MSIS549: ML and AI for Business Applications
Lesson 5

Course Outline

Lesson 1: Introduction to Deep Learning

Lesson 2: Neural Network Fundamentals

Lesson 3: Convolutional Neural Networks

Lesson 4: Recurrent Neural Networks

Lesson 5: Deep Learning Practice

Course Evaluation

https://uw.iasystem.org/survey/222884

Foster School of Business

Recap

Deep Learning, ML and AI

Artificial Intelligence

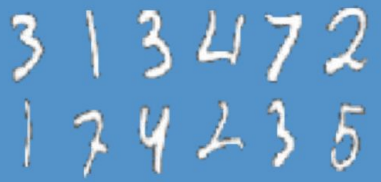
Any technique that enables computers to mimic human behavior

Machine Learning

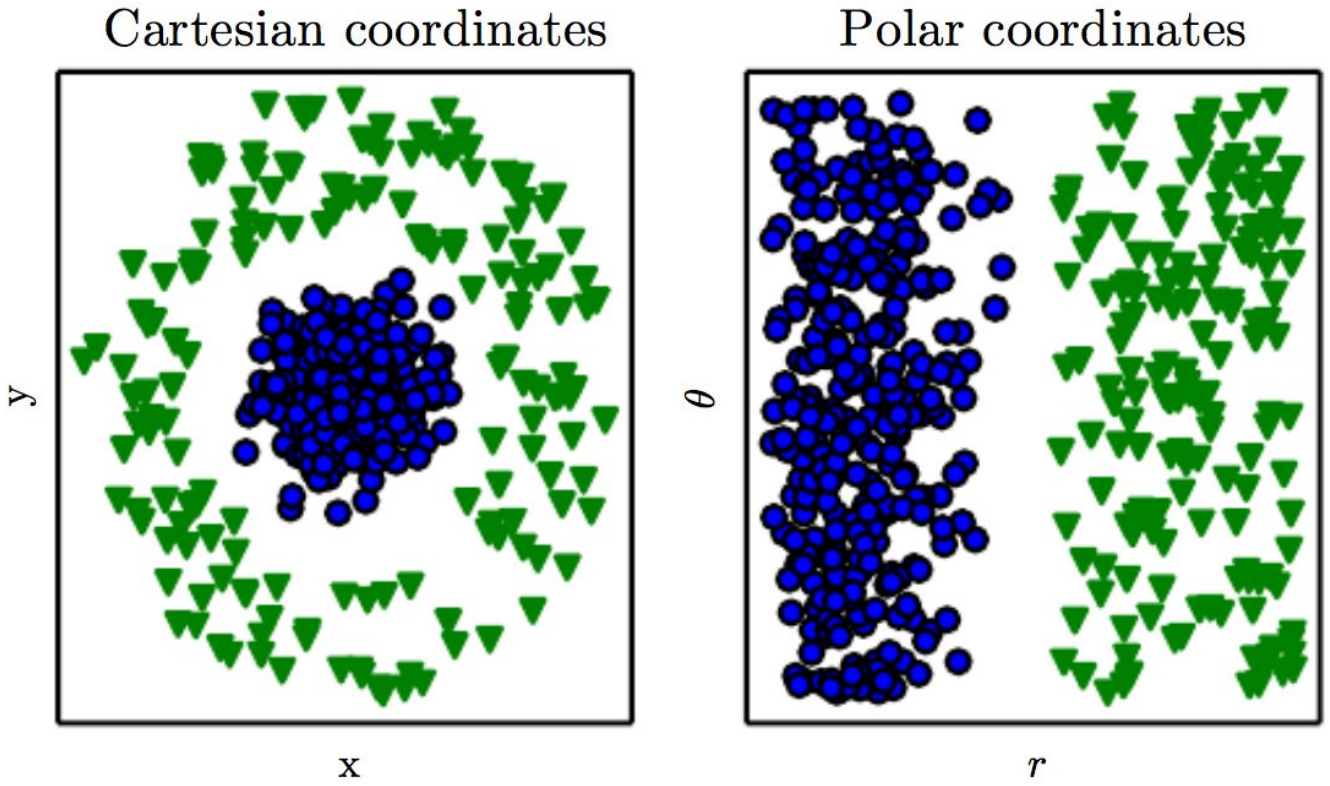
Ability to learn without explicitly being programmed

Deep Learning

Extract patterns from data using neural networks

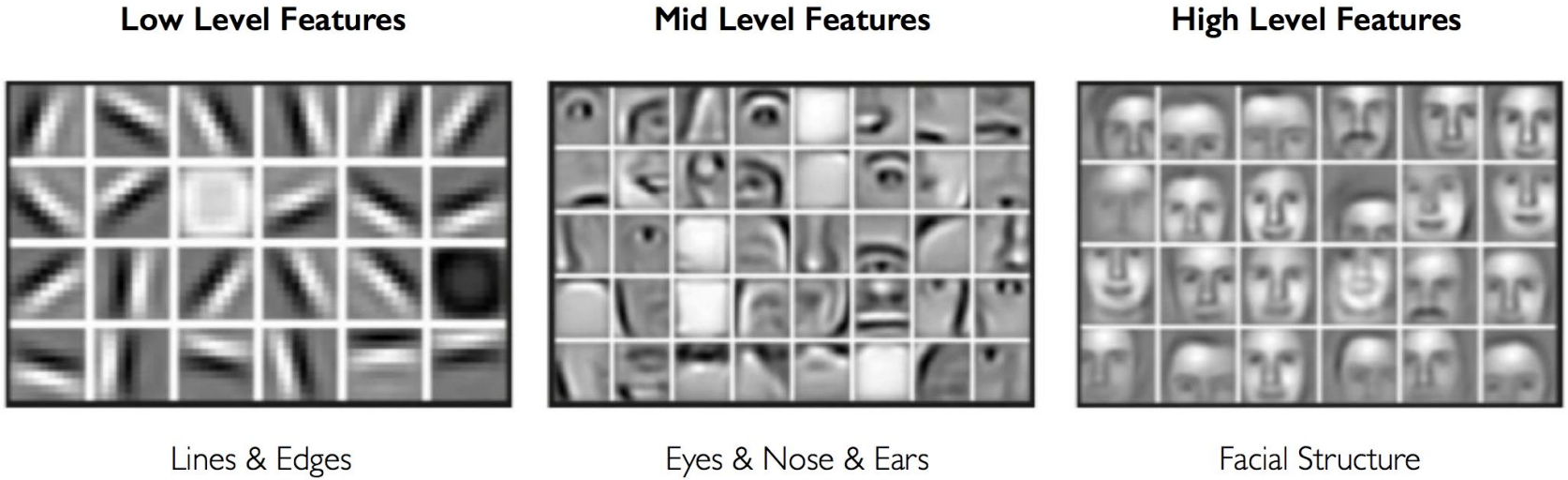


Feature Representation Matters A LOT

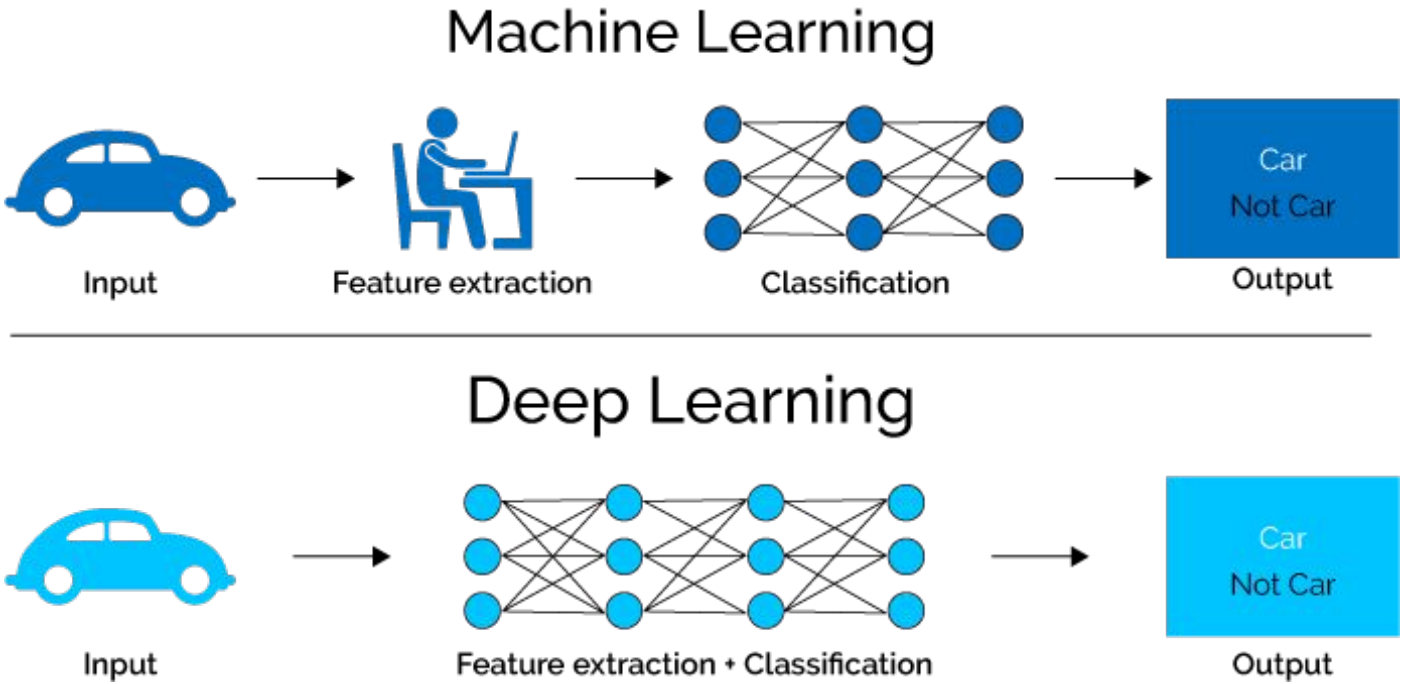


Can we learn the representation?

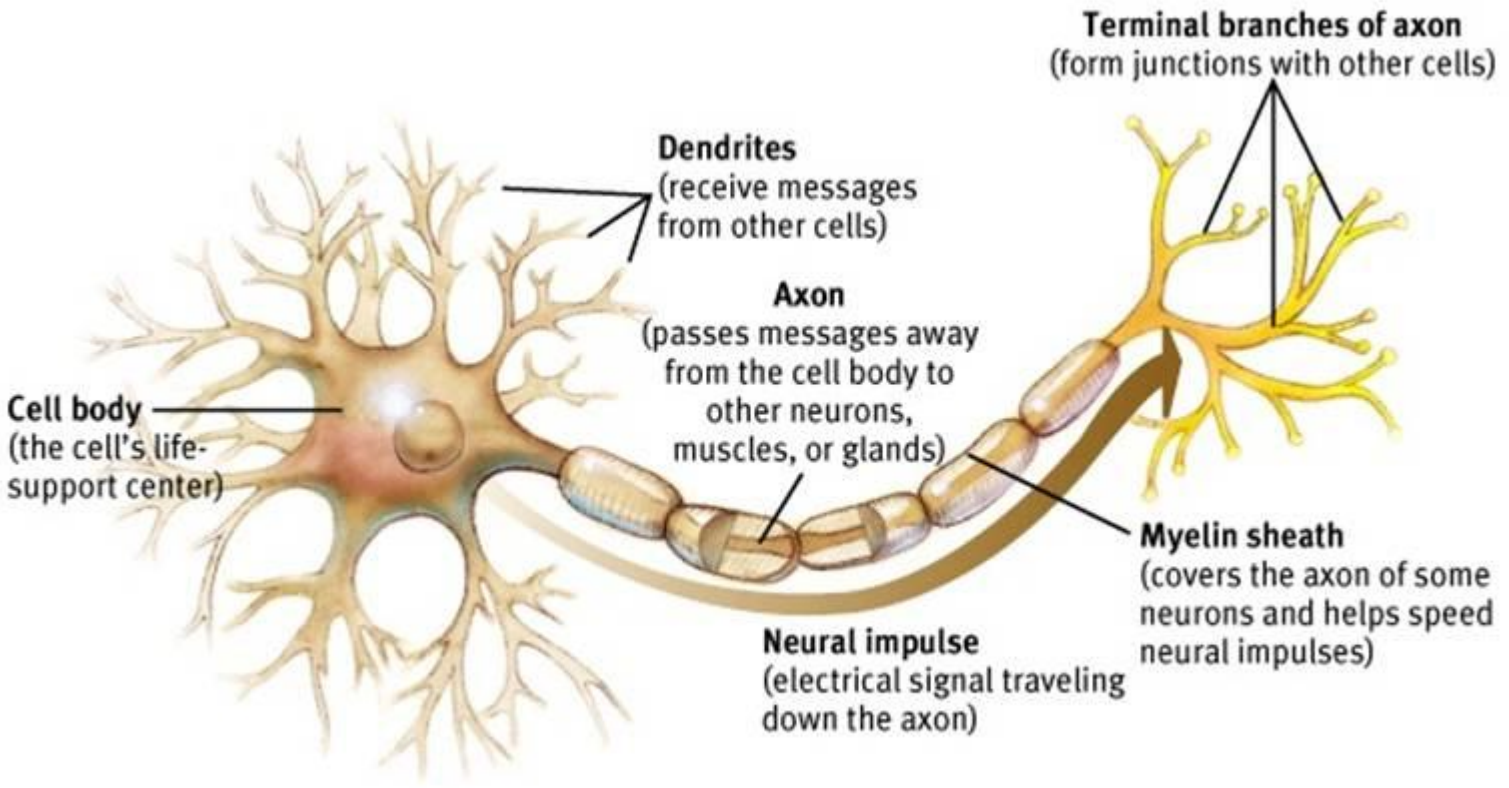
Hand engineered features are time consuming, brittle and not scalable in practice. Can we learn the underlying features directly from data?



