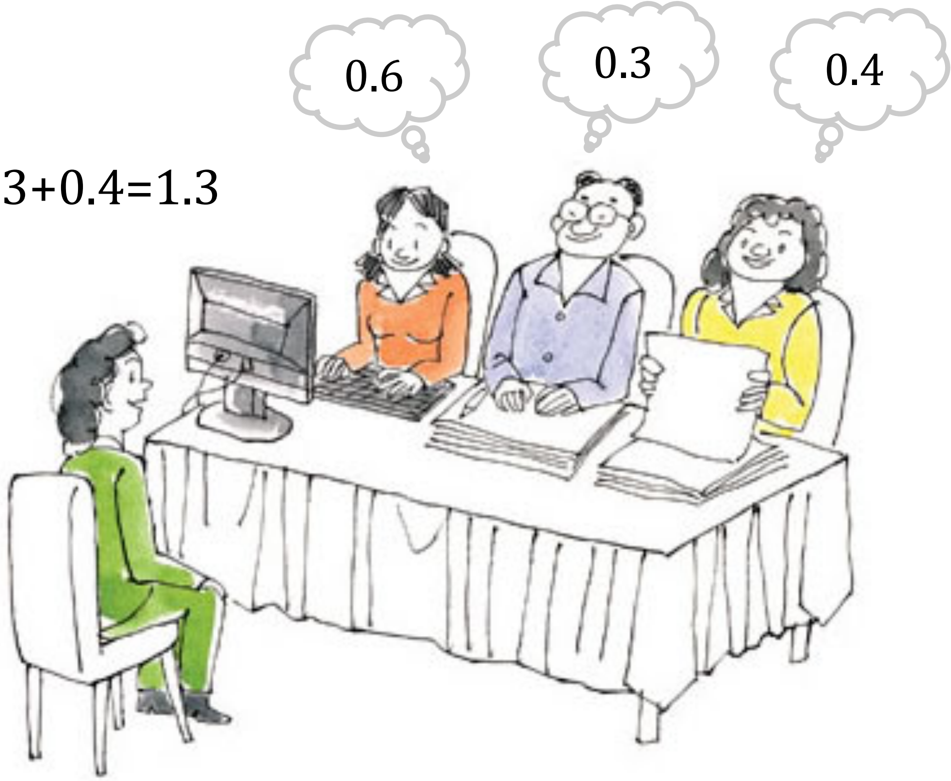
Deep Learning

Lecture 02: Neural Network

Wanxiang Che

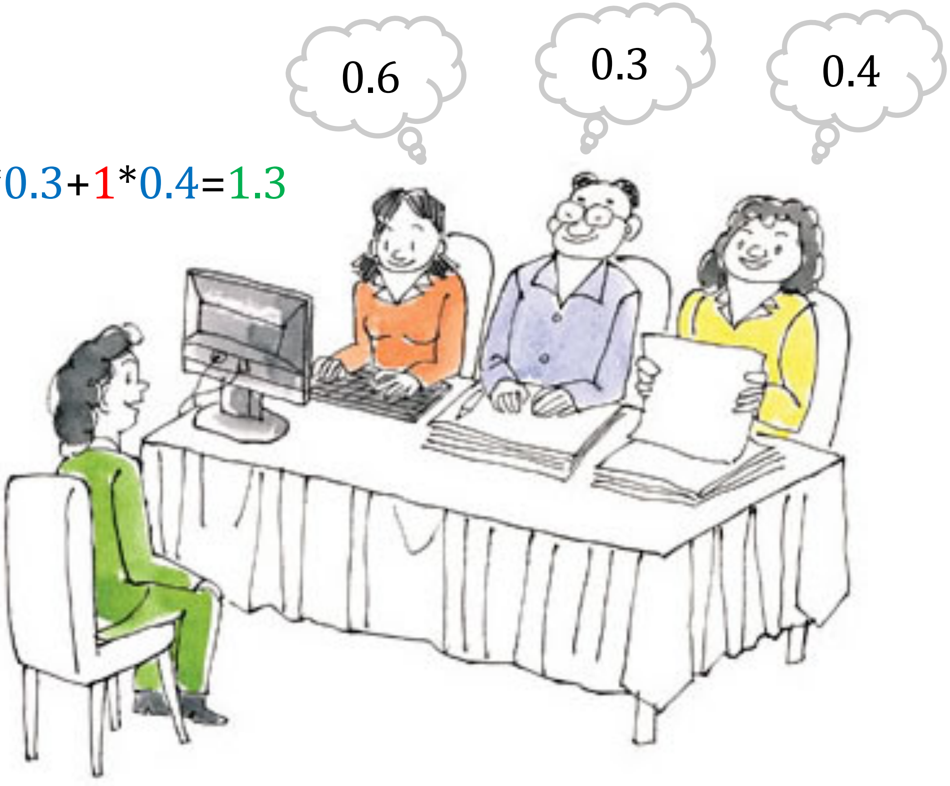
Features

$0.6+0.3+0.4=1.3$



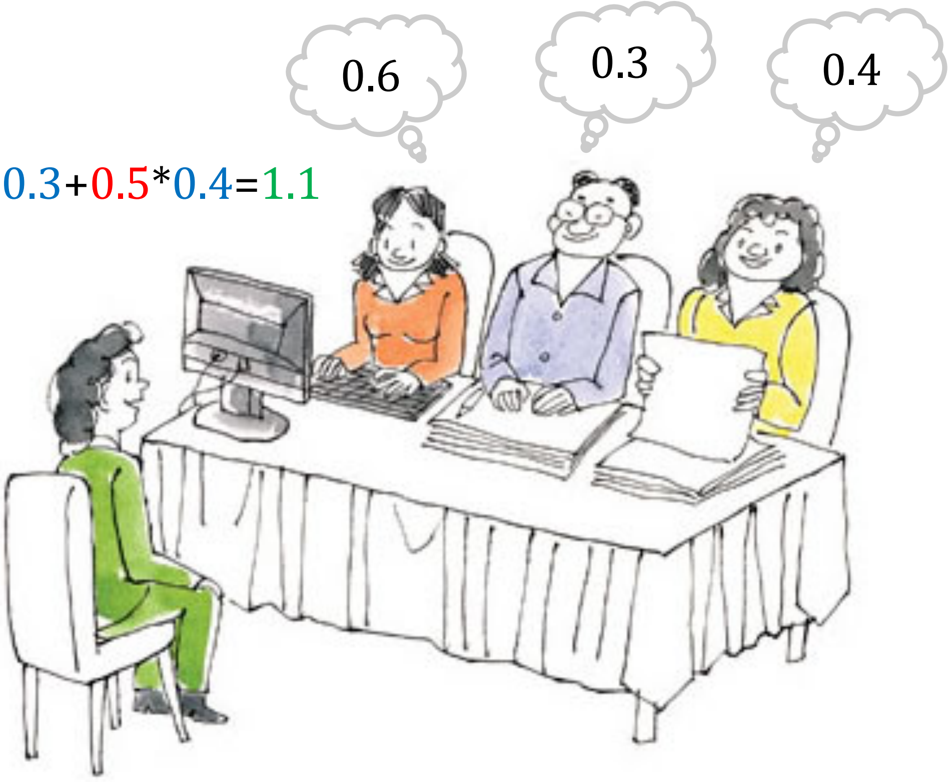
Features

$1*0.6+1*0.3+1*0.4=1.3$



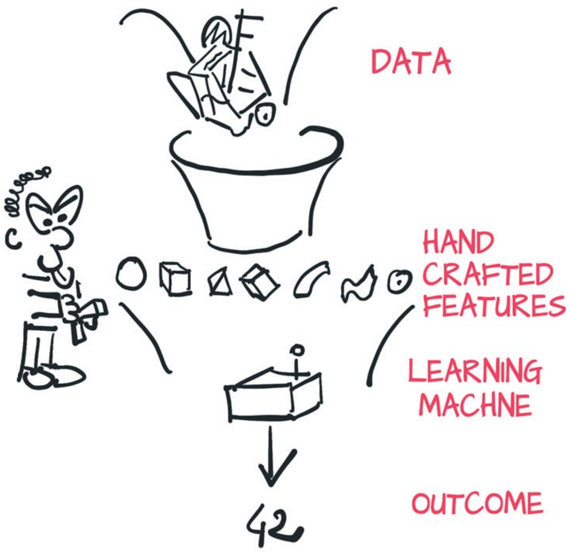
Features

$0.5*0.6+2*0.3+0.5*0.4=1.1$

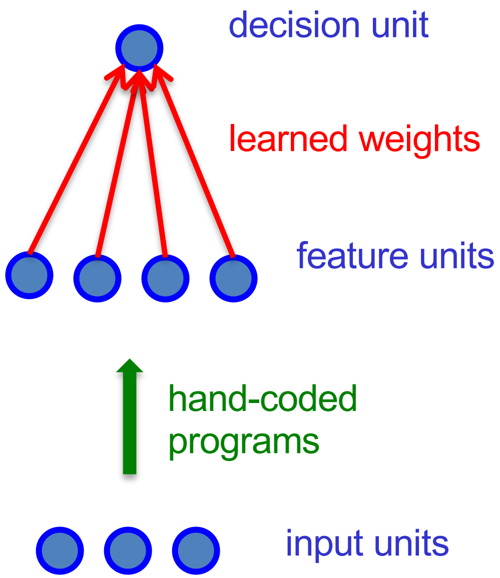


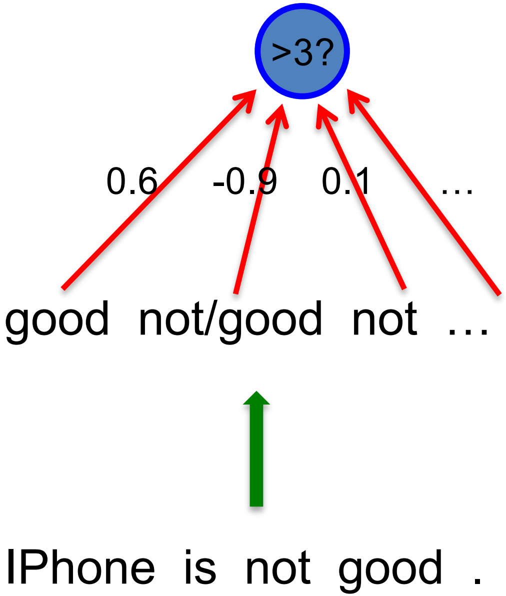
The Standard Paradigm (范式) for ML

1. Convert the raw input vector into a vector of feature activations
   - Use hand-written programs based on common-sense to define the features
2. Weight each of the feature activations to get a single scalar quantity
3. If this quantity is above some threshold, decide that the input vector is a positive example of the target class



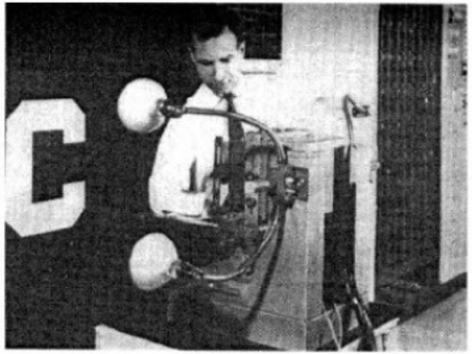
