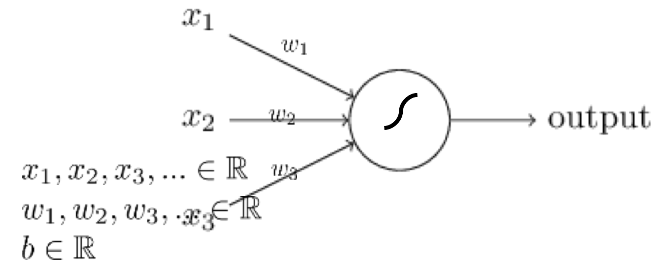
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

Neural Network Basics



- Given several **inputs**:
  and several **weights**:
  and a **bias** value:

- A neuron produces a single output:

$$o_1 = s(\sum_i w_i x_i + b)$$

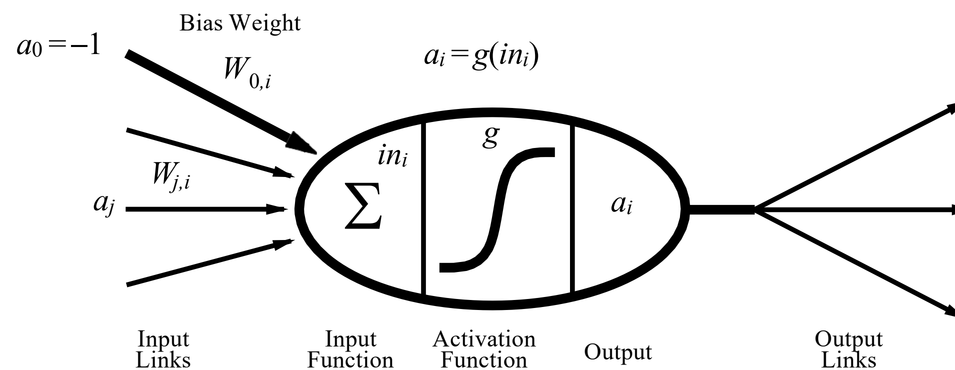
$$\sum_i w_i x_i + b$$

- This sum is called the **activation** of the neuron
- The function *s* is called the **activation function** for the neuron
- The weights and bias values are typically initialized randomly and learned during training

McCulloch–Pitts "unit"

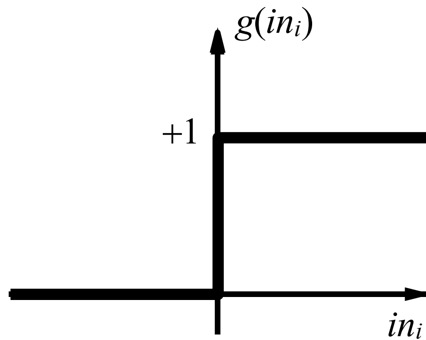
Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$

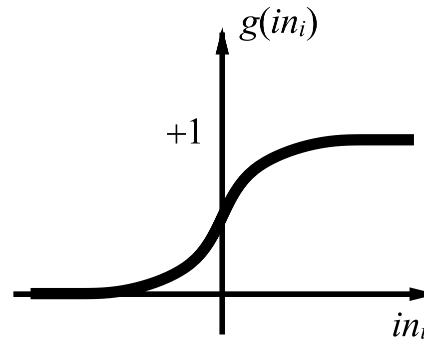


A gross oversimplification of real neurons, but its purpose is
to develop understanding of what networks of simple units can do

Activation functions



(a)



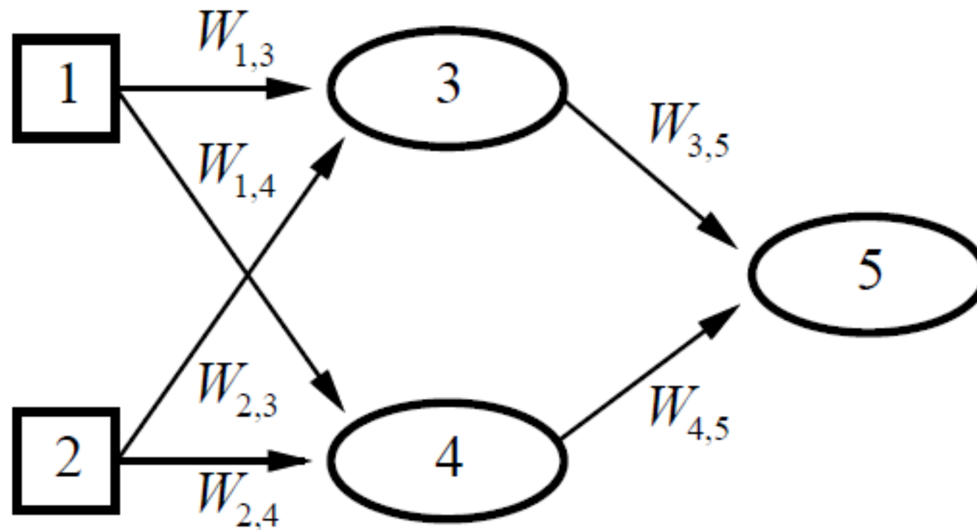
(b)

(a)is a step function or threshold function

(b)is a sigmoid function $1/(1 + e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Feed forward example



Feed-forward network = a parameterized family of nonlinear functions:

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

Adjusting weights changes the function: do learning this way!

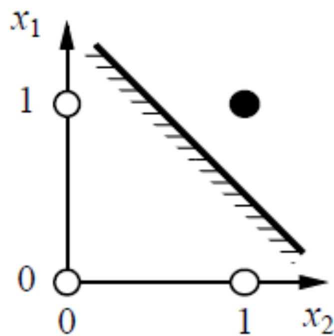
Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)

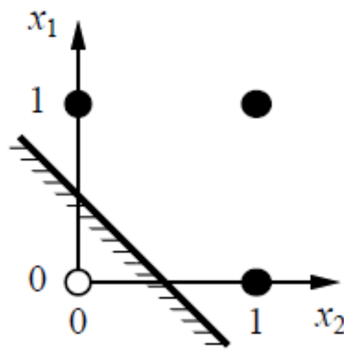
Can represent AND, OR, NOT, majority, etc., but not XOR

Represents a linear separator in input space:

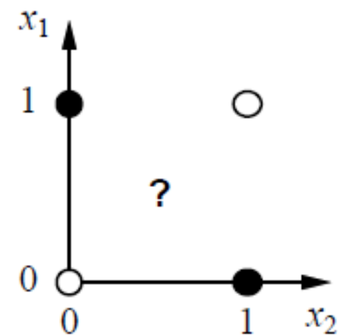
$$\Sigma_j W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$



(a) x_1 **and** x_2



(b) x_1 **or** x_2

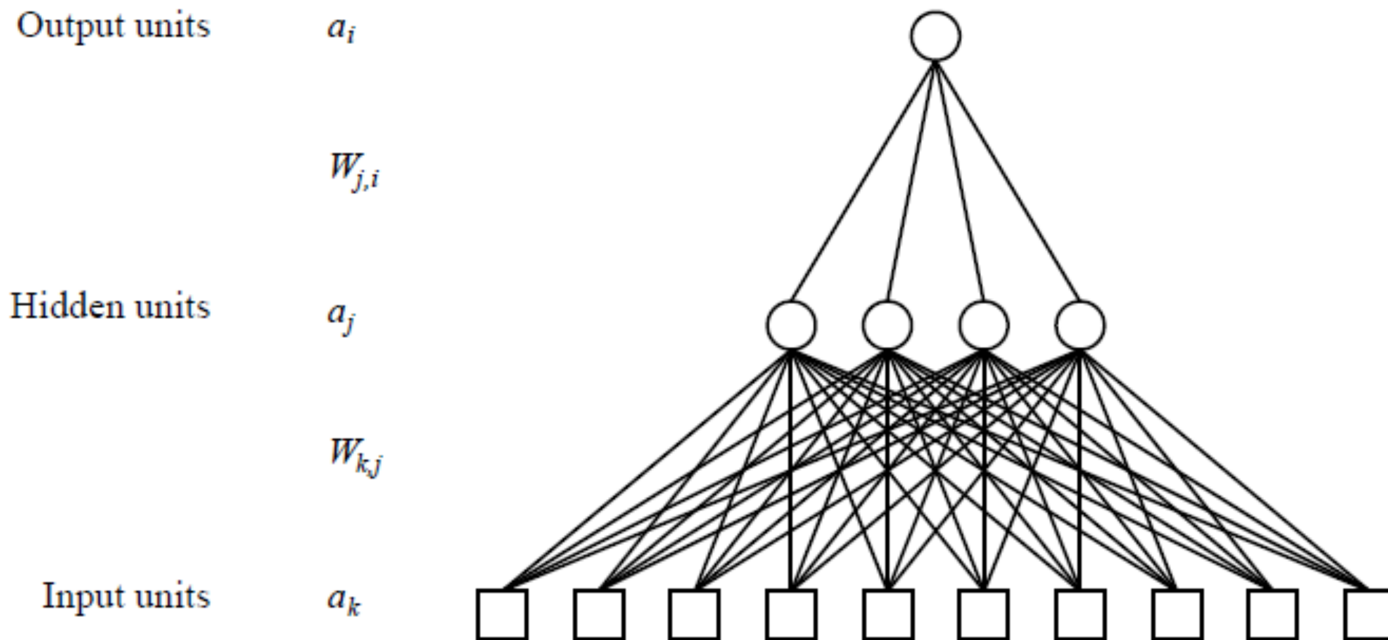


(c) x_1 **xor** x_2

Minsky & Papert (1969) pricked the neural network balloon

Feed Forward Neural Networks

Layers are usually fully connected;
numbers of hidden units typically chosen by hand



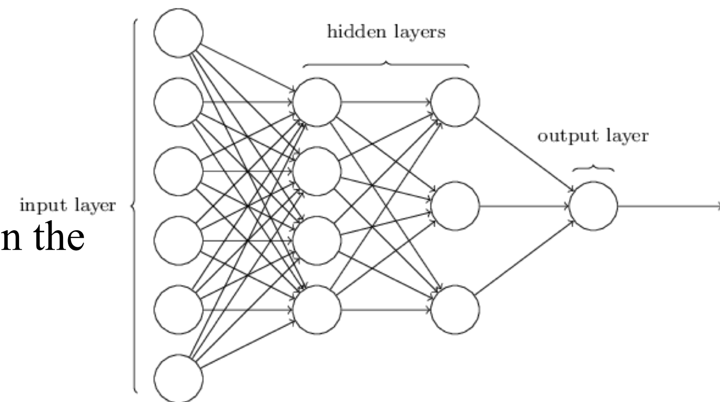
Hidden-Layer

- The hidden layer (L_2, L_3) represent learned non-linear combination of input data
- For solving the XOR problem, we need a hidden layer
  - some neurons in the hidden layer will activate only for some combination of input features
  - the output layer can represent combination of the activations of the hidden neurons
- Neural network with one hidden layer **is a universal approximator**
  - Every function can be modeled as a shallow feed forward network
  - Not all functions can be represented *efficiently* with a single hidden layer $\Rightarrow$ we still need deep neural networks

Going from Shallow to Deep Neural Networks

- Neural Networks can have several hidden layers
- Initializing the weights randomly and training all layers at once does hardly work
- Instead we train layerwise on unannotated data (a.k.a. pre-training):
  - Train the first hidden layer
  - Fix the parameters for the first layer and train the second layer.
  - Fix the parameters for the first & second layer, train the third layer

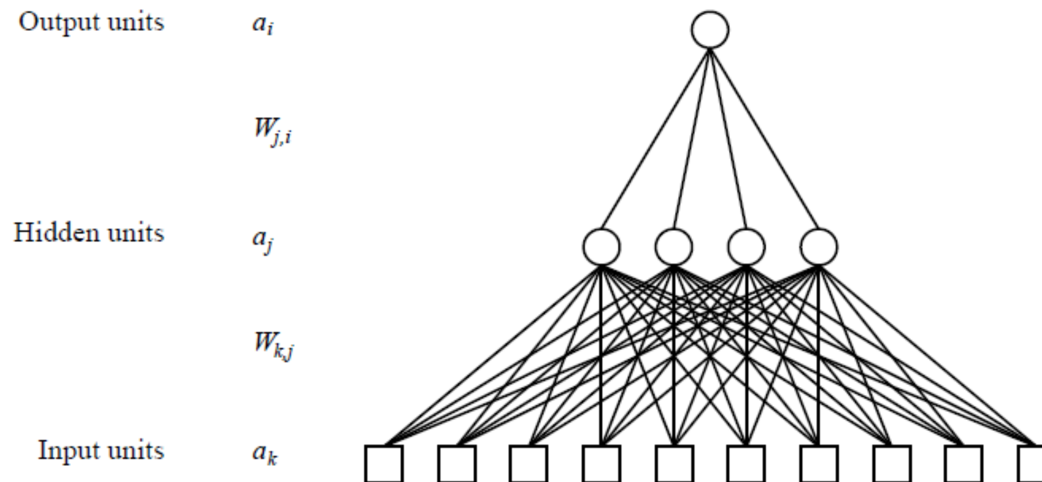
Img-Source: http://neuralnetworksanddeeplearning.com



- After the pre-training, train all layers using your annotated data
- The pre-training on your unannotated data creates a high-level abstractions of the input data
- The final training with annotated data fine tunes all parameters in the network