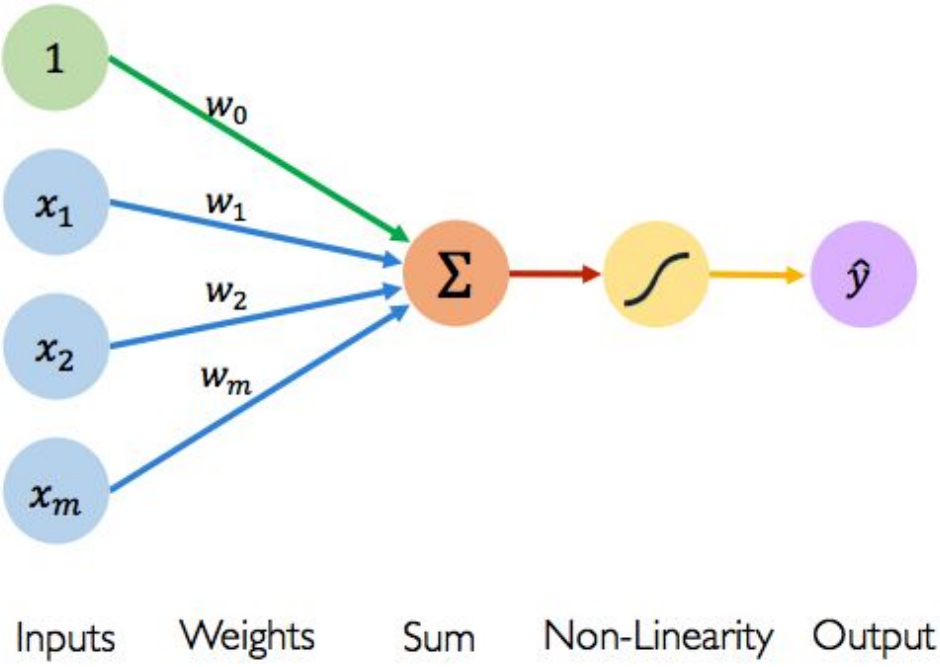
End-to-end learning

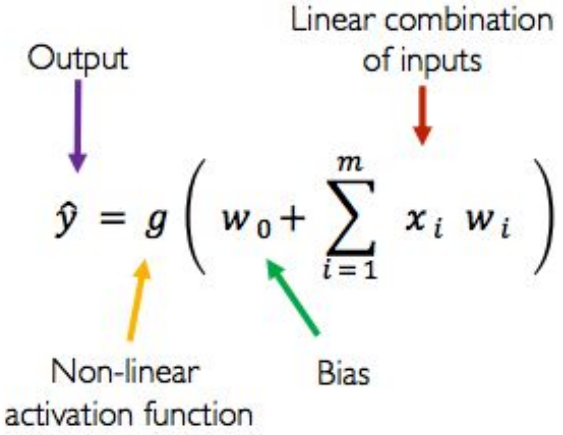


Introduction and History of Neural Networks



The Perceptron: Forward Propagation

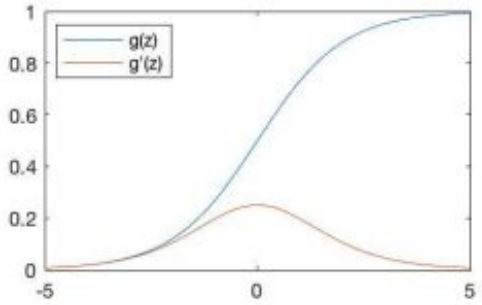




Foster School of Business

Common Activation Functions

Sigmoid Function

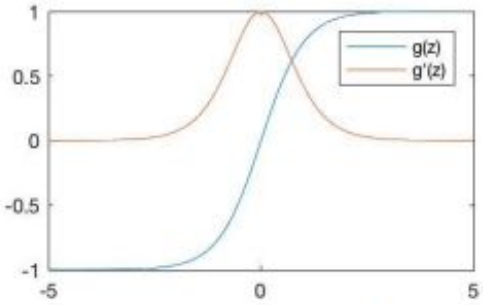


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

tf.nn.sigmoid(z)

Hyperbolic Tangent

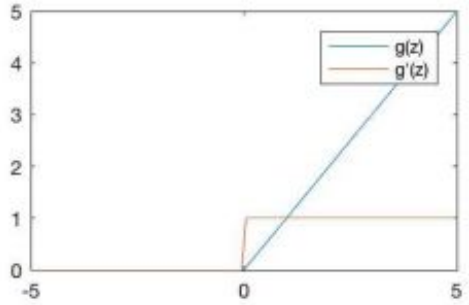


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

tf.nn.tanh(z)

Rectified Linear Unit (ReLU)

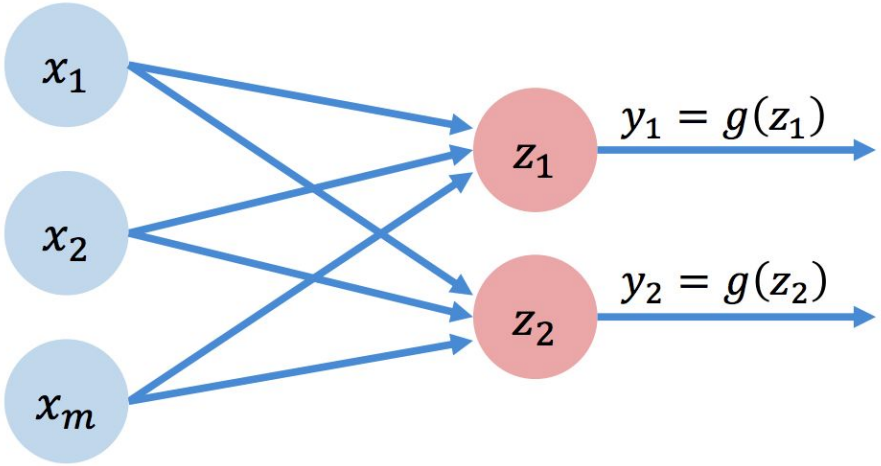


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

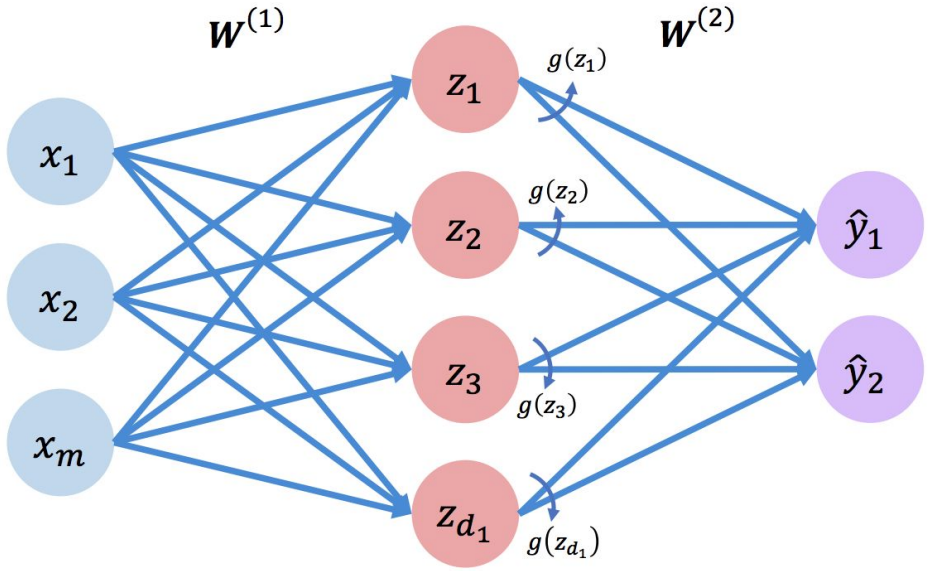
tf.nn.relu(z)

Build Neural Networks with Perceptrons



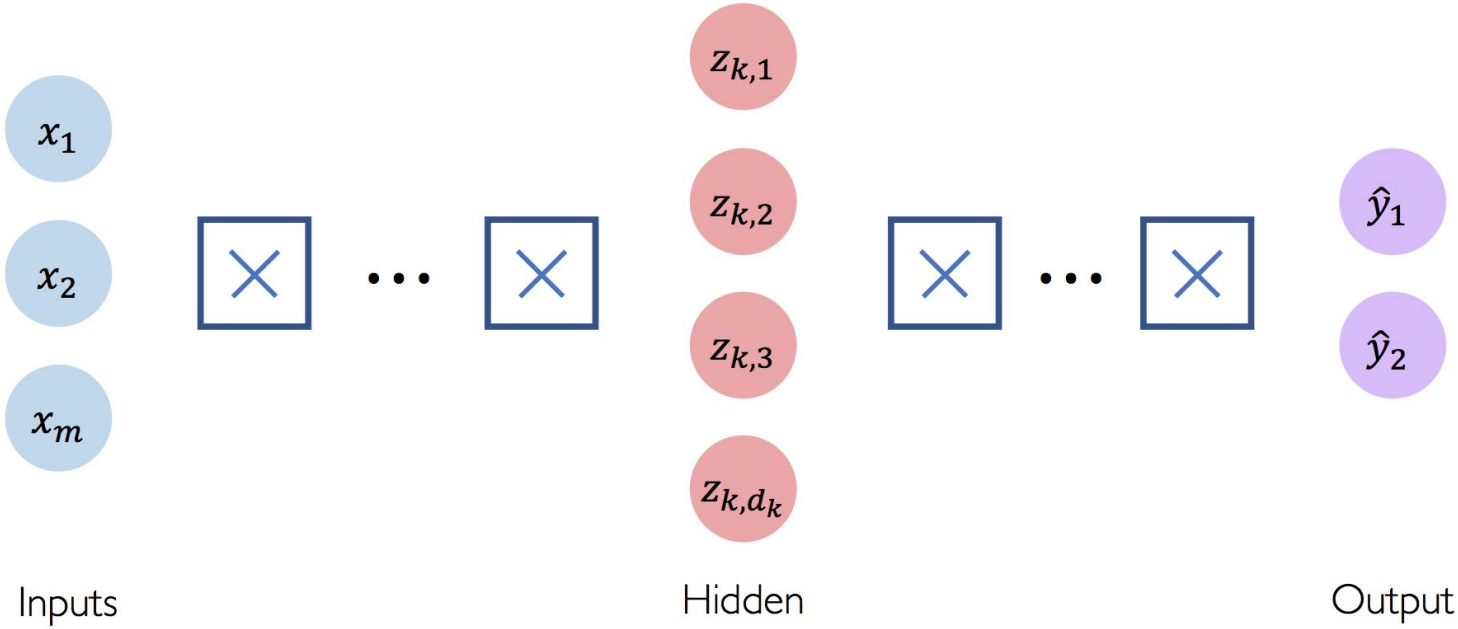
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Build Neural Networks with Perceptrons