The Standard Perceptron (感知器) Architecture

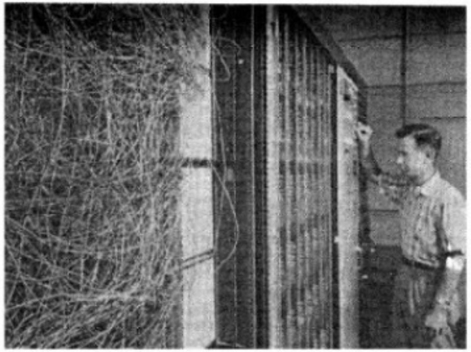


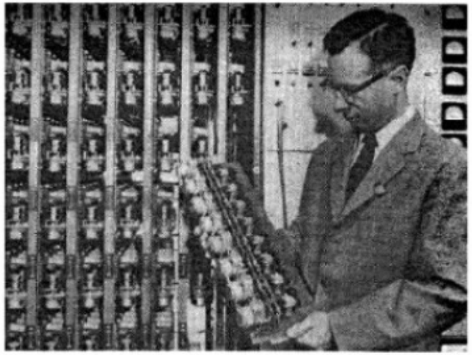


The History of Perceptrons

- Invented in 1957 by Frank Rosenblatt
- In 1969, Minsky and Papert published a book called “Perceptrons” that analysed what they could do and showed their limitations (XOR function)
- Still **widely used today** for tasks with enormous feature vectors, such as NLP







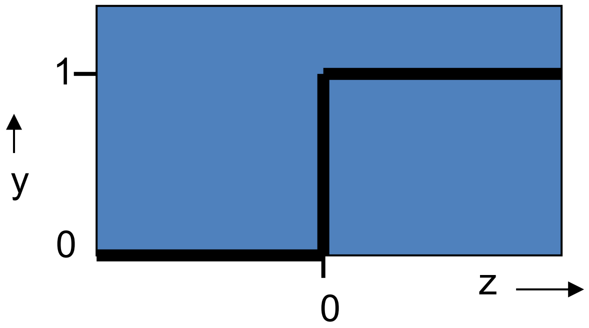
Binary Neurons (Decision Units)

- McCulloch-Pitts (1943)
  - First compute a weighted sum of the inputs from other neurons (plus a bias)
  - Then output a 1 if the weighted sum exceeds 0

Negative threshold

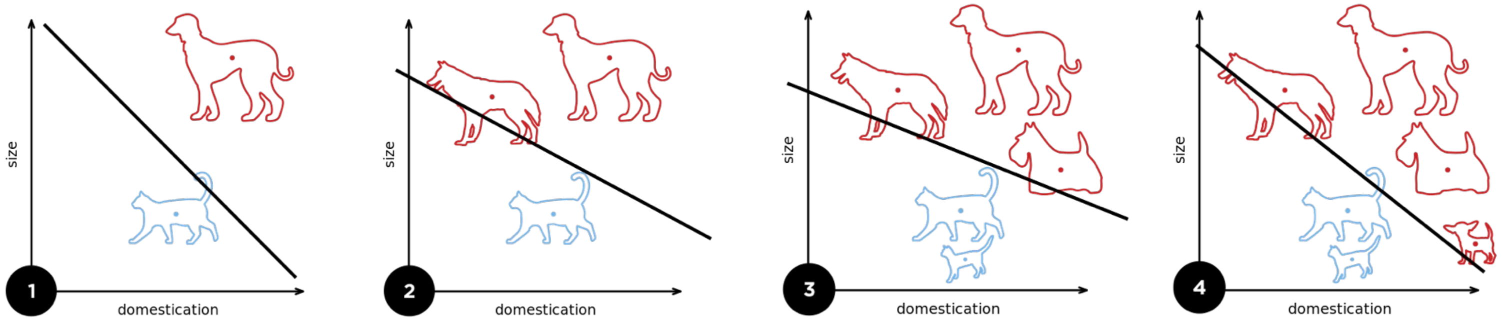
$$z = \boxed{b} + \sum_i x_i w_i$$

$$y = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ otherwise} \end{cases}$$



