Learning

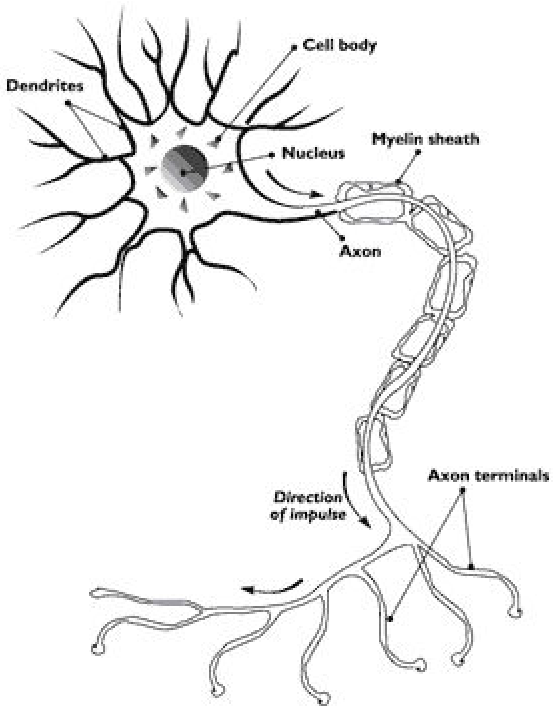
Neural Networks

photo: fisher-price.com

Professor Marie Roch

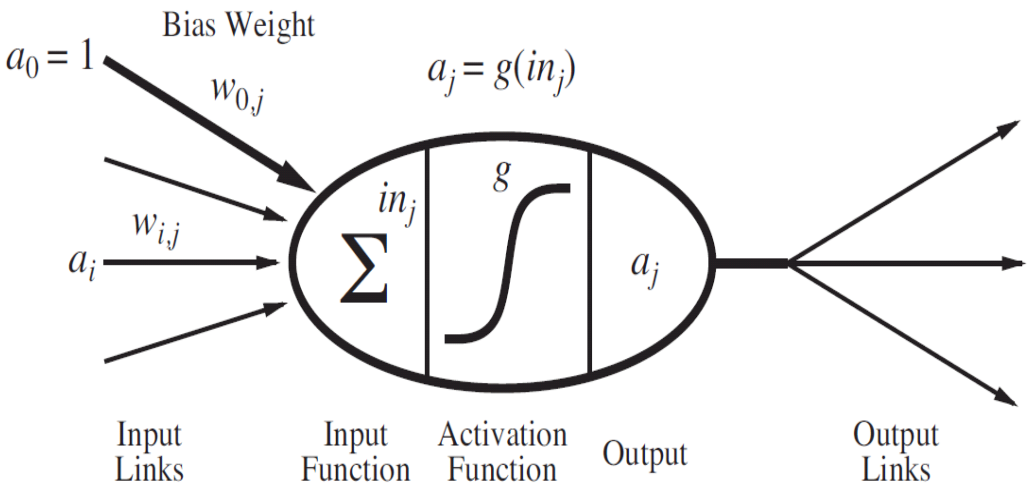
Chapter 19, Russell & Norvig

Connectionist networks (artificial neural networks)