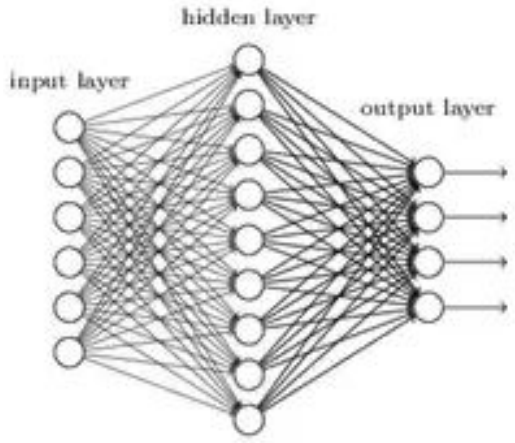
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

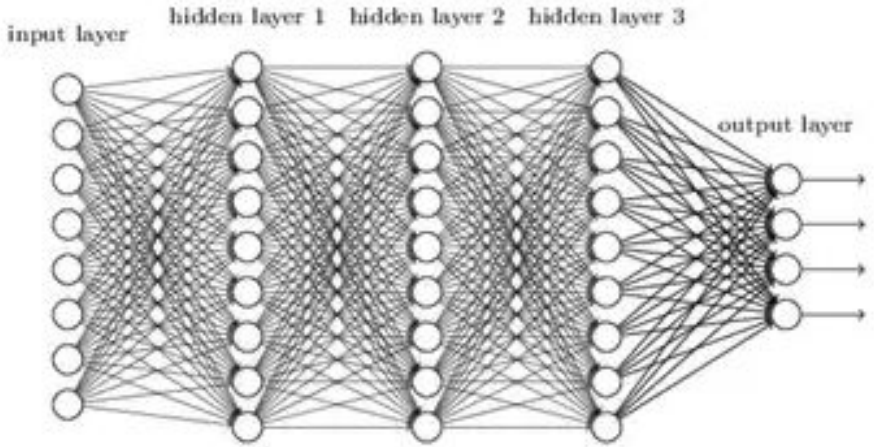
Build Neural Networks with Perceptrons



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Shallow vs. Deep Networks

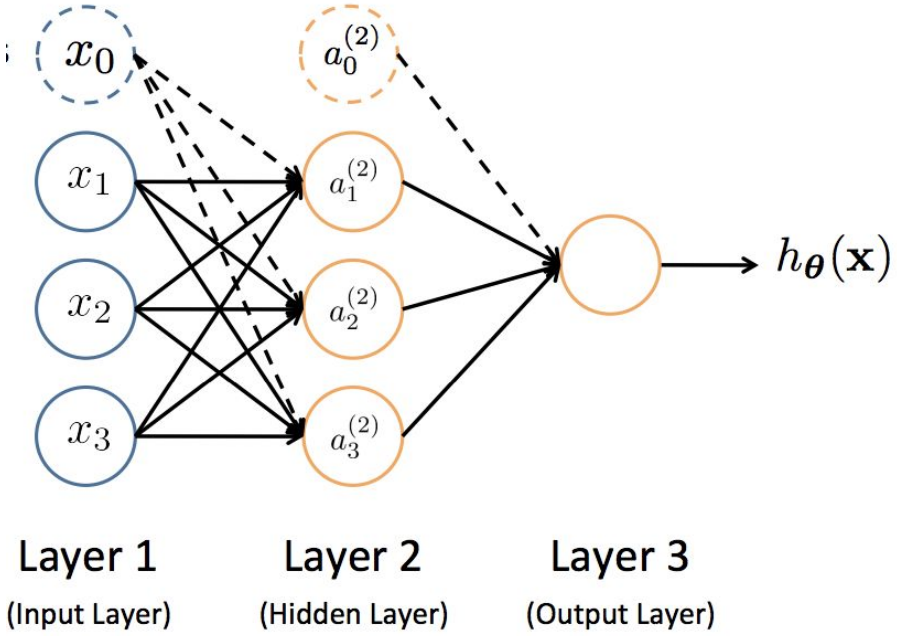




Advantages of Deep Networks:

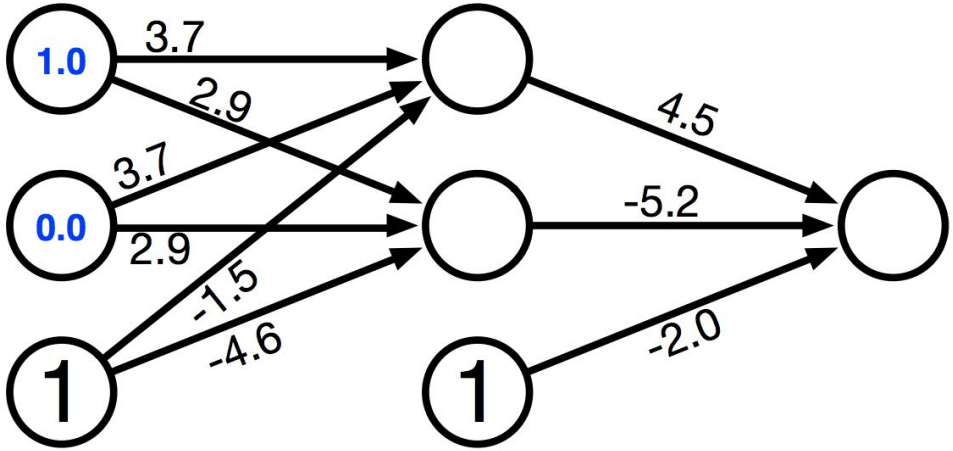
- Deep representation allows for a hierarchy of representations.
- Very wide and shallow networks are very good at memorization while deep networks are better at generalization.
- Often outperform shallow ones with the same computational power.

Universal Approximation Theorem



The universal approximation theorem found that a neural network with one hidden layer can approximate any continuous function (George Cybenko, 1989).

Neural Network Example

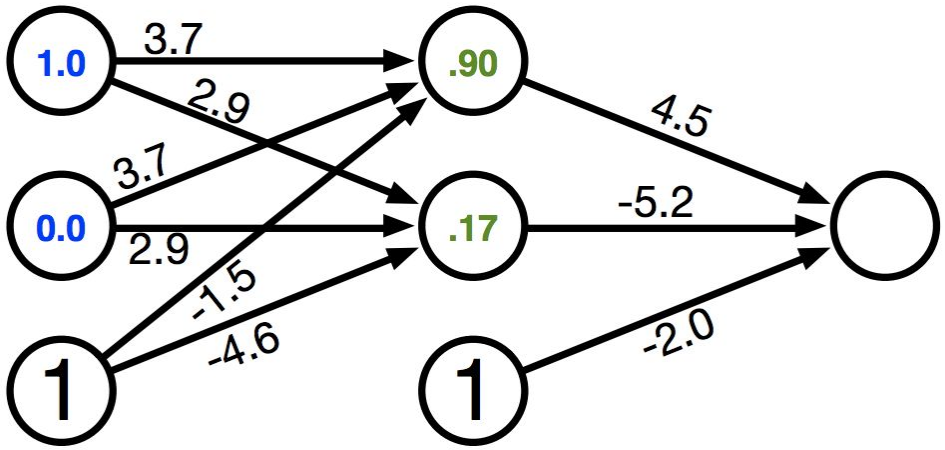


Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$

Neural Network Example

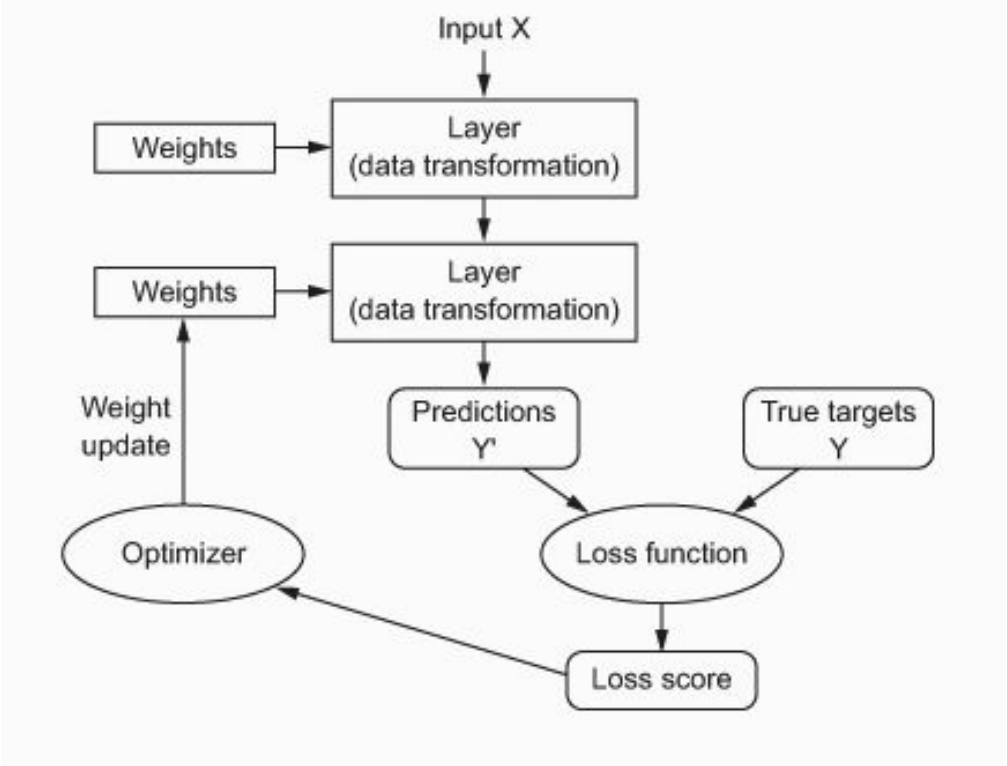


Output computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Training Neural Networks

Learning means finding a combination of model parameters that minimizes a loss function for a given set of training data samples and their corresponding targets.



Loss Optimization

We want to find the network weights that achieve the lowest loss.

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:

$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \ldots\right\}$$

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

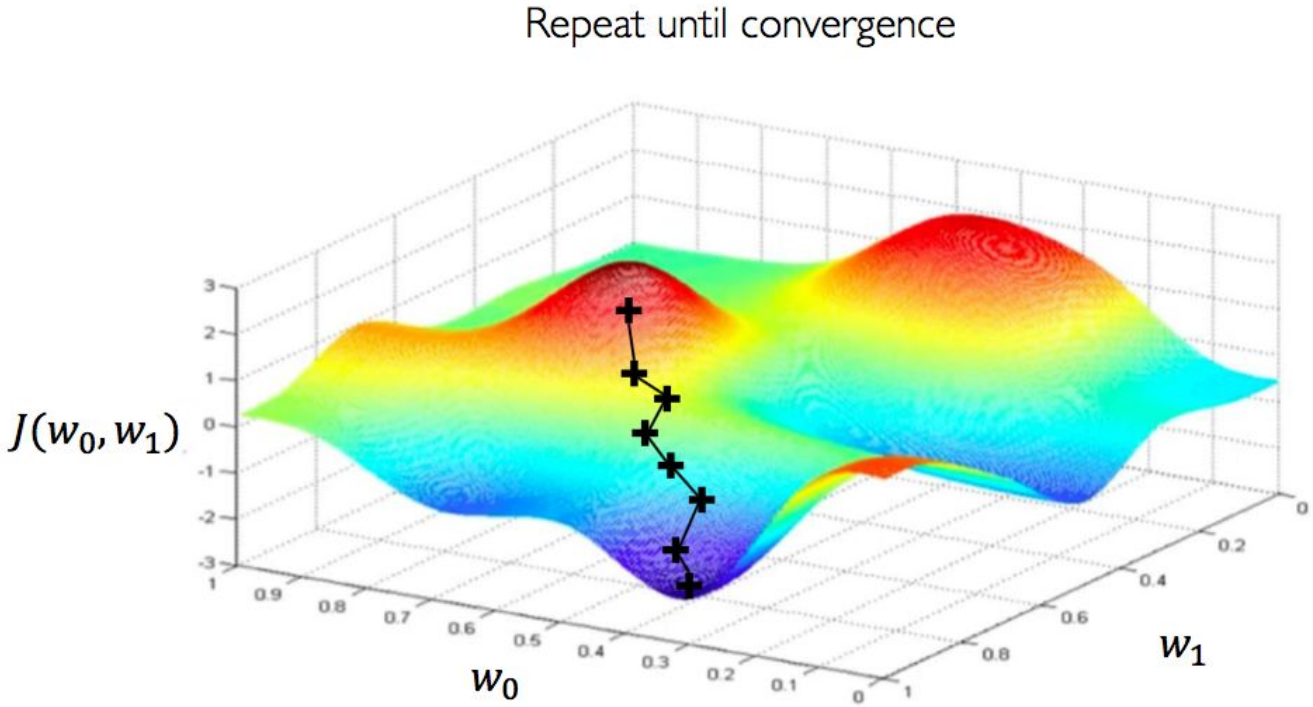
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

