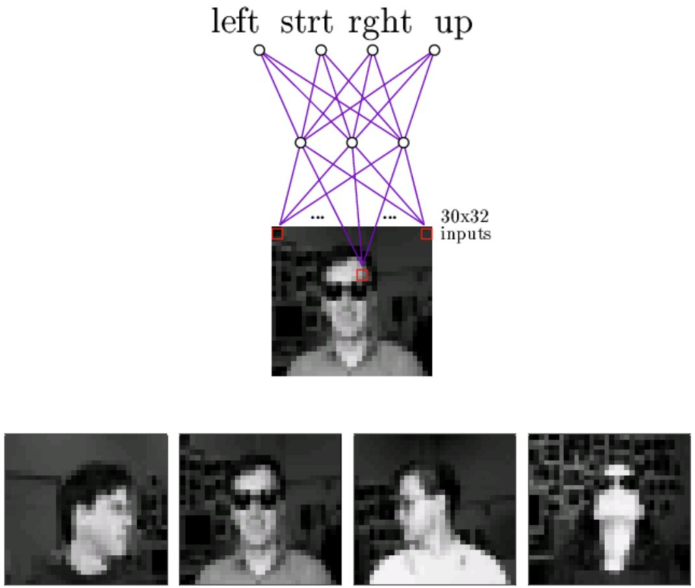
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001



Training

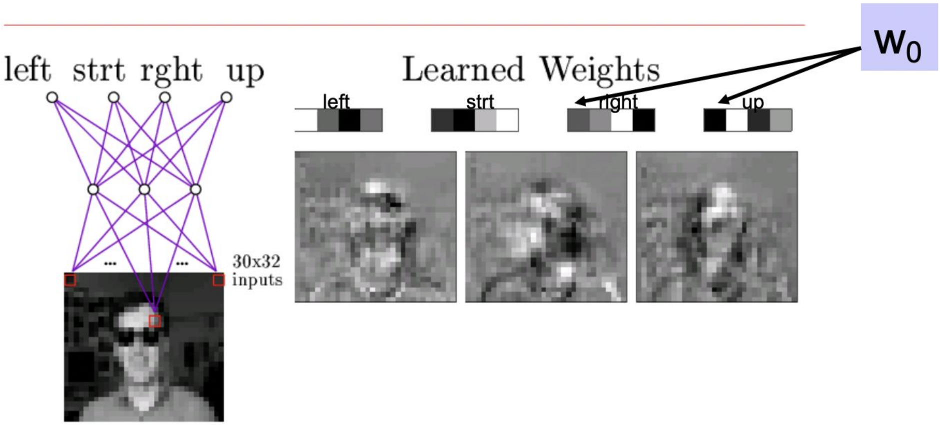


Hidden unit encoding for input 01000000
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
500
1000
1500
2000
2500



Typical input images

90% accurate learning head pose, and recognizing 1-of-20 faces




Typical input images

http://www.cs.cmu.edu/~tom/faces.html

Why training is hard

- Underfitting: use better optimization:

  - use better optimization tools (e.g. batch-normalization, 2nd-order methods).

  - use GPUs, distributed computing.
- Overfitting: use better regularization:

  - unsupervised pre-training

  - stochastic drop-out training
- For many large-scale practical problems, have to scale up:

  - ReLu nonlinearity

  - initialization (e.g. Kaiming He's initialization)

  - stochastic gradient descent

  - momentum, batch-normalization, and drop-out

Preprocessing

- One-hot representation: class 0 or class 1 → (1,0) or (0,1)
- Normalizing the inputs will speed up training (Lecun et al. 1998)

  - could normalization be useful at the level of the hidden layers?
- Batch normalization is an attempt to do that

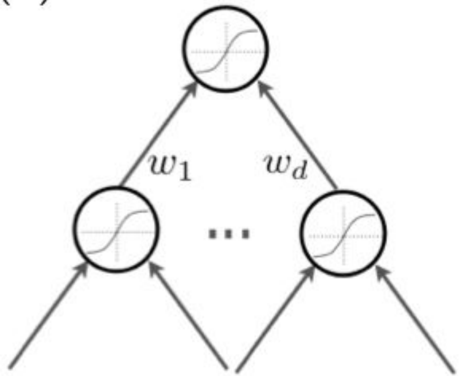
  - each unit's pre-activation is normalized (mean subtraction, stddev division)