Neuron

National Institute on Drug Abuse

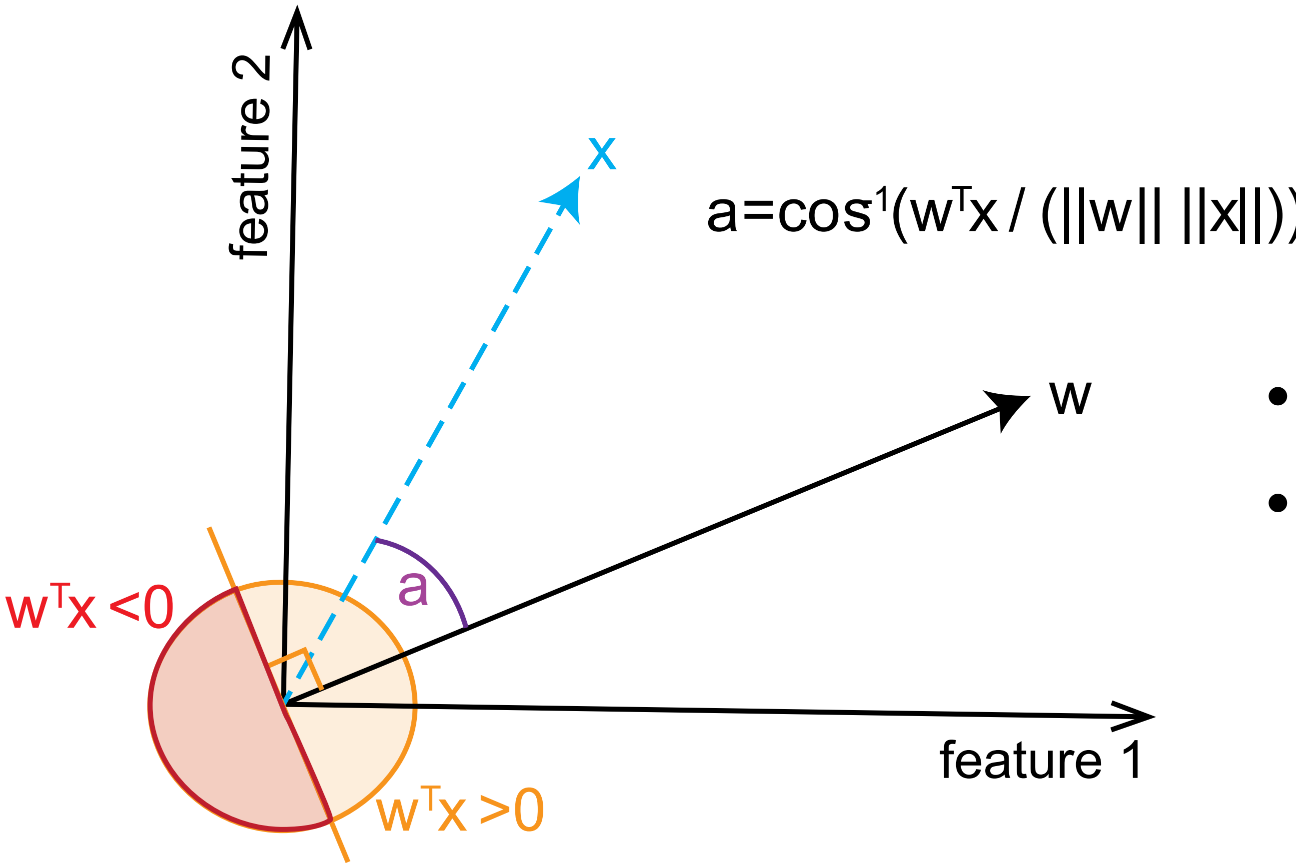


Model of a neuron

Fig. 18.19 R&N

$$a = g(w^T x + b) = g\left(\sum_i w_i x_i + b\right)$$

Remember: interpreting weight vectors

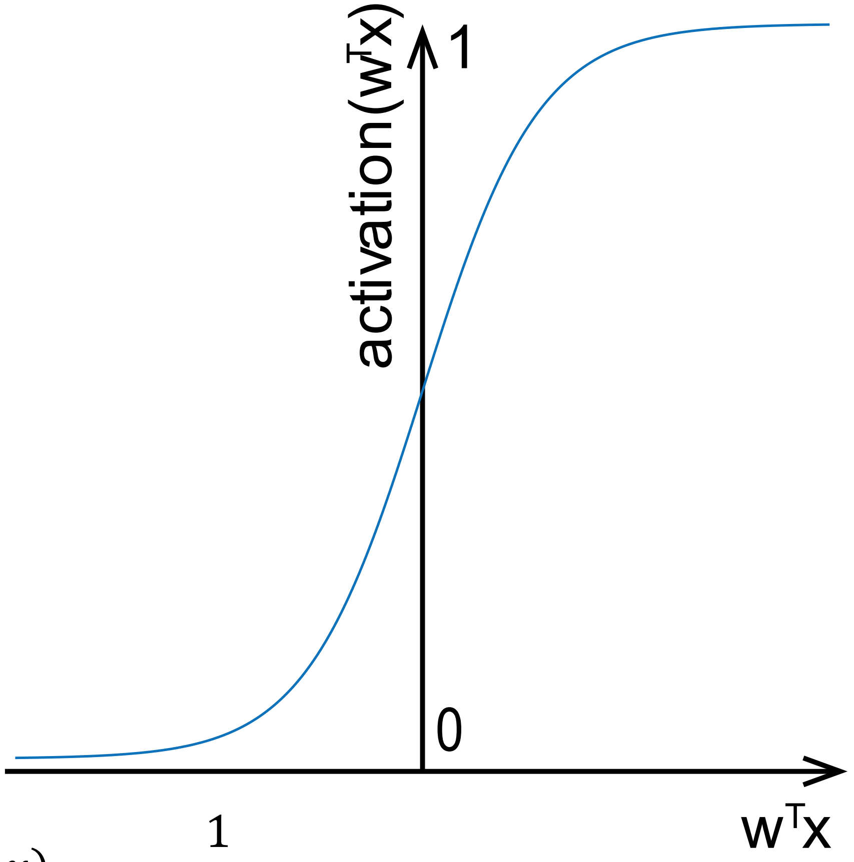


Roch et al. 2021, *Acoustics Today*

- $w^T x \propto \angle a$
- Sign indicates which side of line ⊥ to w vector x falls on

Activation function

- The dot product is passed through an activation function.
- Key ideas about activation functions:
  - nonlinear
  - differentiable
- Common functions:
  - sigmoid or logistic regression (shown)
  - rectified linear unit (ReLU)