Loss Optimization

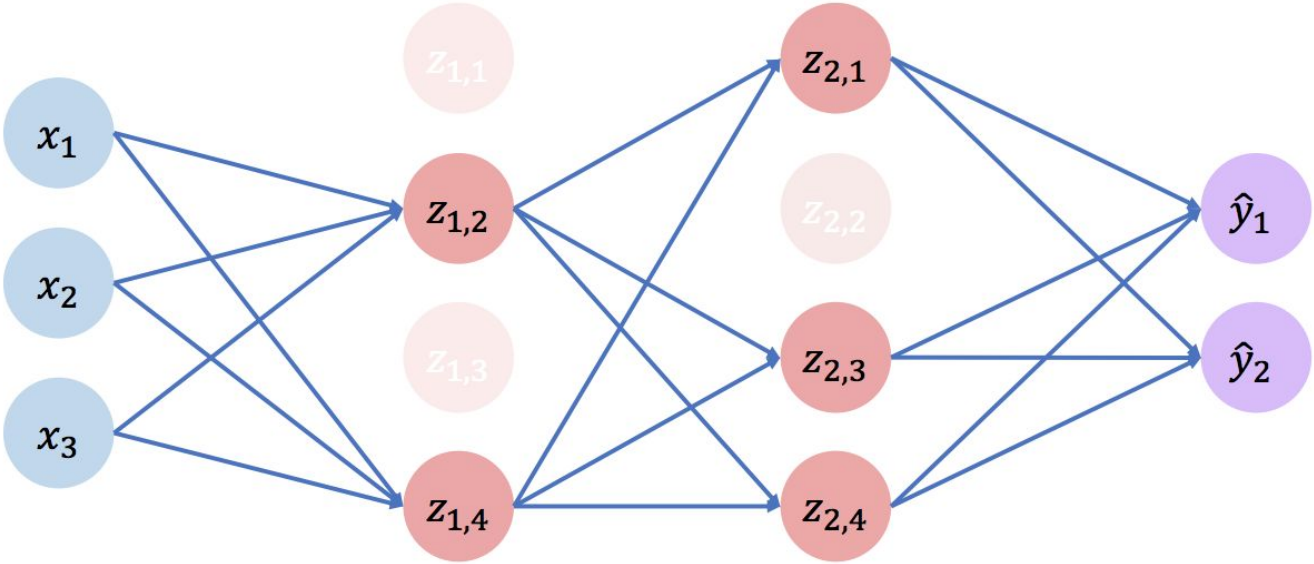
Find the network weights that achieve the lowest loss.



Regularization: Dropout

During training, randomly set some activations to 0

- Typically ‘drop’ 50% of activations in layer
- Forces network to not rely on any 1 node



Regularization: Early Stopping

Stop training before we have a chance to overfit

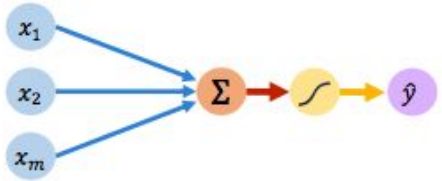
Last-layer Activation and Loss Function

Problem type	Last-layer activation	Loss function
Binary classification	sigmoid	binary_crossentropy
Multiclass, single-label classification	softmax	categorical_crossentropy
Multiclass, multilabel classification	sigmoid	binary_crossentropy
Regression to arbitrary values	None	mse
Regression to values between 0 and 1	sigmoid	mse or binary_crossentropy

Lecture Recap

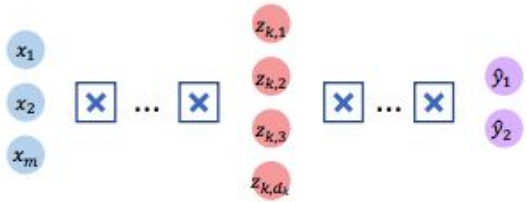
The Perceptron

- Structural building blocks
- Nonlinear activation functions



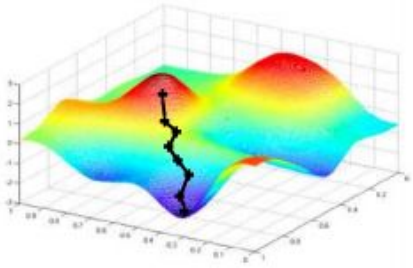
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



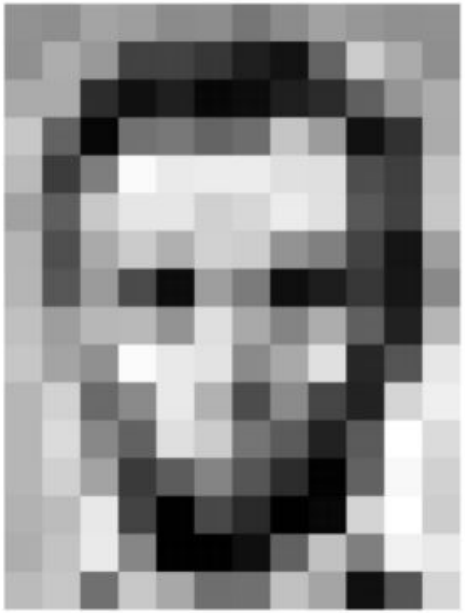
Training in Practice

- Adaptive learning
- Batching
- Regularization



Convolutional Neural Networks (CNNs)

What Computers “See”



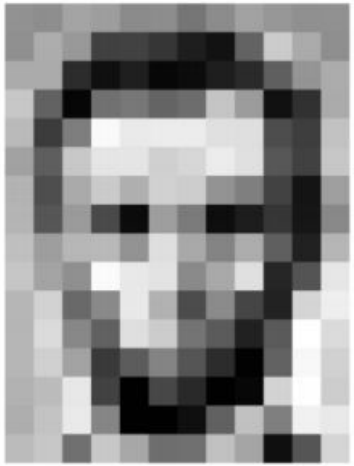
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
196	206	123	207	177	121	123	200	175	13	96	218

What the computer sees

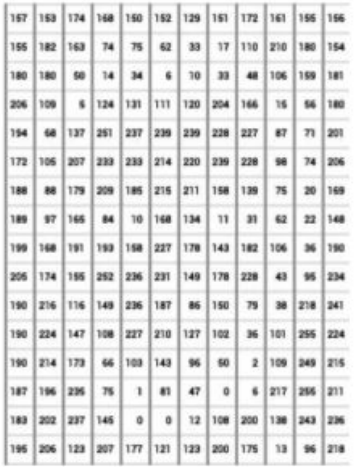
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
196	206	123	207	177	121	123	200	175	13	96	218

An image is just a matrix of numbers [0,255]!
i.e., 1080x1080x3 for an RGB image

Tasks in Computer Vision



Input Image



Pixel Representation

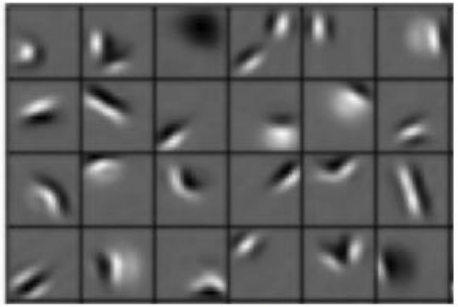
classification

Lincoln	0.8
Washington	0.1
Jefferson	0.05
Obama	0.05

Learning Feature Representations

Can we learn a **hierarchy of features** directly from the data instead of hand engineering?

Low level features



Edges, dark spots

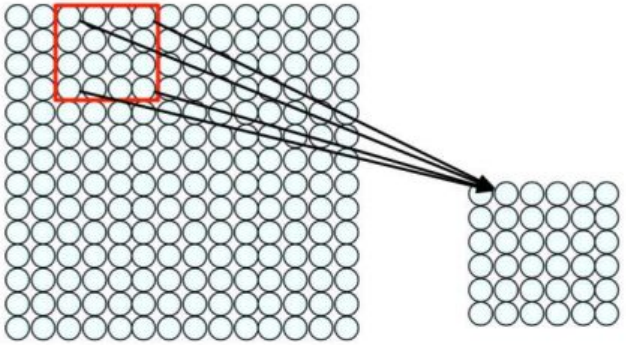
Mid level features

Eyes, ears, nose

High level features

Facial structure

Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This "patchy" operation is **convolution**

