CS60010: Deep Learning

Sudeshna Sarkar

Spring 2019

16 Jan 2019

Simpler Representation

1. lower dimensional representations
   - compress as much information about x as possible in a smaller representation
2. sparse representations
   - embed the dataset into a representation whose entries are mostly zeroes for most inputs
3. independent representations
   - Disentangle the sources of variation underlying the data distribution such that the dimensions of the representation are statistically independent

- The notion of representation is one of the central themes of deep learning

Feedforward Networks

FFN

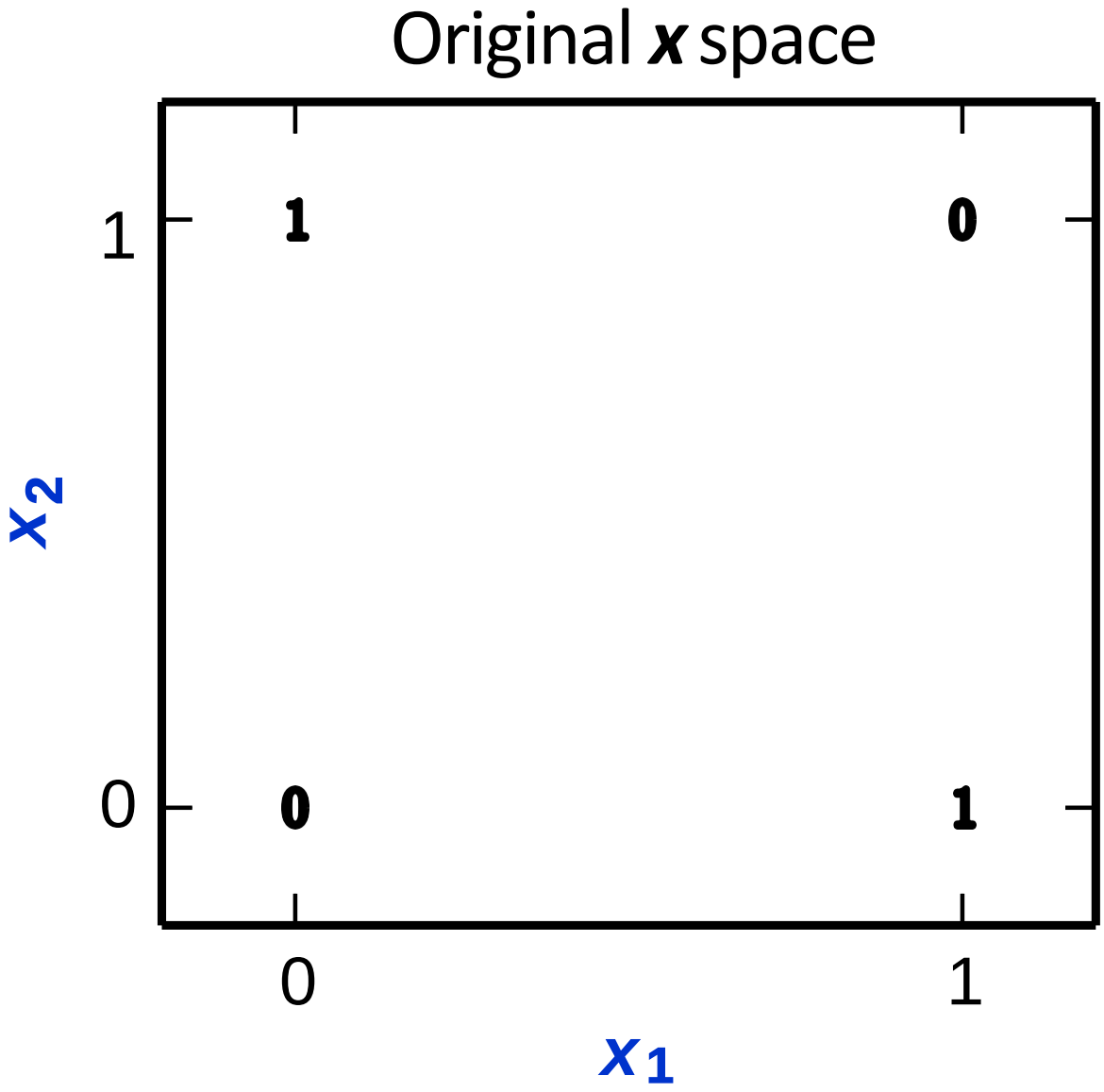
1. Deep feedforward networks
2. Feedforward neural networks
3. Multilayer perceptrons (MLPs)

- Defines a mapping $y = f(x; \theta)$
- Approximates f^*
- Learns θ that result in the best function approximation
- The training data provides us with noisy, approximate examples of f*(x) evaluated at different training points

- FFNs are typically represented by composing together many different functions. The model is associated with a directed acyclic graph.
- For example, we might have three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain, to form