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

Roch et al. 2021, *Acoustics Today*

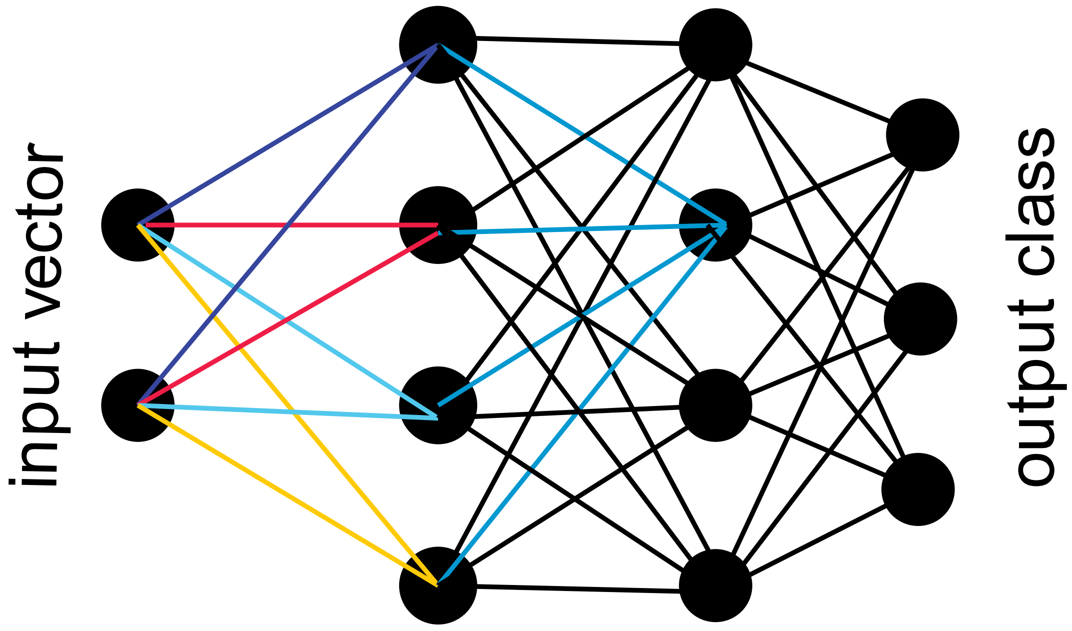
Connectionist networks

- Activations functions for perceptrons are nonlinear:
  - hard threshold
  - logistic regression (frequently called sigmoid function)

- Linking perceptrons together provides complex function modeling capability

Putting it together

- Feature vectors are presented to each node of the network
- Each node computes an output
- Subsequent nodes take previous inputs

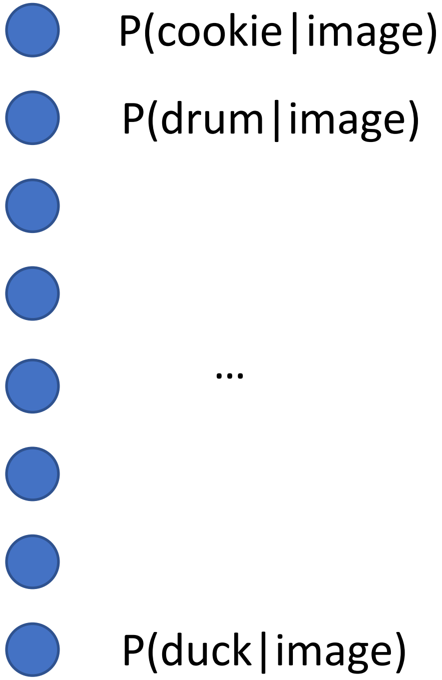


derived from Roch et al. 2021, *Acoustics Today*

Output layer

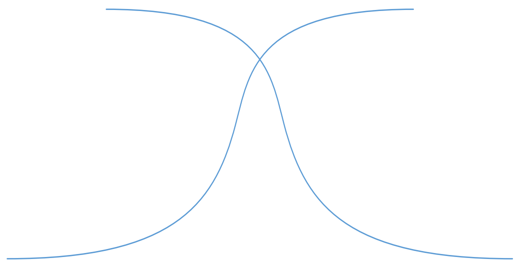
- Output values can be regression targets
- Classification targets
  - probability of one class in a binary class problem
  - set of output vectors with probability of each class
- Labels for training
  - 1 in target category, “one-hot” representation

image: www.teacherspayteachers.com

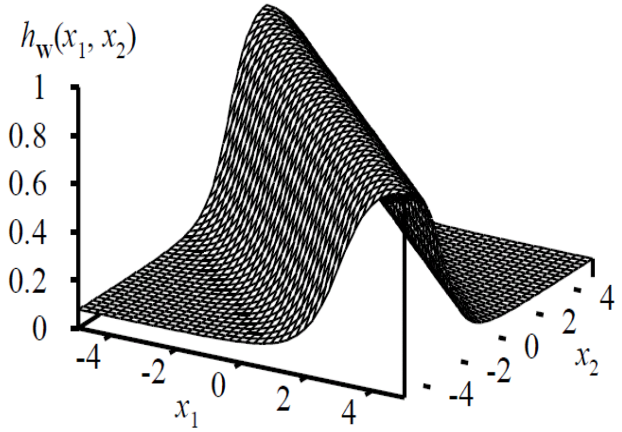


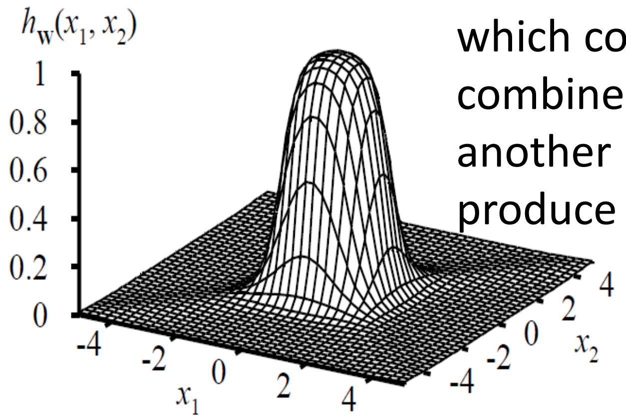
An intuitive view of neural nets

- Suppose we combine two perceptrons whose output functions are reversed



- This could be used to model a ridge in output space





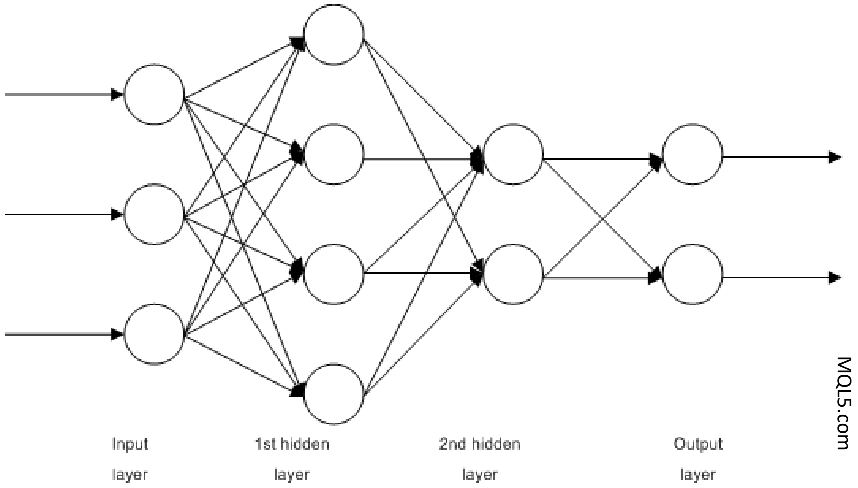
which could be combined with another ridge to produce this