1) Apply a set of weights – a filter – to extract **local features**

2) Use **multiple filters** to extract different features

3) **Spatially share** parameters of each filter

Convolution Filters

Original

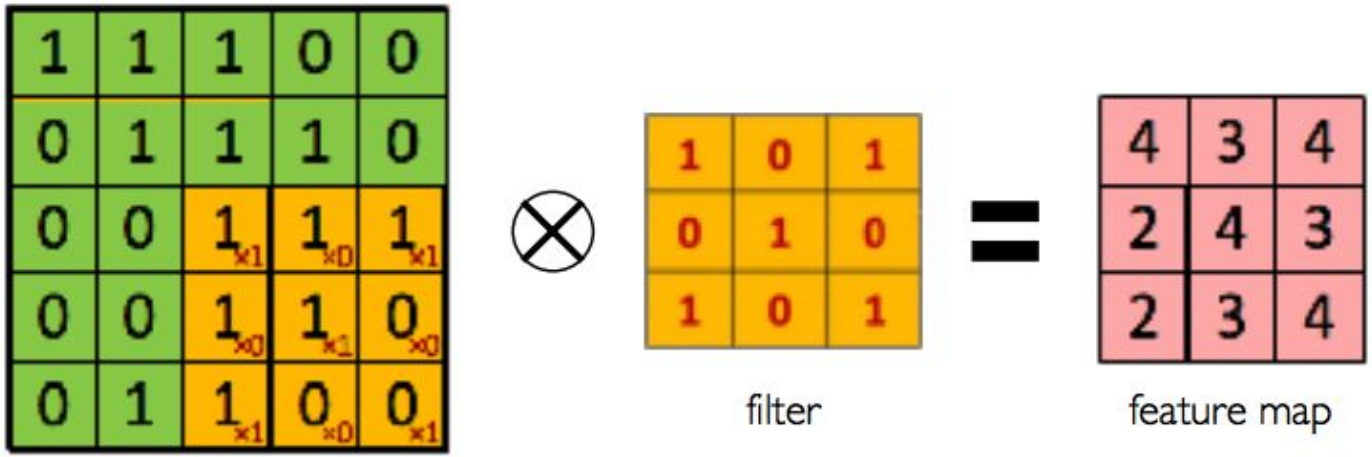
Sharpen

Edge Detect

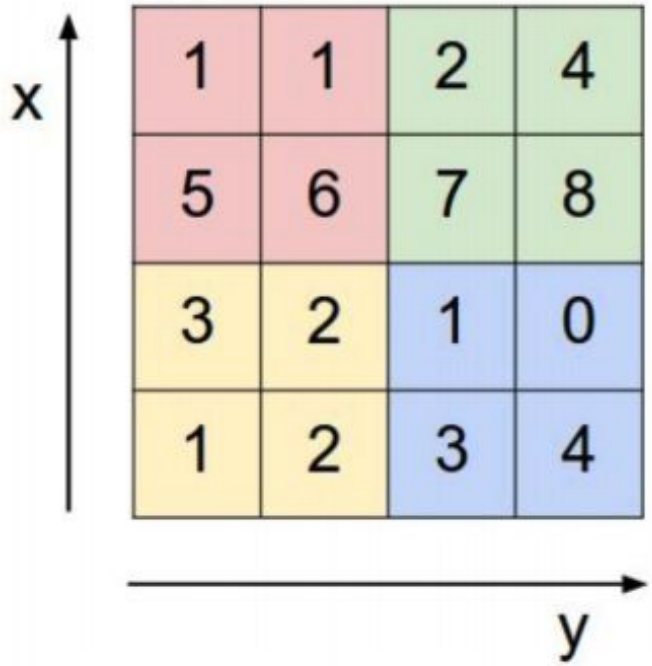
"Strong" Edge Detect

The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



Max Pooling

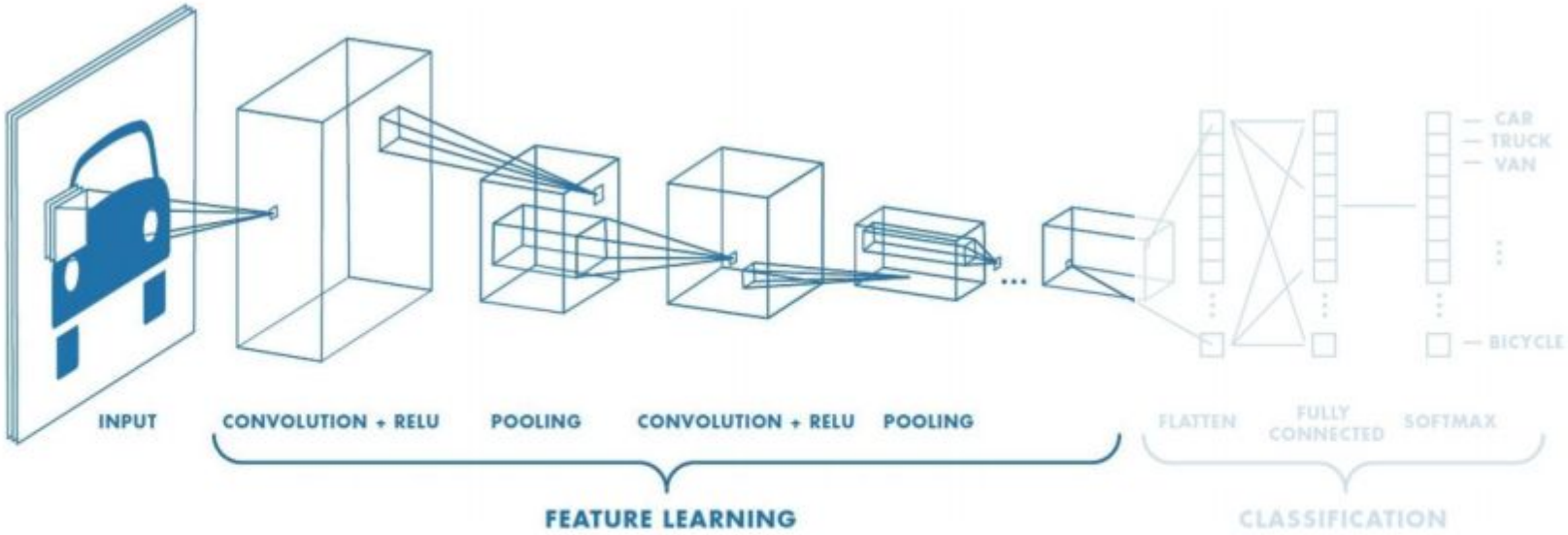


max pool with 2x2 filters
and stride 2

6	8
3	4

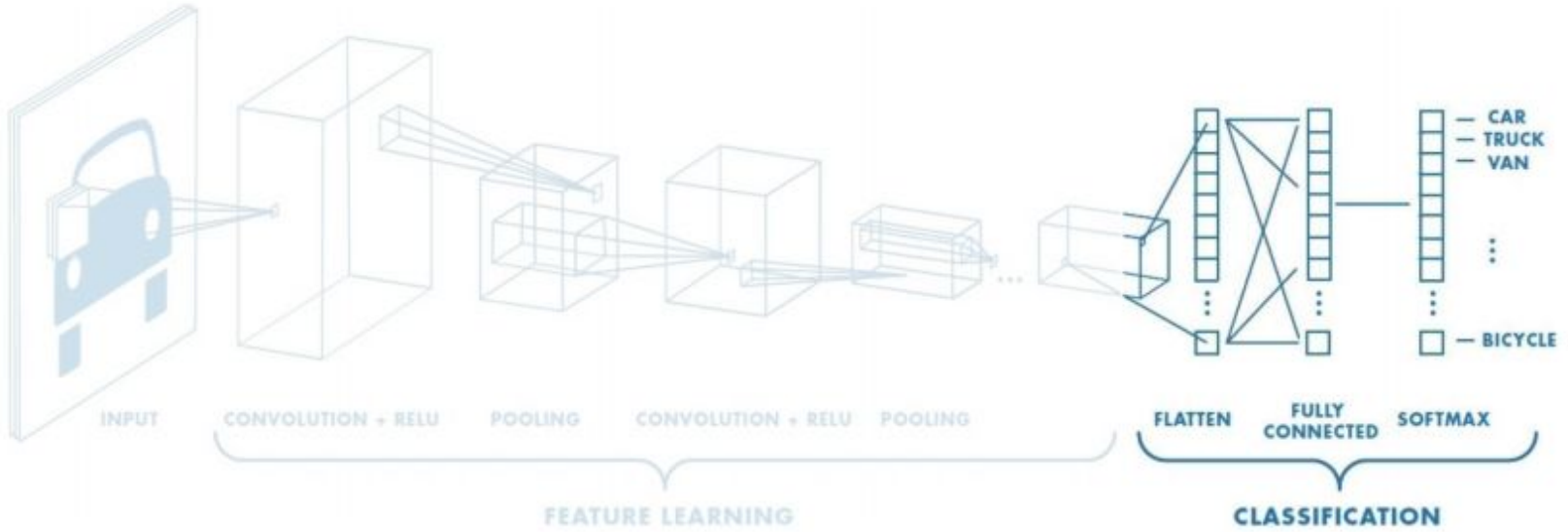
1) Reduced dimensionality
2) Spatial invariance

CNNs for Classification: Feature Learning



1. Learn features in input image through **convolution**
2. Introduce **non-linearity** through activation function (real-world data is non-linear!)
3. Reduce dimensionality and preserve spatial invariance with **pooling**

CNNs for Classification: Class Probabilities

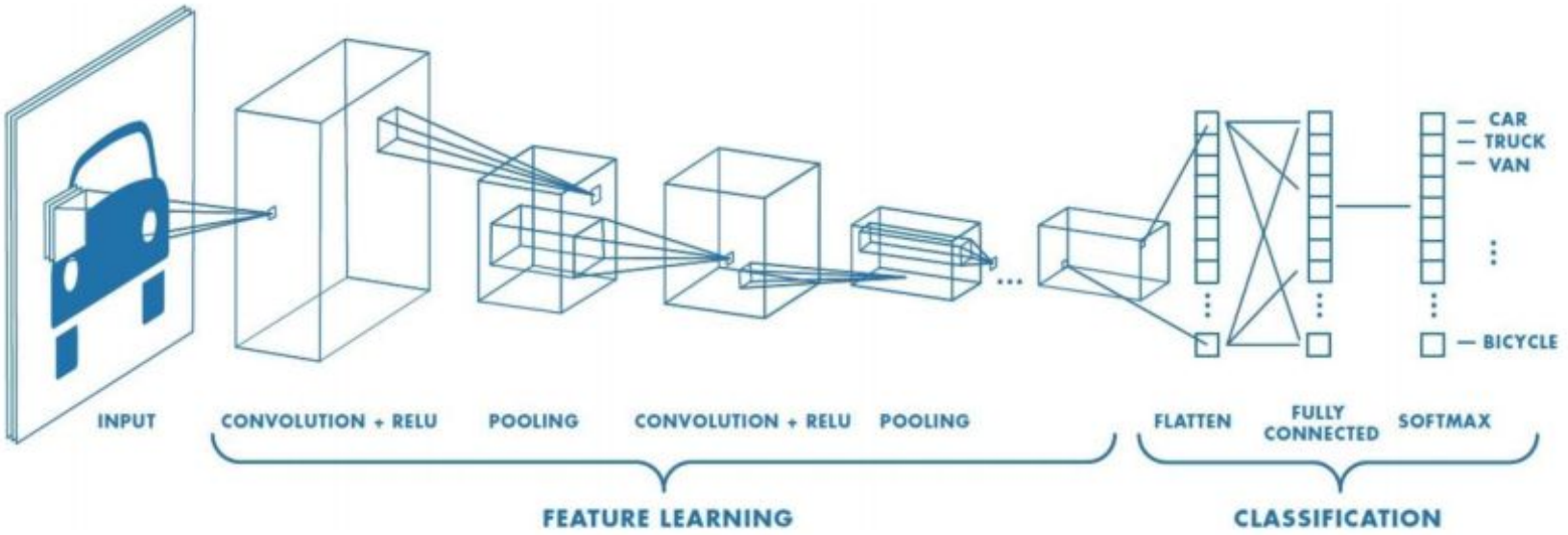


- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Foster School of Business

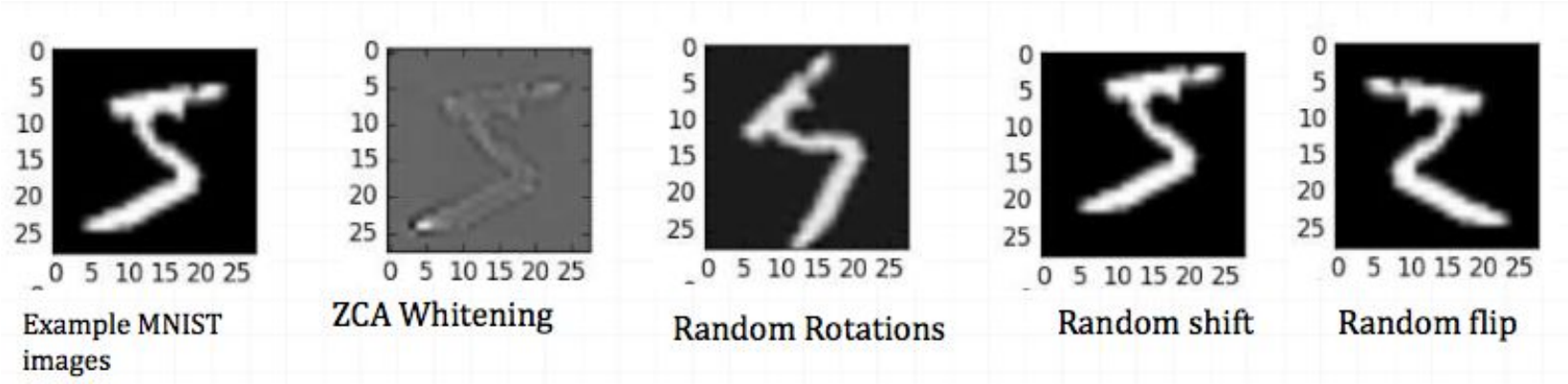
CNNs: Training with Backpropagation



Learn weights for convolutional filters and fully connected layers

Backpropagation: cross-entropy loss

Data Augmentation



- Approaches that alter the training data in ways that change the array representation while keeping the label the same.
- They are a way to artificially expand your dataset. Some popular augmentations people use are gray scales , horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more .

Recurrent Neural Networks (RNNs)

Given an image of a ball,
can you predict where it will go next?

Given an image of a ball,
can you predict where it will go next?

Sequence modeling: predict the next word

"This morning I took my cat for a walk."