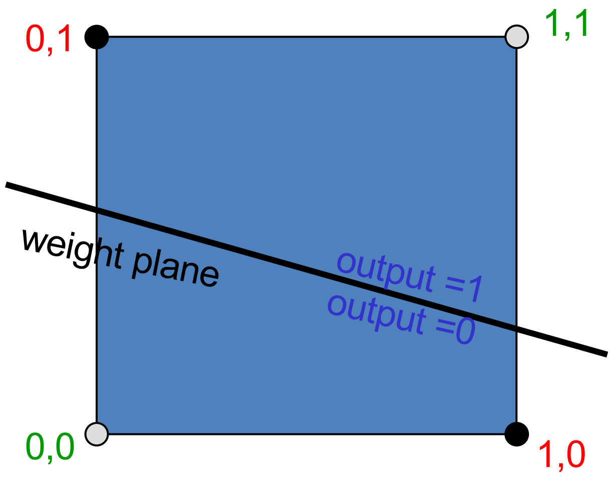
The Perceptron Training Algorithm

1. Initialize the weights $\boldsymbol{w}$
2. For each example n in training set, perform the following steps over the input $\boldsymbol{x}^n$ and desired output t^n
   a. Calculate the output: $y^n = f(\boldsymbol{w} \bullet \boldsymbol{x}^n)$
   b. Update the weights: $\boldsymbol{w} = \boldsymbol{w} + (t^n - y^n)\boldsymbol{x}^n$



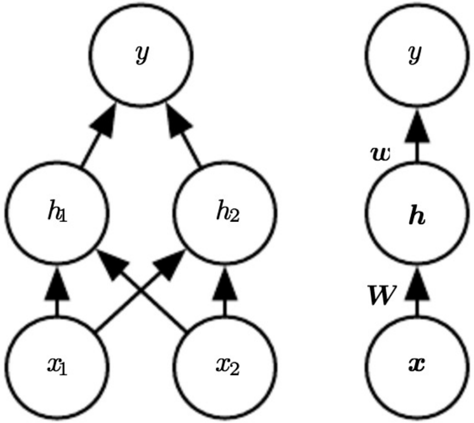
The Limitations of Perceptrons

- The **hand-coded features**
  - Great influence on the performance
  - Need lots of cost to find suitable features
- A **linear classifier** with a hyperplane
  - Cannot separate non-linear data, such as XOR function cannot be learned by a single-layer perceptron



The positive and negative cases cannot be separated by a plane

Learning with Non-linear Hidden Layers

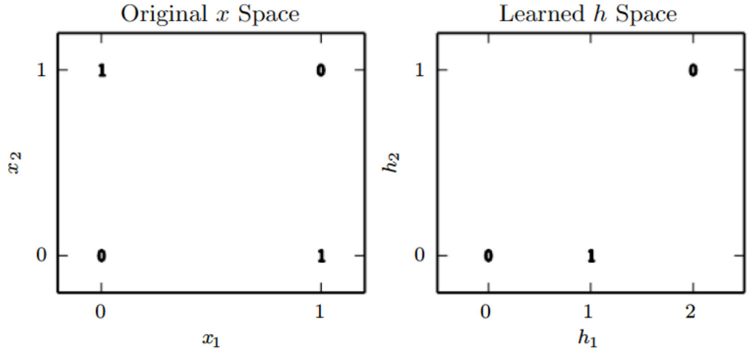


$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

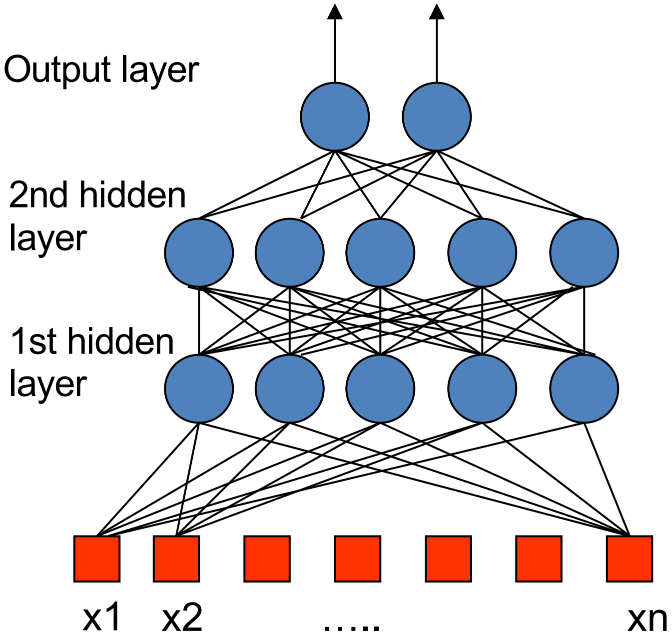
$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

$$b = 0.$$



$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

Multi-layer Perceptron (MLP)

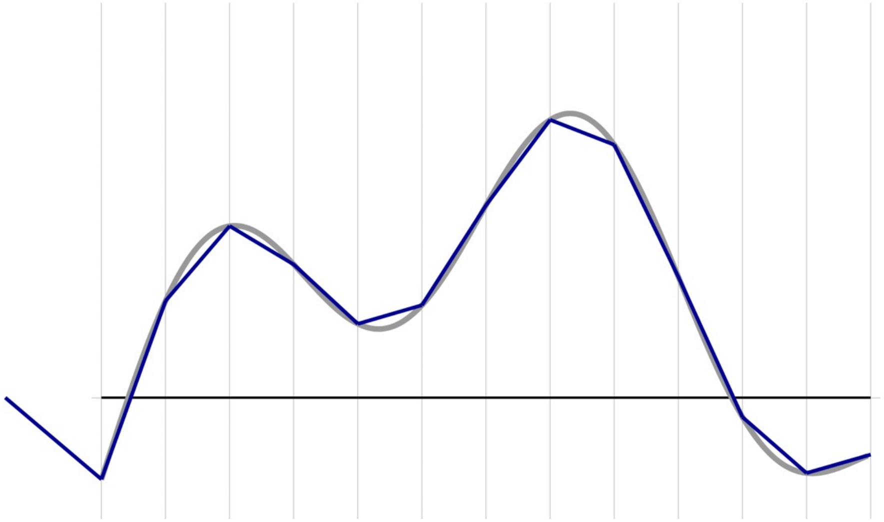


- The information is propagated from the inputs to the outputs
- Learning the weights of hidden units is equivalent to **learning features**
- Networks without hidden layers are very limited in the input-output mappings
  - More layers of linear units do not help. Its still linear
  - Fixed output non-linearities are not enough

Universal approximation

- We can approximate any functions *f* with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \dots$$



Linear Neurons (linear regression)

- The neuron has a **real-valued** output which is a weighted sum of its inputs
- The aim of learning is to minimize the error summed over all training cases
  - The error is the squared difference between the desired output and the actual output.

weight vector

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

neuron's estimate of the desired output

input vector

A Toy Example

- Each day you get lunch at the cafeteria
  - Your diet consists of fish, chips, and ketchup
  - You get several portions of each
- The cashier only tells you the total price of the meal
  - After several days, you should be able to figure out the price of each portion

Solving the Equations Iteratively (迭代)