Fig. 18.23 R&N

Learning in a neural network

- Consider input vector **x**
- Output vector **a**

$\vec{x}$



$\vec{a}$

Multilayer Feed-forward network

Learning in a neural network

- Similar to the regression problem, for output **a** and desired output **y**, we can find the loss gradient for each output node

$$\frac{\partial}{\partial w} Loss(w) = \frac{\partial}{\partial w} |y - h_w(x)|^2 = \frac{\partial}{\partial w} \sum_{k=1}^{D} (y_k - a_k)^2 = \sum_{k=1}^{D} \frac{\partial}{\partial w} (y_k - a_k)^2$$

$$h_w(x) = activation(w^T x)$$

$$a_k = \frac{1}{1 + e^{-w_{output,k} \cdot input}}$$

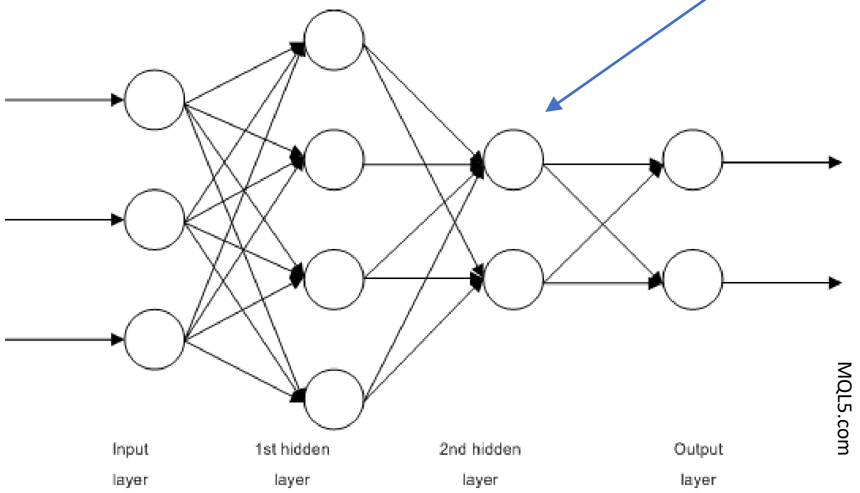
here we assumed a sigmoid activation (other functions possible)

and use the perceptron learning rule for the sum of the gradients at the output layer.

Back-propagation

- What should the targets be for the previous input layer?

$Loss(\vec{w}_{out-1,k})$?



$Loss(\vec{w}_{out,k}, \vec{a}, \vec{y})$

is known

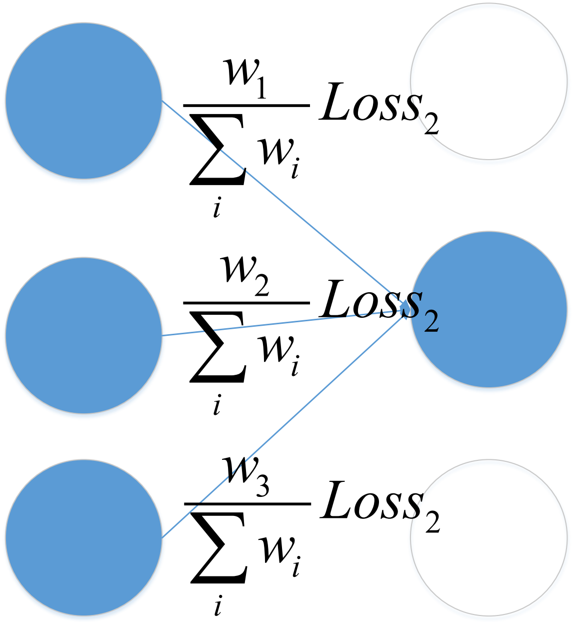
Multilayer Feed-forward network

Back propagating error (overview)

- Error of the k^th output: $Err_k = y_k - a_k$
- We can compute the gradient for any input node (in) and apply the regression rule.
- This gives us a new set of weights for the output node.

Back propagating error (overview)

- After applying the update to the output layer, there still exists loss
- We assign a portion of the loss to each of the input nodes based on their weight.
- This contribution is computed for each node of the current layer