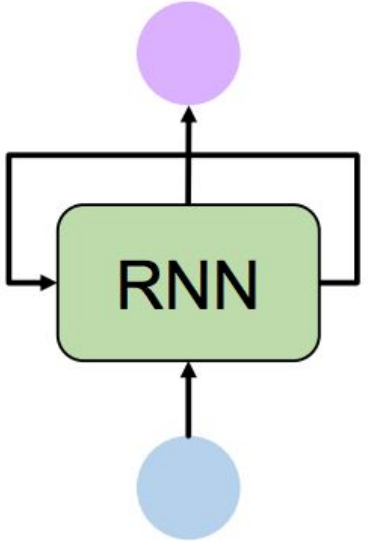
given these words

predict the next word

Sequence modeling: design criteria

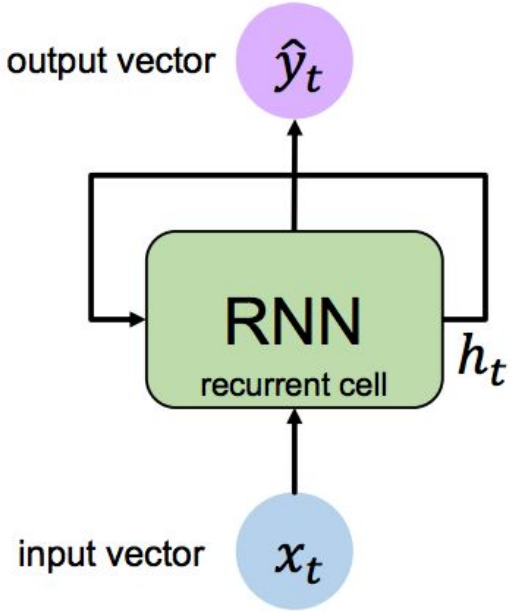
To model sequences, we need to:

1. Handle **variable-length** sequences
2. Track **long-term** dependencies
3. Maintain information about **order**
4. **Share parameters** across the sequence



Today: **Recurrent Neural Networks (RNNs)** as an approach to sequence modeling problems

A recurrent neural network (RNN)



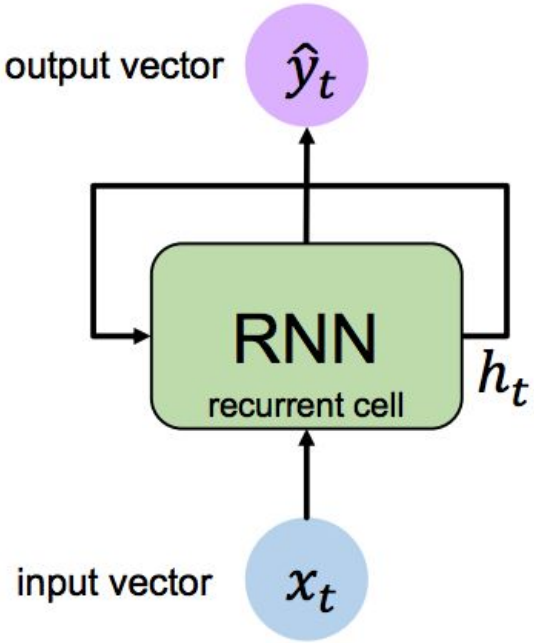
Apply a **recurrence relation** at every time step to process a sequence:

$$\boxed{h_t} = \boxed{f_W}\left(\boxed{h_{t-1}}, \boxed{x_t}\right)$$

new state — function parameterized by W — old state — input vector at time step t

