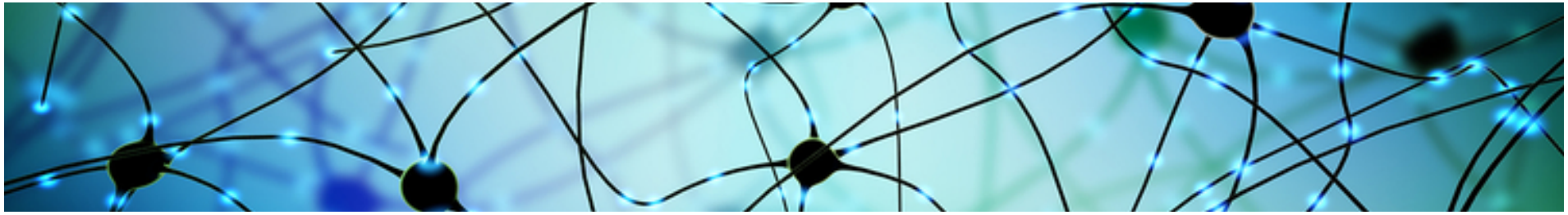
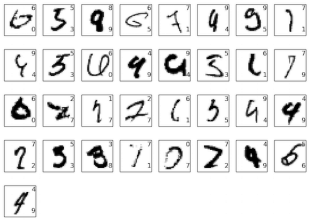


Intro to Practical Neural Networks and Deep Learning (Part 1)

CL Kim



IEEE Boston Computer Society and GBC/ACM Thursday, 15 July 2021 7:00 PM EDT

Reference book (online and free)
http://neuralnetworksanddeeplearning.com/

Neural Networks and Deep Learning

Personal: Please consider donating to author Michael Nielsen

Feedforward Neural Networks

1. Using neural nets to recognize handwritten digits

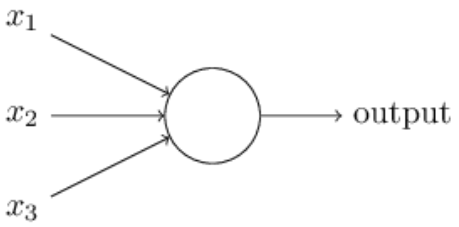
Sigmoid neuron

An artificial neuron

An early artificial neuron model is a *perceptron* — skip in interest of time

Sigmoid neuron

- Has *inputs* x_1, x_2, $\cdots$ where x_k is [0,1] inclusive
- Has *weights* w_1, w_2, $\cdots$ corresponding to the inputs
- Has one overall *bias* b
  - Notion of bias: measure of a *threshold value* needed to *fire* neuron
  - As we'll see, $b \equiv -$threshold (so b is notionally a negative number)



Sigmoid neuron

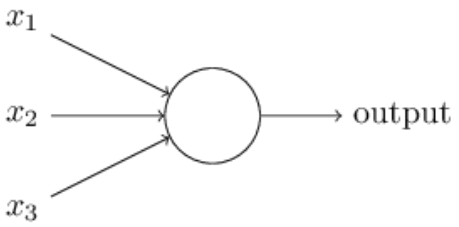
Sigmoid neuron

/2

Weighted input z

Let $z \equiv \sum_k w_k x_k + b$

Or $z \equiv \boldsymbol{w} \cdot \boldsymbol{x} + b$



Sigmoid neuron

$\boldsymbol{w}$ is weight vector, with components $w_1, w_2, \cdots$

$\boldsymbol{x}$ is input vector, with component $x_1, x_2, \cdots$

inner product $\boldsymbol{w} \cdot \boldsymbol{x} \equiv \sum_k w_k x_k$ (In matrix form, *dot* product $\boldsymbol{w}^T \cdot \boldsymbol{x} \equiv \sum_k w_k x_k$)

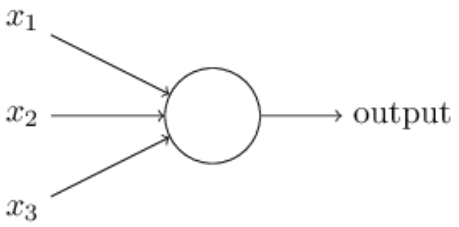
Sigmoid neuron

/3

Output

- Given by an *activation function*
  - It is a function of z or $(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
- For a sigmoid neuron, *output* activation function is
  - *sigmoid* function: $\sigma(z)$ or $\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
    - Also called *logistic* function

output $a = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$



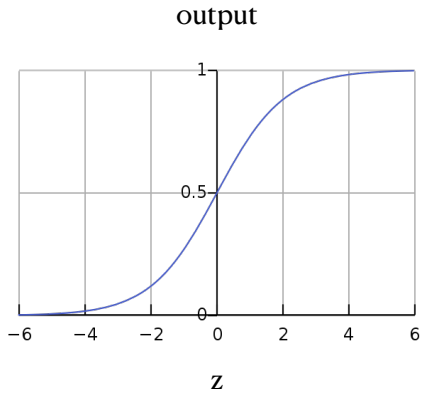
Sigmoid neuron

Sigmoid neuron

/4

Sigmoid function

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-\sum_k w_k x_k - b)}$$



- What is shape of $\sigma(z)$?

  if $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$ is large positive: $e^{-z} \approx 0$ and $\sigma(z) \approx 1$

  if $z = \boldsymbol{w} \cdot \boldsymbol{x} + b$ is large negative: $e^{-z} \to \infty$ and $\sigma(z) \approx 0$

  - It is a "smoothed out" *step function* (~ perceptron output)

    - To have argument $z > 0$, we must have $\boldsymbol{w} \cdot \boldsymbol{x} > -b$ or $\boldsymbol{w} \cdot \boldsymbol{x} >$ threshold

Sigmoid neuron

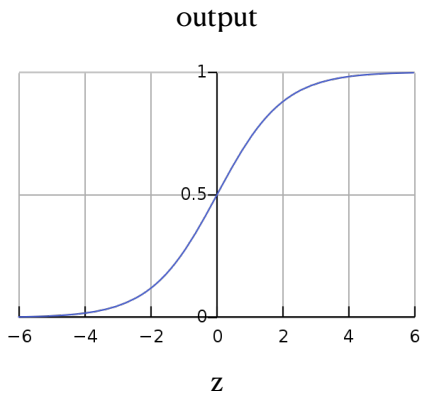
/5

Sigmoid function

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-\sum_k w_k x_k - b)}$$

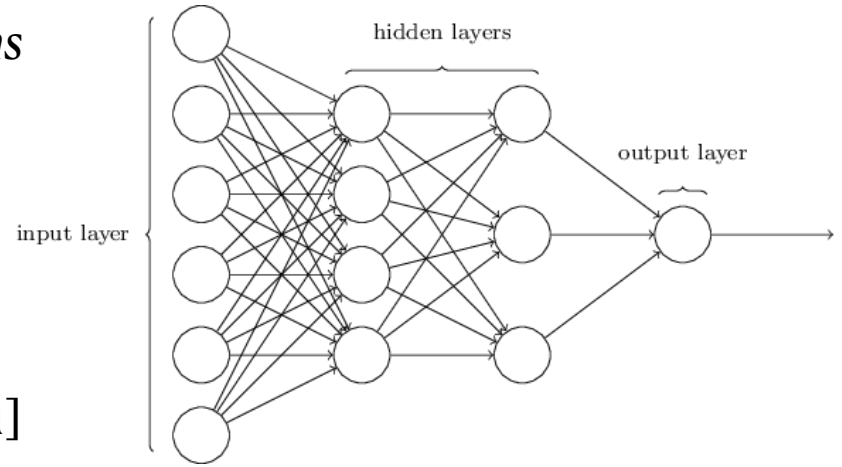
Continuity (no "jump" in value) of σ matters

- No big change Δoutput from small changes Δw_k and Δb



Architecture of Neural Networks

- *input layer* : leftmost layer, informally: *input neurons*
- *output layer* : rightmost layer, has *output neurons*
- *hidden layer* : middle layer, here 2 hidden layers
- E.g. Is handwritten image the digit “9” or not?
  - Image is 28×28 pixels, so 784 input neurons
    - Each input neuron intensity scaled to [0, 1]
  - 1 output neuron; say, if value > 0.5 -> image is “9”
- (Confusingly, sometimes called *multilayer perceptrons*)