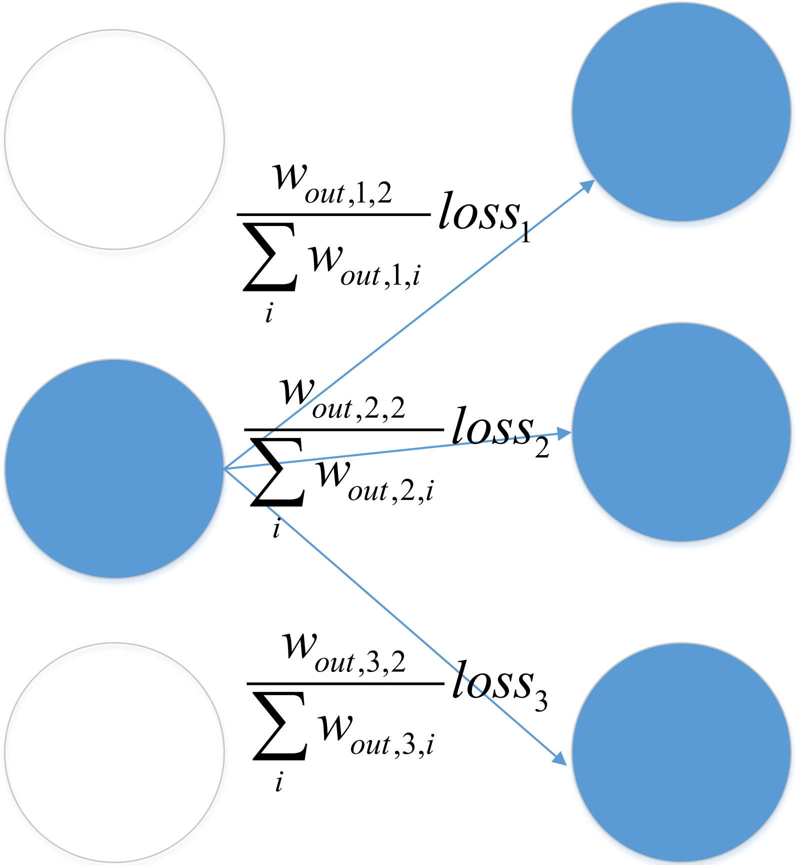


Back propagating error (overview)

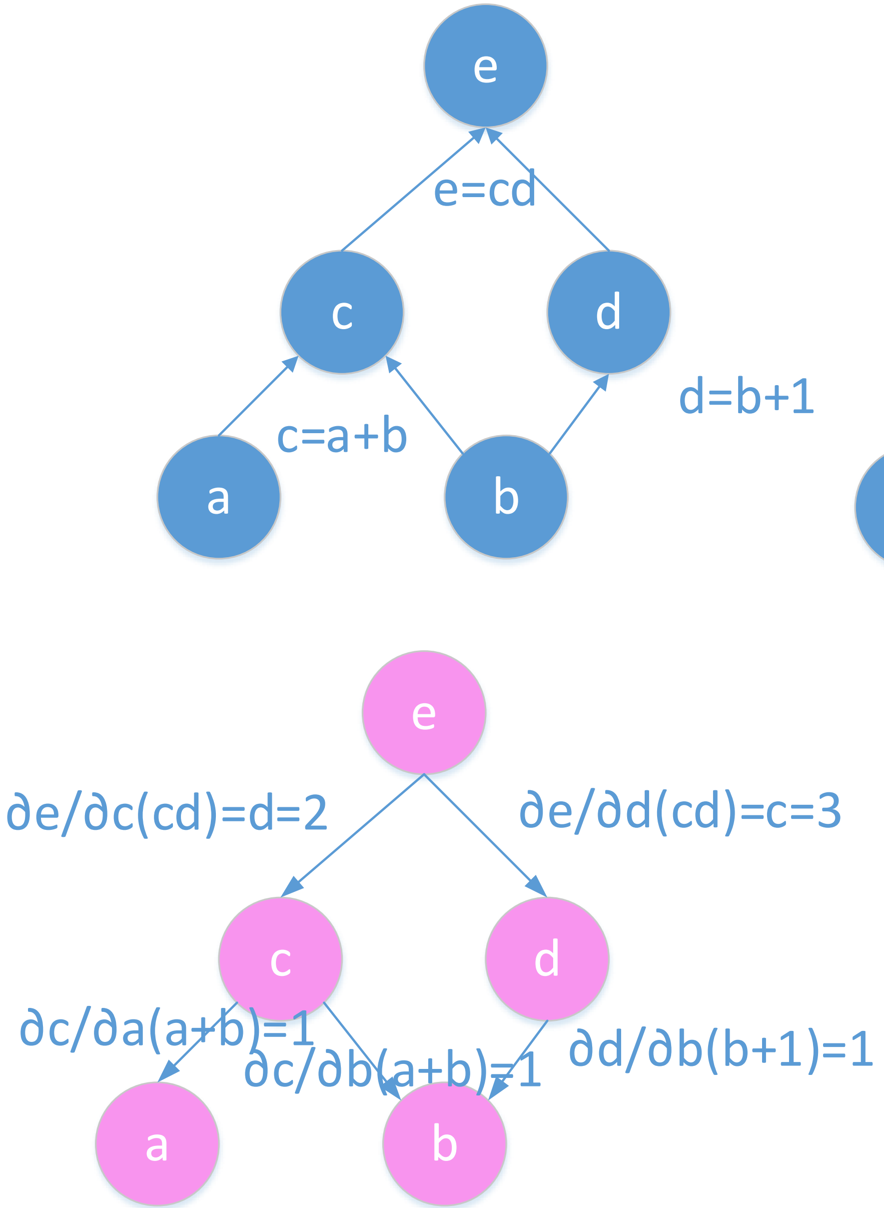
- Now we can look at the sum of losses attributable to each node in the previous layer.

- The sum of these provides us with a loss to minimize.