Note: the same function and set of parameters are used at every time step

RNN state update and output

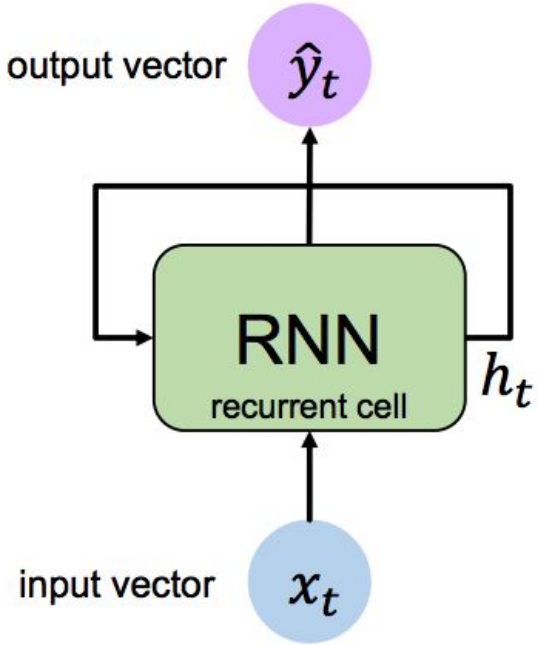


Update Hidden State

$$h_t = \tanh(\boldsymbol{W_{hh}} h_{t-1} + \boldsymbol{W_{xh}} x_t)$$

Input Vector

RNN state update and output



Output Vector

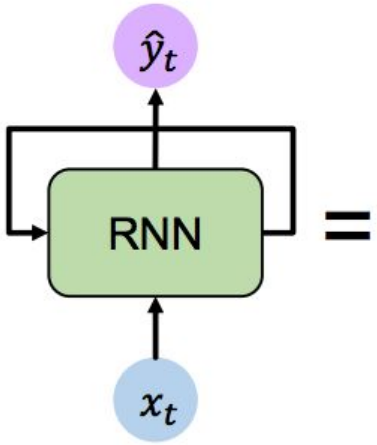
$$\hat{y}_t = \boldsymbol{W_{hy}} h_t$$

Update Hidden State

$$h_t = \tanh(\boldsymbol{W_{hh}} h_{t-1} + \boldsymbol{W_{xh}} x_t)$$

Input Vector

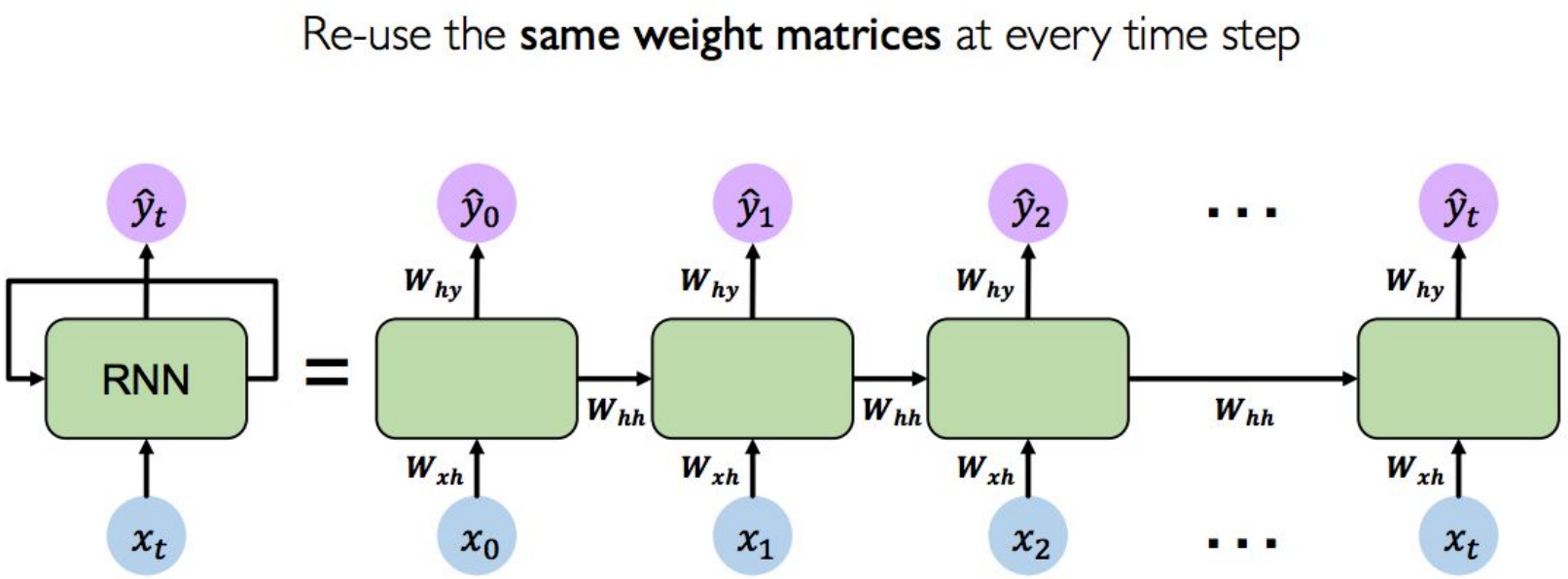
RNNs: computational graph across time



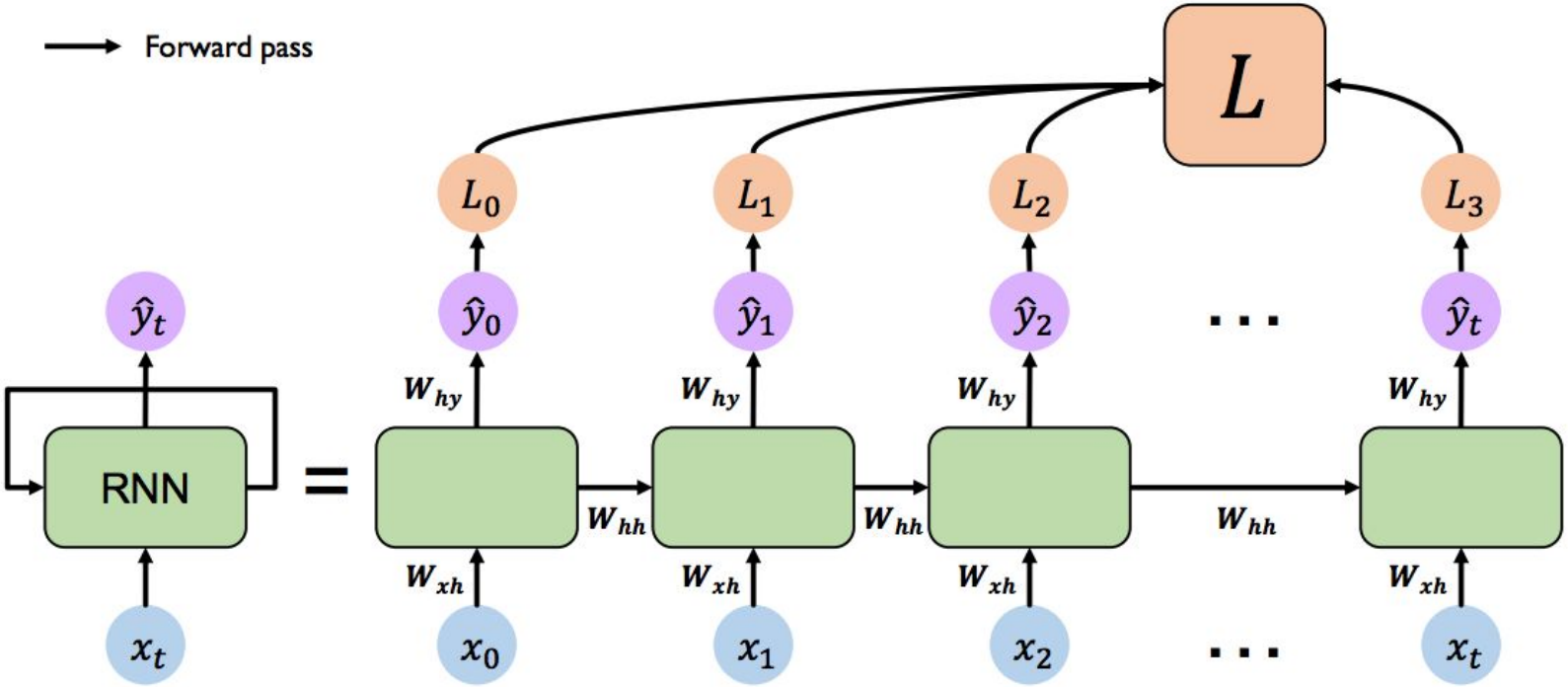
=

Represent as computational graph unrolled across time

RNNs: computational graph across time

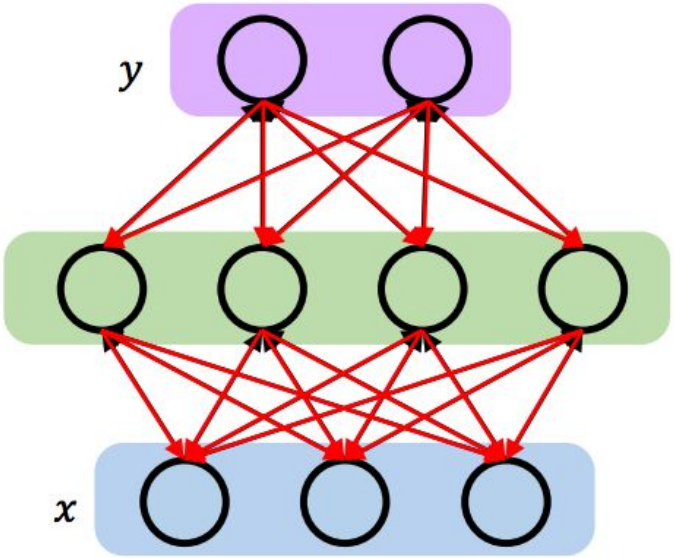


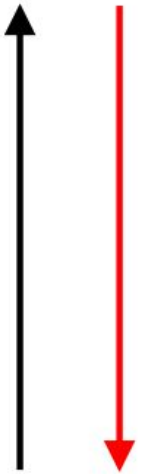
RNNs: computational graph across time



Foster School of Business

Backpropagation in vanilla neural network

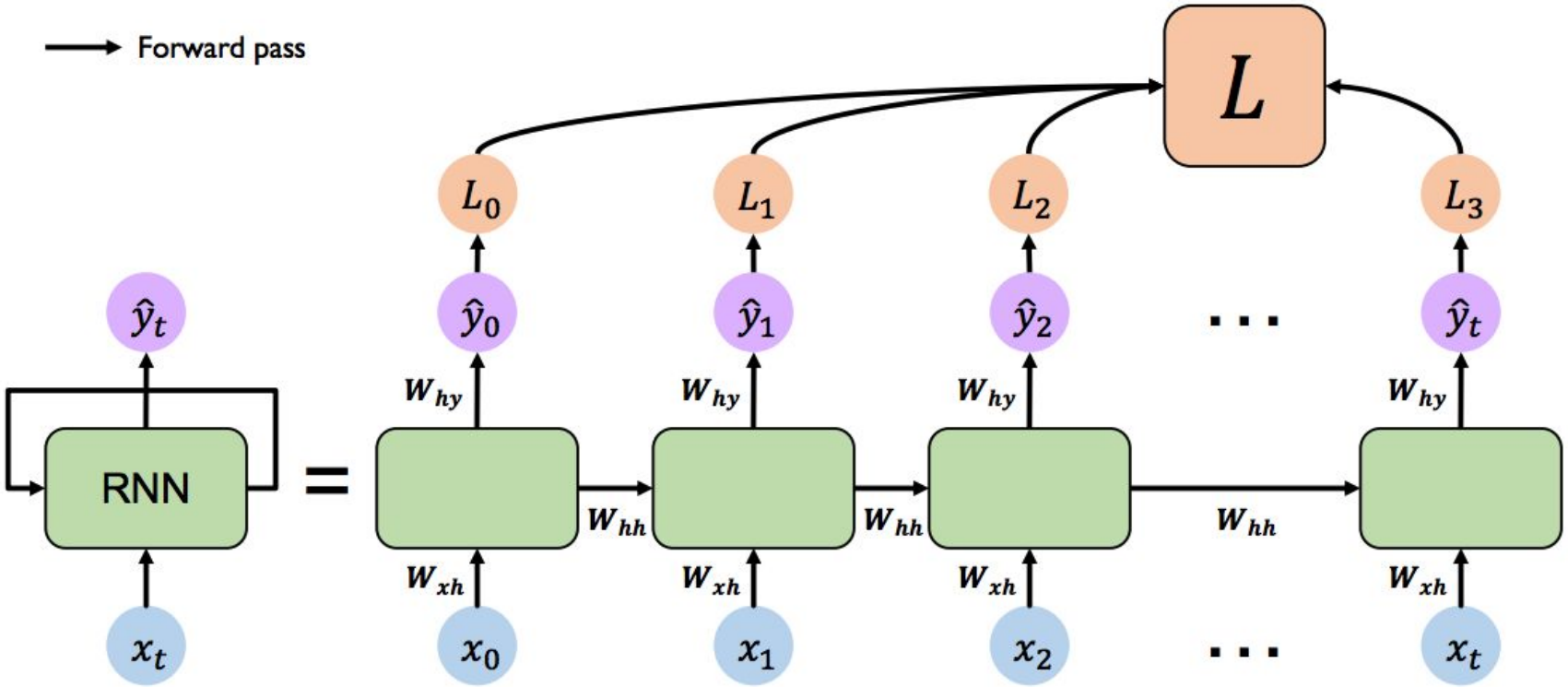




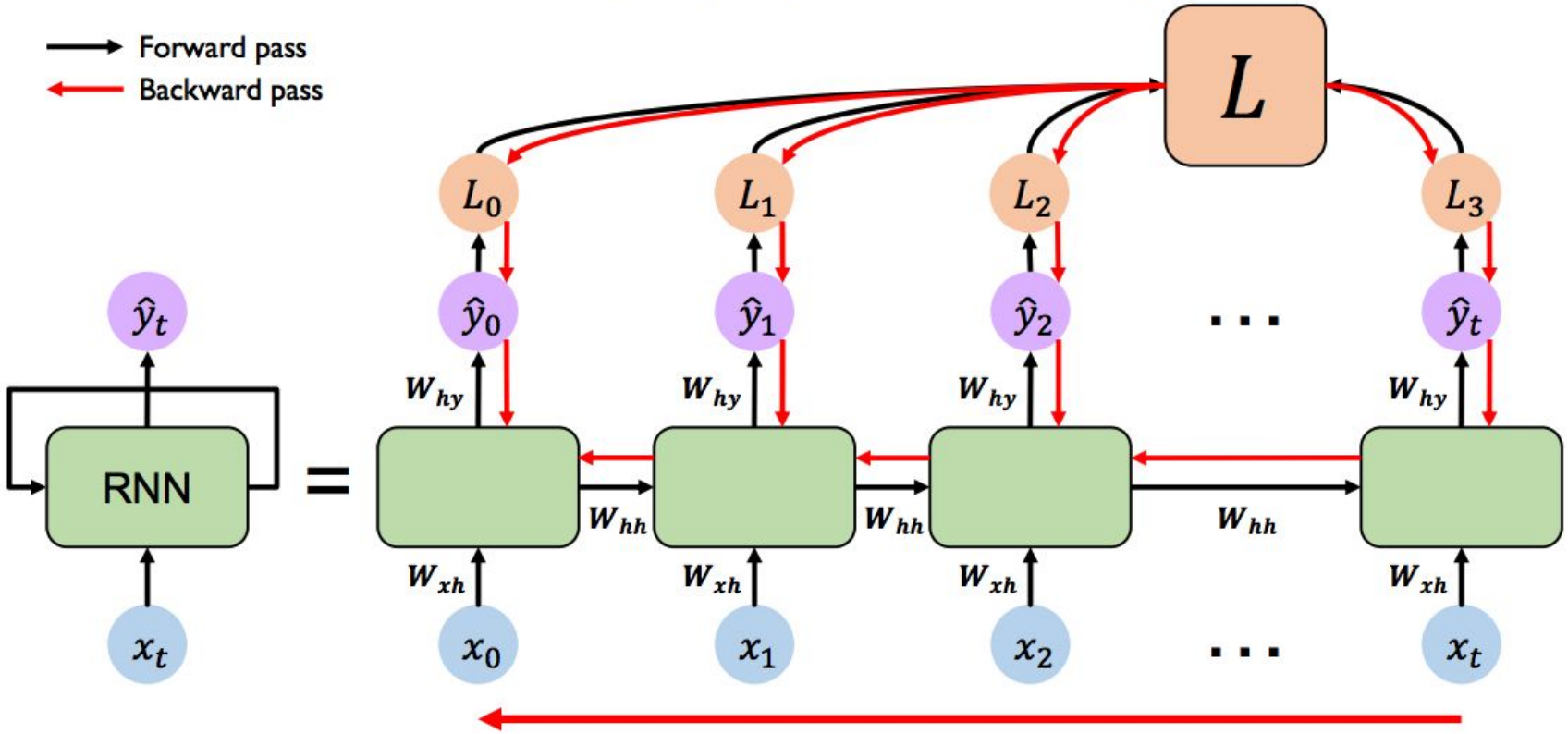
Backpropagation algorithm:

1. Take the derivative (gradient) of the loss with respect to each parameter
2. Shift parameters in order to minimize loss

RNNs: backpropagation through time

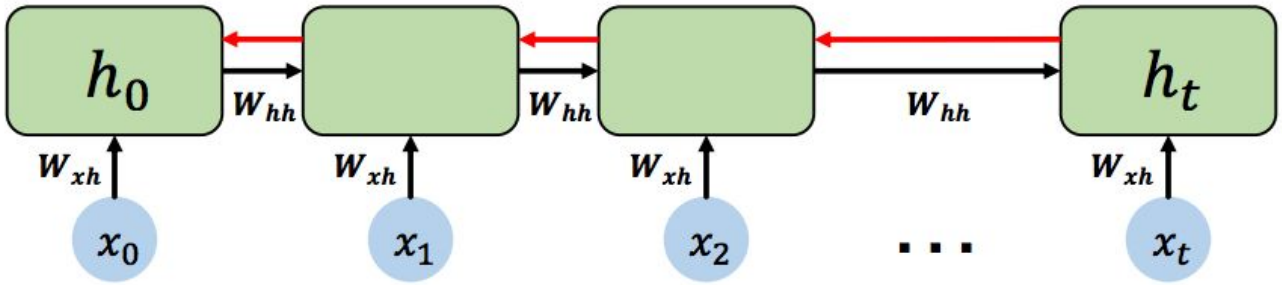


RNNs: backpropagation through time



Foster School of Business

RNN gradient flow: exploding gradients

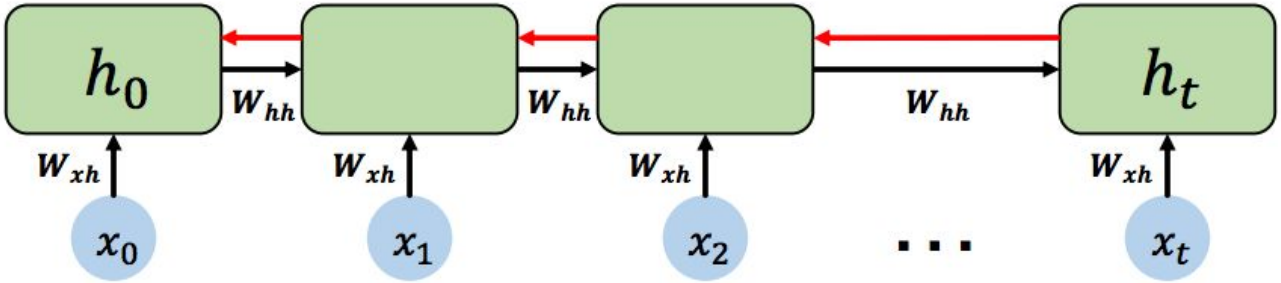


Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Many values > 1:
exploding gradients

Gradient clipping to scale big gradients

RNN gradient flow: vanishing gradients



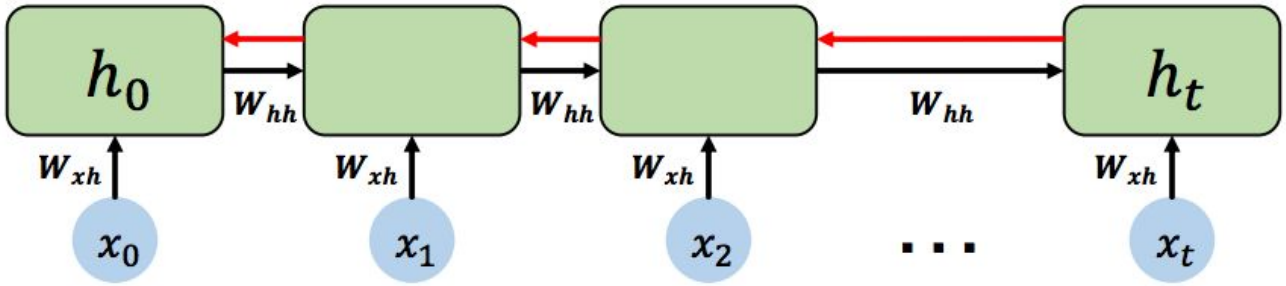
Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Many values > 1:
exploding gradients

Gradient clipping to scale big gradients

Many values < 1:
vanishing gradients

RNN gradient flow: vanishing gradients



Computing the gradient wrt h_0 involves **many factors of W_{hh}** (and repeated f'!)