  - during training, mean and stddev is computed for each minibatch

  - backpropagation takes into account the normalization

  - at test time, the global mean / stddev is used

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$



Columbia University

Initialization of parameters

- Initialize biases to 0
- For weights
  - Can not initialize weights to 0 with tanh activation
    - All gradients would be zero (saddle point)
  - Can not initialize all weights to the same value
    - All hidden units in a layer will always behave the same
    - Need to break symmetry
  - Sample $\mathbf{W}_{i,j}^{(k)}$ from $U\left[-b, b\right]$, where

$$b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$$

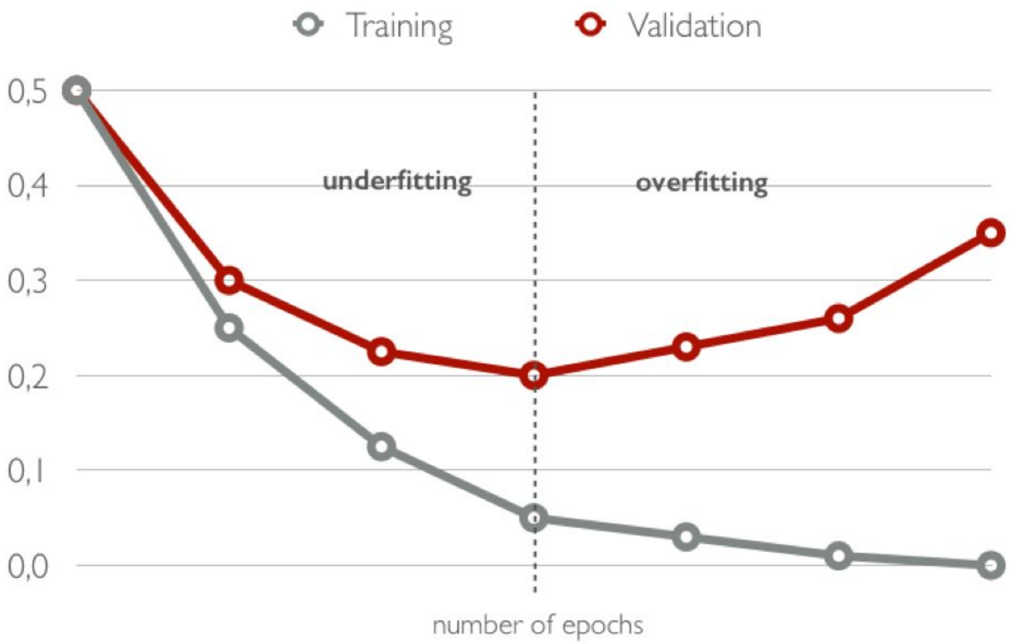
Size of $\mathbf{h}^{(k)}(\mathbf{x})$

Sample around 0 and break symmetry

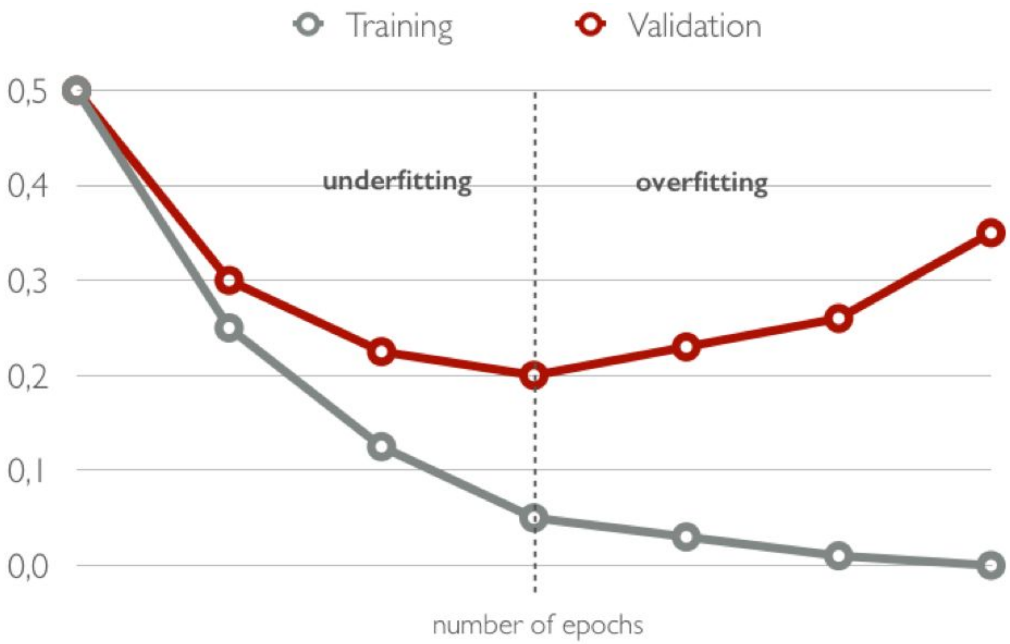
Columbia University

Overfitting

- Overfitting often occurs in applications of neural networks.
- Ways to overcome:
  - Early stopping:
    Stop training process early.
  - Dropout:
    Use random binary masks.



Early stopping



COLUMBIA UNIVERSITY

Dropouts

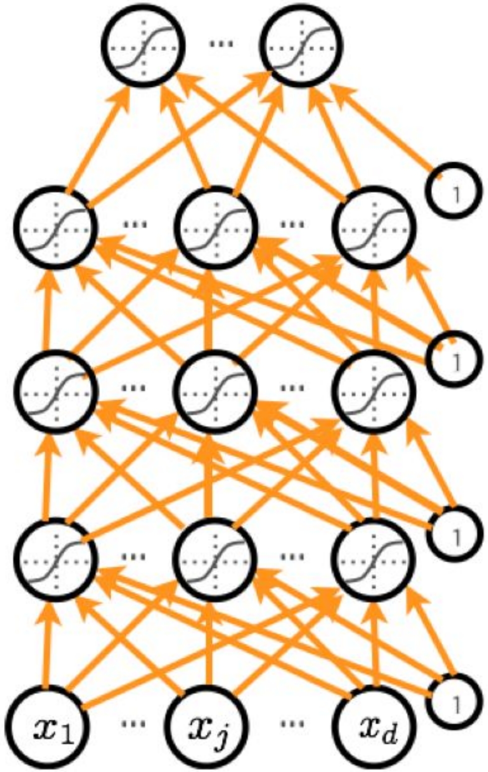
- Cripple neural network by removing hidden units stochastically

  - each hidden unit is set to 0 with probability 0.5

  - hidden units cannot co-adapt to other units

  - hidden units must be more generally useful

- Could use a different dropout probability, but 0.5 usually works well



Model selection

- Training Protocol:
  - Train your model on the Training Set $\mathcal{D}^{\text{train}}$
  - For model selection, use Validation Set $\mathcal{D}^{\text{valid}}$