- Repeat recursively



$w^l_{j,k}$ layer l weight from node k to j



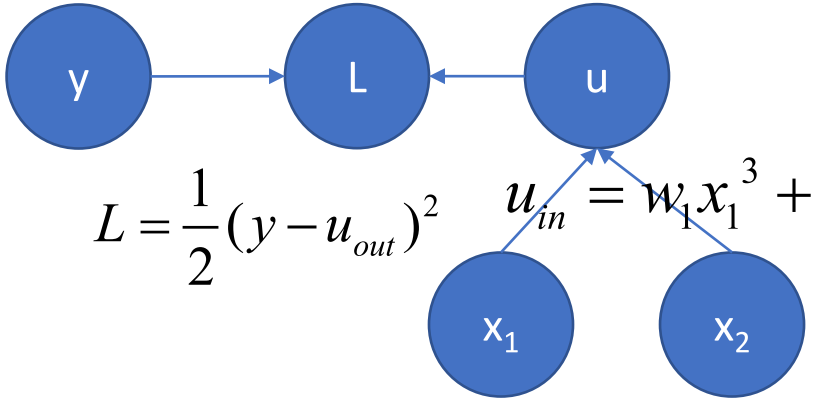
$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 2 \cdot 1 + 3 \cdot 1 = 5$$

Concrete example

Example based on
Christopher Olah's blog post

$$u_{out} = \sigma(u_{in})$$

Activation fn example for backprop



$$L = \frac{1}{2}(y - u_{out})^2$$

$$u_{in} = w_1 x_1^{3} + w_2 x_2$$

partial derivatives

$$\frac{\partial L}{\partial u_{out}} = \frac{2}{2}(y - u_{out})(-1) = u_{out} - y$$

$$\frac{\partial u_{out}}{\partial u_{in}} = \sigma(u_{in})(1 - \sigma(u_{in}))$$

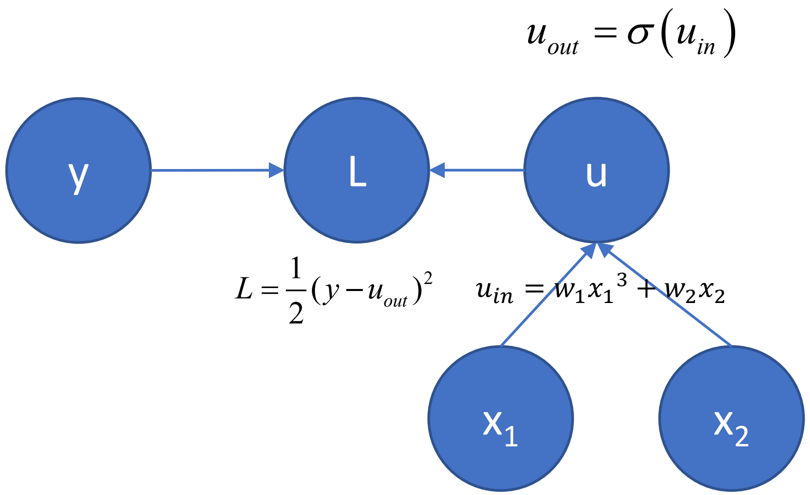
$$\frac{\partial u_{in}}{\partial u_{x_1}} = w_1 3 x_1^2$$

$$\frac{\partial u_{in}}{\partial u_{x_2}} = w_2$$

$$\frac{\partial u_{in}}{\partial u_{w_1}} = x_1^3$$

$$\frac{\partial u_{in}}{\partial u_{w_2}} = x_2$$

Activation fn example for backprop



To update w_1 we use the chain rule:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial u_{out}} \frac{\partial u_{out}}{\partial u_{in}} \frac{\partial u_{in}}{\partial u_{w_1}} = (u_{out} - y)\sigma(u_{in})(1 - \sigma(u_{in}))x_1^3$$

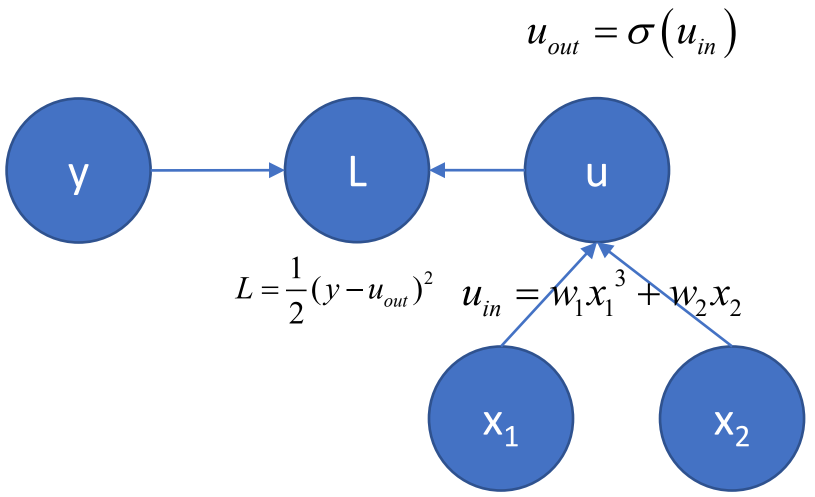
from previous slide

$$\frac{\partial L}{\partial u_{out}} = u_{out} - y$$

$$\frac{\partial u_{out}}{\partial u_{in}} = \sigma(u_{in})(1 - \sigma(u_{in}))$$

$$\frac{\partial u_{in}}{\partial u_{w_1}} = x_1^3$$

Activation fn example for backprop



$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial u_{out}} \frac{\partial u_{out}}{\partial u_{in}} \frac{\partial u_{in}}{\partial u_{w_1}} = (u_{out} - y)\sigma(u_{in})(1 - \sigma(u_{in}))x_1^3$$

Concrete example

$$y = 0, w = \begin{bmatrix} .02 \\ .01 \end{bmatrix}, x = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