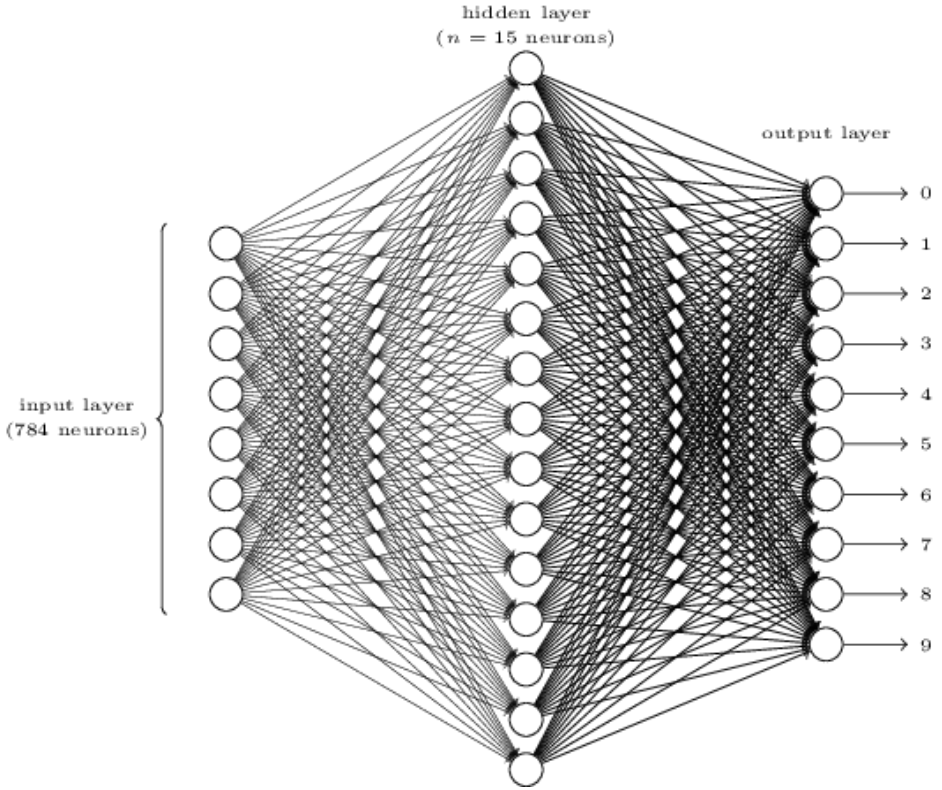
Feedforward Neural Networks

- The output from one layer is used as input to the next layer
  - No loops back, information always fed forward, never fed back
- Other models: feedback loops are possible
  - *Recurrent* neural networks
    - Idea is to have neurons which fire for some limited duration of time only
    - That firing can stimulate other neurons to fire a while later (limited duration)
    - Good for processing sequence data for predictions, e.g. Speech recognition
    - Much more complex, *not covering* here

A Simple Network

to classify a handwritten digit

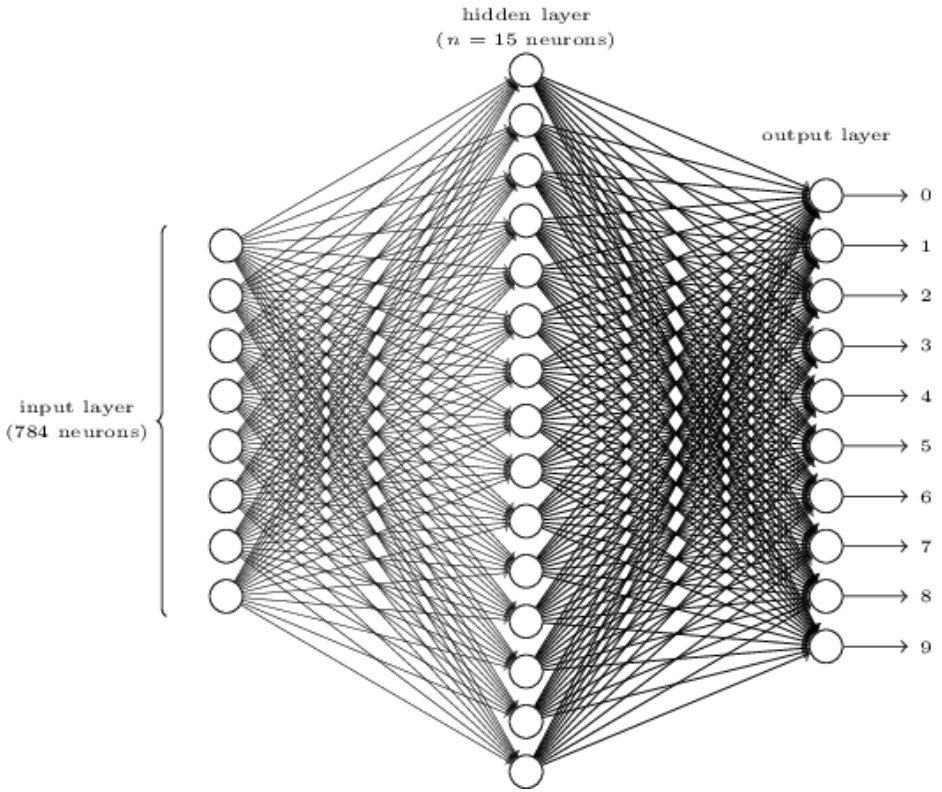
- Some ‘housekeeping’ clarification
  - Each neuron has single output
    - Multiple arrows merely to indicate that output is used as input to several others; here, to all neurons in following layer (*fully connected*)
  - Input neurons in input layer *not* really neurons with output but no input
    - Just a conventional shorthand to represent the input values x_1, x_2, ⋯



A Simple Network

to classify a handwritten digit /2

- Three-layer neural network
- Input layer: $28 \times 28 = 784$ neurons
  - Pixel is greyscale, so input value is [1.0, 0.0] inclusive
- Hidden layer: *n* number of neurons, experiment with different numbers
- Output layer: 10 neurons, #0 to #9
  - Predicted answer is neuron with highest activation value

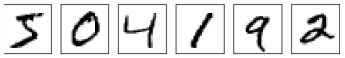


MNIST data set

Learning with gradient descent

Named for a *modified* subset of two *NIST* (National Institute of Standards and Technology) collected data sets.

Scanned handwriting samples from 250 people. Two parts.



A few images from MNIST

- 60,000 images to be used as *training data*
- 10,000 images to be used as *test data*
  - Different set 250 people

MNIST data set

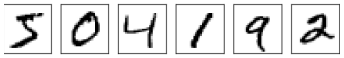
Learning with gradient descent /2

We use notation $\boldsymbol{x}$

- to denote a training input (image)

Regard each $\boldsymbol{x}$ as a $28 \times 28 = 784$-dimension (column) *vector*

- Each component of the vector represents
  - the grey value for a single pixel in the image

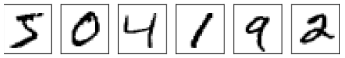


A few images from MNIST

MNIST data set

Learning with gradient descent /3

We use $\boldsymbol{y} = y(\boldsymbol{x})$ denote the corresponding *desired output*, where $\boldsymbol{y}$ is a 10-dimension *vector*



A few images from MNIST

E.g. If training image $\boldsymbol{x}$ depicts digit 7, then $\boldsymbol{y} = y(\boldsymbol{x}) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ is desired output