How to learn the weights

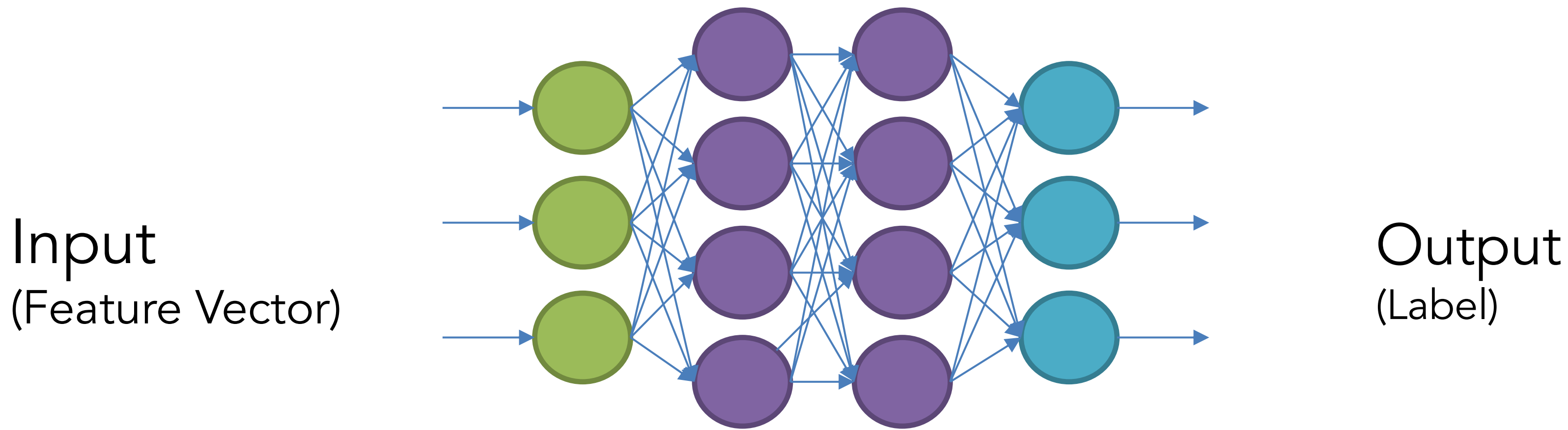
- Initialise the weights i.e. $W_{k,j}$ $W_{j,i}$ with random values
- With input entries we calculate the predicted output
- We compare the prediction with the true output
- The error is calculated
- The error needs to be sent as feedback for updating the weights



BACKPROPAGATION

Slides from Intel

How to Train a Neural Net?



- Put in Training inputs, get the output
- Compare output to correct answers: Look at loss function J
- Adjust and repeat!
- Backpropagation tells us how to make a single adjustment using calculus.

How have we trained before?

- Gradient Descent!

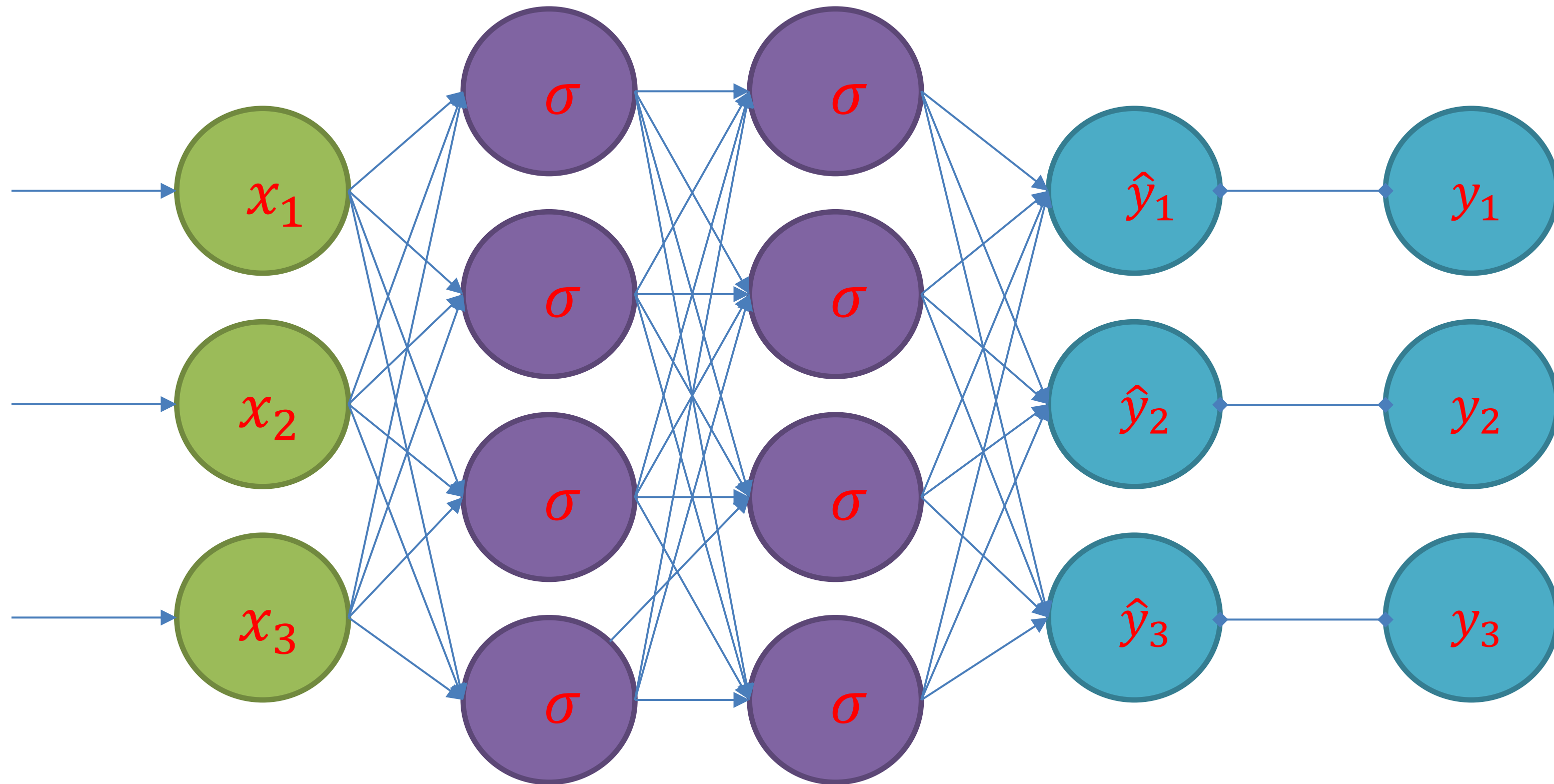
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

How have we trained before?

- Gradient Descent!

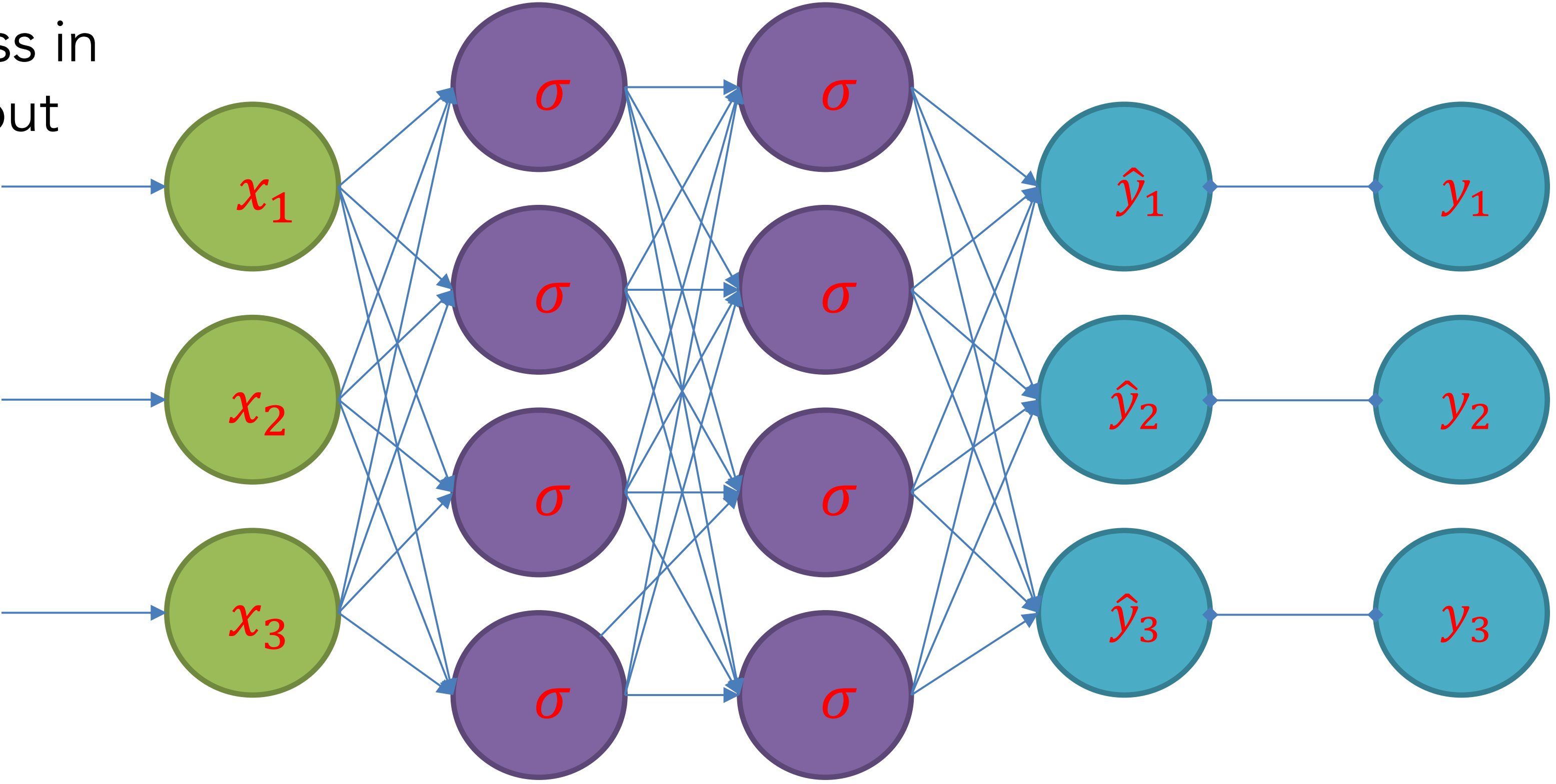
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