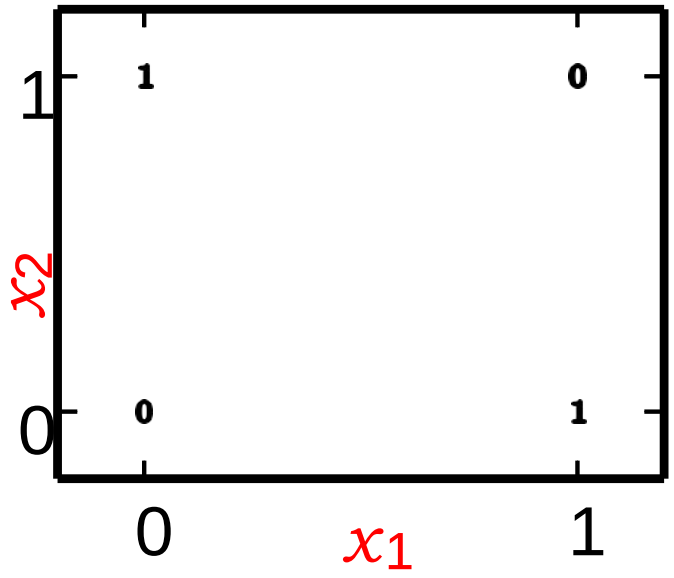
$$f(x) = f^{(1)}(f^{(2)}(f^{(3)}(x)))$$

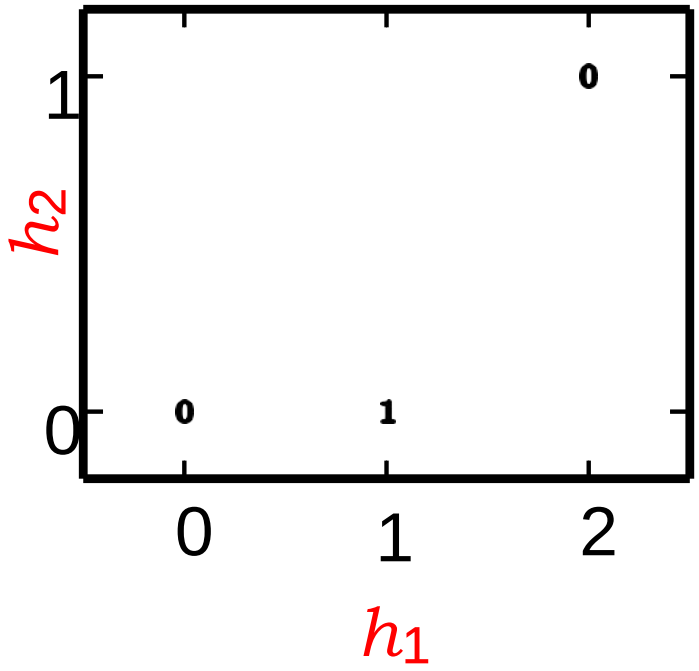
XOR is not linearly separable



Solving X-OR

- $f(x; W, c, w, b) = w^T . max\{0, W^T x + c\} + b$
- $W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$
- $c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$
- $w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$
- $b = 0$





Gradient-Based Learning

- Specify
  - Model
  - Cost (smooth)
  - Minimize cost using gradient descent or related techniques

- The nonlinearity of a neural network causes most interesting loss functions to become nonconvex.
- Stochastic gradient descent applied to nonconvex loss functions has no convergence guarantee and is sensitive to the values of the initial parameters.
- Initialize all weights to small random values. The biases may be initialized to zero.

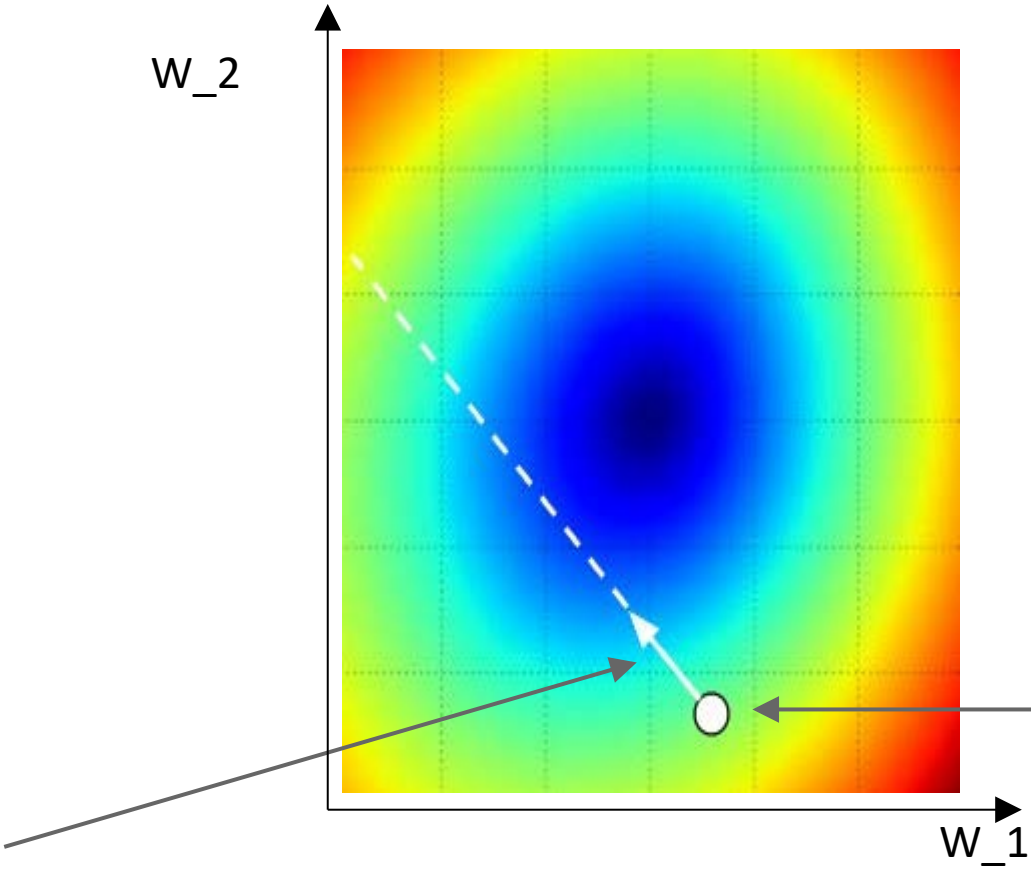
Gradient

- In 1-d, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- In multiple dimensions, the gradient is the vector of (partial derivatives).

Gradient Descent



original W

negative gradient direction

Stochastic Gradient Descent

- Nearly all of deep learning is powered by one very important algorithm: stochastic gradient descent or SGD

$$J(\theta) = \mathbb{E}_{x \sim \hat{p}_{data}} L(x, y, \theta) = \frac{1}{m} \sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta)$$