Cost function

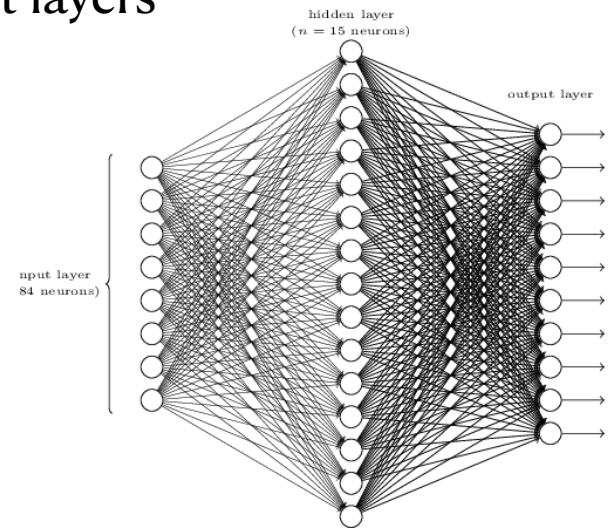
Learning with gradient descent

What we'd like is a (neural network deep learning) algorithm which lets us find

- weights and bias for each neuron in hidden and output layers
- in order that the result from output layer L: $\boldsymbol{a}^L \approx \boldsymbol{y}$
  - for every training input $\boldsymbol{x}$ in the training set

Define a *cost function* (or *loss* or *objective* function), say

$$C(w,b) = \frac{1}{n} \sum_x \frac{\|y(x) - a^L(x,w,b)\|^2}{2}$$



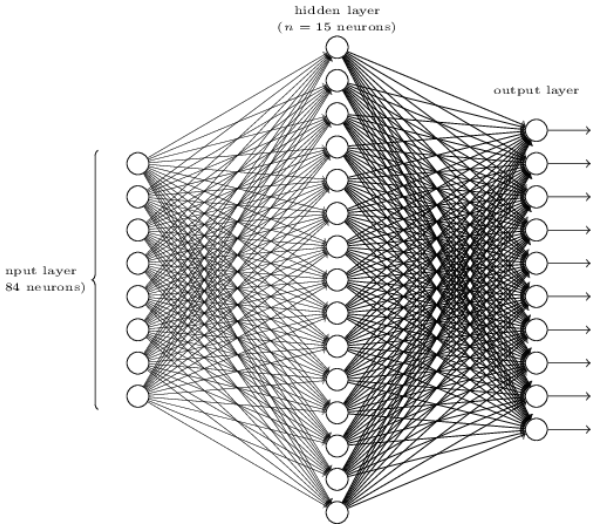
Our three-layer neural network

Cost function

/2

$$C(w,b) = \frac{1}{n}\sum_x \frac{\|y(x) - a^L(x,w,b)\|^2}{2}$$ *quadratic* or *mean squared error* MSE cost function

- $\boldsymbol{w}$ denotes all weights in the network, $\boldsymbol{b}$ all the biases
- n is total number of training inputs $\boldsymbol{x}$
- $\boldsymbol{a}^L$ is a *vector* of output layer L neuron values, when input is $\boldsymbol{x}$
  - So $\boldsymbol{a}^L$ is also 10-dimensional, similar to desired $\boldsymbol{y}$
- $\|\boldsymbol{y} - \boldsymbol{a}^L\|$ denotes usual length function for *vector* $(\boldsymbol{y} - \boldsymbol{a}^L)$
- $\div 2$ seems only for *convenience*, when differentiating numerator



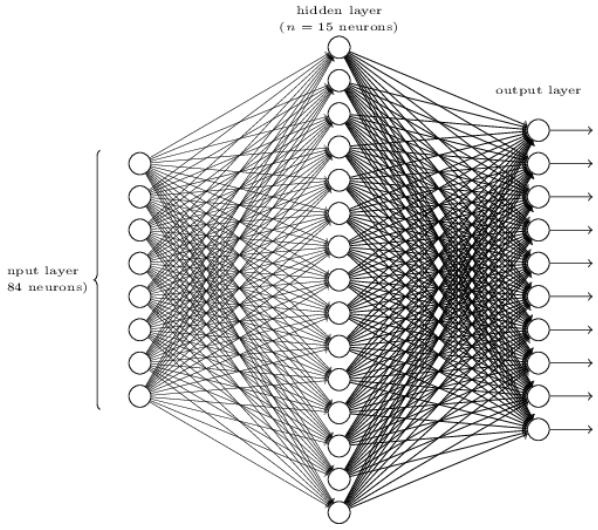
Our three-layer neural network

Cost function

/3

$$C(w,b) = \frac{1}{n}\sum_x \frac{\|y(x) - a^L(x,w,b)\|^2}{2}$$ *quadratic* or *mean squared error* MSE cost function

- Note: $C(w,b)$ is non-negative
- When: $C(w,b) \approx 0$, then $\boldsymbol{a}^L \approx \boldsymbol{y}$ for all training inputs $\boldsymbol{x}$
  - Found "good" values for each neuron's weights and bias
- Training algorithm's aim is to minimize cost $C(w,b)$
  - as a function of the weights and bias of each neuron
- Can use *gradient descent* if cost function is "smooth"



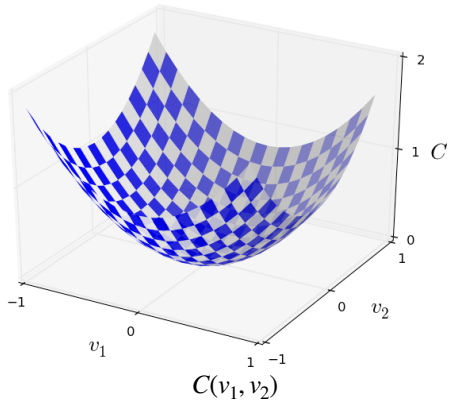
Our three-layer neural network

Minimizing cost function

Learning with gradient descent

Suppose we're trying to minimize a function $C(\boldsymbol{v})$ of *many variables* $\boldsymbol{v} = v_1, v_2, \ldots$

- C must be a real-valued (*returns a scalar*) function, but could be any function
- We've replaced $\boldsymbol{w}$ and $\boldsymbol{b}$ notations by $\boldsymbol{v}$ to represent any multiple variables, say v_1, v_2
- Can try using calculus to find minimum analytically
  - Nightmare if network has >> 10,000+ of w_k and b
- So imagine descending down slope of a "valley" shaped C
  - Move small amounts Δv_1 and Δv_2 in those directions



Minimizing cost function

/2

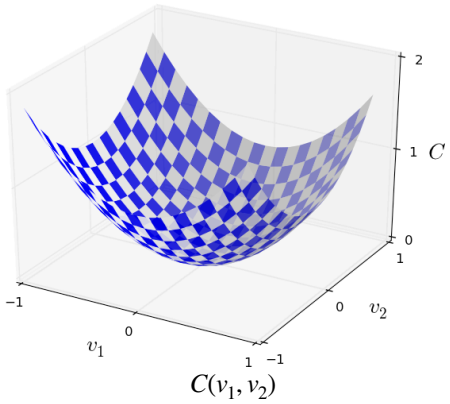
Calculus tells us C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$ Note: ΔC is a scalar

Define: gradient of C, $\nabla \boldsymbol{C} \equiv \begin{bmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{bmatrix}$, $\boldsymbol{\Delta v} \equiv \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \end{bmatrix}$

$$\Delta C \approx \nabla \boldsymbol{C} \cdot \boldsymbol{\Delta v}$$

- OK to view $\nabla \boldsymbol{C}$ as a (gradient) *vector* whose components are the partial derivatives (or view ∇ as differential operator)



Minimizing cost function

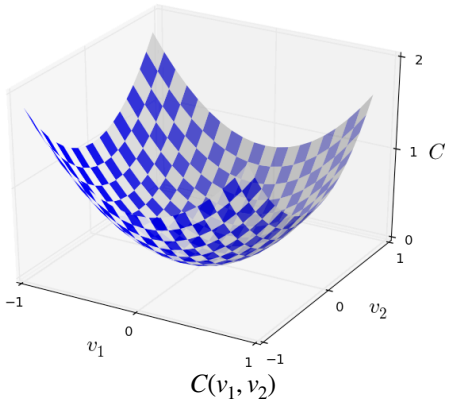
/4

$$\Delta C \approx \nabla \boldsymbol{C} \cdot \boldsymbol{\Delta v} = \boldsymbol{\Delta v} \cdot \nabla \boldsymbol{C} = \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{bmatrix}$$

- To minimize C, want $\Delta C < 0$
- Choose $\boldsymbol{\Delta v} = -\eta \nabla \boldsymbol{C}$ where η (*learning rate*) is small, positive

$$\begin{aligned} \Delta C &\approx -\eta \nabla \boldsymbol{C} \cdot \nabla \boldsymbol{C} \\ &= -\eta \, \|\nabla \boldsymbol{C}\|^2 \quad \text{both terms after minus sign} \geq 0 \\ &\leq 0 \end{aligned}$$