Feedforward Neural Network



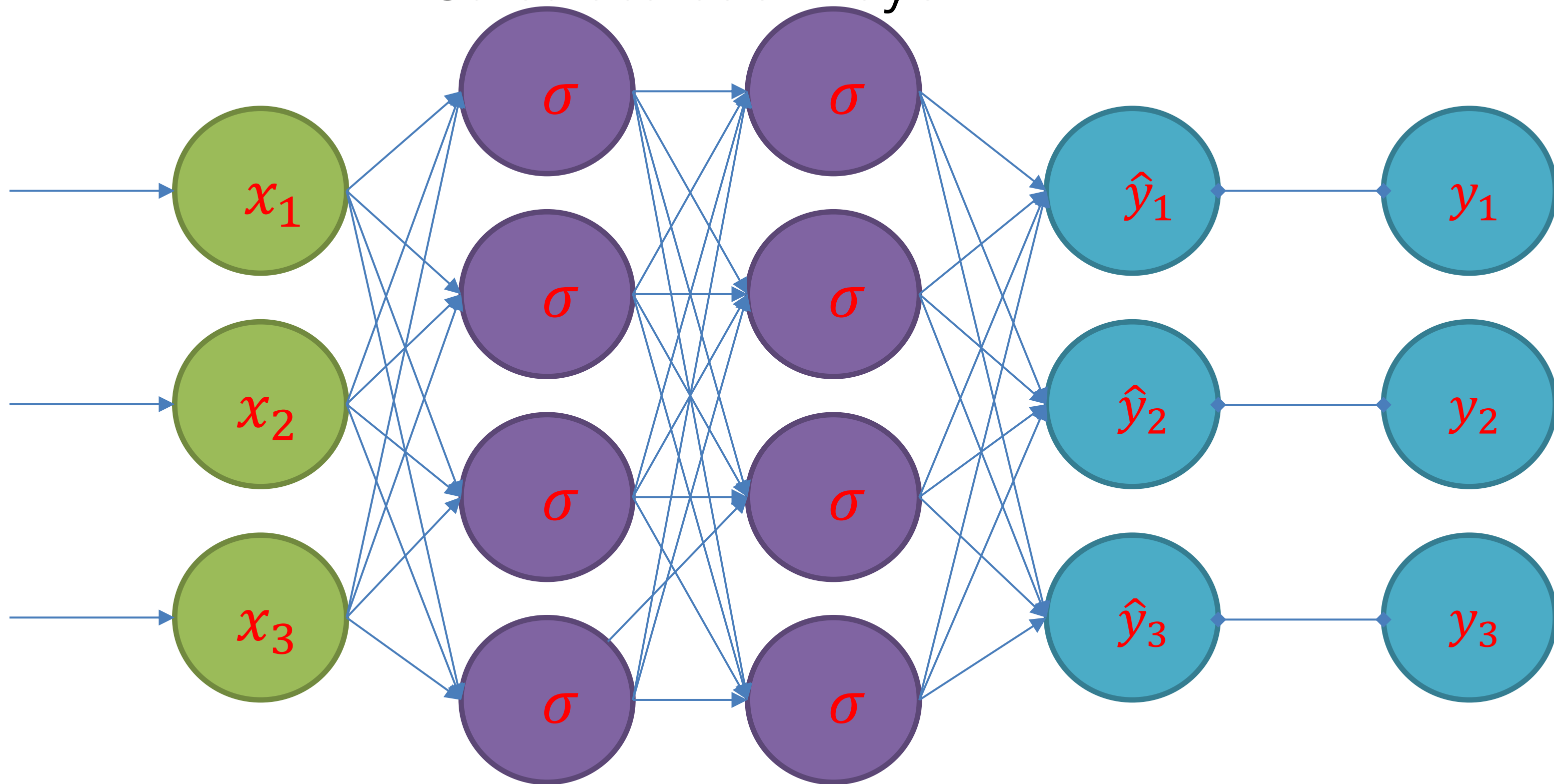
Forward Propagation

Pass in Input

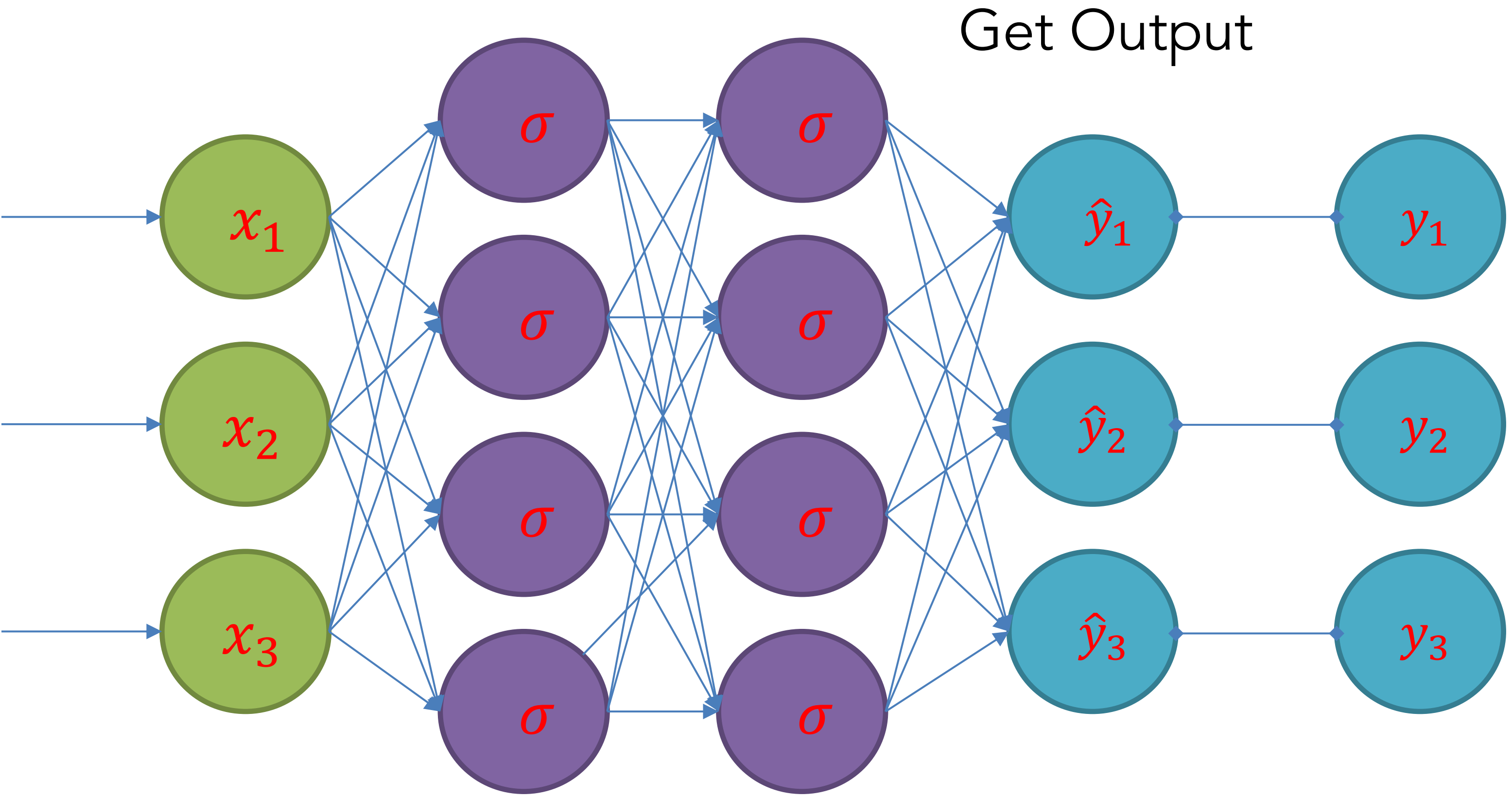


Forward Propagation

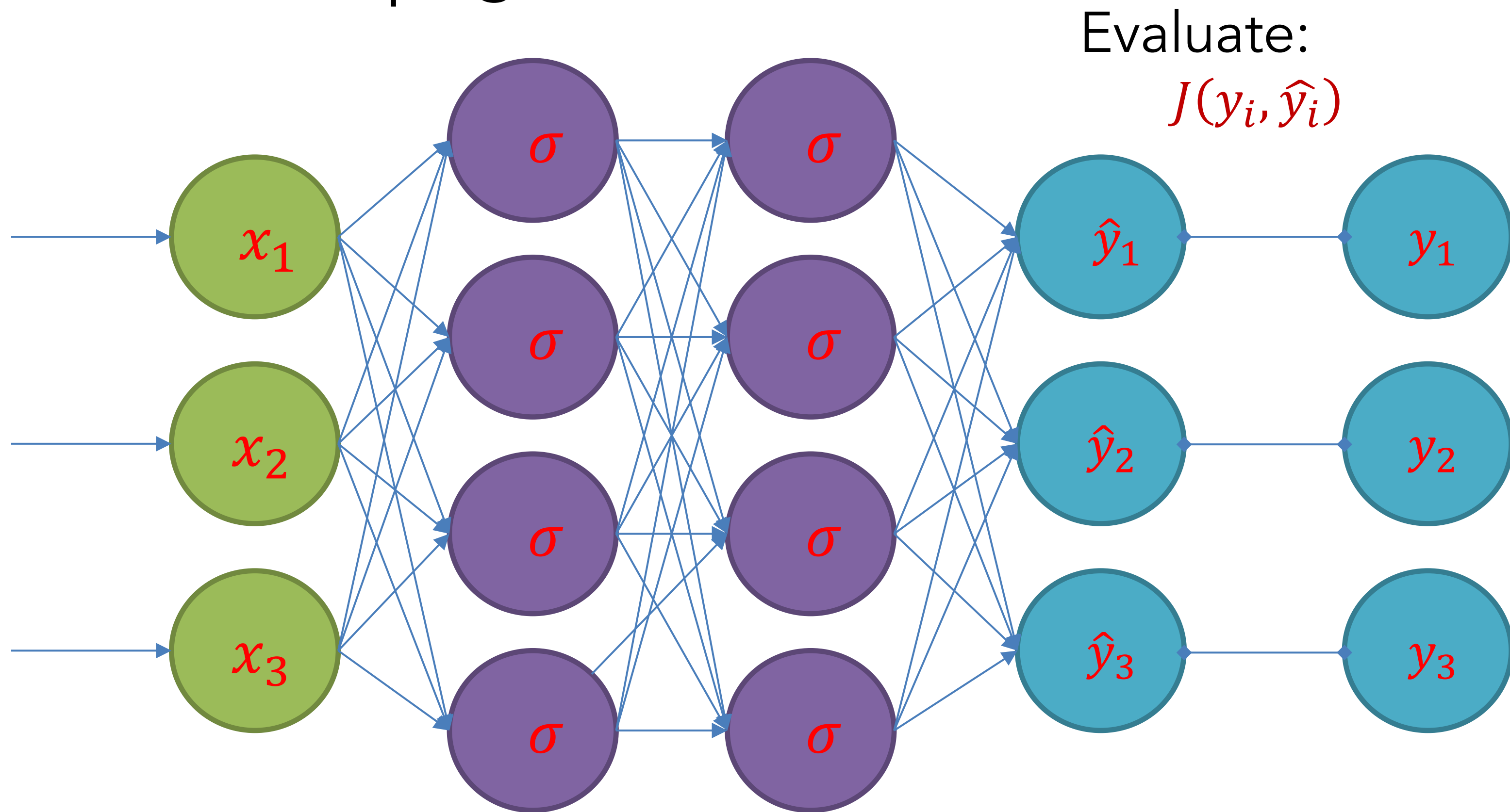
Calculate each Layer



Forward Propagation



Forward Propagation



How have we trained before?

- Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

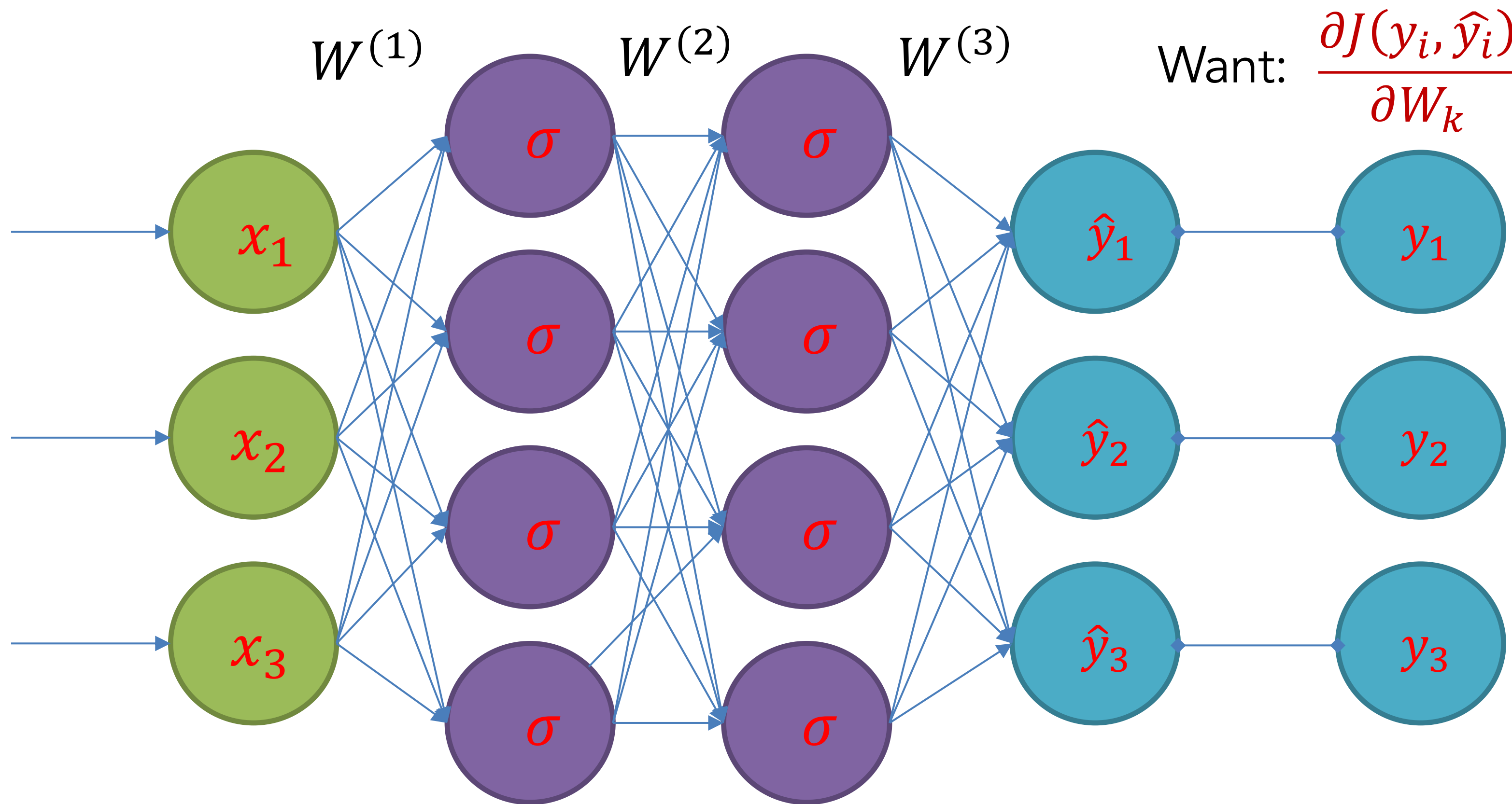
How to Train a Neural Net?

- How could we change the weights to make our Loss Function lower?
- Think of neural net as a function $F: X \rightarrow Y$
- F is a complex computation involving many weights W_k
- Given the structure, the weights “define” the function F (and therefore define our model)
- Loss Function is $J(y, F(x))$

How to Train a Neural Net?

- Get $\frac{\partial J}{\partial W_k}$ for every weight in the network.
- This tells us what direction to adjust each W_k if we want to lower our loss function.
- Make an adjustment and repeat!