where L is the per-example loss $L(x, y, \theta) = -\log p(y|x; \theta)$

SGD

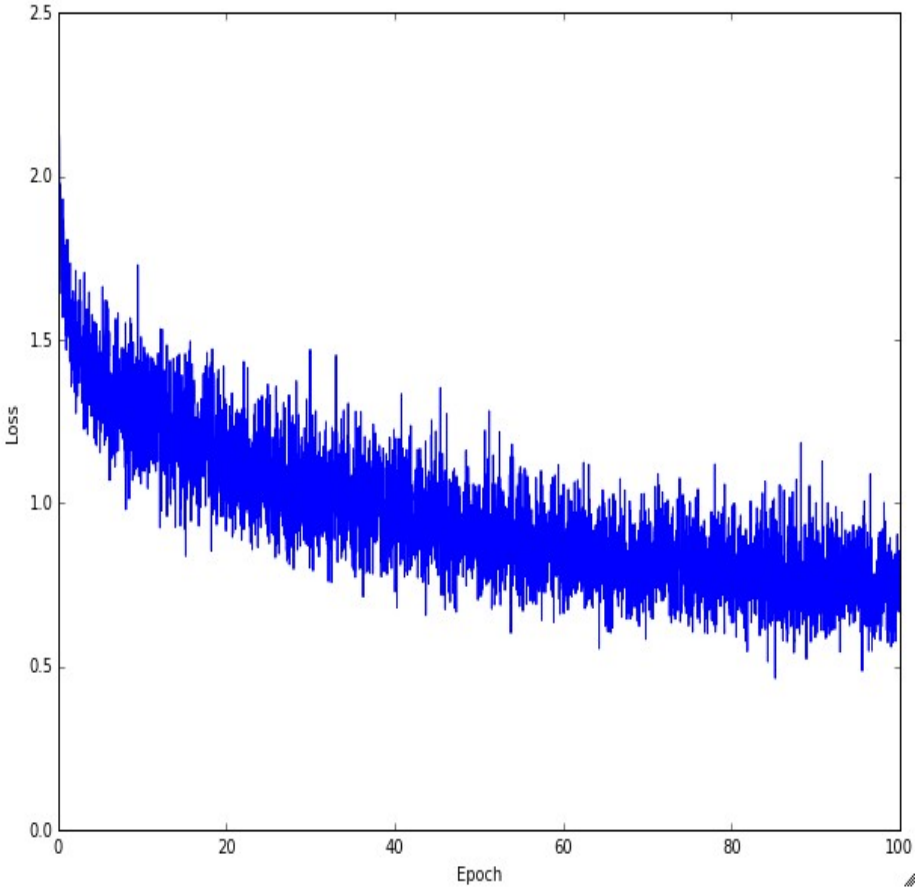
- gradient descent requires computing

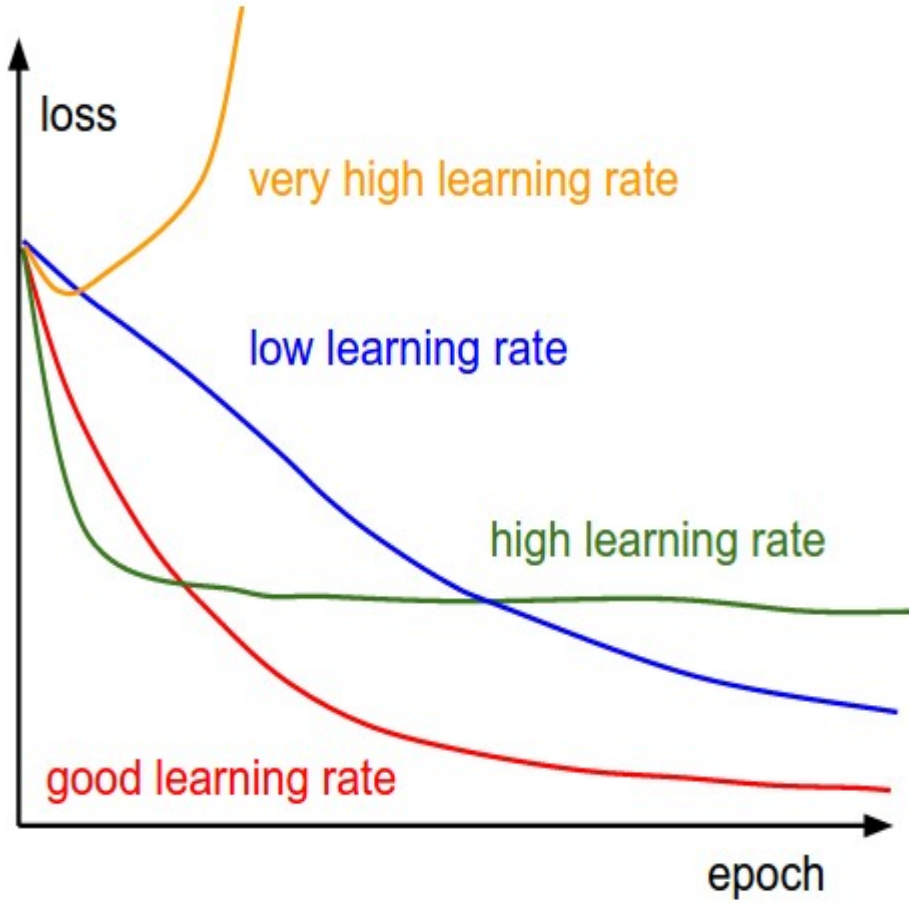
$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