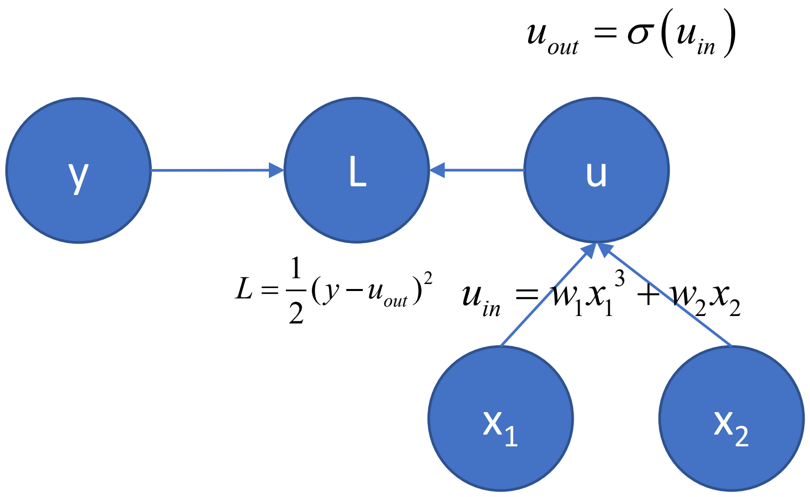
implies

$$u_{in} = w_1 x^3 + w_2 x_2$$
$$= .02 \cdot 3^3 + .01 \cdot 5 = .59$$
$$u_{out} = \frac{1}{1 + e^{-u_{in}}} = \frac{1}{1 + e^{-.59}} = .6434$$
$$L = \frac{1}{2}(y - u_{out})^2$$
$$= .5 \cdot (0 - .6434)^2 = .2070$$

$$\frac{\partial L}{\partial u_{out}} = (u_{out} - y) \cdot -1 = .6434 - 0 = .6434$$
$$\frac{\partial u_{out}}{\partial u_{in}} = \sigma(u_{in})(1 - \sigma(u_{in})) = \sigma(.59)(1 - \sigma(.59))$$
$$= \frac{1}{1 + e^{-.59}}\left(1 - \frac{1}{1 + e^{-.59}}\right) = .6434(1 - .6434) = .2294$$
$$\frac{\partial u_{in}}{\partial u_{w_1}} = x_1^3 = 3^3 = 27$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial u_{out}} \frac{\partial u_{out}}{\partial u_{in}} \frac{\partial u_{in}}{\partial u_{w_1}} = .6434 \cdot .2294 \cdot 27 = 3.9851$$

Activation fn example for backprop



$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial u_{out}} \frac{\partial u_{out}}{\partial u_{in}} \frac{\partial u_{in}}{\partial u_{w_1}} = (u_{out} - y)\sigma(u_{in})(1 - \sigma(u_{in}))x_1^3$$

Suppose we have a learning rate *ε=.01:* $w_1 = w_1 - \epsilon \frac{\partial L}{\partial w_1} = .02 - .01 \cdot 3.9851 = -0.0199$

Update of w_2 is left as an exercise, but loss with only w_1 changed:

$$y = 0, w = \begin{bmatrix} -.02 \\ .01 \end{bmatrix}, x = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

implies

$$u_{in} = w_1 x^3 + w_2 x_2 = -.02 \cdot 3^3 + .01 \cdot 5 = -0.49$$

$$u_{out} = \frac{1}{1 + e^{-u_{in}}} = \frac{1}{1 + e^{-(-0.49)}} = 0.3799$$

$$L = \frac{1}{2}(y - u_{out})^2 = .5 \cdot (0 - 0.3799\ )^2 = 0.0722 < \text{old L=.2070}$$

Overfit regressions

- Not a problem for univariate linear regression
- Problematic for multivariate
- Regularization provides penalties for increasing complexity

$$Cost(h_w) = EmpLoss(h) + \lambda Complexity(h)$$

- Common regularization: L_p penalties

$$Complexity(h_w) = L_p(w) = \sum_i |(w_i)|^p$$

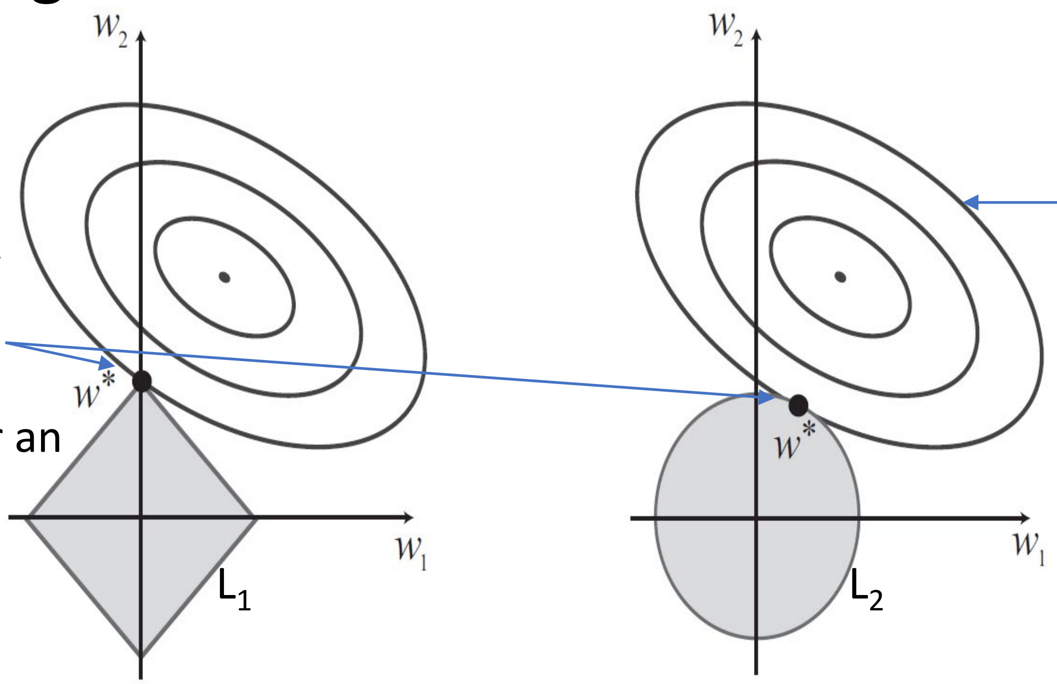
- we regularize by picking the minimal cost hypothesis.