- So, from $\boldsymbol{v}_{now} \rightarrow \boldsymbol{v}_{new} = \boldsymbol{v}_{now} - \eta \nabla \boldsymbol{C}$ to move towards global C *minimum*



Gradient Descent

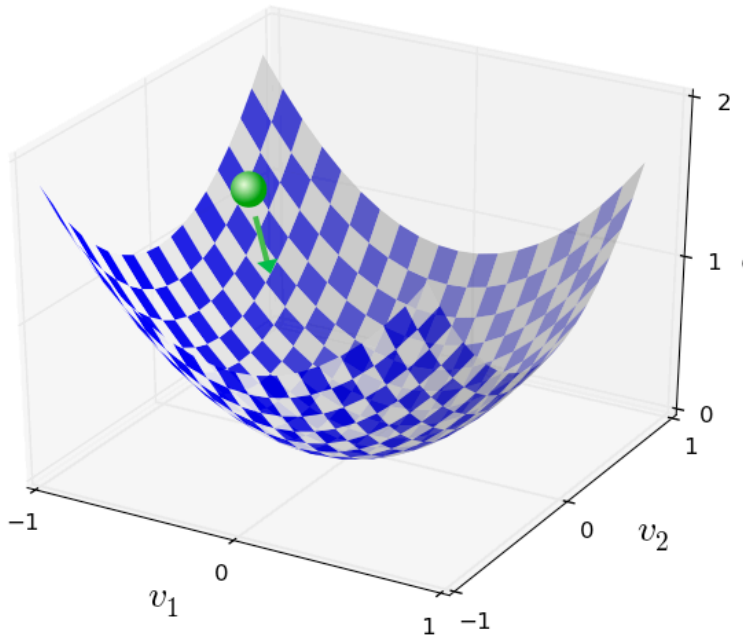
Another viewpoint of the rule

The rule is a way of taking small steps in the *direction* which does the *most* to immediately decrease C

- We want to make a move $\boldsymbol{\Delta v}$ so as to make ΔC *most negative*, equivalent to *minimizing* $\Delta C \approx \nabla \boldsymbol{C} \cdot \boldsymbol{\Delta v}$
- If we also *constrain* $\|\boldsymbol{\Delta v}\| = \epsilon$ for a small $\epsilon > 0$
- Can be proved: $\boldsymbol{\Delta v} = -\eta \nabla \boldsymbol{C}$ where $\eta = \epsilon / \|\nabla \boldsymbol{C}\|$

$$\boldsymbol{\Delta v} = \frac{-\epsilon}{\|\nabla \boldsymbol{C}\|} \nabla \boldsymbol{C} = -\epsilon \frac{\nabla \boldsymbol{C}}{\|\nabla \boldsymbol{C}\|}$$

$$= -\epsilon \text{ (unit vector in direction of } \nabla \boldsymbol{C})$$



https://math.stackexchange.com/questions/1688662/tricky-proof-of-a-result-of-michael-nielsens-book-neural-networks-and-deep-lea/1945507#1945507

Gradient Descent in a Neural Network

Learning with gradient descent

Use gradient descent to find/learn the *weights* w_k and *bias* b for each neuron

- Which would tend to minimize the cost function C
- Choose $\boldsymbol{\Delta v} = -\eta \nabla \boldsymbol{C}$ where η (*learning rate*) is small, positive

So here: $\boldsymbol{\Delta v} \equiv \begin{bmatrix} \Delta w_k \\ \Delta b \end{bmatrix} = -\eta \nabla \boldsymbol{C} \equiv -\eta \begin{bmatrix} \frac{\partial C}{\partial w_k} \\ \frac{\partial C}{\partial b} \end{bmatrix} = \begin{bmatrix} -\eta \frac{\partial C}{\partial w_k} \\ -\eta \frac{\partial C}{\partial b} \end{bmatrix}$

Gradient descent update rule: $w_k \rightarrow w_{new,k} = w_k - \eta \frac{\partial C}{\partial w_k}$ and $b \rightarrow b_{new} = b - \eta \frac{\partial C}{\partial b}$

Stochastic Gradient Descent

Learning with gradient descent

Gradient descent update rule: $w_k \rightarrow \ w_{new,k} = w_k - \eta \frac{\partial C}{\partial w_k}$ and $b \rightarrow \ b_{new} = b - \eta \frac{\partial C}{\partial b}$

One problem: $C(w,b) = \frac{1}{n} \sum_x \frac{\|y(x) - a^L(x,w,b)\|^2}{2}$

- The quadratic cost function C is an average over sum of all cost C_x for each training input x
- To compute the gradient $\nabla \boldsymbol{C}$, for example the $\frac{\partial C}{\partial w_k}$ component, we would also need to compute that $\frac{\partial C_x}{\partial w_k}$ separately for each training input x then average the sum over all x
- For large training data set, this can take long time, and learning occurs slowly

Stochastic Gradient Descent

Learning with gradient descent /2

stochastic gradient descent is used to speed up learning

- Idea is to estimate the gradient ∇C by computing each ∇C_{X_j}
  for a sample of randomly chosen (stochastic) training inputs X_j
  then averaging over this sample to get an estimate of the true gradient ∇C
  - Pick out a small(er) number m of randomly chosen training inputs $X_1, X_2, \ldots, X_m$ referred to as a *mini-batch*.

Stochastic Gradient Descent

Learning with gradient descent /3

- Average value of the ∇C_{X_j} from the mini-batch will be

  $\approx$ the average over *all* ∇C_x provided the sample size m is large enough

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \quad \text{where } \Sigma_x \text{ is over entire training data set}$$

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j} \qquad \text{where we estimate overall } \nabla C \text{ from the random mini-batch}$$

Stochastic Gradient Descent

Learning with gradient descent /4

Reminder: $\boldsymbol{v} \rightarrow \ \boldsymbol{v}_{new} = \boldsymbol{v} - \eta \nabla \boldsymbol{C}$

Stochastic gradient descent update rule

- Pick out a randomly chosen mini-batch of training inputs X_j and train with those

$$w_k \rightarrow \ w_{new,k} = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \quad \text{and} \quad b \rightarrow \ b_{new} = b - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b}$$

  where the sums are over all the training examples in current mini-batch

- Then pick out another random mini-batch, and train (do w_k and b updates) with those
- Until we've exhausted the training data inputs, which completes an *epoch* of training
- At which point we start over with a new training epoch

Stochastic Gradient Descent

Learning with gradient descent /6

E.g. if we have a training set of size $n = 60{,}000$ as in MNIST and choose a mini-batch size of (say) $m = 10$

- Get a factor of 6,000 speedup in estimating the gradient (in each update pass)
- The estimate won't be perfect because of statistical fluctuations
- But it doesn't need to be perfect
- All we care about is moving in a general direction that will help decrease cost C
- In practice, stochastic gradient descent is a commonly used and powerful technique for learning in neural networks

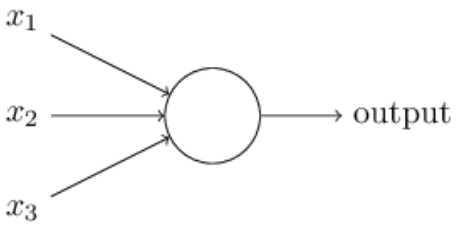
Sigmoid neuron

/2

Weighted input z

Let $z \equiv \sum_k w_k x_k + b$

Or $z \equiv \boldsymbol{w} \cdot \boldsymbol{x} + b$ <==



Sigmoid neuron

$\boldsymbol{w}$ is weight vector, with components $w_1, w_2, \cdots$

$\boldsymbol{x}$ is input vector, with component $x_1, x_2, \cdots$

inner product $\boldsymbol{w} \cdot \boldsymbol{x} \equiv \sum_k w_k x_k$ (In matrix form, *dot* product $\boldsymbol{w}^T \cdot \boldsymbol{x} \equiv \sum_k w_k x_k$)