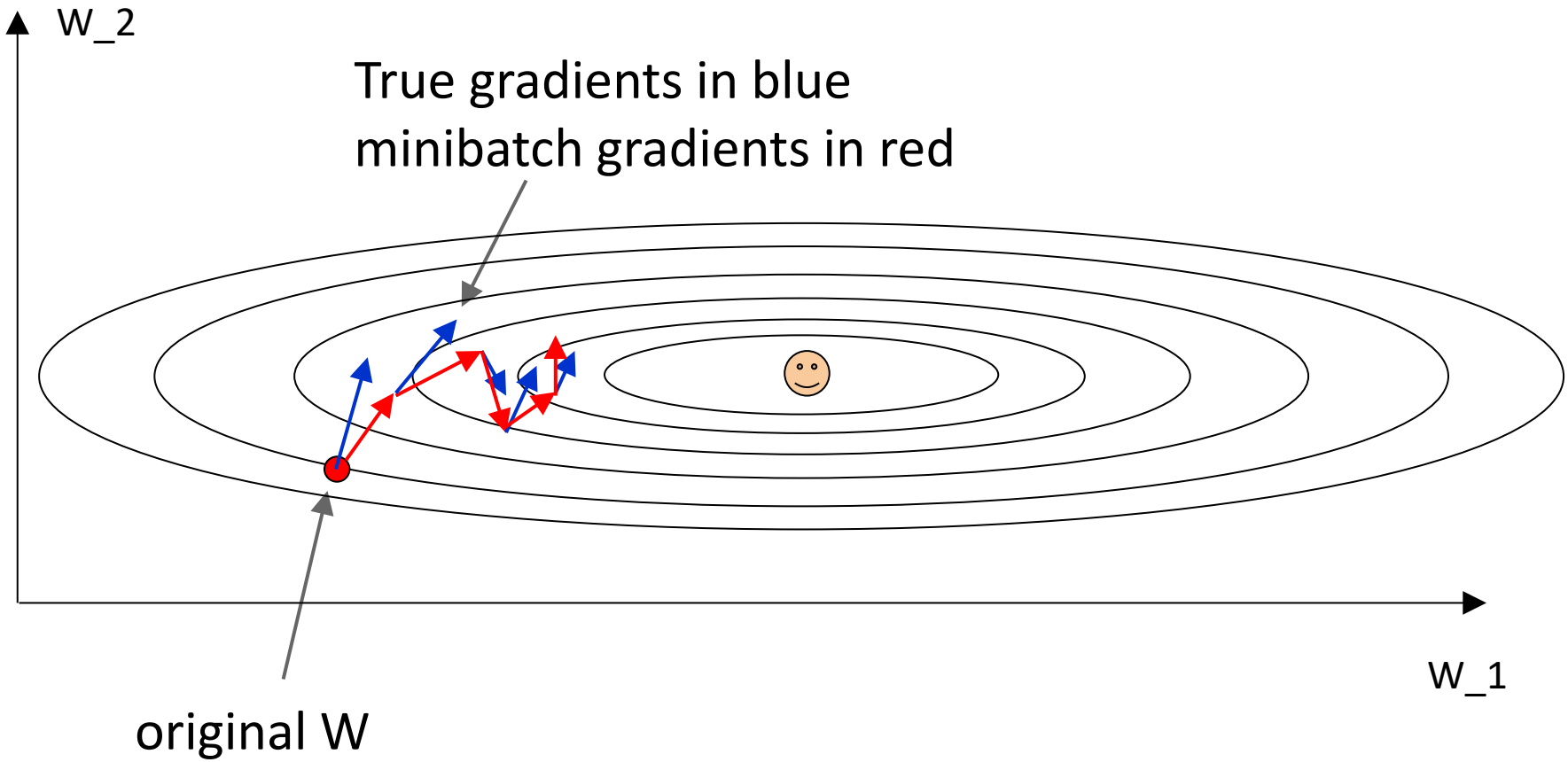
- The insight of stochastic gradient descent is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples.
- We can sample a minibatch of examples of size m'
- The estimate of the gradient is formed as

$$g = \frac{1}{m'}\sum_{i=1}^{m'} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

The effects of step size (or “learning rate”)







Gradients are noisy but still make good progress on average

Cost Functions

- In most cases, our parametric model defines a distribution $p(y|x;\theta)$
- Use the principle of maximum likelihood

- The cost function is often the negative log-likelihood
- equivalently described as the cross-entropy between the training data and the model distribution.

$$J(\theta) = -E_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x)$$

- .

Conditional Distributions and Cross-Entropy

$$J(\theta) = -E_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x)$$

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$
- For example, if $p_{model}(y|x) = \mathcal{N}(\mathrm{y}; \mathrm{f}(\mathrm{x}, \theta), \mathrm{I})$ then we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2} E_{x,y\sim\hat{p}_{data}} \|y - f(x;\theta)\|^2 + \mathrm{Const}$$

- For predicting median of Gaussian, the equivalence between maximum likelihood estimation with an output distribution and minimization of mean squared error holds

- The gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.
- The negative log-likelihood helps to avoid this problem for many models.

Evaluating the gradient numerically

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```python
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

Slides based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

$$\nabla_W L = \ldots$$

Challenges Motivating Deep Learning

- The failure of traditional algorithms to generalize well on such AI tasks as speech recognition, object recognition NLP
- High-dimensional data
- Generalization in high-dimensional spaces.

The Curse of Dimensionality

- The number of possible distinct configurations of a set of variables increases exponentially as the number of variables increases
- The number of possible configurations of x is much larger than the number of training examples.
- We assume that the data was generated by the composition of factors or features, potentially at multiple levels in a hierarchy
- The exponential advantages conferred by the use of deep, distributed representations counter the exponential challenges posed by the curse of dimensionality

Manifold Learning

- A manifold is a connected region.
- Mathematically, it is a set of points, associated with a neighborhood around each point.
- The definition of a neighborhood surrounding each point implies the existence of transformations that can be applied to move on the manifold from one position to a neighboring one.
- In ML it tends to be used more loosely to designate a connected set of points that can be approximated well by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space.
- Each dimension corresponds to a local direction of variation

Manifold Learning

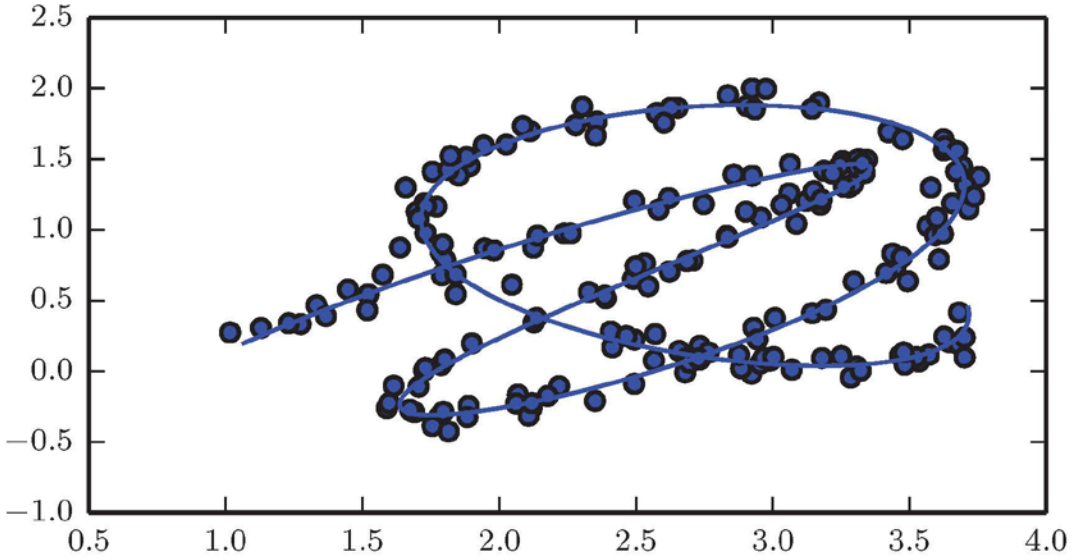


Figure 5.11

- Manifold learning algorithms assume that
  - most of R^n consists of invalid inputs
  - interesting inputs occur only along a collection of manifolds containing a small subset of points,
  - with interesting variations in the output of the learned function occurring only along directions that lie on the manifold,
  - or with interesting variations happening only when we move from one manifold to another.