- Each meal price gives a linear constraint on the prices of the portions

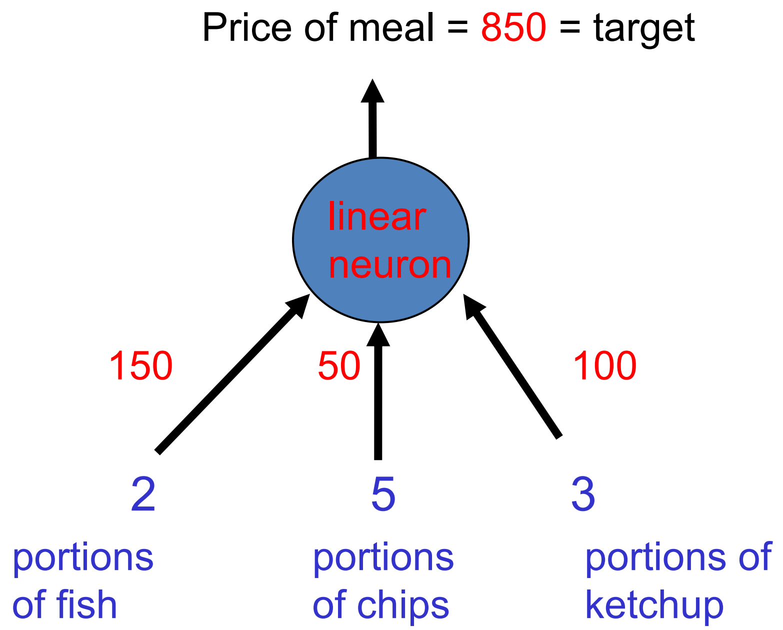
$$price = x_{fish} w_{fish} + x_{chips} w_{chips} + x_{ketchup} w_{ketchup}$$

- The prices of the portions are like the weights in of a linear neuron

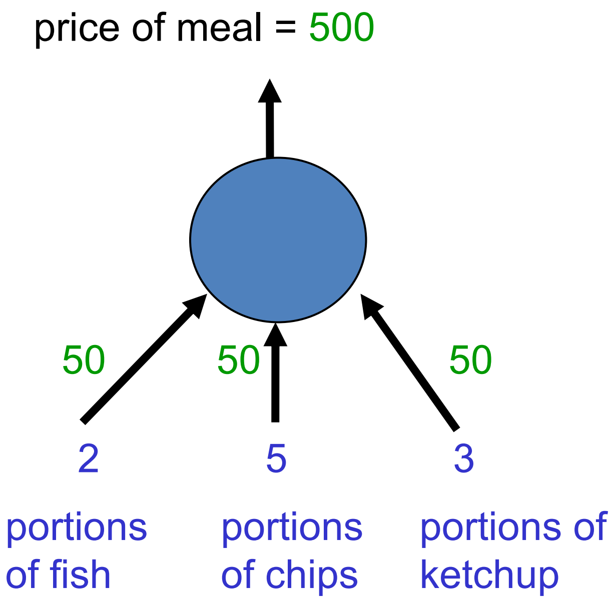
$$\mathbf{w} = (w_{fish}, w_{chips}, w_{ketchup})$$

- The iterative approach
  - Start with **guesses** for the weights and then adjust the guesses slightly to give a **better** fit to the prices given by the cashier

The True Weights Used by the Cashier



With an Random Initial Weights



- Residual error = 850 – 500 = 350
- The “delta-rule” for learning

$$\Delta w_i = \varepsilon \, x_i \, (t - y)$$

- With a learning rate: 1/35, the weight changes are +20, +50, +30
- New weights of 70, 100, 80.
  - Notice that the weight for chips got worse!

Implementation in Python

- ```python
  eta = 1 / 35.0
  ws = [50, 50, 50]

  train = (
    ((2, 5, 3), 850), ((1, 4, 7), 1050), ((2, 3, 5), 950), ((3, 6, 9), 1650), ((7, 4, 1), 1350),
    )

  for _ in range(100):
    for xs, t in train:
      y = sum([w * x for w, x in zip(ws, xs)])
      delta_ws = [eta * x * (t - y) for x in xs]
      ws = [w + delta_w for w, delta_w in zip(ws, delta_ws)]

  print ws
  ```