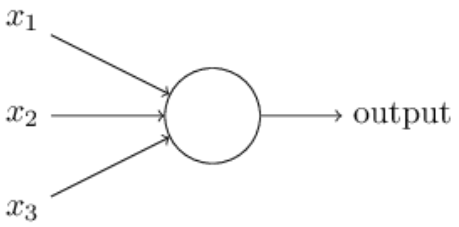
Sigmoid neuron

/3

Output

- Given by an *activation function*
  - It is a function of z or $(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
- For a sigmoid neuron, *output* activation function is
  - *sigmoid* function: $\sigma(z)$ or $\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$
    - Also called *logistic* function

output $a = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ <==



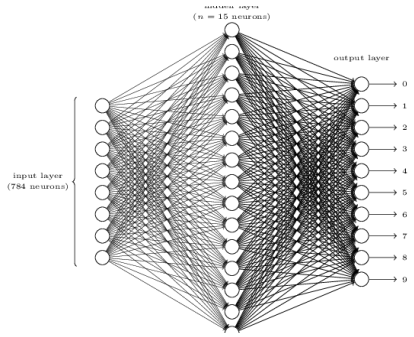
Sigmoid neuron

Implementing our network to classify digits

/4

- Reminder: $w^{l=3rd\ layer}$ = `weights[1]` a 10 by 30 matrix
  - As we'll see, it stores the weights associated with the neurons in the third layer
  - We can write the *output* **activation** of third layer: $a^{l=3} = \sigma\left(w^{l=3}\, a^{l=2} + b^{l=3}\right)$

$$a^{l=3} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,k}^{l=3} & \cdots & \cdots & w_{1\,30}^{l=3} \\ w_{j\,1}^{l=3} & w_{j\,k}^{l=3} & \cdots & \cdots & w_{j\,30}^{l=3} \\ \vdots & & & & \\ w_{10\,1}^{l=3} & w_{10\,k}^{l=3} & \cdots & \cdots & w_{10\,30}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_k^{l=2} \\ \vdots \\ \vdots \\ a_{30}^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \\ b_j^{l=3} \\ \vdots \\ b_{10}^{l=3} \end{bmatrix}\right)$$



Our three-layer neural network

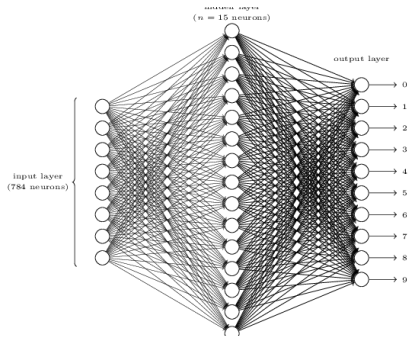
  - $w_{j\,k}^{l=3} = weights[1]_{j\,k}$ is one of the weights of the j^{th} neuron in the *third* layer, in particular, the weight for the output from the k^{th} neuron in the *second* layer

Implementing our network to classify digits

/5

$$a^{l=3} = \sigma\left(w^{l=3}\, a^{l=2} + b^{l=3}\right)$$

$$= \sigma\left(\begin{bmatrix} w^{l=3}_{1\,1} & w^{l=3}_{1\,k} & \cdots & \cdots & w^{l=3}_{1\,30} \\ w^{l=3}_{j\,1} & w^{l=3}_{j\,k} & \cdots & \cdots & w^{l=3}_{j\,30} \\ \vdots & & & & \\ w^{l=3}_{10\,1} & w^{l=3}_{10\,k} & \cdots & \cdots & w^{l=3}_{10\,30} \end{bmatrix} \begin{bmatrix} a^{l=2}_{1} \\ a^{l=2}_{k} \\ \vdots \\ \vdots \\ a^{l=2}_{30} \end{bmatrix} + \begin{bmatrix} b^{l=3}_{1} \\ b^{l=3}_{j} \\ \vdots \\ b^{l=3}_{10} \end{bmatrix}\right)$$



Our three-layer neural network

- Denote: $b^{l=3}$ = `biases[1]` a <u>10 by 1</u> matrix
  - $b^{l=3}_{j} = biases[1]_{j}$ is the bias for the j^{th} neuron in the third layer

2. How the backpropagation algorithm works

Gradient Descent in a Neural Network

Learning with gradient descent

Use gradient descent to find/learn the *weights* w_k and *bias* b for each neuron

- Which would tend to minimize the cost function C
- Choose $\boldsymbol{\Delta v} = -\eta \nabla \boldsymbol{C}$ where η (*learning rate*) is small, positive

So here: $\boldsymbol{\Delta v} \equiv \begin{bmatrix} \Delta w_k \\ \Delta b \end{bmatrix} = -\eta \nabla \boldsymbol{C} \equiv -\eta \begin{bmatrix} \frac{\partial C}{\partial w_k} \\ \frac{\partial C}{\partial b} \end{bmatrix} = \begin{bmatrix} -\eta \frac{\partial C}{\partial w_k} \\ -\eta \frac{\partial C}{\partial b} \end{bmatrix}$

Gradient descent update rule: $w_k \rightarrow w_{new,k} = w_k - \eta \frac{\partial C}{\partial w_k}$ and $b \rightarrow b_{new} = b - \eta \frac{\partial C}{\partial b}$

Backpropagation Algorithm

How to compute gradient of the cost function

- Introduced in 70's, came into own in famous 1986 *Rumelhart, Hinton, & Williams* paper
- At its heart: gives expression for the partial derivatives of the cost function C_x

  $\frac{\partial C_x}{\partial w_k}$ and $\frac{\partial C_x}{\partial b}$ for one training example x

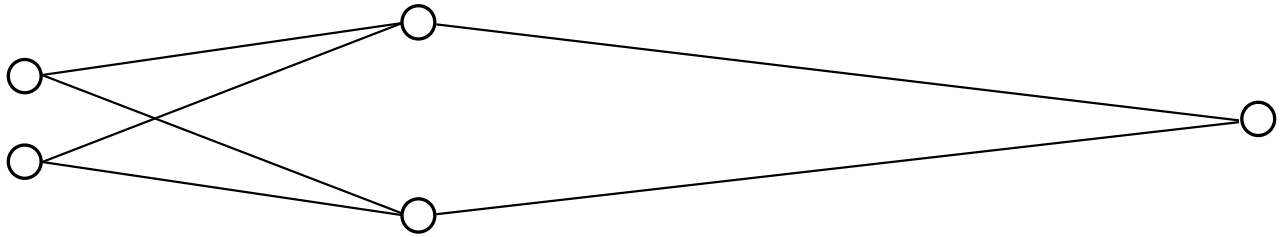
  Or, with notation to specify the j^{th} neuron in the l^{th} layer

  $\frac{\partial C_x}{\partial w^l_{jk}}$ and $\frac{\partial C_x}{\partial b^l_j}$ for one training example x
- Fast algorithm to compute those expressions

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : Layer 1 Layer 2 Output : Layer 3

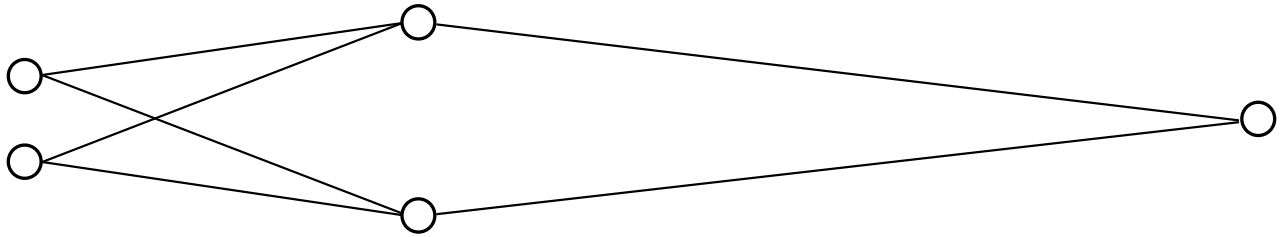


$$\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \begin{bmatrix} a_1^{l=3} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \end{bmatrix} \right)$$

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : **Layer 1** Layer 2 Output : Layer 3