Manifold hypothesis

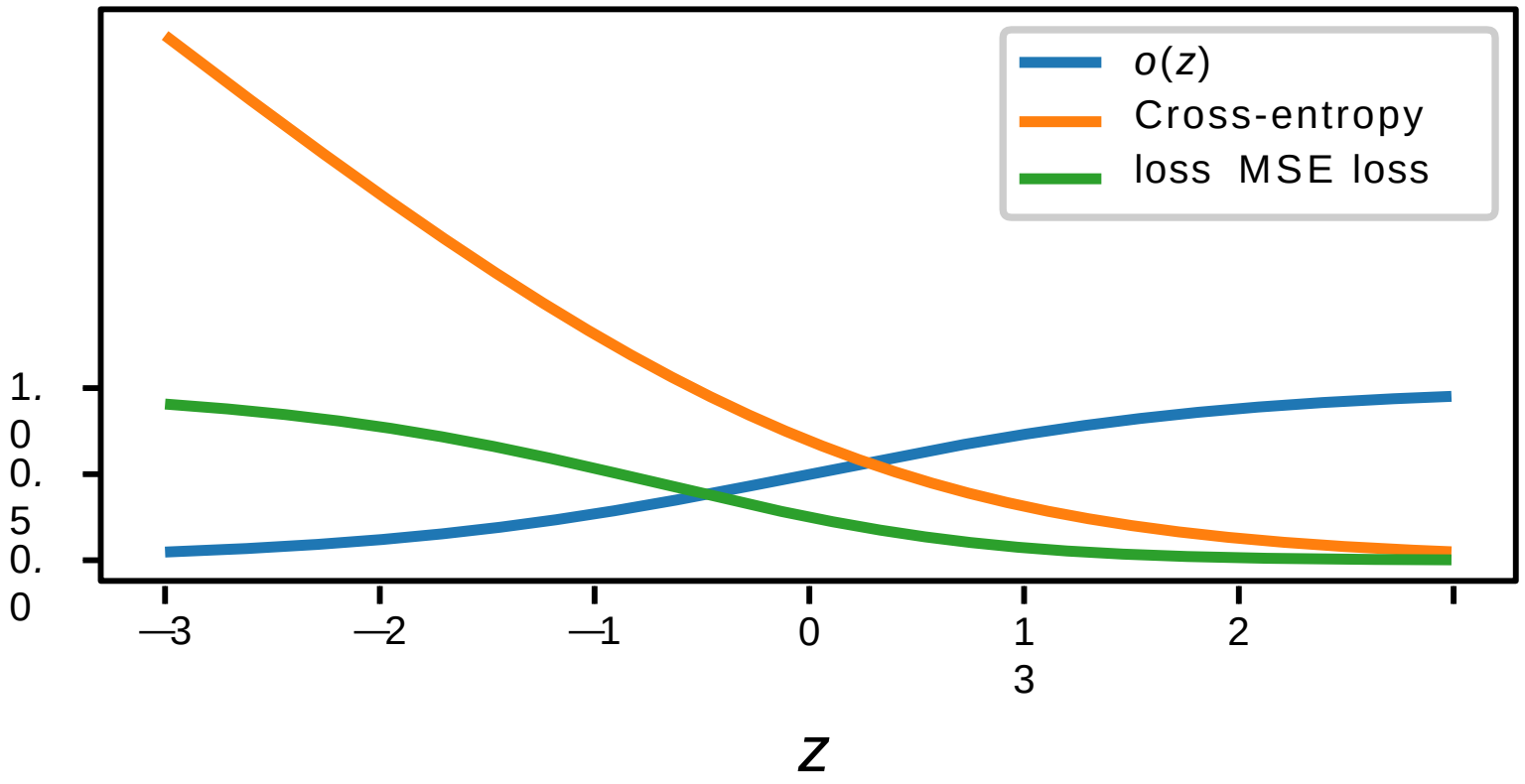
- We argue that in the context of AI tasks, such as those that involve processing images, sounds, or text, the manifold assumption is at least approximately correct.
- Obs1: the probability distribution over images, text strings, and sounds that occur in real life is highly concentrated
- Obs2. we can also imagine such neighborhoods and transformations informally.
  - images
  - Images: we can think of transformations that allow us to trace out a manifold in image space: we can gradually dim or brighten the lights, gradually move or rotate objects in the image, gradually alter the colors on the surfaces of objects, etc.
  - It remains likely that there are multiple manifolds involved in most applications. For example, the manifold of images of human faces may not be connected to the manifold of images of cat faces.

- When the data lies on a low-dimensional manifold, it can be most natural for machine learning algorithms to represent the data in terms of coordinates on the manifold, rather than in terms of coordinates in Rn.

Output Types

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	GAN, VAE, FVBN	Various

Sigmoid output with target of 1



Sigmoid units

- Task: Predict a binary variable y
- Sigmoid unit : $\hat{y} = \sigma(w^T h + b)$
- Cost:

$$J(\theta) = -\log p(y|x) = -\log \sigma((2y-1)(w^T h + b))$$

Softmax units

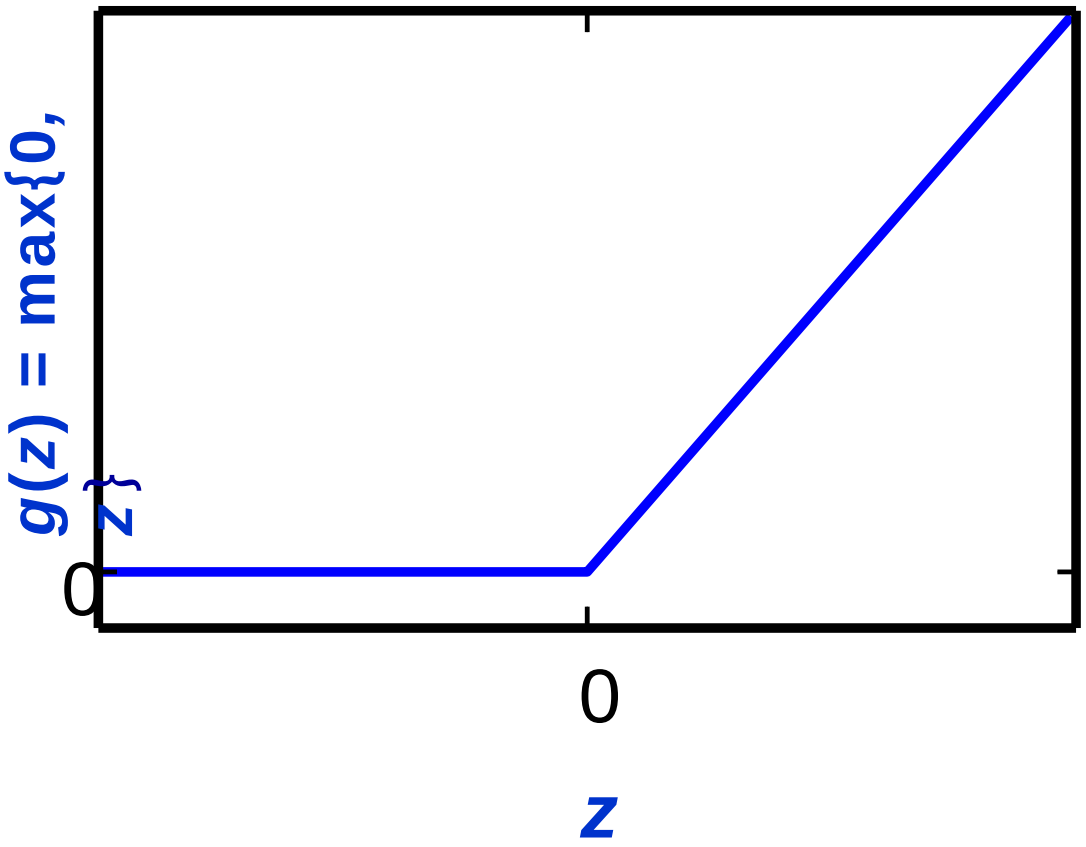
- Task: Predict a vector $\hat{y}$ with $\hat{y}_i = p(y = i|x)$
- Linear layer produces unnormalized log probabilities: $z = w^T h + b$
- Softmax:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Log of the softmax:

$$\text{logsoftmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

Rectified Linear Activation



Positives:

- Gives large and consistent gradients (does not saturate) when active
- Efficient to optimize, converges much faster than sigmoid or tanh

Negatives:

- Units "die" i.e. when inactive they will never update

Rectified linear units are an excellent default choice of hidden unit. Use ReLUs, 90% of the time

Universal Approximator Theorem

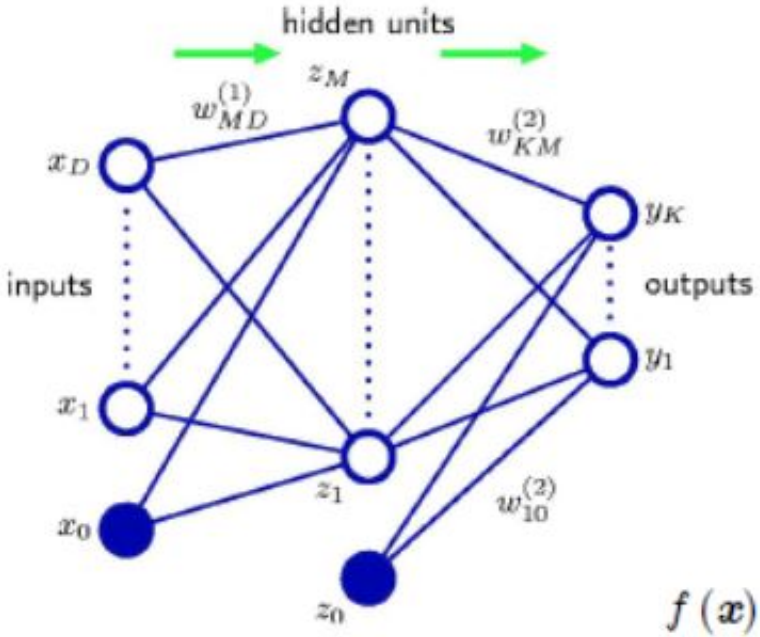
- One hidden layer is enough to *represent* (not *learn*) an approximation of any function to an arbitrary degree of accuracy
- So why deeper?
  - Shallow net may need (exponentially) more width
  - Shallow net may overfit more

- http://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html
- https://blog.goodaudience.com/neural-networks-part-1-a-simple-proof-of-the-universal-approximation-theorem-b7864964dbd3
- http://neuralnetworksanddeeplearning.com/chap4.html

Hidden Layer

- Behavior of other layers is not directly specified by the data
- Learning algorithm must decide how to use those layers to produce value that is close to y
- Training data does not say what individual layers should do
- Since the desired output for these layers is not shown, they are called hidden layers

A net with depth 2: one hidden layer



K outputs $y_1,..y_K$ for a given input $\mathbf{x}$
Hidden layer consists of M units

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

$$f(\boldsymbol{x}) = f^{(2)}[f^{(1)}(\boldsymbol{x})]$$

$f^{(1)}$ is a vector of M dimensions and
$f^{(2)}$ is a vector of K dimensions

$$f_m{}^{(1)} = z_m = h(\boldsymbol{x}^T \boldsymbol{w}^{(1)}),\ m=1,..M$$
$$f_k{}^{(2)} = \sigma(\boldsymbol{z}^T \boldsymbol{w}^{(2)}),\ k=1,..K$$

Depth versus Width

- Going deeper makes network more expressive
  - It can capture variations of the data better.
  - Yields expressiveness more efficiently than width
- Tradeoff for more expressiveness is increased tendency to overfit
  - You will need more data or additional regularization
    - network should be as deep as training data allows.
  - But you can only determine a suitable depth by experiment.
    - Also computation increases with no. of layers.