Feedforward Neural Network



Calculus to the Rescue

- Use calculus, chain rule, etc. etc.
- Functions are chosen to have “nice” derivatives
- Numerical issues to be considered

Punchline

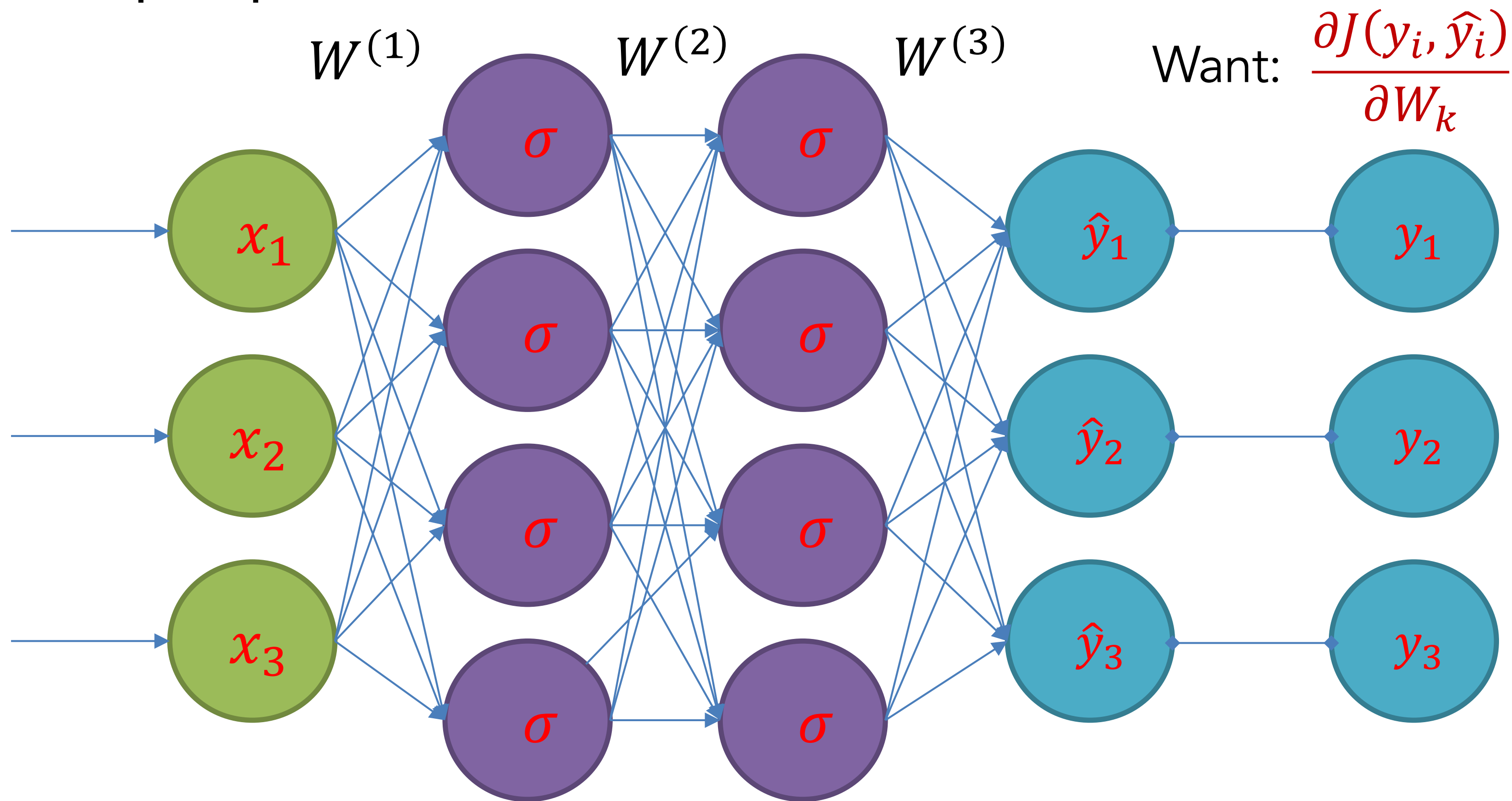
$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot a^{(2)}$$

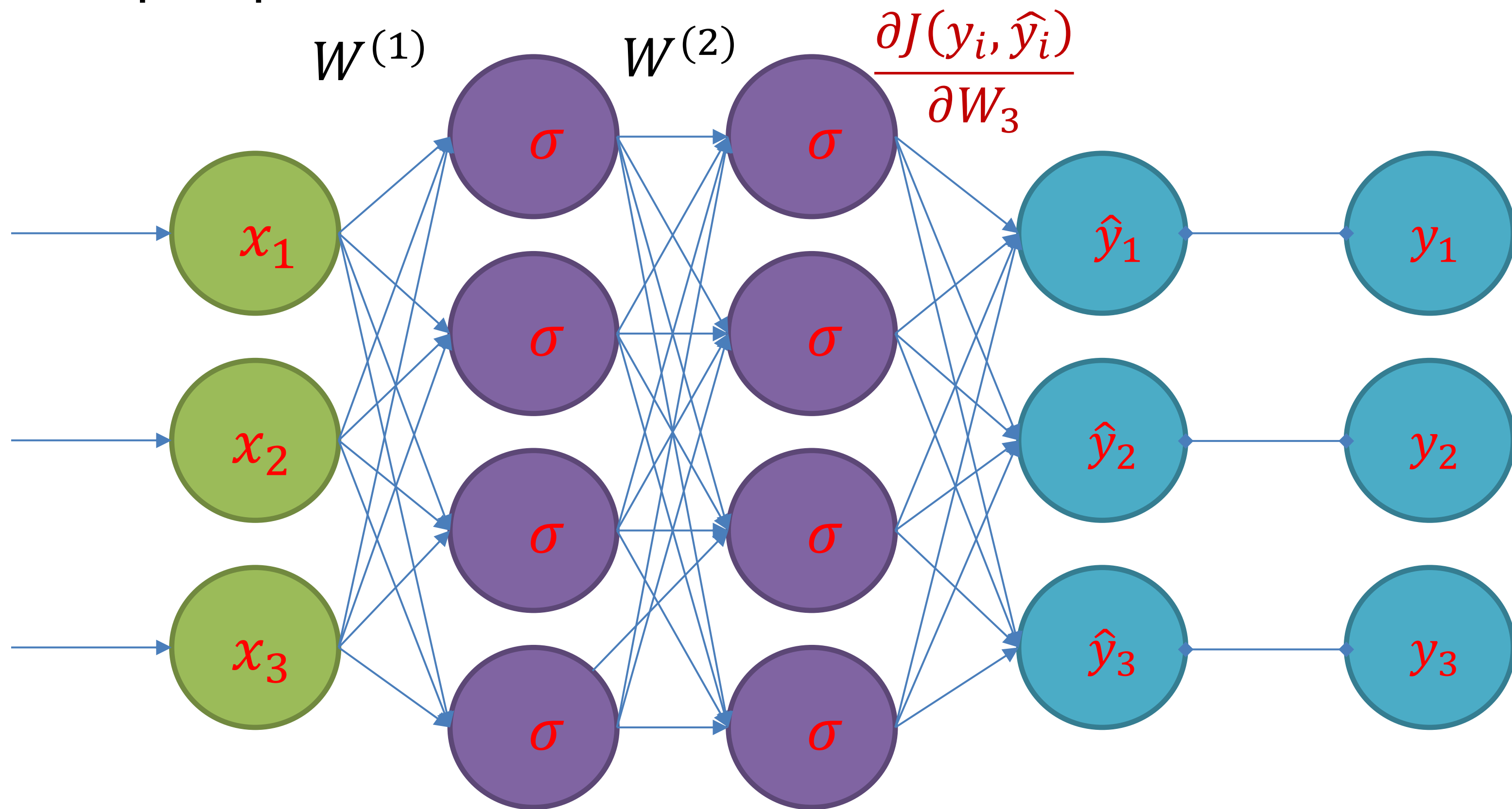
$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

- Recall that: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Though they appear complex, above are easy to compute!

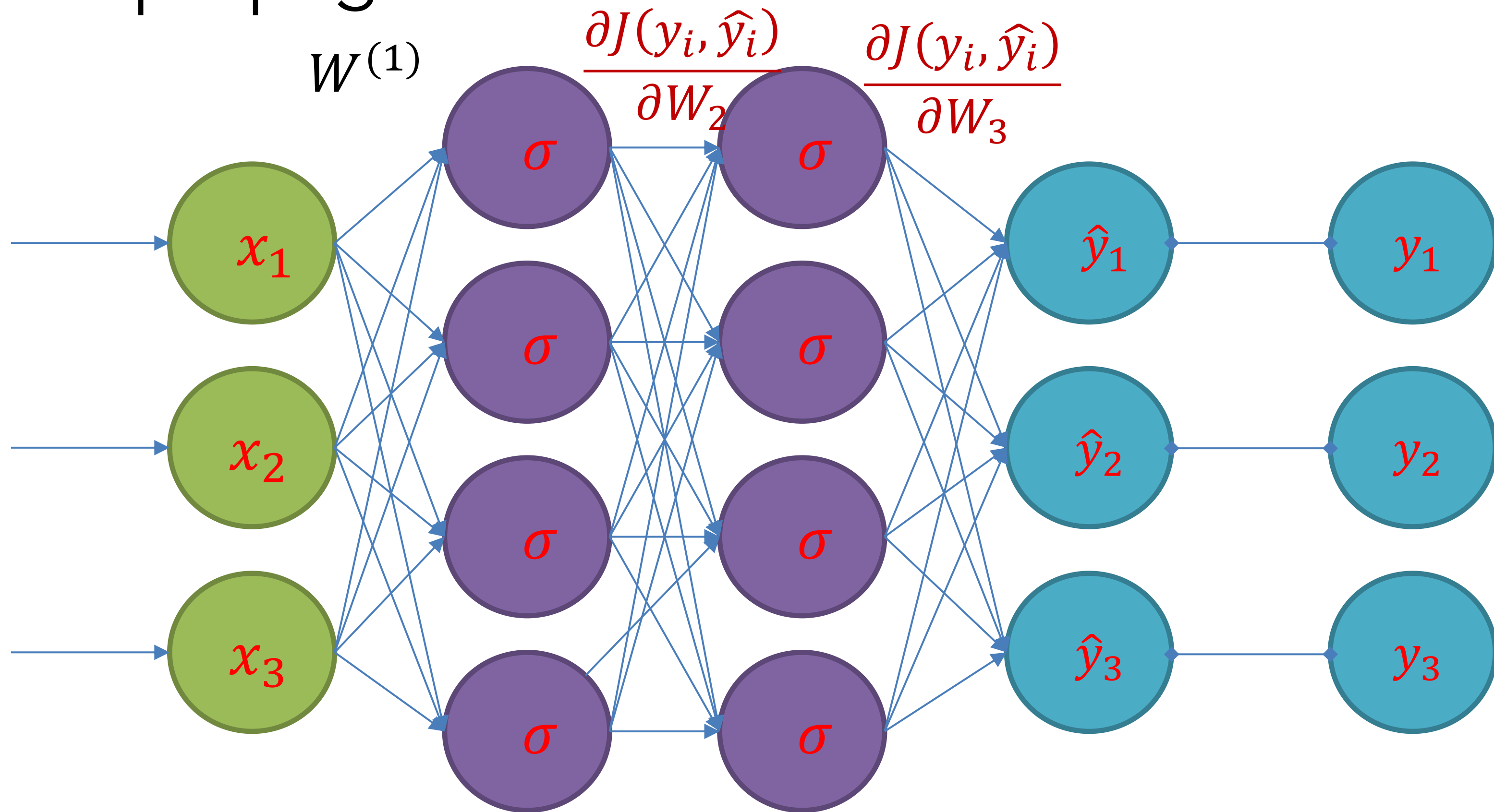
Backpropagation



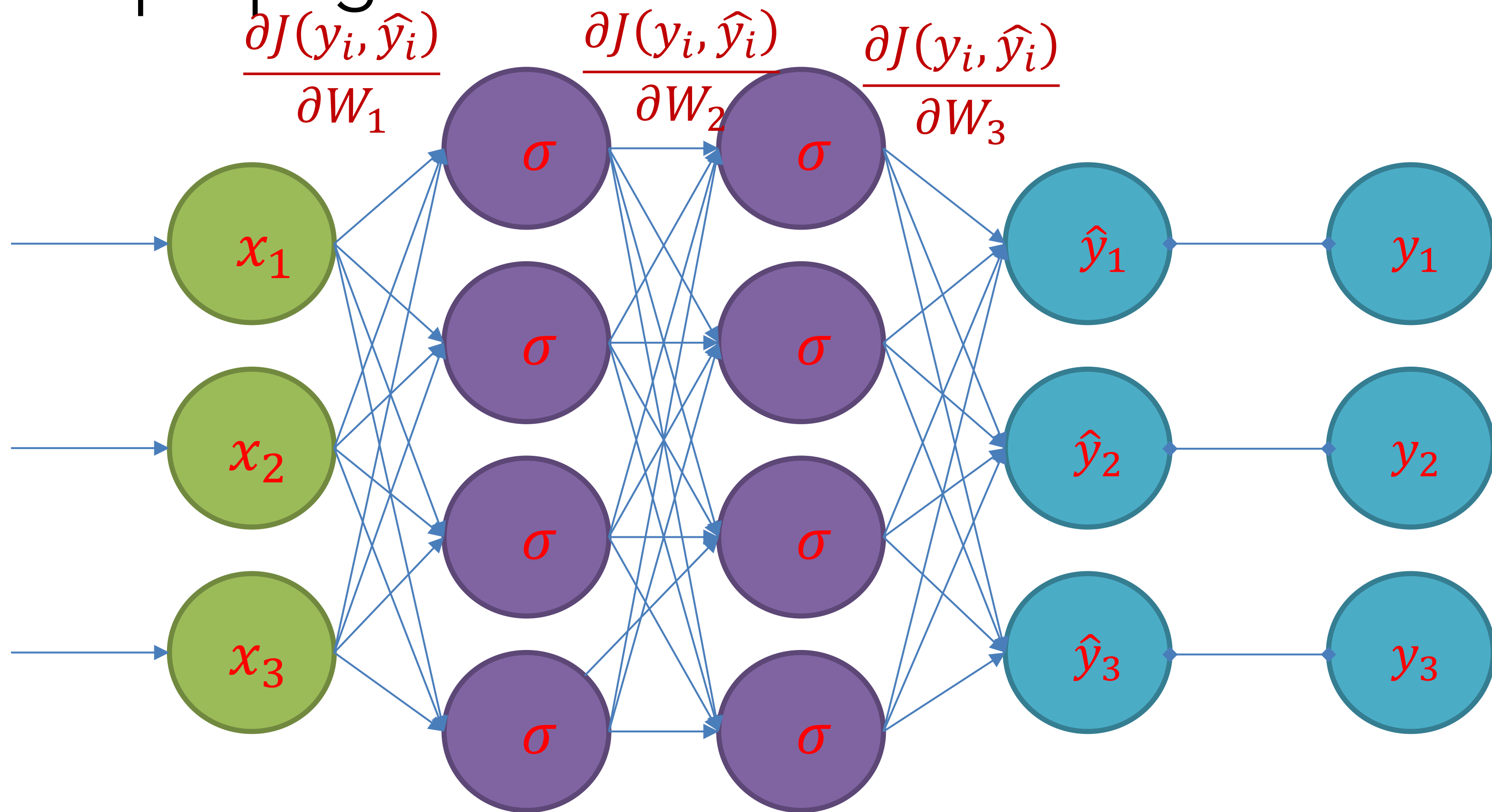
Backpropagation



Backpropagation



Backpropagation



How have we trained before?

- Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

Vanishing Gradients

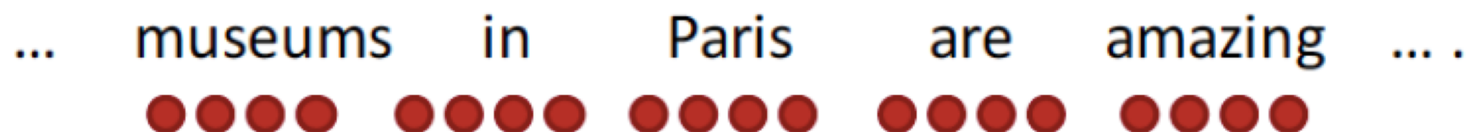
Recall that:

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

- Remember: $\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right) \leq .25$
- As we have more layers, the gradient gets very small at the early layers.
- This is known as the “vanishing gradient” problem.
- For this reason, other activations (such as ReLU) have become more common.

Window Classification

Example: Classify ‘Paris’ in the context of this sentence with window length 2:



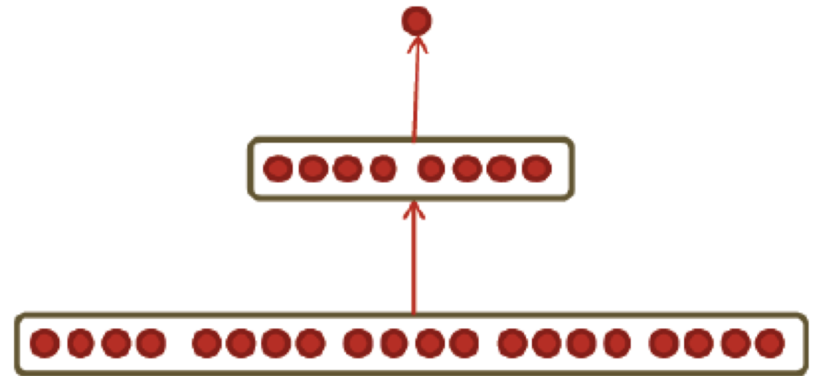
$$X_{window} = [\; x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing} \;]^T$$

Resulting vector $x_{window} \in R^{5d}$ is a column vector.

Feed-forward computation

$$s = U^T f(Wx + b) \qquad x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$$

$$\begin{aligned} s &= U^T a \\ a &= f(z) \\ z &= Wx + b \end{aligned}$$



Maximum Margin Objective Function

Idea

Ensure that the score computed for “true” labeled data points is higher than the score computed for “false” labeled data points.

- s = score(museums in Paris are amazing)
- s_c = score(Not all museums in Paris)

Maximum Margin Objective Function

Objective

Maximize $(s - s_c)$ or to minimize $(s_c - s)$. One possible objective function: minimize $J = max(s_c - s, 0)$

What is the problem with this?

- Does not attempt to create a margin of safety. We would want the "true" labeled data point to score higher than the "false" labeled data point by some positive margin Δ.
- We would want error to be calculated if $(s - s_c < \Delta)$ and not just when $(s - s_c < 0)$. The modified objective: minimize $J = max(\Delta + s_c - s, 0)$

Maximum Margin Objective Function

- Objective for a single window: $J = max(1 + s_c - s, 0)$
- Each window with a location at its center should have a score +1 higher than any window without a named entity at its center.
- $s = U^T f(Wx + b)$, $s_c = U^T f(Wx_c + b)$
- Assuming cost J is > 0, compute the derivatives of s and s_c with respect to the involved variables: U, W, b, x

Training with backpropagation

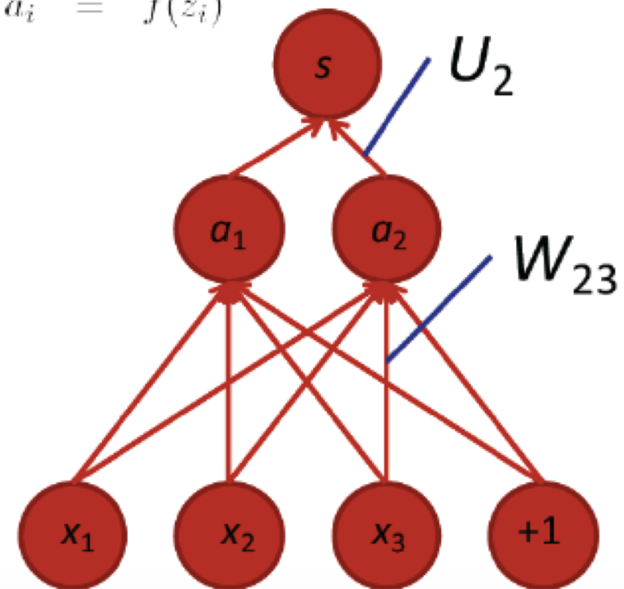
Derivative of weight W_{ij}:

$$\frac{\partial}{\partial W_{ij}} U^T a \quad \rightarrow \quad \frac{\partial}{\partial W_{ij}} U_i a_i$$

$$z_i = W_{i\cdot}x + b_i = \sum_{j=1}^{3} W_{ij} x_j + b_i$$

$$a_i = f(z_i)$$

$$\begin{aligned}
U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i \frac{\partial a_i}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\
&= U_i \frac{\partial f(z_i)}{\partial z_i} \frac{\partial z_i}{\partial W_{ij}} \\
&= U_i f'(z_i) \frac{\partial z_i}{\partial W_{ij}} \\
&= U_i f'(z_i) \frac{\partial W_{i\cdot}x + b_i}{\partial W_{ij}}
\end{aligned}$$



Derivative continued ...

$$
\begin{aligned}
U_i \frac{\partial}{\partial W_{ij}} a_i &= U_i f'(z_i) \frac{\partial W_{i\cdot} x + b_i}{\partial W_{ij}} \\
&= U_i f'(z_i) \frac{\partial}{\partial W_{ij}} \sum_k W_{ik} x_k \\
&= \underbrace{U_i f'(z_i)} x_j \\
&= \delta_i \quad x_j
\end{aligned}
$$

Local error signal (δ_i)

Local input signal (x_j)

where $f'(z) = f(z)(1 - f(z))$ for logistic f

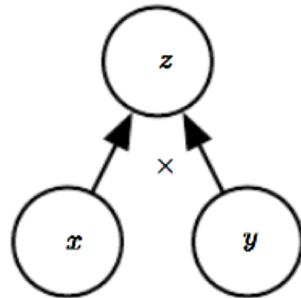
From single weight W_{ij} to full W:

$$\frac{\partial s}{\partial W_{ij}} = \delta_i x_j$$

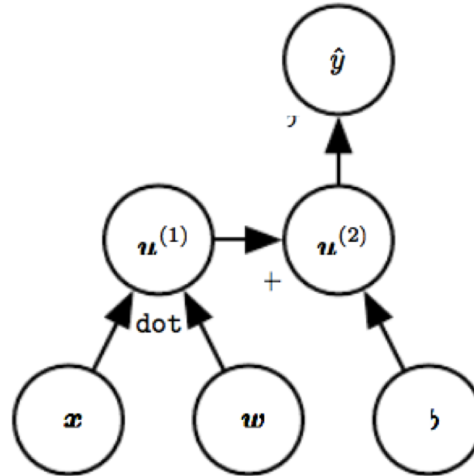
- We want all combinations of $i = 1, 2, \ldots$ and $j = 1, 2, 3, \ldots$
- Solution: Outer product

$$\frac{\partial J}{\partial W} = \delta x^T$$

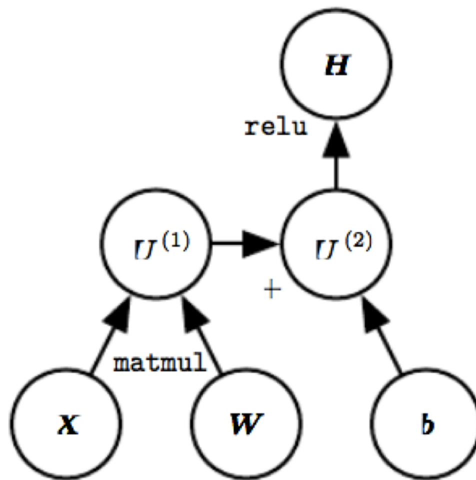
Computation Graphs

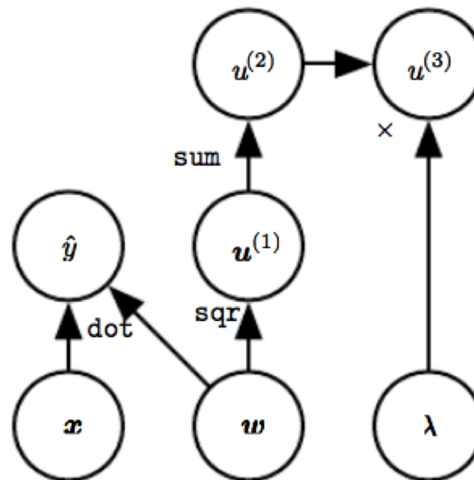


(a)



(b)





AUTOENCODERS

Autoencoders

Autoencoders

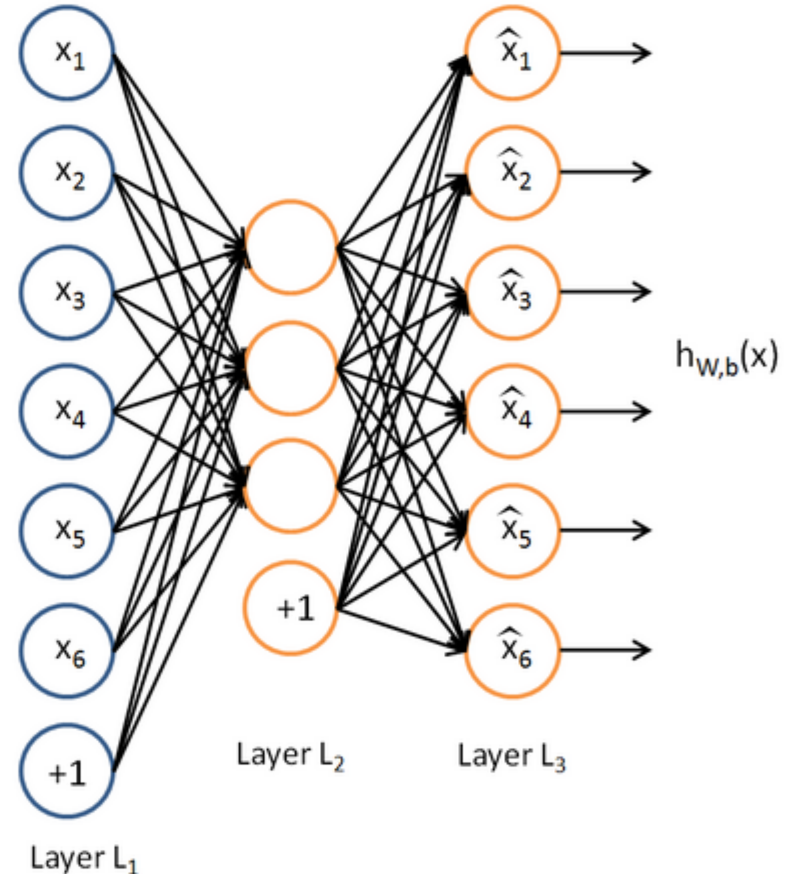
- Unsupervised Learning Algorithm
- Given an input *x*, we learn a *compressed* representation of the input, which we then try to reconstruct
- In the simpliest form: Feed forward network with hidden size < input size.
- We then search for parameters such that:

$$\hat{x} \approx x$$

  for all training examples
- The error function is:

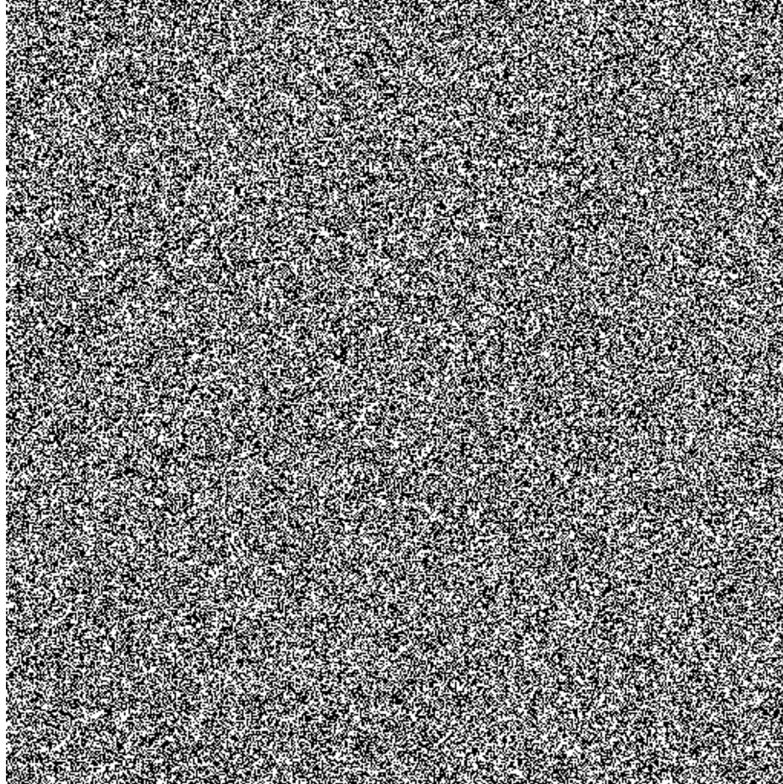
$$E(x, W, b) = ||\hat{x} - x||_2$$

- Once we finished training, we are interested in the compressed representation, i.e. the values of the hidden units



Source: http://ufldl.stanford.edu/wiki/index.php/Autoencoders_and_Sparsity

Why would we use autoencoders?