- [149.9999854696171, 50.00004609118093, 99.99997795027406]
  ```

Deriving the delta rule

- Define the error (loss/cost) as the squared residuals summed over all training cases (损失/代价函数)
- Now differentiate to get error derivatives (导数) for weights
- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases

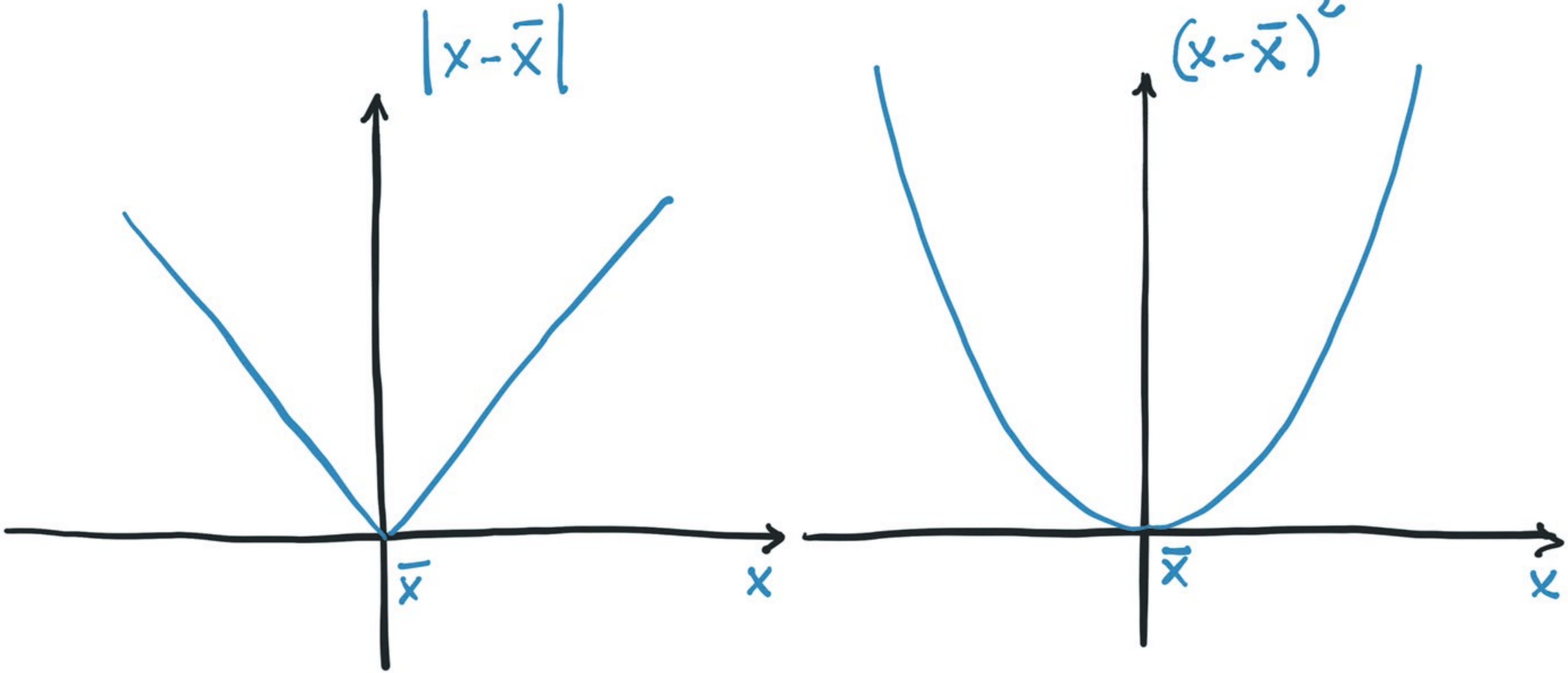
$$\Rightarrow \quad E = \frac{1}{2} \sum_{n \in training} (t^n - y^n)^2$$

$$\Rightarrow \quad \frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{dE^n}{dy^n}$$

$$= -\sum_n x_i^n (t^n - y^n)$$

$$\Rightarrow \quad \Delta w_i = -\varepsilon \frac{\partial E}{\partial w_i} = \sum_n \varepsilon \, x_i^n (t^n - y^n)$$

Absolute vs. Squared Residuals



General Optimizing (Learning) Algorithms

- Gradient Descent (梯度下降)

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}; \boldsymbol{\theta})$$

- Stochastic Gradient Descent (SGD, 随机梯度下降)
  - Minibatch SGD (m > 1), Online GD (m = 1)

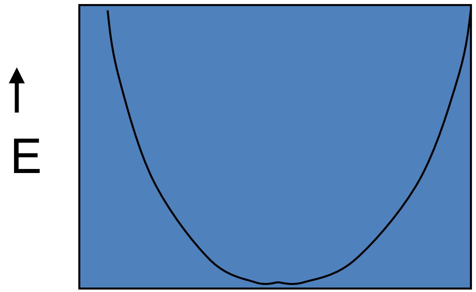
Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

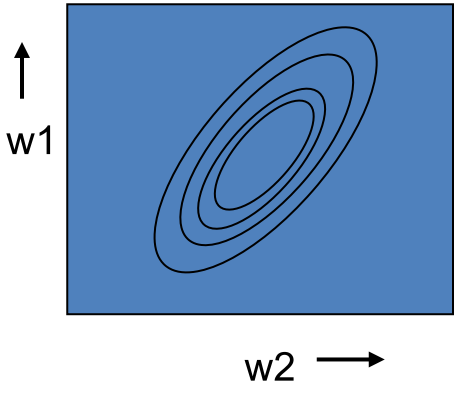
---

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while $\boxed{\text{stopping criterion not met}}$ **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

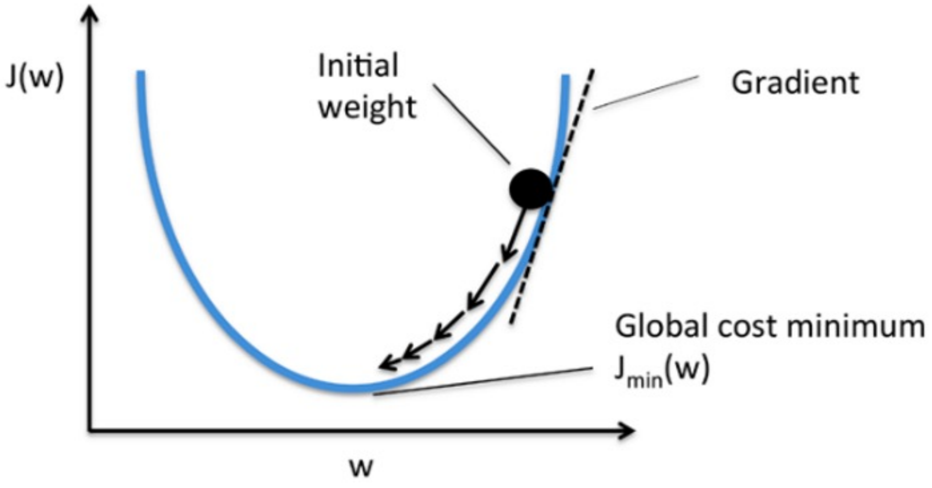
The Error Surface

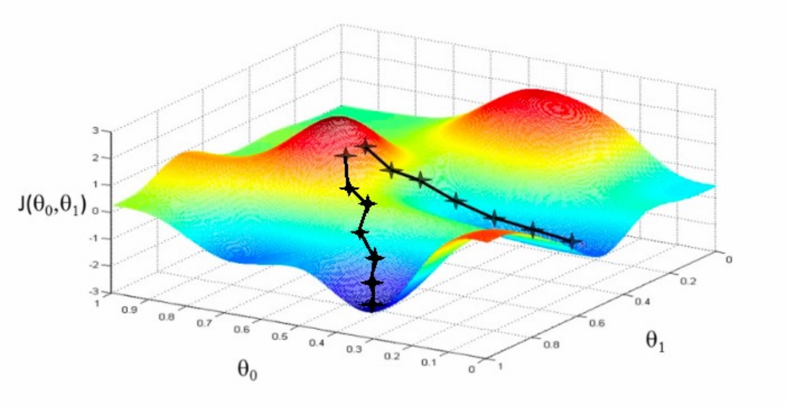
- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error
  - For a linear neuron with a squared error, it is a quadratic bowl
  - Vertical cross-sections are parabolas
  - Horizontal cross-sections are ellipses
- For multi-layer, non-linear nets the error surface is much more complicated.

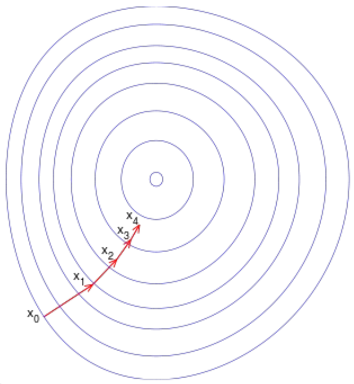




Illustrations of Gradient Descent





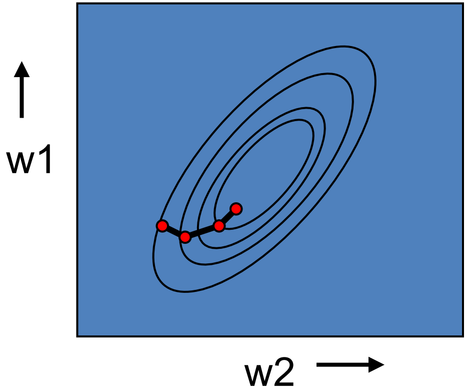


Behavior of the Iterative Learning Procedure

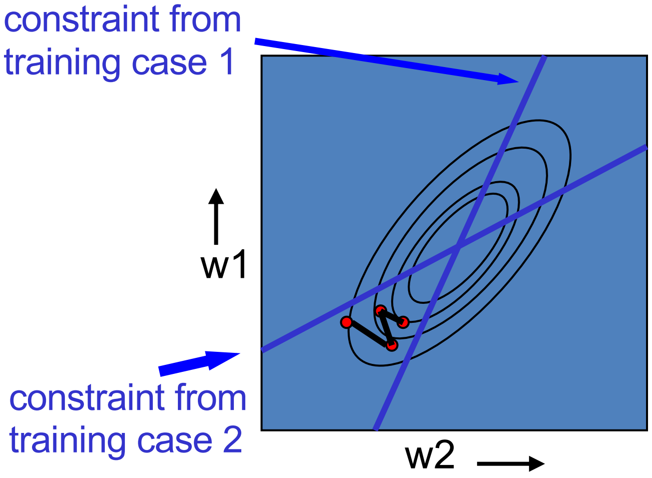
- Does the learning procedure eventually get the right answer?
  - There **may be no perfect** answer
  - By making the learning rate small enough we can get as close as we desire to the best answer
- How quickly do the weights converge to their correct values?