Exponential Gains from depth

Efficiency of MLP

- Multilayer perceptrons have these properties:
  1. They are universal approximators
     - i.e., can approximate most functions given enough hidden units upto any non-zero tolerance
  2. Functions are represented by smaller deep networks compared to shallow networks
  - No of linear regions carved out with d inputs, depth l and n units per hidden layer is exponential in l

$$O\left[\binom{n}{d}^{d(l-1)} n^d\right]$$

  - Decrease in model size leads to statistical efficiency
- Similar results apply to other distributed hidden representations

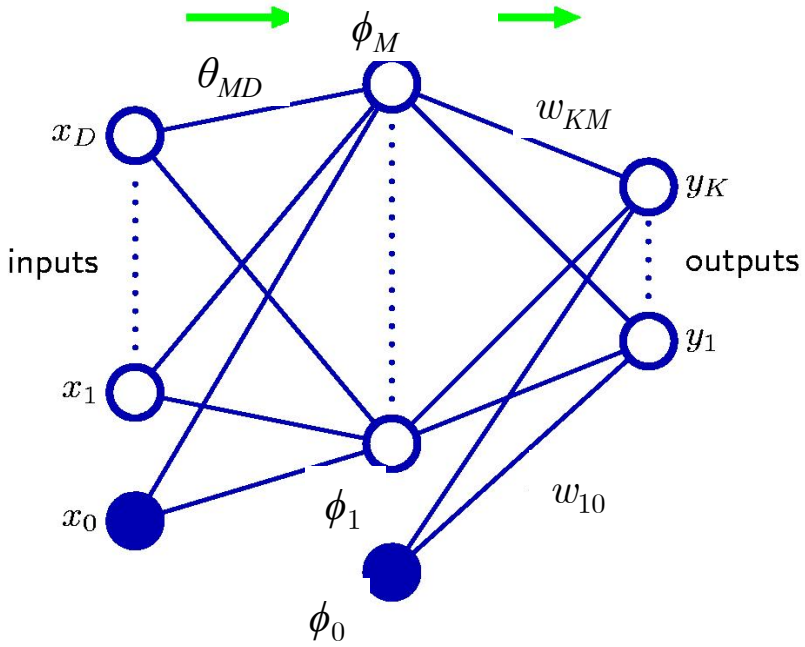
Expressive power of deep networks

- There are families of functions that can be:
  1. Efficiently represented by architecture of depth k
  2. But would require an exponential number of hidden units (wrt input size) with insufficient depth (depth 2 or depth k-1)

Minimum depth required

- Theoretical result based on using a Sum-Product Network (SPN)
  - Which use polynomial circuits
  - Models compute a probability distribution over a set of random variables
- Showed that there exist probability distributions for which a minimum depth of SPN is required to avoid needing an exponentially large model
  - Later showed that there are significant differences between every two finite depths of SPN
  - Some constraints used to make SPNs tractable may limit their representational power

Extend Linear Methods to Learn ϕ



K outputs $y_1,..y_K$ for a given input $\boldsymbol{x}$
Hidden layer consists of M units

$$y_k(\boldsymbol{x};\boldsymbol{\theta},\mathbf{w}) = \sum_{j=1}^{M} w_{kj}\phi_j\left(\sum_{i=1}^{D}\theta_{ji}x_i + \theta_{j0}\right) + w_{k0}$$

$$y_k = f_k(\boldsymbol{x};\boldsymbol{\theta},\mathrm{w}) = \boldsymbol{\phi}(\boldsymbol{x};\boldsymbol{\theta})^T \mathrm{w}$$

Can be viewed as a generalization of linear models

- Nonlinear function f_k with $M+1$ parameters $\mathbf{w}_k = (w_{k0},..w_{kM})$ with
- M basis functions, ϕ_j $j=1,..M$ each with D parameters $\boldsymbol{\theta}_j = (\theta_{j1},..\theta_{jD})$
- Both w_k and $\boldsymbol{\theta}_j$ are learnt from data

Approaches to Learning ϕ

- Parameterize the basis functions as $\phi(\boldsymbol{x};\boldsymbol{\theta})$
  - Use optimization to find $\boldsymbol{\theta}$ that corresponds to a good representation
- Approach can capture benefit of first approach (fixed basis functions) by being highly generic
  - By using a broad family for $\phi(\boldsymbol{x};\boldsymbol{\theta})$
- Can also capture benefits of second approach
  - Human practitioners design families of $\phi(\boldsymbol{x};\boldsymbol{\theta})$ that will perform well
  - Need only find right function family rather than precise right function

Importance of Learning ϕ

- Learning ϕ is discussed beyond this first introduction to feed-forward networks
  - It is a recurring theme throughout deep learning applicable to all kinds of models
- Feedforward networks are application of this principle to learning deterministic mappings form x to y without feedback
- Applicable to
  - learning stochastic mappings
  - functions with feedback
  - learning probability distributions over a single vector

Plan of Discussion: Feedforward Networks

1. A simple example: learning XOR
2. Design decisions for a feedforward network
   - Many are same as for designing a linear model
     - Basics of gradient descent
       - Choosing the optimizer, Cost function, Form of output units
   - Some are unique
     - Concept of hidden layer
       - Makes it necessary to have activation functions
     - Architecture of network
       - How many layers , How are they connected to each other, How many units in each later
     - Learning requires gradients of complicated functions
       - Backpropagation and modern generalizations

ML for XOR: linear model doesn't fit

- Treat it as regression with MSE loss function

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in X}\left(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta})\right)^2 = \frac{1}{4}\sum_{n=1}^{4}\left(f^*(\boldsymbol{x}_n) - f(\boldsymbol{x}_n;\boldsymbol{\theta})\right)^2$$

  - Usually not used for binary data
  - But math is simple

Alternative is Cross-entropy $J(\boldsymbol{\theta})$

$$J(\boldsymbol{\theta}) = -\ln p(\boldsymbol{t}\,|\,\boldsymbol{\theta}) = -\sum_{n=1}^{N}\left\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\right\}$$

$$y_n = \sigma(\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}_n)$$

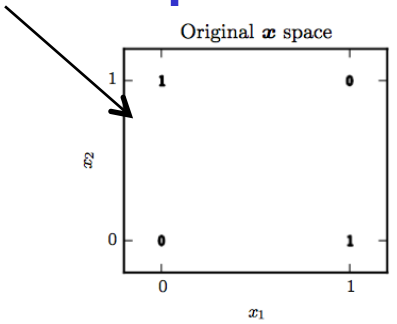
- We must choose the form of the model
- Consider a linear model with $\boldsymbol{\theta} = \{\boldsymbol{w}, b\}$ where

$$f(\boldsymbol{x};\boldsymbol{w},b) = \boldsymbol{x}^T\boldsymbol{w} + b$$

  - Minimize $J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{n=1}^{4}\left(t_n - \boldsymbol{x}_n^T\boldsymbol{w} - b\right)^2$ to get closed-form solution
    - Differentiate wrt $\boldsymbol{w}$ and b to obtain $\boldsymbol{w} = \mathbf{0}$ and $b = ½$
      - Then the linear model $f(\boldsymbol{x};\boldsymbol{w},b) = ½$ simply outputs 0.5 everywhere
  - Why does this happen?