- How does a randomly generated image look like?

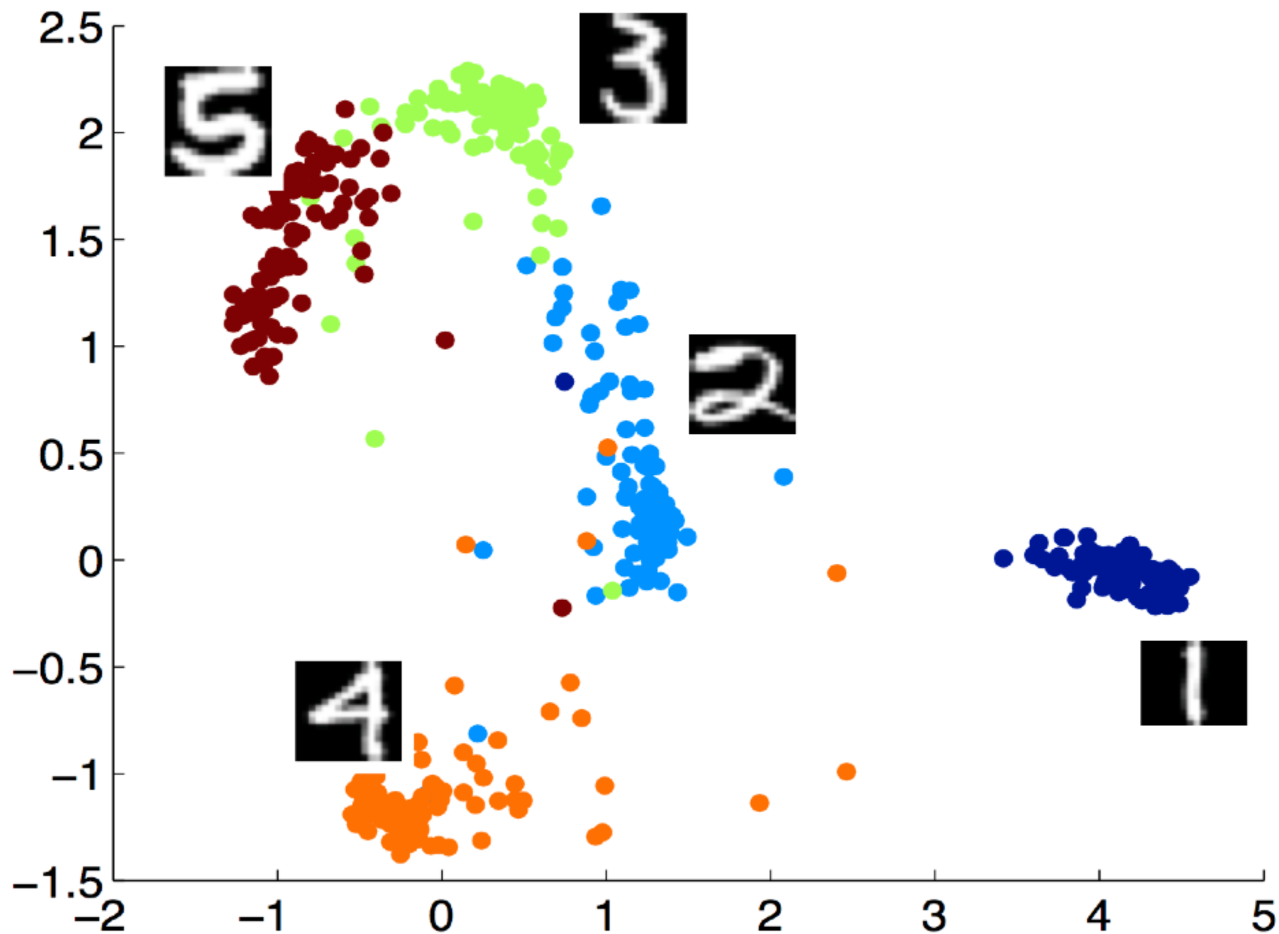
Why would we use autoencoders?

- What would be the probability to get an image like this from random sampling?

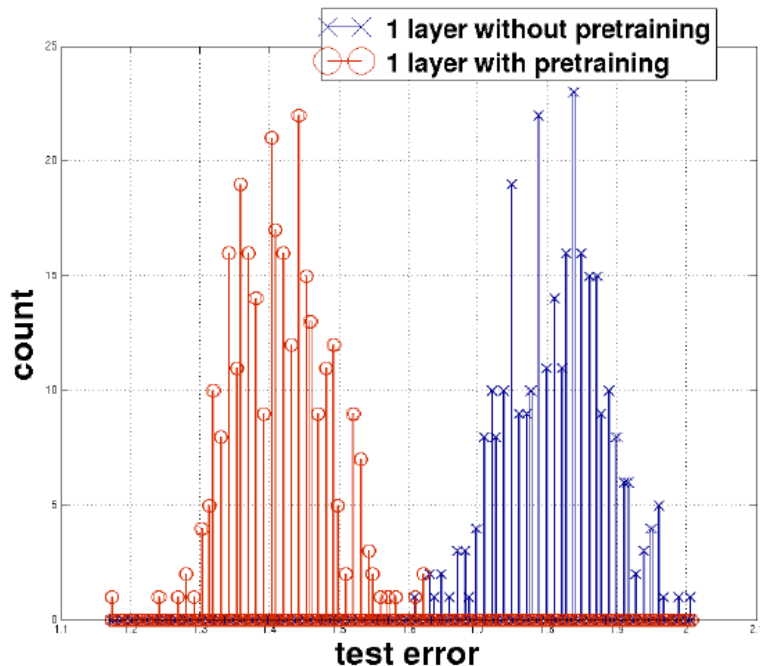
Why would we use autoencoders?

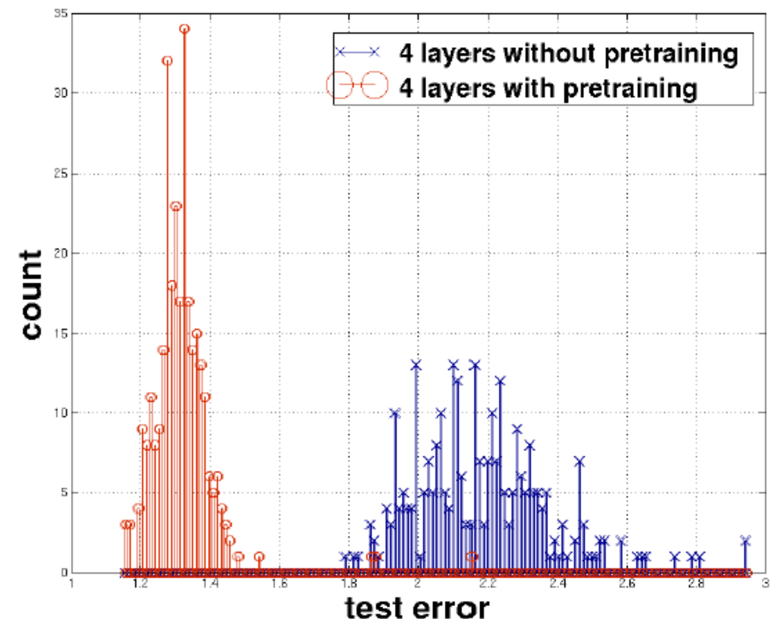
- Produce a compressed representation of a high-dimensional input (for example images)
- The compression is lossy. Learning drives the encoder to be a good compression in particular for training examples
- For random input, the reconstruction error will be high
- The autoencoder learns to abstract properties from the input. What defines a natural image? Color gradients, straight lines, edges etc.
- The abstract representation of the input can make a further classification task much easier

Dimension-Reduction can simplify classifcation tasks – MNIST Task



Dimension-Reduction can simplify classifcation tasks – MNIST Task





- Histogram-plot of test error on the MNIST hand written digit recognition.
- Comparison of neural network with and without pretraining

Source: Erhan et al, 2010, Why Does Unsupervised Pre-training Help Deep Learning?

Autoencoders vs. PCA

- Principal component analysis (PCA) converts a set of correlated variables to a set of linearly uncorrelated variables called *principal components*
- PCA is a standard method to break down high-dimensional vector spaces, e.g. for information extraction or visualization
- However, PCA can only capture linear correlations

PCA

Encoder:

$$f_\theta(x) = Wx$$

Decoder:

$$g_\theta(y) = W'y$$

Autoencoders

Encoder:

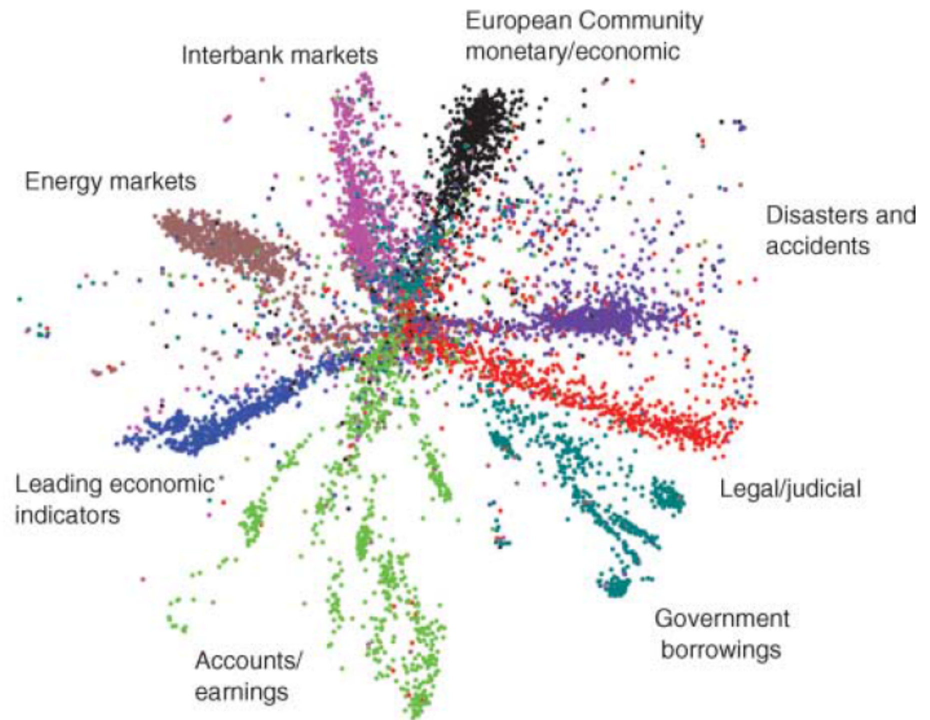
$$f_\theta(\mathbf{x}) = s(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

Decoder:

$$g_{\theta'}(\mathbf{y}) = s(\mathbf{W}'\mathbf{y} + \mathbf{b}'),$$

Autoencoders vs. PCA - Example

- Articles from Reuter corpus were mapped to a 2000 dimensional vector, using the 2000 most common word stems



Deep Autoencoder

Source: Hinton et al., Reducing the Dimensionality of Data with Neural Networks

How to ensure the encodes does *not* learn the identity function?