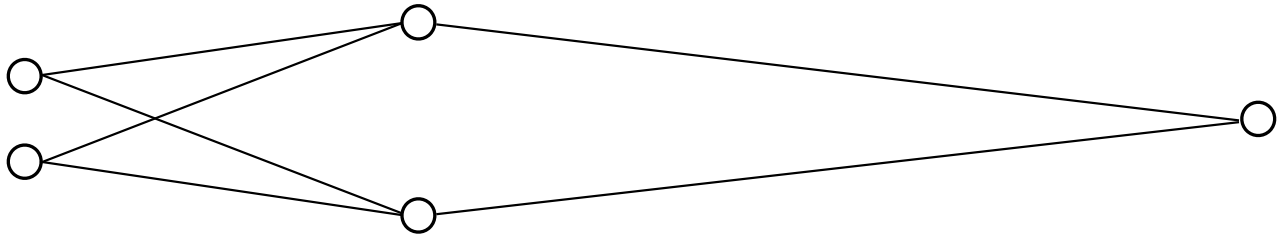


$$\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} = \overset{x}{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \begin{bmatrix} a_1^{l=3} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \end{bmatrix} \right)$$

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : **Layer 1** Layer 2 Output : Layer 3

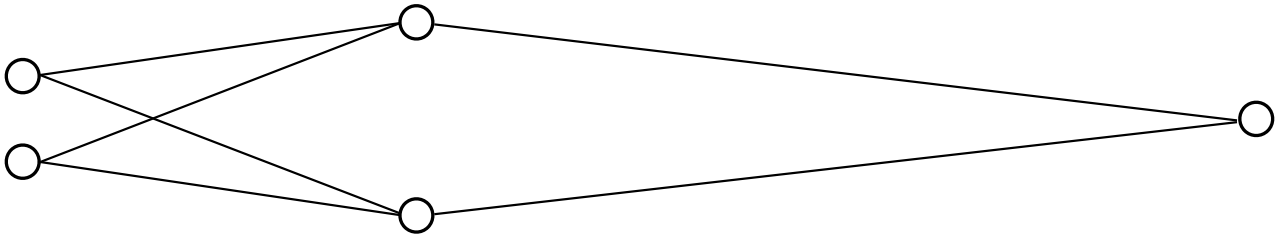


$$\underset{a}{\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix}} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \begin{bmatrix} a_1^{l=3} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \end{bmatrix} \right)$$

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : Layer 1 **Layer 2** Output : Layer 3

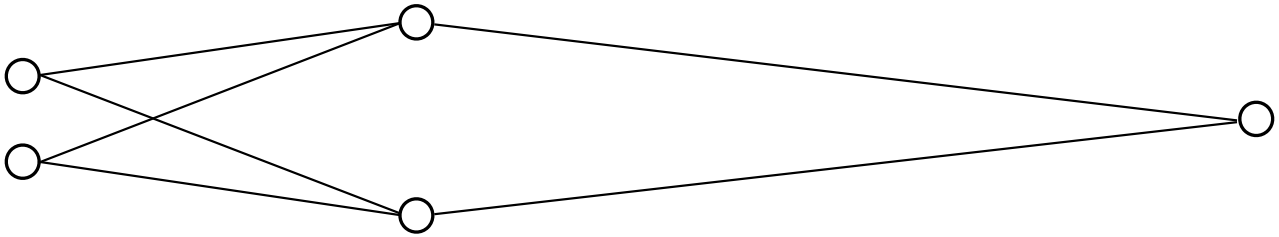


$$\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\underbrace{\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix}}_{z} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \begin{bmatrix} a_1^{l=3} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \end{bmatrix} \right)$$

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : Layer 1 **Layer 2** Output : Layer 3



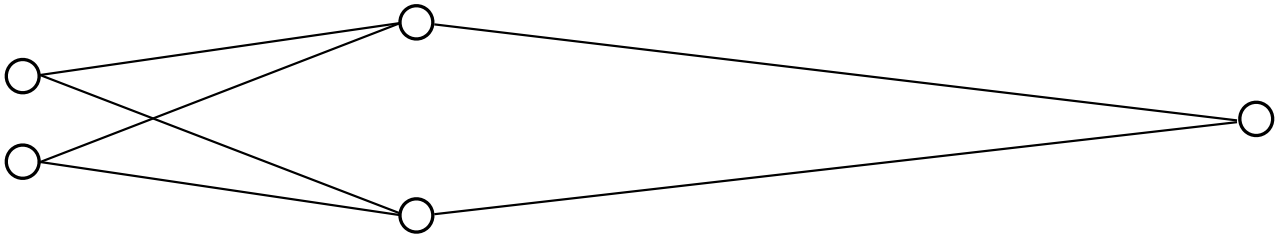
$$\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \begin{bmatrix} a_1^{l=3} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} + \begin{bmatrix} b_1^{l=3} \end{bmatrix} \right)$$

a

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : Layer 1 Layer 2 Output : **Layer 3**

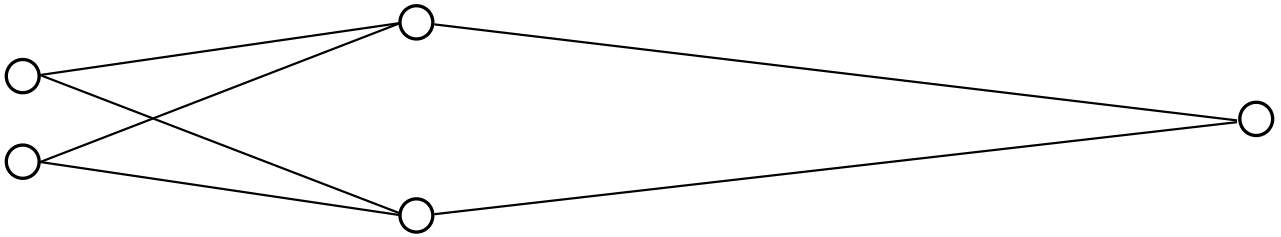


$$\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \left[a_1^{l=3}\right] = \sigma\left(\overbrace{\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix}}^{z} + \left[b_1^{l=3}\right] \right)$$

Backpropagation Algorithm Overview

Feedforward phase - using the activation function

Input : Layer 1 Layer 2 Output : **Layer 3**



$$\begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow[\text{to layer } 2]{\text{layer } 1} \begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=2} & w_{1\,2}^{l=2} \\ w_{2\,1}^{l=2} & w_{2\,2}^{l=2} \end{bmatrix} \begin{bmatrix} a_1^{l=1} \\ a_2^{l=1} \end{bmatrix} + \begin{bmatrix} b_1^{l=2} \\ b_2^{l=2} \end{bmatrix} \right) \xrightarrow[\text{to layer } 3]{\text{layer } 2} \begin{bmatrix} a_1^{l=3} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1\,1}^{l=3} & w_{1\,2}^{l=3} \end{bmatrix} \overset{a}{\begin{bmatrix} a_1^{l=2} \\ a_2^{l=2} \end{bmatrix}} + \begin{bmatrix} b_1^{l=3} \end{bmatrix} \right)$$

Backpropagation Algorithm Overview

Backpropagate the Error phase - using 4 fundamental eqns of backpropagation

Define the *error* δ_j^l of neuron j in layer l : $\delta_j^l \equiv \dfrac{\partial C}{\partial z_j^l}$ because can get $\dfrac{\partial C}{\partial w_{jk}^l}$ and $\dfrac{\partial C}{\partial b_j^l}$ fast

1. $\delta_j^L = \dfrac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$ If using quadratic cost function $\delta_j^L = (a_j^L - y_j)\,\sigma'(z_j^L)$
2. $\delta_j^l = \displaystyle\sum_k w_{kj}^{l+1}\delta_k^{l+1}\;\sigma'(z_j^l) = ((W_{:,j}^{l+1})^T\delta^{l+1})\;\sigma'(z_j^l)$
3. $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$
4. $\dfrac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$

Backpropagation Algorithm Overview

Backpropagate the Error phase - using 4 fundamental eqns of backpropagation

Define the *error* δ_j^l of neuron j in layer l : $\delta_j^l \equiv \dfrac{\partial C}{\partial z_j^l}$ because can get $\dfrac{\partial C}{\partial w_{jk}^l}$ and $\dfrac{\partial C}{\partial b_j^l}$ fast