Regularization of regression

L_1 tends to produce sparse models with many zero weights.

L_2 tends to keep weights small overvall

L1 complexity constraint is more likely to intersect near an axis.



Isolines for loss function, loss is lowest at center

Fig. 18.14 R&N

- Minimizing Cost is equivalent to minimizing loss with constraint that complexity $\leq$ some constant.
- Complexity increases as w* moves away from the origin

Inputs

- Common to use a normalization on inputs
  - Learn the transform on the training data
  - Apply it to all data

- Commonly used transform
  - z-score normalization
  - Implemented in scikit learn's StandardScaler class

Outputs and loss functions

- Regression, commonly uses
  - sigmoid activation function
  - log mean squared error loss
- Classification
  - softmax activation on one-hot class outputs
  - cross-entropy loss

Neural net summary

- Supervised learner
  - Training labels either
    - High value for class (n classes → n output nodes)
    - Encoding of class information
    - Regression targets
  - Iterative training typically using a gradient descent algorithm (e.g. back propagation)
- Classification
  - Present features to input nodes
  - Interpret output nodes for category
- Caveats
  - Subject to overfit without appropriate regularization

Keras κέρας

- Library designed to simplify neural net specification
- Originally designed to work with several neural net packages including Tensorflow
- Now part of the official Tensorflow distribution
- Advantages
    - High-level specification of neural nets and other computation.
    - Transparent GPU vs non-GPU programming
    - Rapid specification

http://vem.quantumunlimited.org/the-gates-of-horn/

Keras concepts : Models

Models can be:

- Specified: Functionality is specified by invoking model methods, e.g. add a new layer of N nodes.
- Compiled: A compile method writes the back-end code to generate the model
- Fitted: Optimization step where weights are learned
- Evaluated: Tested on new data

Keras concepts : Models

We can use a Sequential model for a feed-forward network

```
from tensorflow.keras.models import Sequential
model = Sequential()
```

Keras concepts: Layers

- Layers can be added to a model
- Dense layers
  - compute $f(W^T x+b)$
  - user specifies
    - number of units
    - input/output tensor shapes
      (tensors are N-dimensional arrays)
    - activation functions
    - other options…

A Keras model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, InputLayer

model = Sequential()

# Three category prediction with 2 hidden layers
# and 30 features, categorical output (3 categories)
model.add(InputLayer(shape=(30,)))   # Note (30,) is a tuple w/one element
model.add(Dense(10, activation='relu'))
model.add(Dense(10, activation='relu'))
# Output probability of each category
model.add(Dense(3, activation='softmax'))
```

code for Tensorflow 2+

```
# Create the computational graph
# Specify type of gradient descent, loss metric, and
# measurement metric
model.compile(optimizer = "Adam",
              loss = "categorical_crossentropy",
              metrics = ["accuracy"])

# Not needed: prints architecture summary
model.summary()

# We need examples and labels for supervised learning
# examples:  samples X features numpy.array
examples = get_features()  # you write this

# samples X 1 vector of our 3 categories
labels = get_labels()  # you write this
```

```
from tensorflow.keras.utils import to_categorical

# Our network uses a Multinoulli distribution to
# output one of three choices.  Our labels are scalars,
# we need to convert these to vectors:
# 0 -> [1 0 0], 1 -> [0 1 0], 2 -> [0 0 1]
# this is sometimes called a “one-hot” vector

onehotlabels = to_categorical(labels)

# train the model
# 10 passes (epochs) over data, mini-batch size 100 examples
model.fit(examples, labels, batch_size=100, epochs=10)
```

Using a trained model

- To predict outputs

```
results = model.predict(examples)
```

- results is Nx3 probabilities
  context
- What are the following?
  - `np.sum(results, axis=1)`
  - `np.argmax(results, axis=1)`

Using a trained model

- To evaluate performance

```
# Returns list of metrics
results = model.evaluate(test_examples, test_labels)

# model.metrics_names tells us what was measured
# here:  ['loss', 'categorical_accuracy’]

print(results[1])   # accuracy
# In some fields, it is common to report error: 1 - accuracy
```

Regularization in keras

L1/L2 regularization is available as classes in keras

```
from tensorflow.keras.layers import Dense
from tensorflow.keras import regularizers
layer = Dense(
    units=64,
    kernel_regularizer=regularizers.L2(0.001)
)
```

kernel regularizer regularizes the weights w (other types of regularizers are supported, but not used as often)

Neural net summary

- Disadvantages
  - frequently hard to interpret
  - Many parameters require large data sets
  - Doesn't do well with imbalanced examples
  - Slow to train
  - Overfits easily and regularization is important
- Advantages
  - Flexible, nonlinear learner
  - Deep architectures are very powerful

photo: Flickr user Tb240904