Identify Function

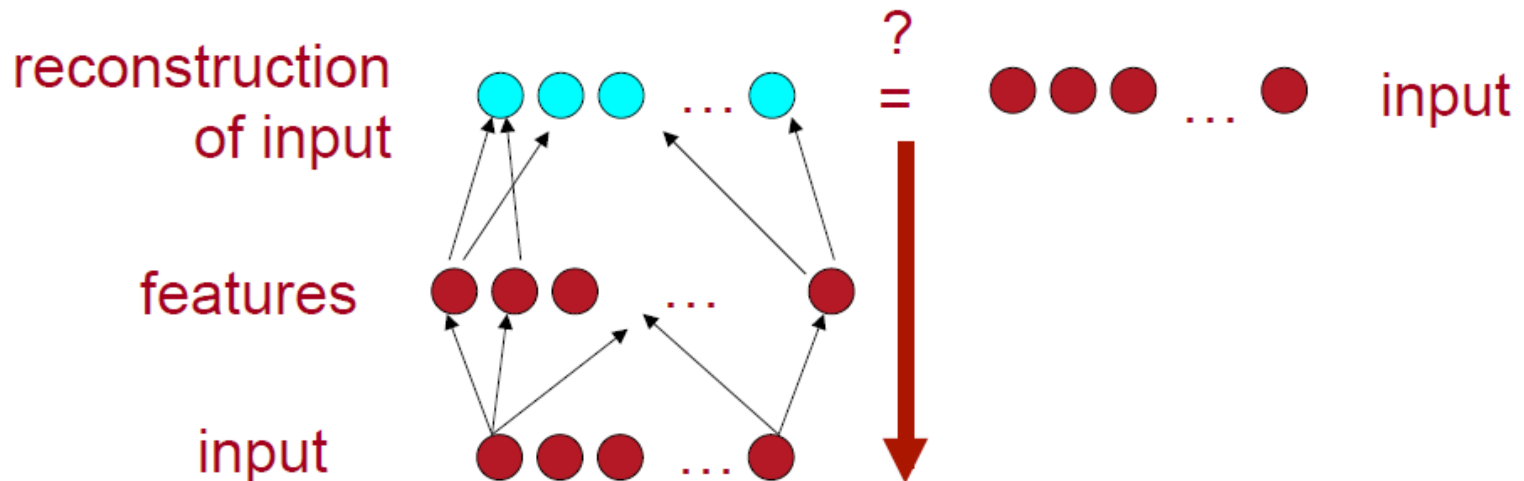
- Learning the identity function would not be helpful
- Different approaches to ensure this:
  - Bottleneck constraint: The hidden layer is (much) smaller than the input layer
  - Sparse coding: Forcing many hidden units to be zero or near zero
  - Denoising encoder: Add randomness to the input and/or the hidden values

Denoising Encoder

- Create some random noise ε
- Compute $\hat{x} = f(x + \varepsilon)$
- Reconstruction Error: $\hat{x} \approx x?$
- Alternatively: Set some of the neurons (e.g. 50%) to zero
- The noise forces the hidden layer to learn more robust features

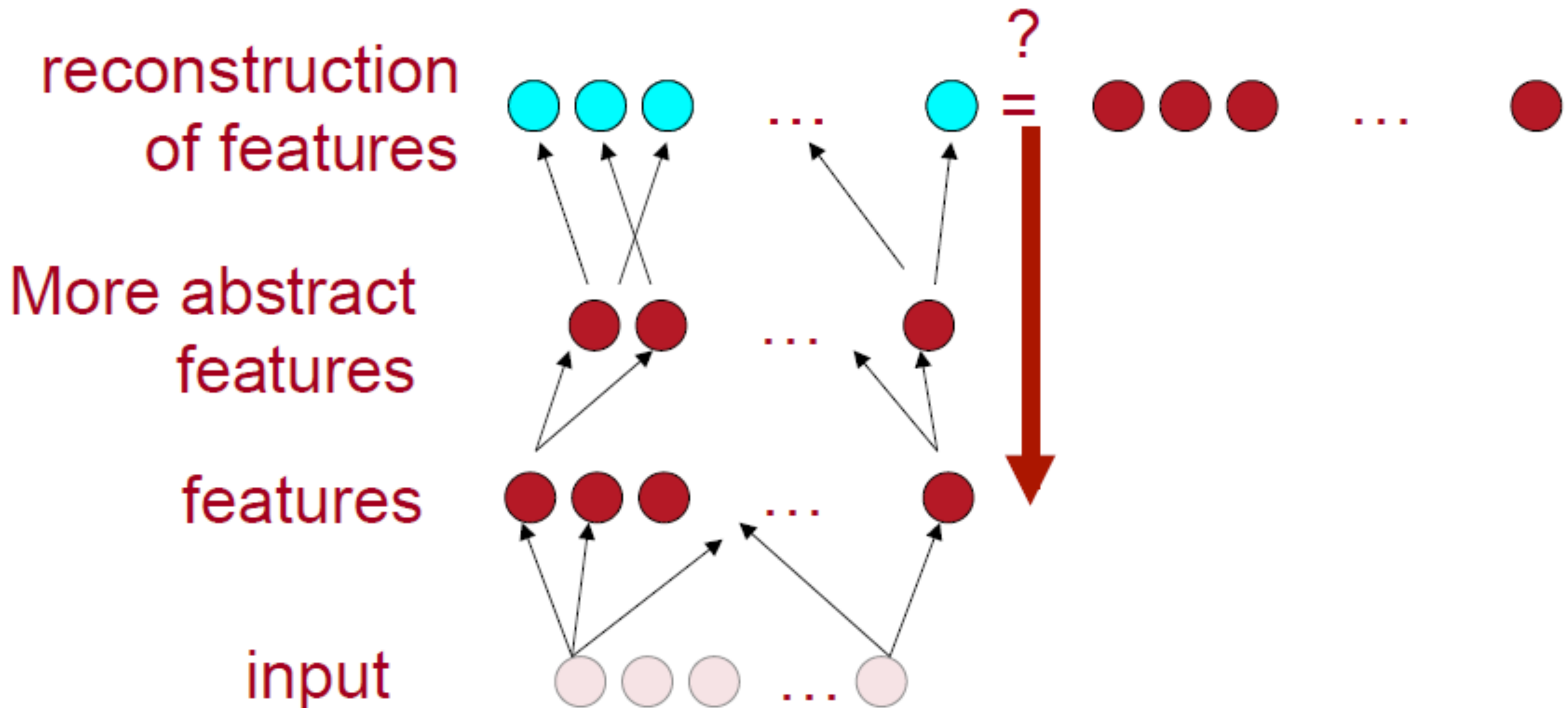
Stacking Autoencoders

- We can stack multiple hidden layers to create a deep autoencoder
- These are especially suitable for highly non-linear tasks
- The layers are trained layer-wise – one at a time



Step 1: Train single layer autoencoder until convergence

Stacking Autoencoders



Step 2: Add additional hidden layer and train this layer by trying to reconstruct the output of the previous hidden layer. Previous layers are will not be changed. Error function: $||\hat{h}_1 - h_1||_2$

Stacking Autoencoders – Fine-tuning

- After pretraining all hidden layers, the deep autoencoder is fine-tuned

Unsupervised Fine-Tuning:

- Apply back propagation to the complete deep autoencoder
- Error-Function:

$$E(x, W^{(1)}, W^{(2)}, \ldots) = ||\hat{x} - x||_2$$

- Further details, see Hinton et al.
- *(It appears that supervised fine-tuning is more common nowadays)*

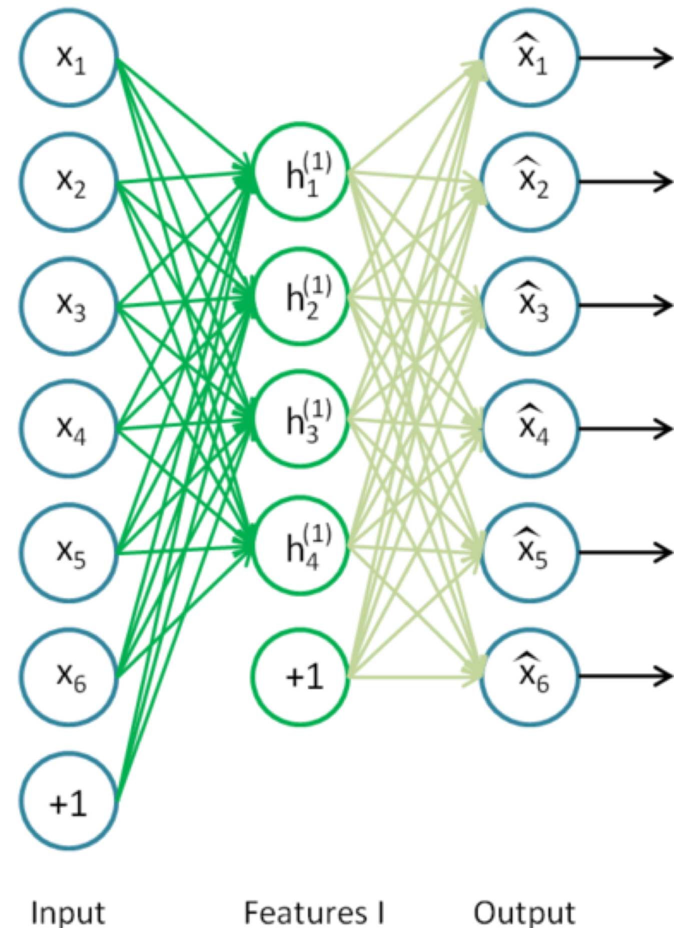
Supervised Fine-Tuning:

- Use your classification task to fine-tune your autoencoders
- A softmax-layer is added after the last hidden layer
- Weights are tuned by using back prograpagtion.
- See next slides for an example or http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders

Stacking Autoencoders - Example

Pretrain first autoencoder

- Train an autoencoder to get the first weight matrix $W^{(1)}$ and first bias vector $b^{(1)}$
- The second weight matrix, connecting the hidden and the output units, will be disregarded after the first pretraining step
- Stop after a certain number of iterations

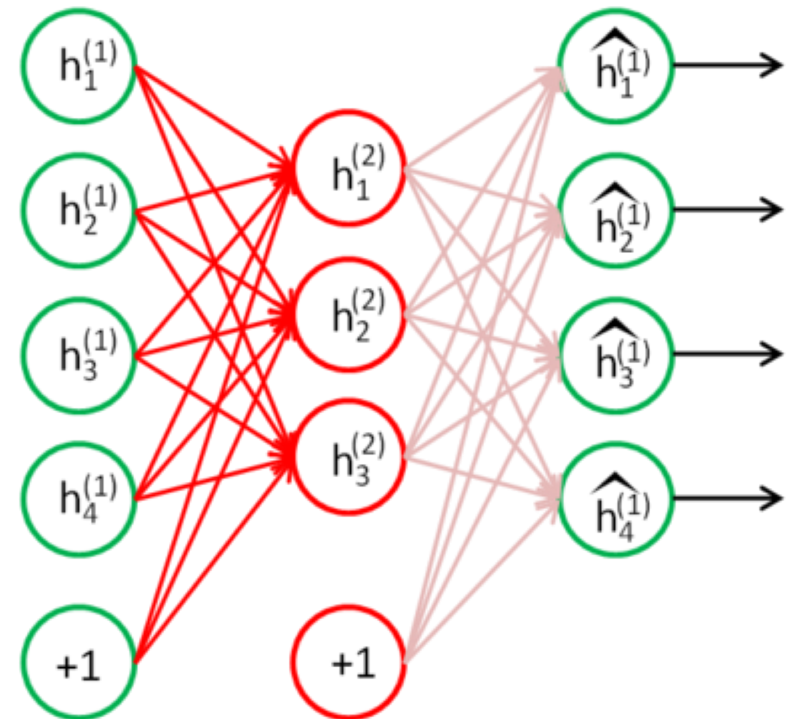


Source: http://ufldl.stanford. edu/wiki/

Stacking Autoencoders - Example

Pretrain second autoencoder

- Use the values of the previous hidden units as input for the next autoencoder.
- Train as before

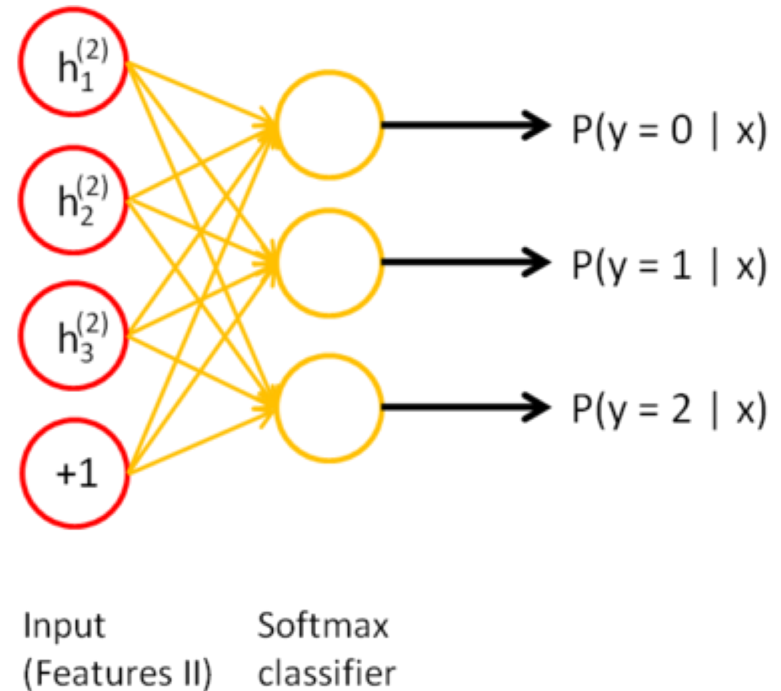


Source: http://ufldl.stanford. edu/wiki/

Stacking Autoencoders - Example

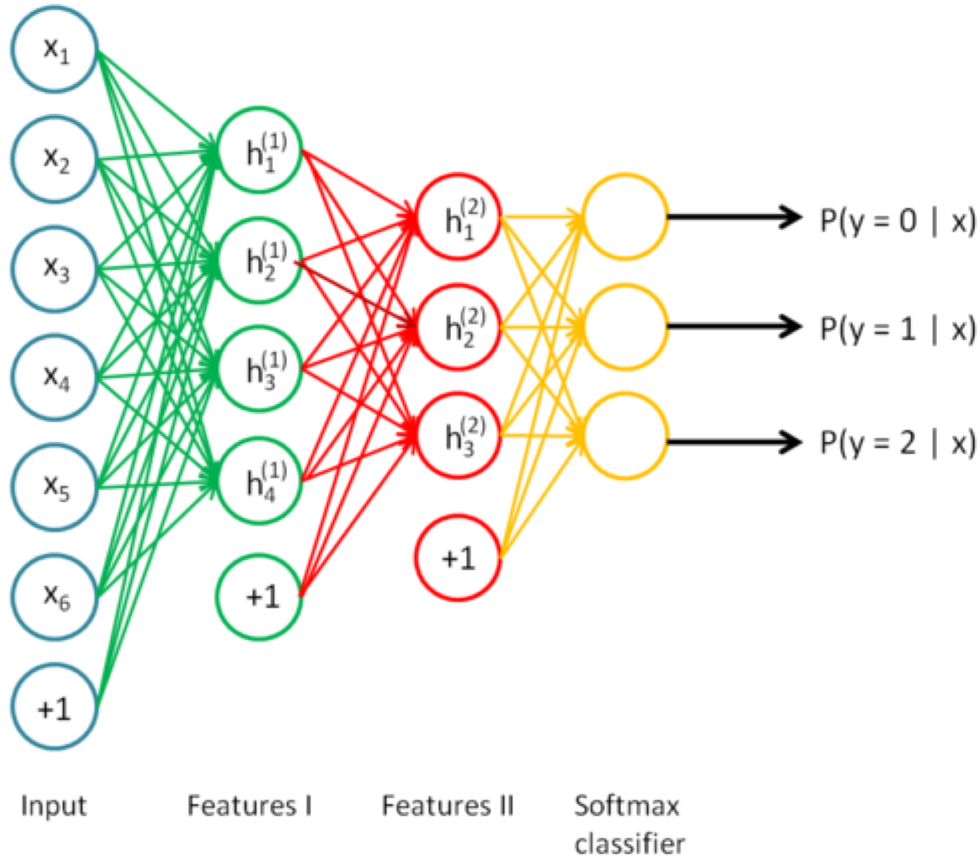
Pretrain softmax layer

- After second pretraining finishes, add a softmax layer for your classification task
- Pretrain this layer using back propagation