  - It can be very slow if two input dimensions are **highly correlated**. If you almost always have the same number of portions of ketchup and chips, it is hard to decide how to divide the price between ketchup and chips.

Online versus batch learning

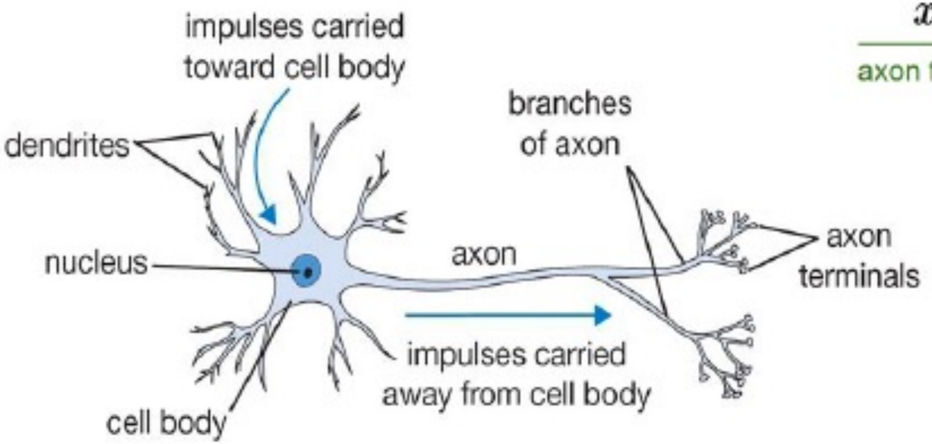
- The simplest kind of batch learning does steepest descent on the error surface.
  – This travels perpendicular to the contour lines.

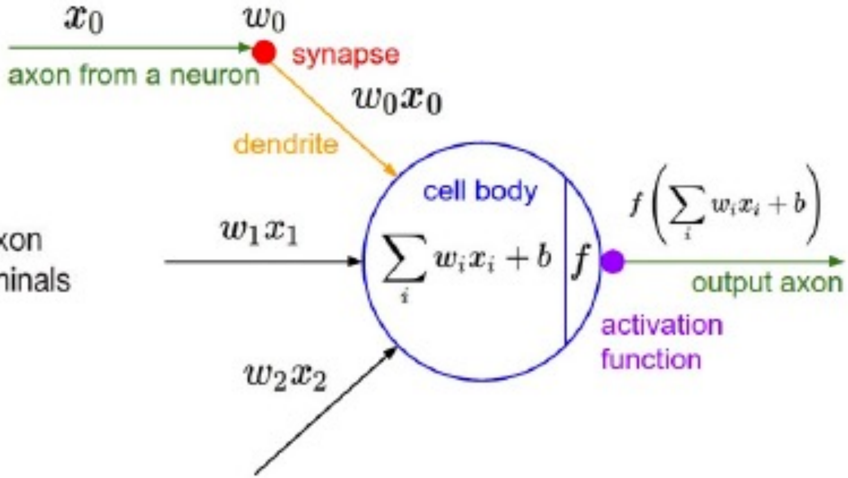


- The simplest kind of online learning zig-zags around the direction of steepest descent:



Non-linear Neurons

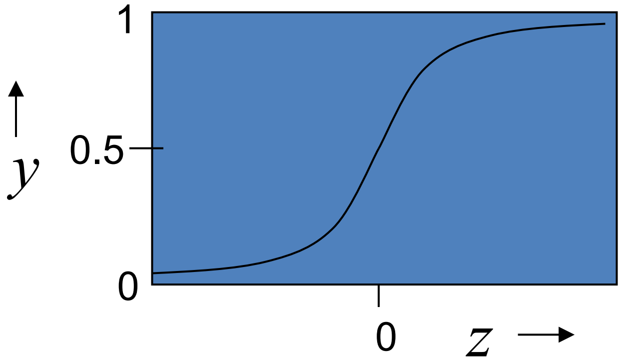




Logistic Regression (sigmoid function)

- These give a real-valued output that is a **smooth and bounded** function of their total input
  - They have nice derivatives which make learning easy.

$$z = b + \sum_i x_i w_i \qquad y = \frac{1}{1 + e^{-z}}$$



The Derivatives of a Logistic Regression

- The derivatives of the logit, z, with respect to the inputs and the weights are very simple:

$$z = b + \sum_i x_i w_i$$

$$\frac{\partial z}{\partial w_i} = x_i$$

- The derivative of the output with respect to the logit is simple if you express it in terms of the output:

$$y = \frac{1}{1 + e^{-z}}$$

$$\frac{dy}{dz} = y\,(1 - y)$$

The Derivatives of a Logistic Regression

$$y = \frac{1}{1+e^{-z}} = (1+e^{-z})^{-1}$$

$$\frac{dy}{dz} = \frac{-1(-e^{-z})}{(1+e^{-z})^2} = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right) = y(1-y)$$

because $\dfrac{e^{-z}}{1+e^{-z}} = \dfrac{(1+e^{-z})-1}{1+e^{-z}} = \dfrac{(1+e^{-z})}{1+e^{-z}} - \dfrac{1}{1+e^{-z}} = 1-y$

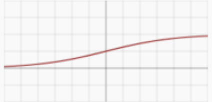
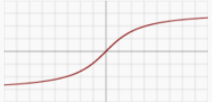
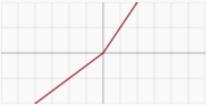
The Derivatives of a Logistic Neuron

- Using the chain rule to get the derivatives needed for learning the weights of a logistic unit
- To learn the weights we need the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i}\frac{dy}{dz} = x_i\, y\,(1-y)$$

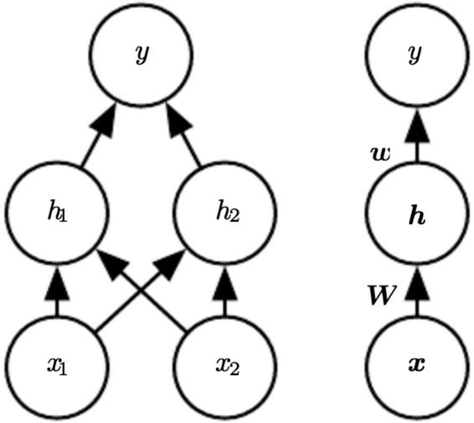
$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E}{\partial y^n} = -\sum_n \underbrace{\boxed{x_i^n}}_{\text{delta-rule}}\;\underbrace{\boxed{y^n\,(1-y^n)}}_{\text{extra term = derivative of logistic}}\;\underbrace{\boxed{(t^n-y^n)}}_{\text{delta-rule}}$$

More Activation Functions

Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)[7]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[8]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

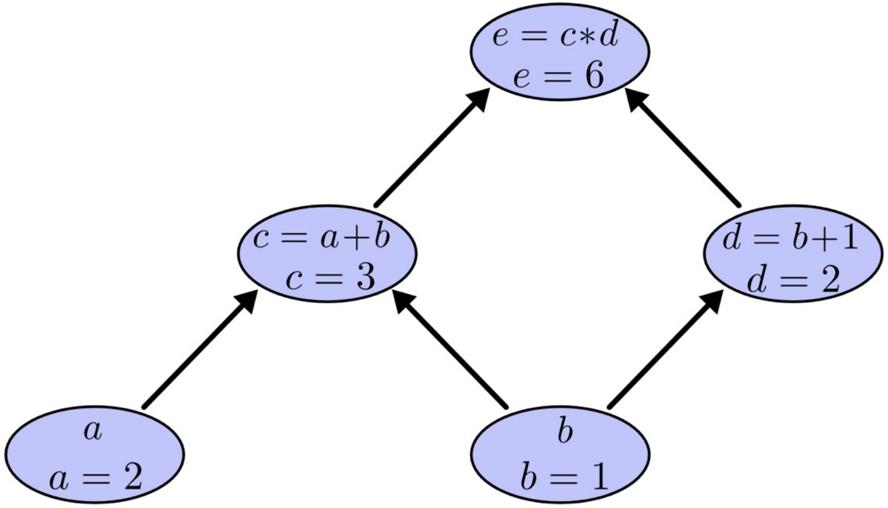
Learning with Hidden Units

- Don't know what the hidden units' errors
- Gradient Computation in DL Libraries
  - TensorFlow
    - tf.gradients(ys, xs)
  - Theano
    - theano.tensor.grad(y, x)

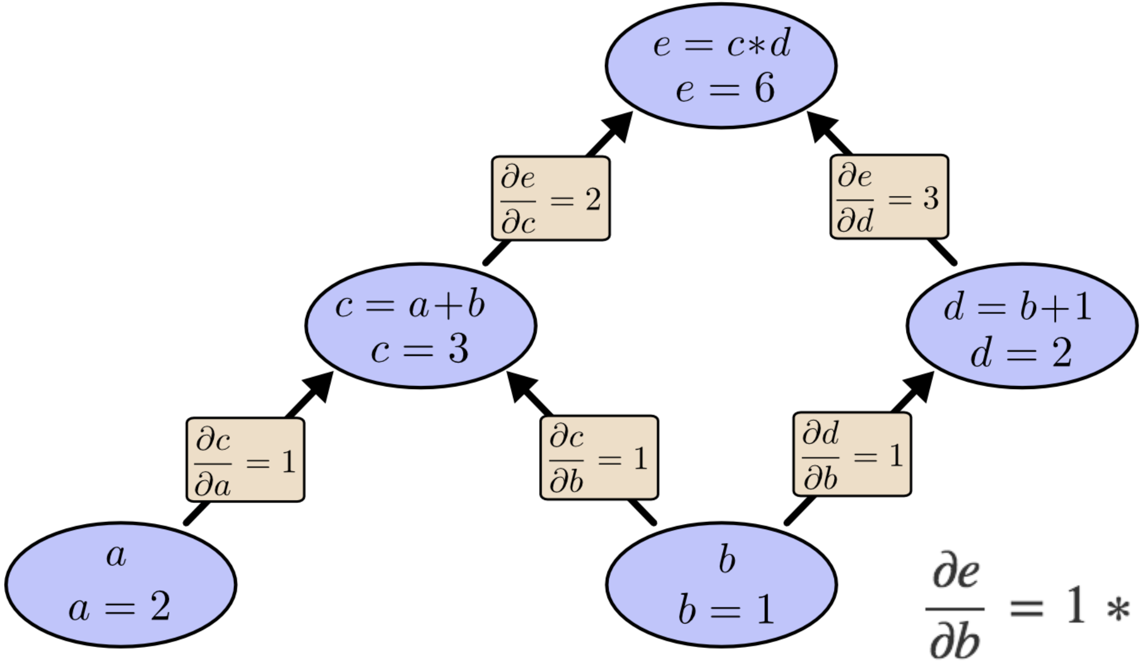


Computational Graphs

- Describing Mathematical Expressions
- For example
  - e = (a + b) * (b + 1)
    - c = a + b, d = b + 1, e = c * d
  - If a = 2, b = 1



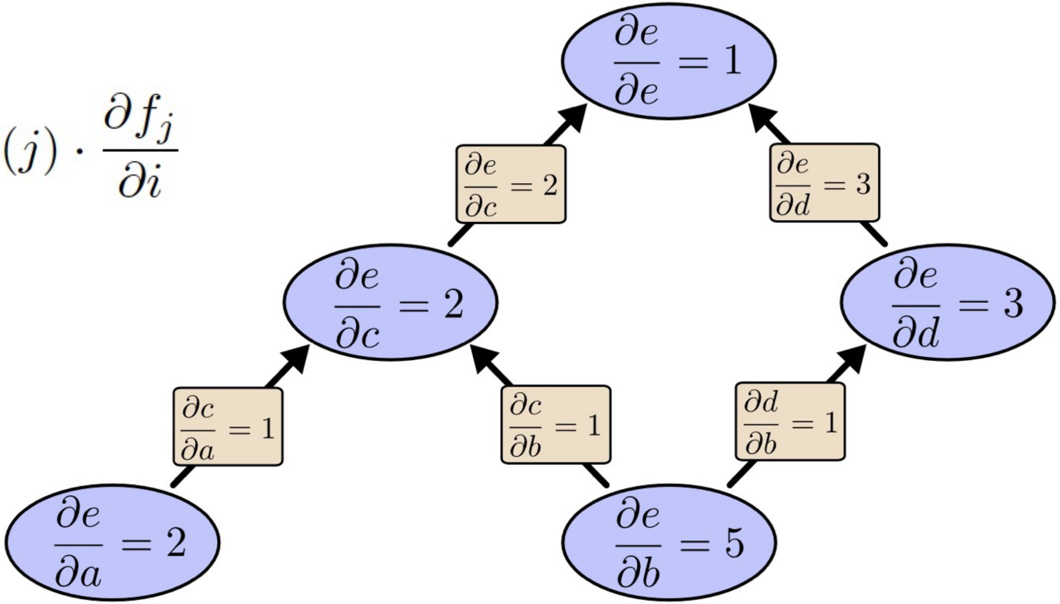
Derivatives on Computational Graphs



$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

Computational Graph Backward Pass (Backpropagation)

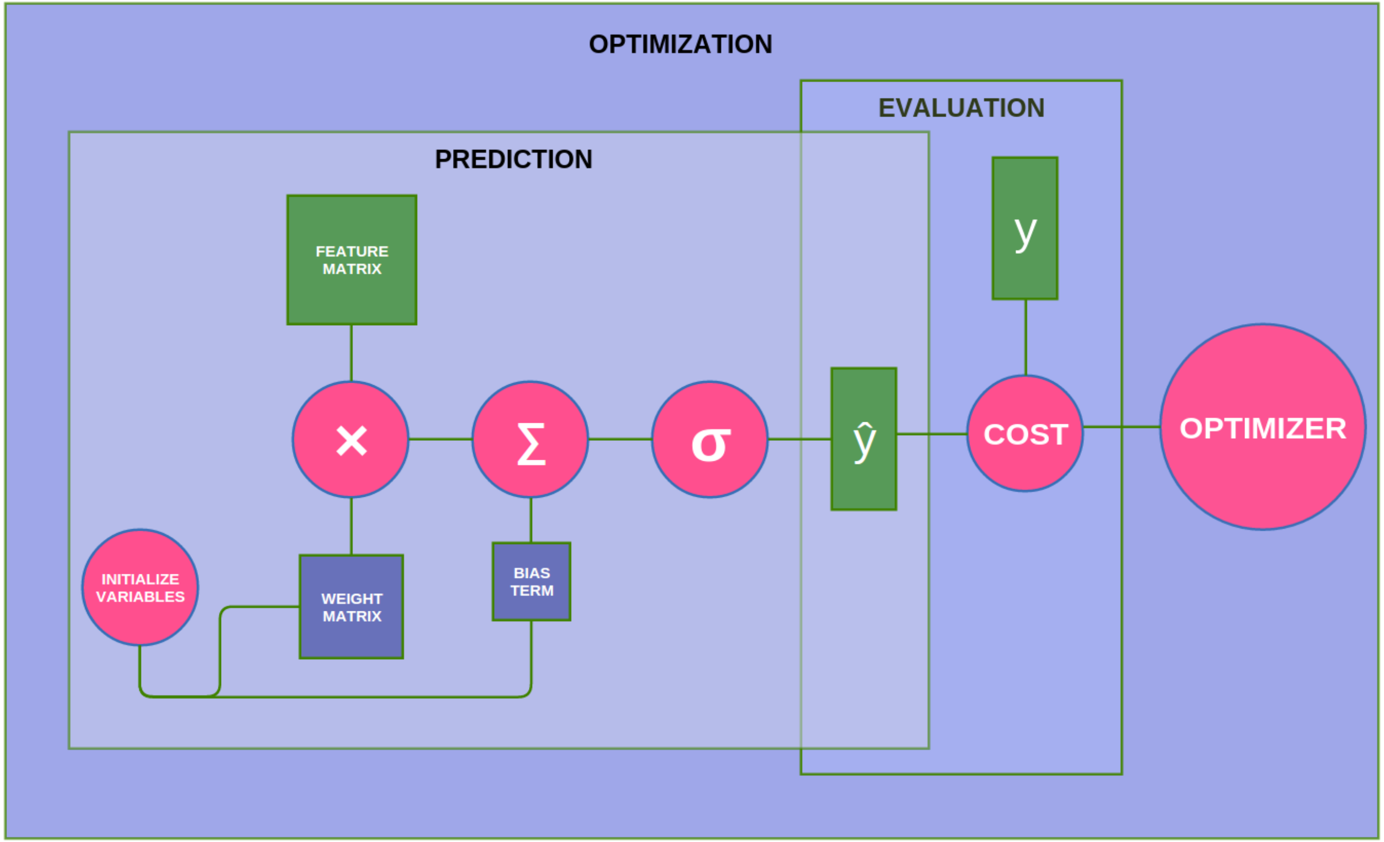
1: $d(N) \leftarrow 1$
2: **for** i = N-1 to 1 **do**
3: $\quad d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$



Quiz

- Draw the computational graph of:
  - $e = a + b + ab + b^2$
  - $a = 2, b = 1$
- Backpropagate on the graph

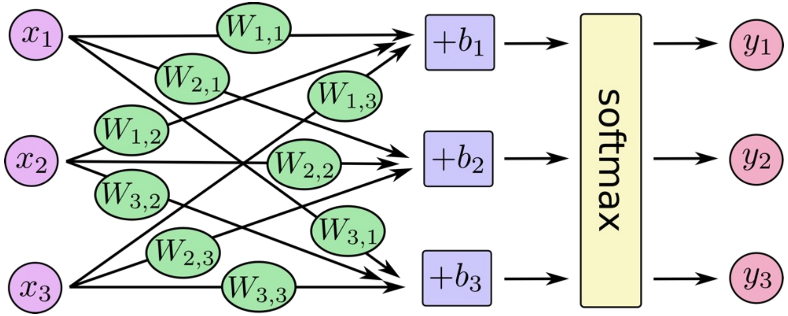
Computational Graph of Logistic Regression



MNIST Example

- 10 digit OCR (Multi-class classification)
- Input layer
  - – 28 by 28 pixel, 784 neurons
- Output layer
  - – 10 neurons, digits 0, 1, …, 9

Softmax Regression (Multinomial Logistic Regression)

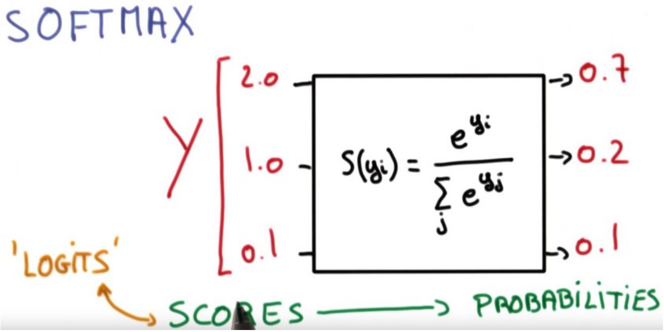


$$y = \mathrm{softmax}(Wx + b)$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$



Cross-entropy Loss/Cost Function (交叉熵损失函数)

- A better (faster) loss function

$$L_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

  - $\boldsymbol{y} = y_1, \ldots, y_K$ is a vector representing the true multinomial distribution
  - $\hat{\boldsymbol{y}} = \hat{y}_1, \ldots, \hat{y}_k$ is the network's output by softmax function
- Also referred to as *negative log likelihood*

$$L_{cross-entropy(\text{hard classification})}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_t)$$

Implementation with Keras

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

batch_size = 128
num_classes = 10
epochs = 20

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Implementation with Keras