1. $\delta_j^L = \dfrac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$ If using quadratic cost function $\delta_j^L = (a_j^L - y_j)\,\sigma'(z_j^L)$ <== e.g. $L = 3$
2. $\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\,\sigma'(z_j^l) = ((W_{:,j}^{l+1})^T\delta^{l+1})\,\sigma'(z_j^l)$
3. $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$
4. $\dfrac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$

Backpropagation Algorithm Overview

Backpropagate the Error phase - using 4 fundamental eqns of backpropagation

Define the *error* δ_j^l of neuron j in layer l : $\delta_j^l \equiv \dfrac{\partial C}{\partial z_j^l}$ because can get $\dfrac{\partial C}{\partial w_{jk}^l}$ and $\dfrac{\partial C}{\partial b_j^l}$ fast

1. $\delta_j^L = \dfrac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$ If using quadratic cost function $\delta_j^L = (a_j^L - y_j)\,\sigma'(z_j^L)$

2. $\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\,\sigma'(z_j^l) = ((W_{:,j}^{l+1})^T\delta^{l+1})\,\sigma'(z_j^l)$

3. $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$ <== $l = 3$

4. $\dfrac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ <== $l = 3$

Backpropagation Algorithm Overview

Backpropagate the Error phase - using 4 fundamental eqns of backpropagation

Define the *error* δ_j^l of neuron j in layer l : $\delta_j^l \equiv \dfrac{\partial C}{\partial z_j^l}$ because can get $\dfrac{\partial C}{\partial w_{jk}^l}$ and $\dfrac{\partial C}{\partial b_j^l}$ fast

1. $\delta_j^L = \dfrac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$ If using quadratic cost function $\delta_j^L = (a_j^L - y_j)\,\sigma'(z_j^L)$
2. $\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\,\sigma'(z_j^l) = ((W_{:,j}^{l+1})^T\delta^{l+1})\,\sigma'(z_j^l)$ <== $l = 2$
3. $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$
4. $\dfrac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$

Backpropagation Algorithm Overview

Backpropagate the Error phase - using 4 fundamental eqns of backpropagation

Define the *error* δ_j^l of neuron j in layer l : $\delta_j^l \equiv \dfrac{\partial C}{\partial z_j^l}$ because can get $\dfrac{\partial C}{\partial w_{jk}^l}$ and $\dfrac{\partial C}{\partial b_j^l}$ fast

1. $\delta_j^L = \dfrac{\partial C}{\partial a_j^L}\,\sigma'(z_j^L)$ If using quadratic cost function $\delta_j^L = (a_j^L - y_j)\,\sigma'(z_j^L)$

2. $\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\,\sigma'(z_j^l) = ((W_{:,j}^{l+1})^T\delta^{l+1})\,\sigma'(z_j^l)$

3. $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$ <== $l = 2$

4. $\dfrac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ <== $l = 2$

4 fundamental eqns of backprop

Some insights

- Consider a_k^{l-1} in BP4 when it is small
  - The gradient term $\partial C/\partial w$ also tends to be small
- Consider $\sigma'(z_j^L)$ in BP1 or $\sigma'(z_j^l)$ in BP2 when $\sigma'(z_j) \approx 0$
  - which is when the σ function is flat or *saturated* at $\sigma(z_j) \approx 0$ or ≈ 1 (low or high activation)
    - $\partial C/\partial b$ and/or $\partial C/\partial w$ also tend to be small
    - A bias or a weight of a neuron will tend to *learn slowly* if the neuron is near saturation
- *Recap*: the weight of a neuron tends to learn slowly
  - if either its input neuron has low activation a_k^{l-1}
  - or the neuron's output has saturated

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

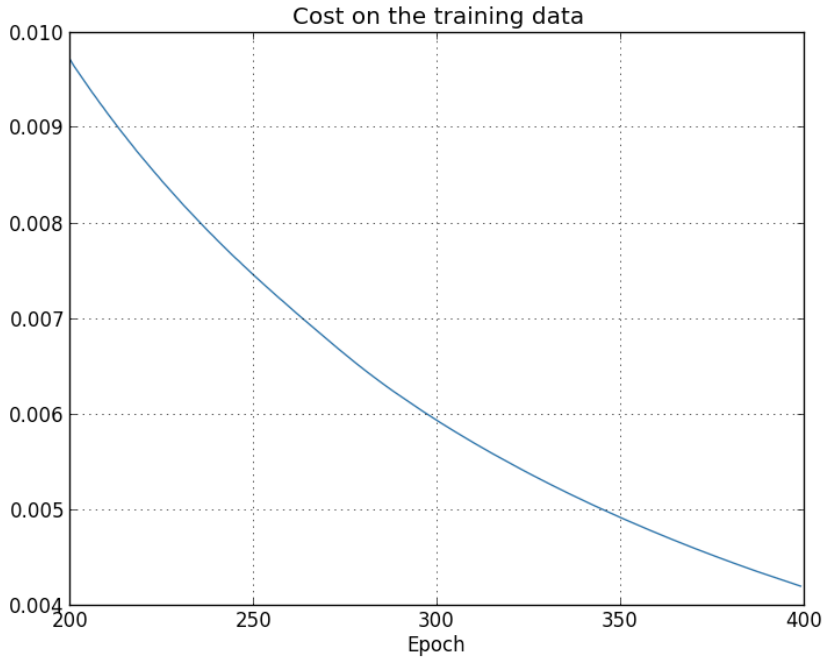
$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{BP3}$$

$$\frac{\partial C}{\partial w_{j\,k}^l} = a_k^{l-1} \delta_j^l \tag{BP4}$$

3. Improving the way neural networks learn

Overfitting

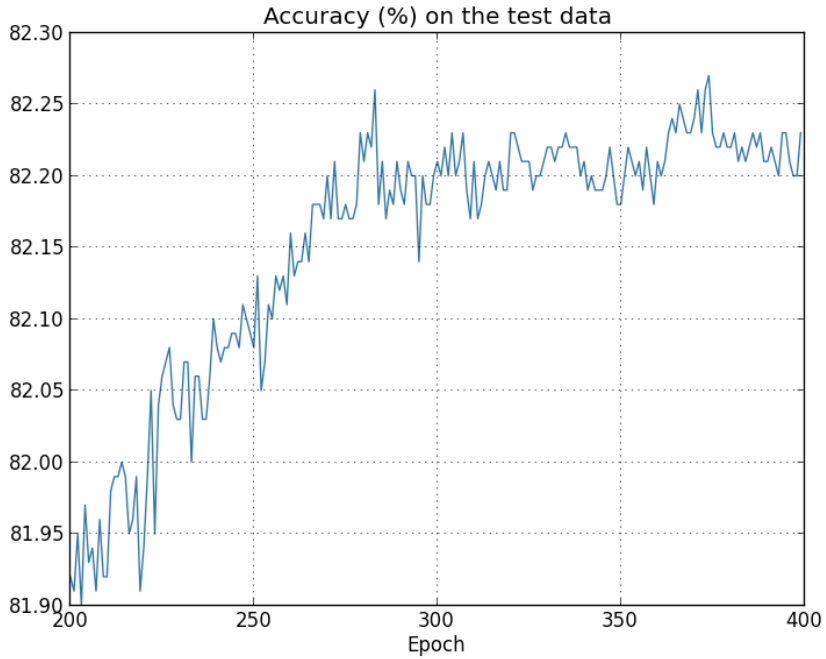
- Models with large enough number of (independent) parameters can describe almost any data set of a given size
  - Will work well for the existing data
  - But will fail to generalize to new input it hasn't been exposed to before
- Here's graph of *training* cost, with our 30 hidden neuron network, with its (28x28 x 30) + 30 + (30x10) + 10 = 23,860 parameters; but using just first 1,000 training images
  - Seems decreasing cost, up to epoch 400