Source: http://ufldl.stanford. edu/wiki/

Stacking Autoencoders - Example



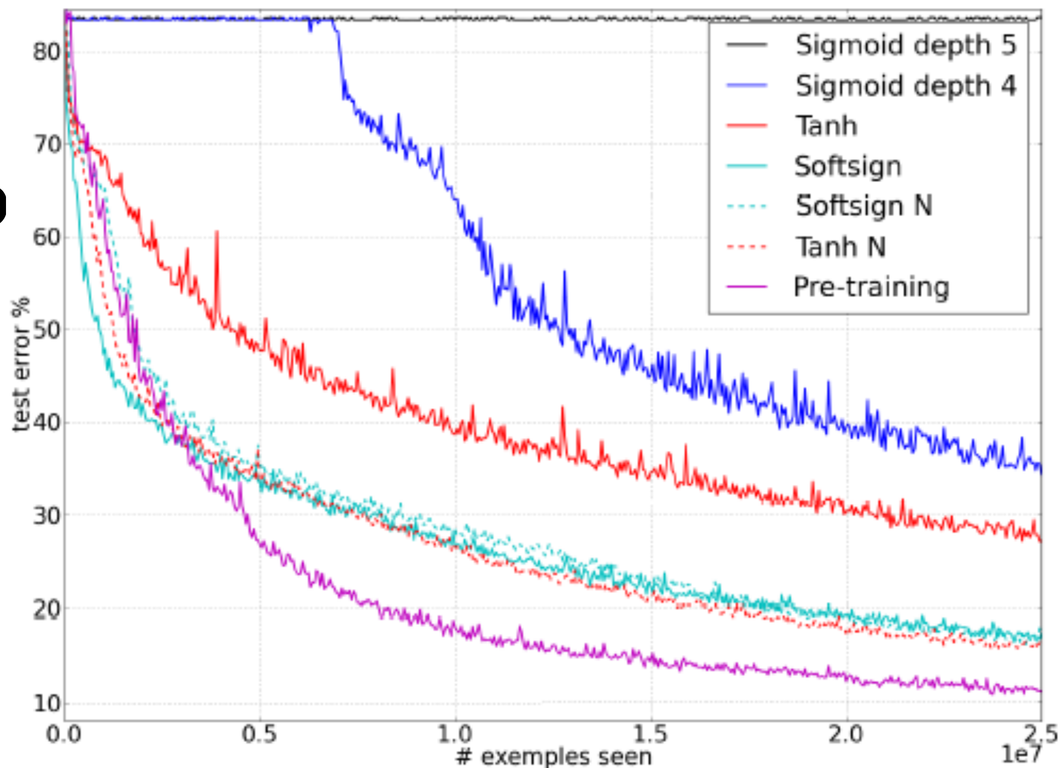
Fine-tuning

- Plug all layers together
- Compute the costs based on the actual input x
- Update **all** weights using backpropagation

Source: http://ufldl.stanford. edu/wiki/

Is pre-training really necessary?

- Xavier Glorot and Yoshua Bengio, 2010, *Understanding the difficulty of training deep feedforward neural networks*
- With the ri initializatio ng decreases



Is pre-training really necessary?

- Pre-training achieves two things:
  - It makes optimization easier
  - It reduces overfitting
- Pre-training is not required to make optimization work, if you have enough data
  - Mainly due to a better understanding how initialization works
- Pre-training is still very effective on small datasets

– More information: https://www.youtube.com/watch?v=vShMxxqtDDs

Dropout in Neural Networks

Inspired by Hinton
https://www.youtube.com/watch?v=vShMxxqtDDs

For details:
Srivastava, Hinton et al., 2014, *Dropout: A Simple Way to Prevent Neural Networks from Overtting*

Ensemble Learning

- Create many different models and combine them at test time to make prediction

- Averaging over different models is very effective against overfitting

- Random Forest
  - A single decision trees is not very powerful
  - Creating hundreds of different trees and combine them

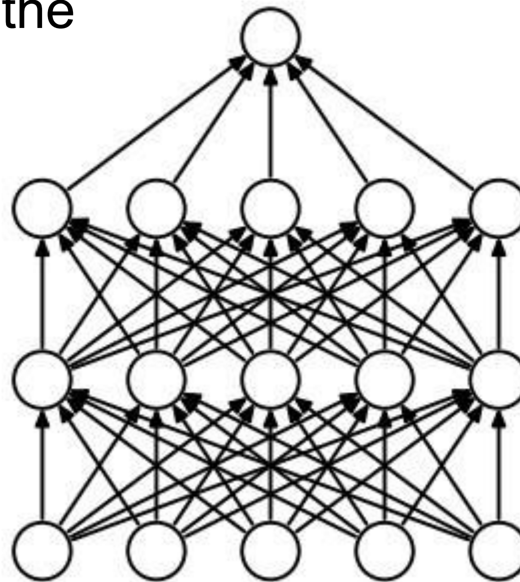
- Random forests works really well
  - Several Kaggle competitions, e.g. Netflix, were won by random forests

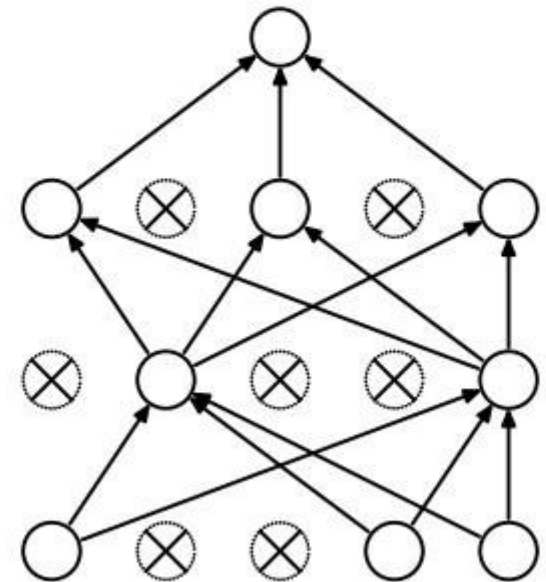
Model Averaging with Neural Nets

- We would like to do massive model averaging
  - Average over 100, 1.000, 10.000 or 100.000 models
- Each net takes a long time to train
  - We don’t have enough time to learn so many models
- At test time, we don’t want to run lots of large neural nets
- We need something that is more efficient
  - Use dropouts!

Dropout

- Each time present a training example, we dropout 50% of the hidden units
- With this, we randomly sample over 2^H different architectures
  - H: Number of hidden units
- All architectures share the same weights


(a) Standard Neural Net


(b) After applying dropout.

Img source: http://cs231n.github.io/

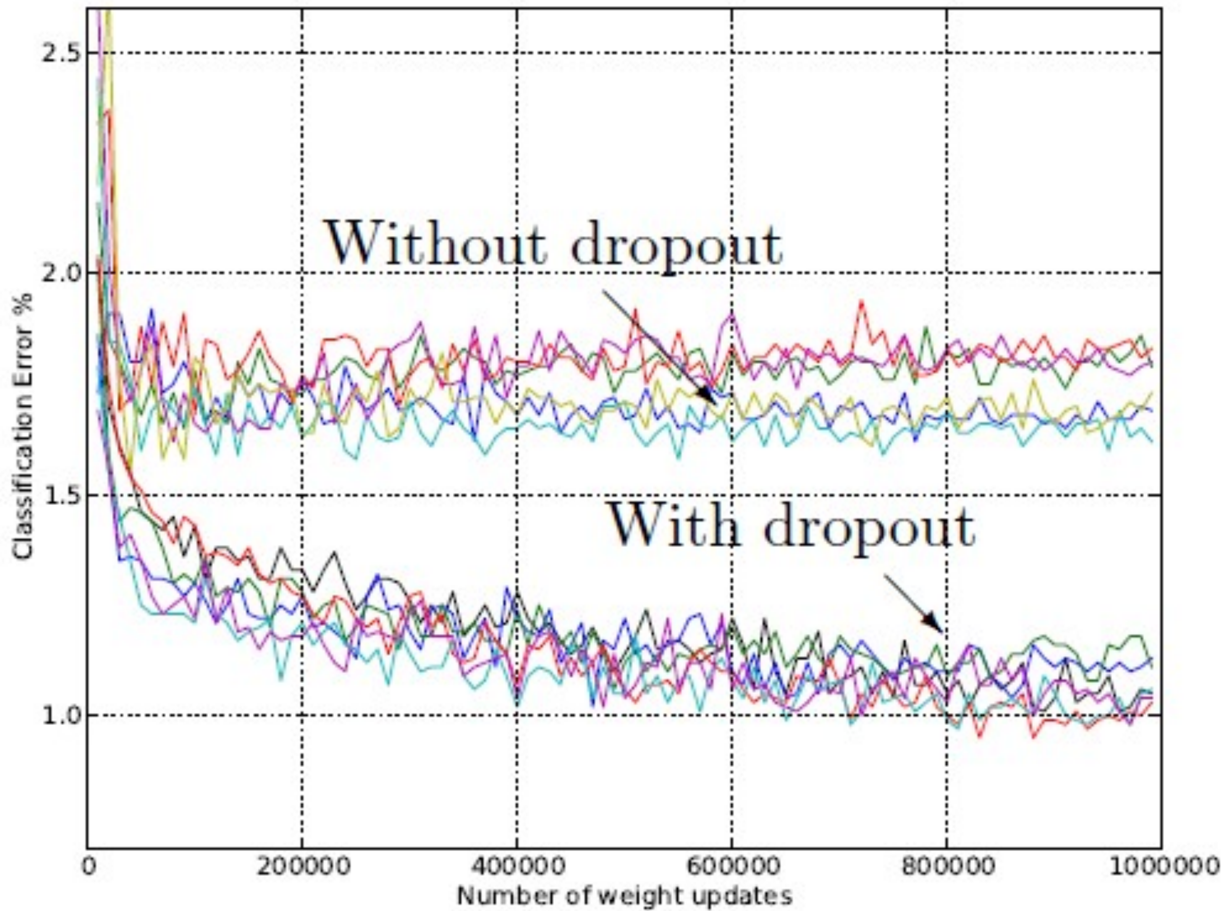
Dropout

- With H hidden units, we sample from 2^H different models
  - Only few of the models get ever trained and they only get 1 training example
- Sharing of weights means that every model is strongly regularized
  - Much better than L1 and L2 regularization, which pulls weights towards zero
  - It pulls weights towards what other models need
  - Weights are pulled towards sensible values
- This works in experiments extremely well

Dropout – at test time

- We could sample many different architectures and average the output
  - This would be way too slow
- Instead: Use all hidden units and half their outgoing weights
  - Computes the geometric mean of the prediction of all 2^H models
  - We can use other dropout rates than p=0.5. At test time, multiply weights by 1-p
- Using this trick, we train and use trillions of “different” models
- For the input layer:
  - We could apply dropout also to the input layer
  - The probability should be then smaller than 0.5
  - This is known as denoising autoencoder
  - Currently this cannot be implemented in out-of-the-box Keras

How well does dropout work?



Classification error on MNIST dataset

Source: Srivastava et al, 2014, Drouput A Simple Way to Prevent Neural Networks from Overtting

How well does dropout work?

- If your deep neural network is significantly overfitting, dropout will reduce the number of errors a lot

- If your deep neural network is not overfitting, you should be using a bigger one
  - Our brain: #parameters >> #experiences
  - Synapses are much cheaper then experiences

Another way to think about Dropout

- In a fully connected neural network, a hidden unit knows which other hidden units are present
  - The hidden unit co-adapt with them for the training data
  - But big, complex conspiracies are not robust -> they fail at test time

- In the dropout scenario, each unit has to work with different sets of co-workers
  - It is likely that the hidden unit does something individually useful
  - It still tries to be different from its co-workers