```
model = keras.Sequential()
model.add(keras.layers.Dense(num_classes, activation='softmax', input_shape=(784,)))

model.summary()

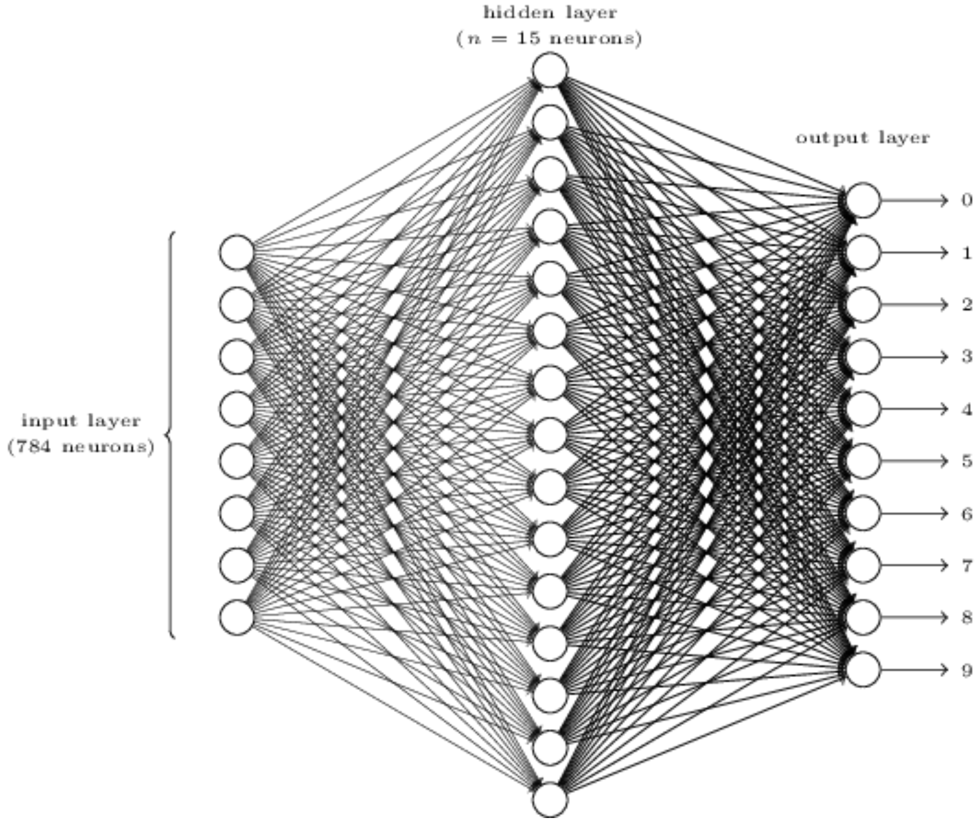
sgd = keras.optimizers.SGD(lr=0.01)

model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

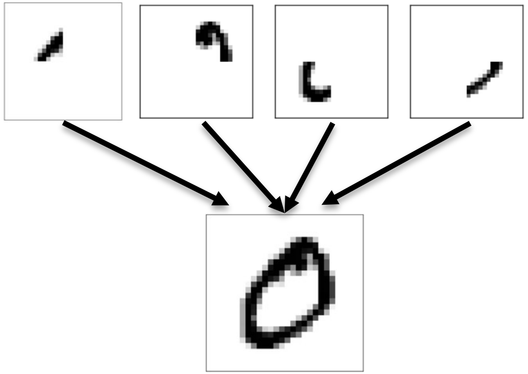
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Acc: ~91%

Multiple Layer Neural Networks



- What are those hidden neurons doing?
  - Maybe represent outlines



Multiple Layer Neural Networks

```
model = keras.Sequential()
model.add(keras.layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(keras.layers.Dense(num_classes, activation='softmax'))
```

Acc: ~**94%**

How to improve performance further?

- Optimization issues
  - How do we use the error derivatives on individual cases to discover a good set of weights?
- Generalization issues
  - How do we ensure that the learned weights work well for cases we did not see during training?

Summary

- Perceptron
- Binary Neuron
- Linear Regression
- Non-linear Neurons
  - Logistic Regression
- Multi-layer Perceptron
- Computation Graphs and Backpropagation
- Softmax Regression