    - Hyper-parameter search: hidden layer size, learning rate, number of iterations/epochs, etc.
  - Estimate generalization performance using the Test Set $\mathcal{D}^{\text{test}}$
- Remember: Generalization is the behavior of the model on **unseen examples**.

Optimization

- SGD with momentum, batch-normalization, and dropout usually works very well
- Pick learning rate by running on a subset of the data

  - Start with large learning rate & divide by 2 until loss does not diverge

  - Decay learning rate by a factor of ~100 or more by the end of training

  - Use ReLU nonlinearity

  - Initialize parameters so that each feature across layers has similar variance. Avoid units in saturation.
- Use adapted learning rate

Summary:

- Actively used to model distributed computation in brain
- Highly non-linear regression/classification
- Vector-valued inputs and outputs
- Potentially millions of parameters to estimate - overfitting
- Hidden layers learn intermediate representations – how many to use?

- Prediction – Forward propagation
- Gradient descent (Back-propagation), local minima problems

- Coming back in new form as deep networks

  - Try different/deeper architecture

Columbia University

References

- Christopher Bishop: Pattern Recognition and Machine Learning, Chapter 5
- Ziv Bar-Joseph, Tom Mitchell, Pradeep Ravikumar and Aarti Singh: CMU 10-701
- Ryan Tibshirani: CMU 10-725
- Ruslan Salakhutdinov: CMU 10-703
- https://towardsdatascience.com/neural-network-architectures-156e5bad51ba
- https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Columbia University