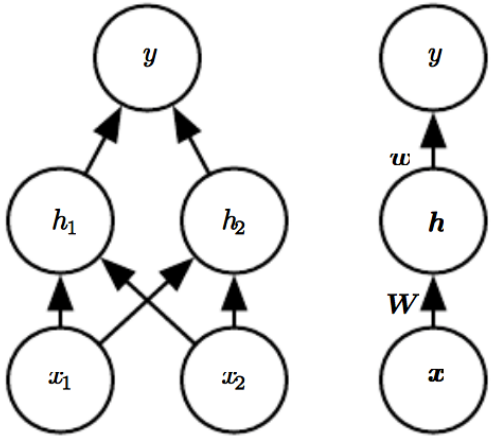
Linear model cannot solve XOR

- Bold numbers are values system must output
  - When $x_1=0$, output has to increase with x_2
  - When $x_1=1$, output has to decrease with x_2



  - Linear model $f(\boldsymbol{x};\boldsymbol{w},b)= x_1w_1+x_2w_2+b$ has to assign a single weight to x_2, so it cannot solve this problem
- A better solution:
  - use a model to learn a different representation
    - in which a linear model is able to represent the solution
  - We use a simple feedforward network
    - one hidden layer containing two hidden units

Feedforward Network for XOR



- Introduce a simple feedforward network
  - with one hidden layer containing two units
- Same network drawn in two different styles
  - Matrix W describes mapping from $\boldsymbol{x}$ to $\boldsymbol{h}$
  - Vector $\boldsymbol{w}$ describes mapping from $\boldsymbol{h}$ to y
  - Intercept parameters b are omitted

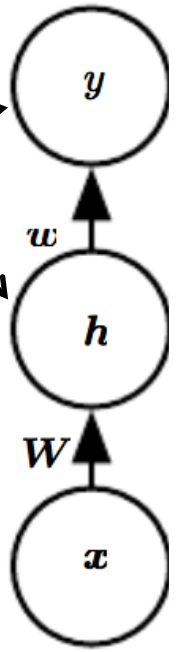
Functions computed by Network

- Layer 1 (hidden layer): vector of hidden units h computed by function $f^{(1)}(\boldsymbol{x};\ W, \boldsymbol{c})$
  - $\boldsymbol{c}$ are bias variables
- Layer 2 (output layer) computes

  $$f^{(2)}(\boldsymbol{h};\ \boldsymbol{w}, b)$$

  - w are linear regression weights
  - Output is linear regression applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$
- Complete model is

  $$f(\boldsymbol{x};\ W, \boldsymbol{c}, \boldsymbol{w}, \boldsymbol{b}) = f^{(2)}(f^{(1)}(x))$$



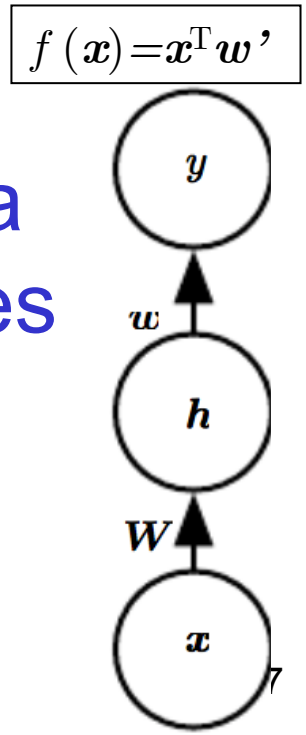
Linear vs Nonlinear functions

- If we choose both $f^{(1)}$ and $f^{(2)}$ to be linear, the total function will still be linear $f(\boldsymbol{x})=\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}'$
  - Suppose $f^{(1)}(\boldsymbol{x})= W^{\mathrm{T}}x$ and $f^{(2)}(\boldsymbol{h})=\boldsymbol{h}^{\mathrm{T}}\boldsymbol{w}$
  - Then we could represent this function as

$$f(\boldsymbol{x})=\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}' \text{ where } \boldsymbol{w}'=W\boldsymbol{w}$$

- Since linear is insufficient, we must use a nonlinear function to describe the features
  - We use the strategy of neural networks
  - by using a nonlinear activation function

$$\boldsymbol{h}=g(W^{\mathrm{T}}\boldsymbol{x}+\boldsymbol{c})$$

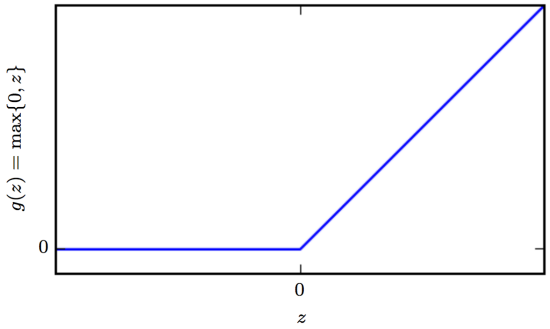


Activation Function

- In linear regression we used a vector of weights $\boldsymbol{w}$ and scalar bias b $\boxed{f(\boldsymbol{x};\boldsymbol{w},b)=\boldsymbol{x}^T\boldsymbol{w}+b}$
  - to describe an affine transformation from an input vector to an output scalar
- Now we describe an affine transformation from a vector $\boldsymbol{x}$ to a vector $\boldsymbol{h}$, so an entire vector of bias parameters is needed
- Activation function g is typically chosen to be applied element-wise $\boldsymbol{h}_i=g(x^TW_{:,i}+c_i)$

Default Activation Function

- Activation: $g(z) = \max\{0, z\}$
  - Applying this to the output of a linear transformation yields a nonlinear transformation
  - However function remains close to linear
    - Piecewise linear with two pieces
    - Therefore preserve properties that make linear models easy to optimize with gradient-based methods
    - Preserve many properties that make linear models generalize well



A principle of CS:
Build complicated systems from minimal components.

A Turing Machine Memory needs only 0 and 1 states.

We can build Universal Function approximator from ReLUs