Overfitting

/2

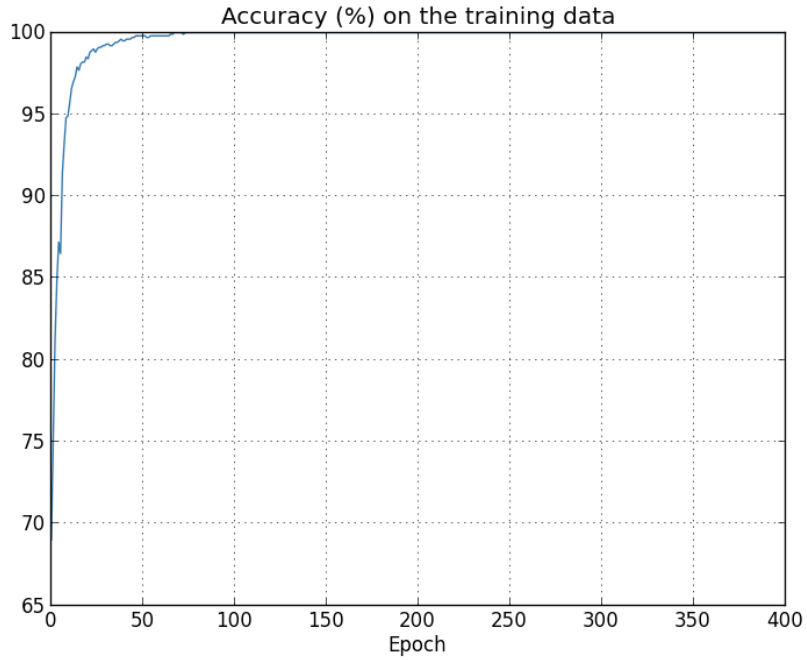
- But classification accuracy on the *test data* set gradually slows down and pretty much stops improving around epoch 280, at around 82%
- What our network learns after epoch 280 no longer generalizes to the test data; it's not useful learning
- We say the network is *overfitting* or *overtraining* beyond epoch 280



Overfitting

/3

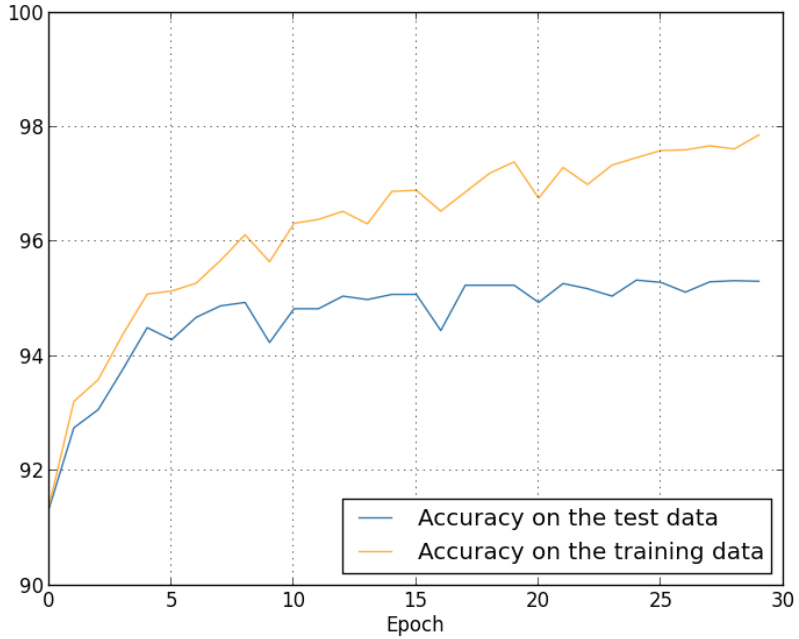
- Another sign of overfitting may be seen in the classification accuracy on the *training data*
  - Accuracy rises to 100%; our network correctly classifies all 1000 training images; but accuracy tops out around 82% on test data
- Our network really is learning about peculiarities of the training set, and not about recognizing digits in general



Increase training data

To reduce Overfitting

- We were training with *1,000* training images
- Let's use full training set of 50,000 for 30 epochs (here, comparing with test data, not validation data, so result more comparable)
  - The classification accuracy gap is still there peaking at 97.86% - 95.33% = 2.53%, but is much smaller than the ~18%; overfitting is still going on but greatly reduced
- Increasing *size* of the *training data* is one of best ways of reducing overfitting
  - But can be expensive or difficult to acquire
  - So not always a practical option



L2 or Weight decay Regularization

Regularization techniques to reduce Overfitting

- Idea of *L2 regularization* or *weight decay regularization* is to add extra term to the cost function, a term called the *regularization term*

Can write regularized cost function $C = C_O + \frac{\lambda}{2n} \sum_w w^2$ where C_O is original c.f.

- Second term is sum of the squares of all the weights in the network (but doesn't include the biases, will touch on why later)
  - Scaled by factor $\lambda/2n$
    - $\lambda > 0$ is the *regularization parameter*; n is, as usual, size of our training set

L2 or Weight decay Regularization

Regularization techniques to reduce Overfitting /9

Some closing words on (L2) regularization

- It's an empirical fact that regularized neural networks usually generalize better than unregularized networks
  - But we don't have an entirely satisfactory systematic understanding of what's going on, merely incomplete heuristics and rules of thumb

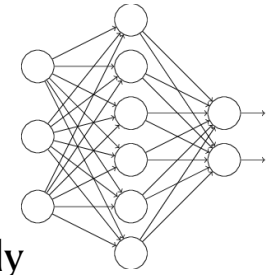
Regularization term doesn't include biases $C = C_0 + \frac{\lambda}{2n} \sum_w w^2$

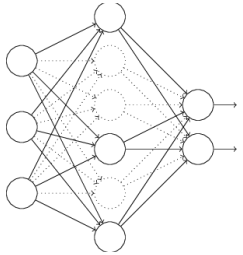
- Empirically, regularizing biases often doesn't change the results very much
  - We don't need to worry about large biases enabling our network to learn the noise in our training data, because a large bias doesn't make a neuron sensitive to its inputs as weights do
  - Large biases make it easier for neurons to saturate, which is sometimes desirable

Dropout

Other regularization techniques to reduce Overfitting

- In dropout we modify the network itself
- Suppose we're trying to train a network with training inputs
  - Forward propagate, backpropagate to do gradient descent, over mini-batches
- With dropout, first choose a *random* half of the hidden neurons to delete temporarily
  Leave input and output neurons untouched
  - Forward propagate and backpropagate through such a modified network
  - After a mini-batch of examples, update appropriate weights, biases
  - Restore dropout neurons; repeat process, deleting a new random subset
  - When we actually run the full network, twice as many hidden neurons are active
    - To compensate, we halve the weights outgoing from the hidden neurons





Artificially expanding the training data

Other regularization techniques to reduce Overfitting

- Suppose we take an MNIST training image of a “5” and rotate it
  - At the pixel level, it’s quite different to any image in MNIST

- Powerful and widely used idea to expand MNIST training data
  - Rotating, Translating, Skewing
  - “Elastic distortions” intended to emulate the random oscillations in hand muscles
    - Increased accuracy up to 99.3%
- General principle is to expand training data
  - Apply operations to reflect real-world variation
    - Speech: add background noise, speed up or slow down
- Sometimes instead of adding noise, may be more efficient to clean up the input by first applying noise reduction

Running demo code

- Why using Docker container
  - In order to avoid needing to give out separate instructions on how to install Python and needed packages to run the book's code on different platforms/flavors (such as Mac, Windows, Linux), it seemed easier to just give one set of instructions on how to create a docker container and how to run the demo code in it.
  - Hopefully, Docker is sufficiently ubiquitous nowadays so that installing and running docker on different platforms should be well documented.
- Book's code is in my forked GitHub repository

  https://github.com/clkim/DeepLearningPython35
- Instructions are in the README

Neural Networks and Deep Learning — A Practical Introduction

Live Webinar class: Saturday, October 16, 2021, 9:00 AM - 12:30 PM EDT

Organizer: GBCACM Instructor: CL Kim

- Simple (Python) Network to classify a handwritten digit
- Learning with Gradient Descent
- Backpropagation algorithm
- Improving neural networks
- Overfitting and Regularization

- *Improving how neural networks learn*
  - Cross-entropy cost function
  - Softmax activation function and log-likelihood cost function
  - Rectified Linear Unit
- *Overfitting and Regularization*
  - L2 regularization
  - Dropout
  - Artificially expanding data set