Largest singular value > 1:
exploding gradients

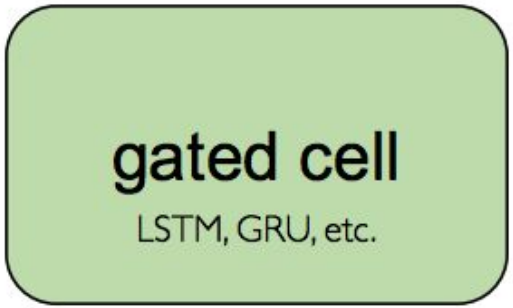
Gradient clipping to scale big gradients

Largest singular value < 1:
vanishing gradients

1. Activation function
2. Weight initialization
3. Network architecture

Solution #3: gated cells

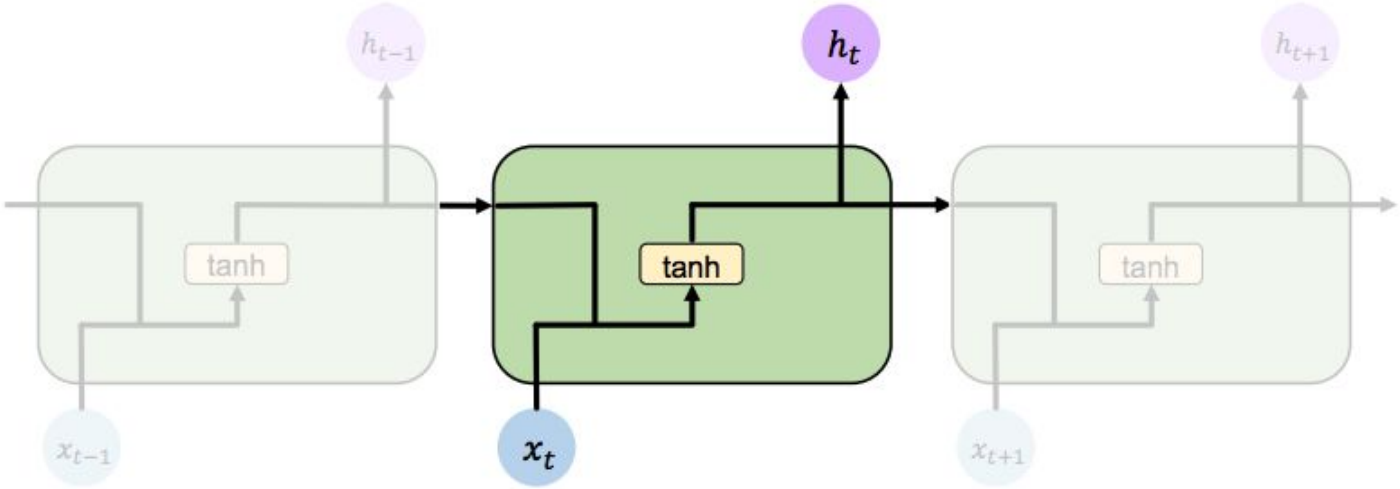
Idea: use a more **complex recurrent unit with gates** to control what information is passed through



Long Short Term Memory (LSTMs) networks rely on a gated cell to track information throughout many time steps.

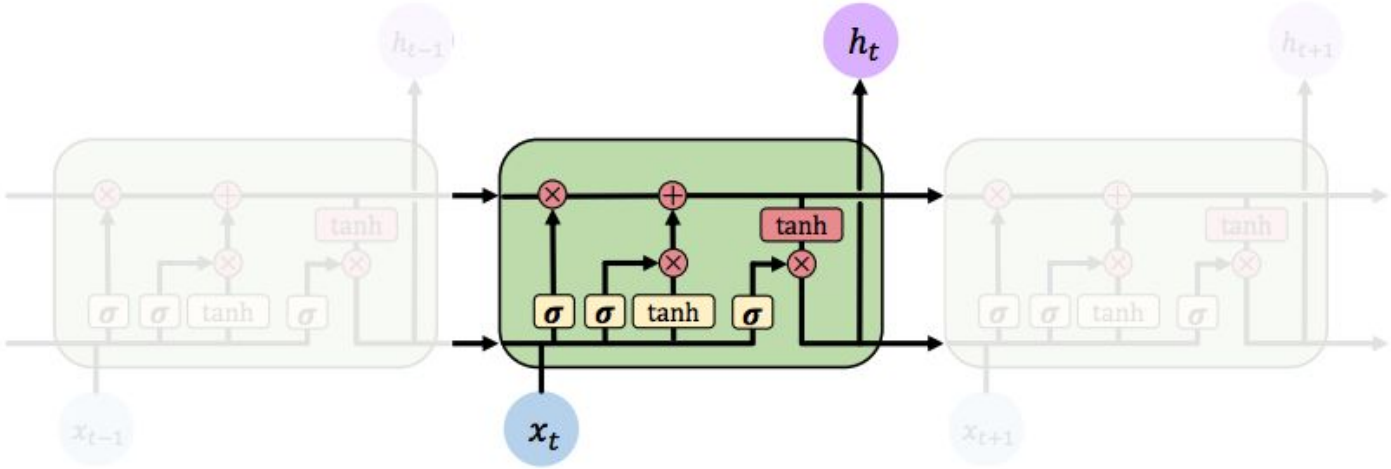
Standard RNN

In a standard RNN, repeating modules contain a **simple computation node**



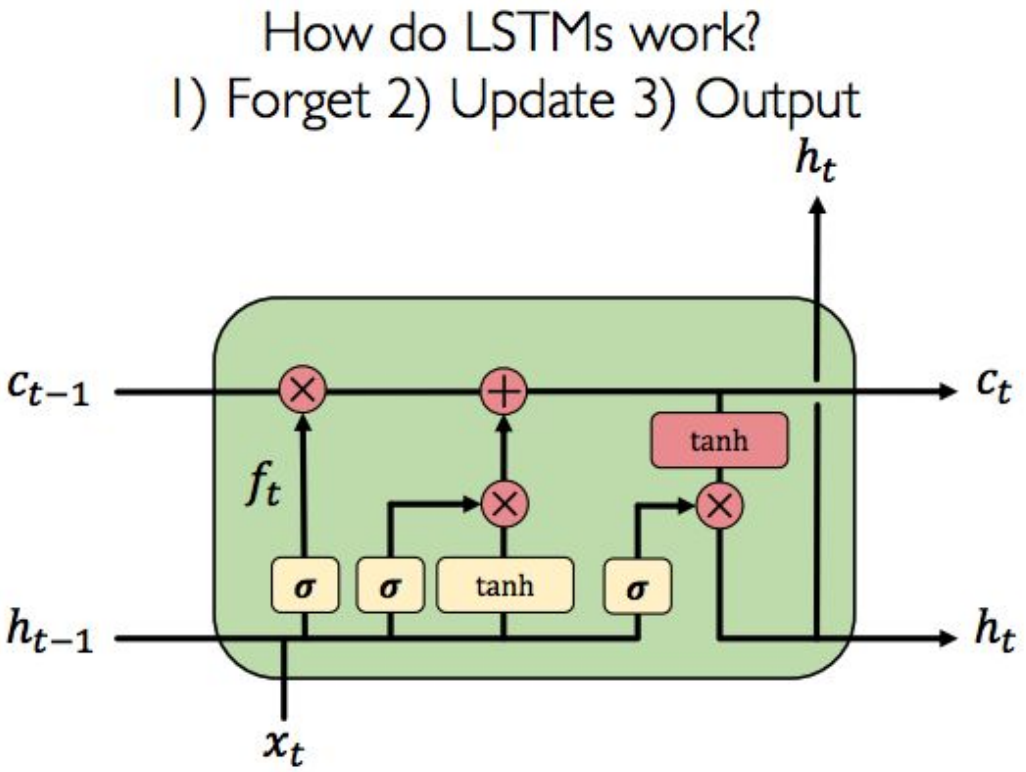
Long Short Term Memory (LSTMs)

LSTM repeating modules contain **interacting layers** that **control information flow**

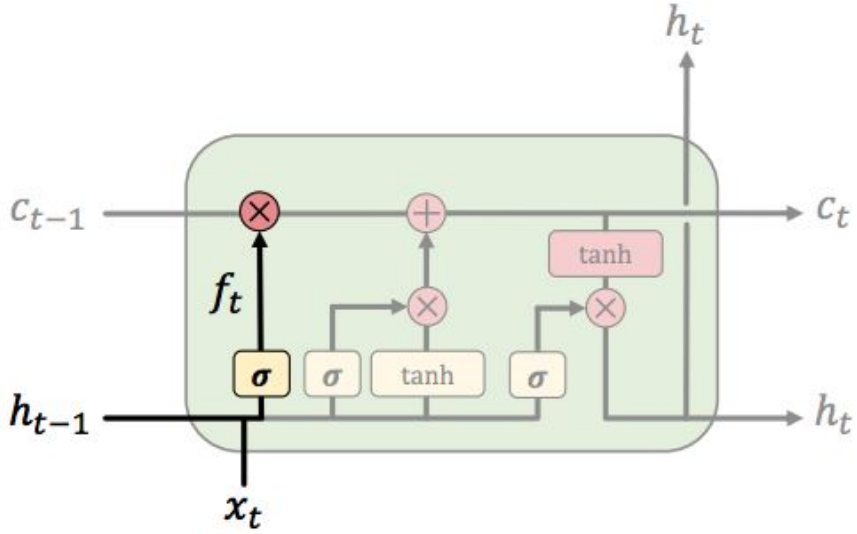


LSTM cells are able to track information throughout many timesteps

Long Short Term Memory (LSTMs)



LSTMs: forget irrelevant information

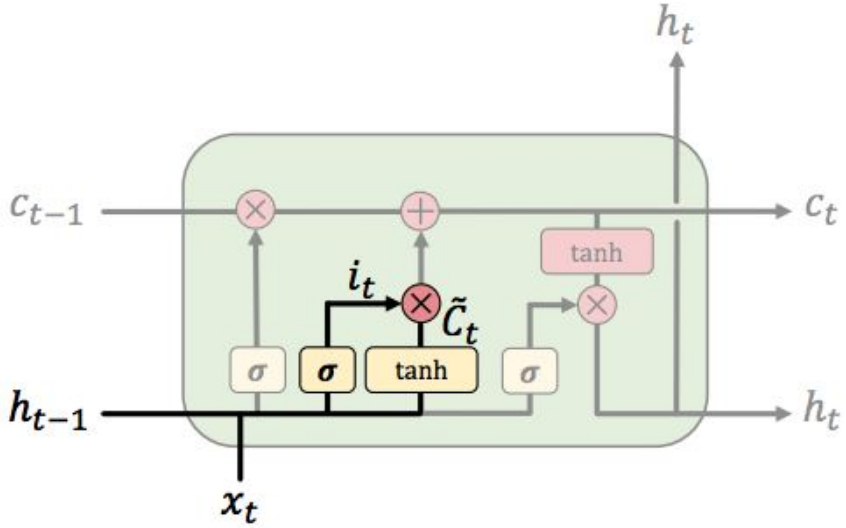


$$f_t = \sigma(W_i[\, h_{t-1}, x_t \,] + b_f)$$

- Use previous cell output and input
- Sigmoid: value 0 and 1 – "completely forget" vs. "completely keep"

ex: Forget the gender pronoun of previous subject in sentence.

LSTMs: identify new info to be stored



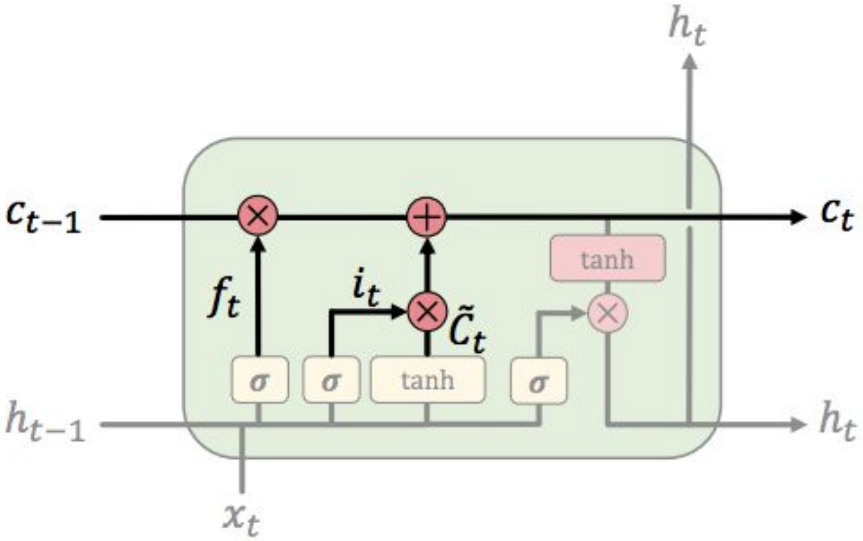
$$i_t = \sigma(W_i[\, h_{t-1}, x_t \,] + b_i)$$

$$\tilde{C}_t = \tanh(W_C[\, h_{t-1}, x_t \,] + b_C)$$

- Sigmoid layer: decide what values to update
- Tanh layer: generate new vector of "candidate values" that could be added to the state

ex: Add gender of new subject to replace that of old subject.

LSTMs: update cell state

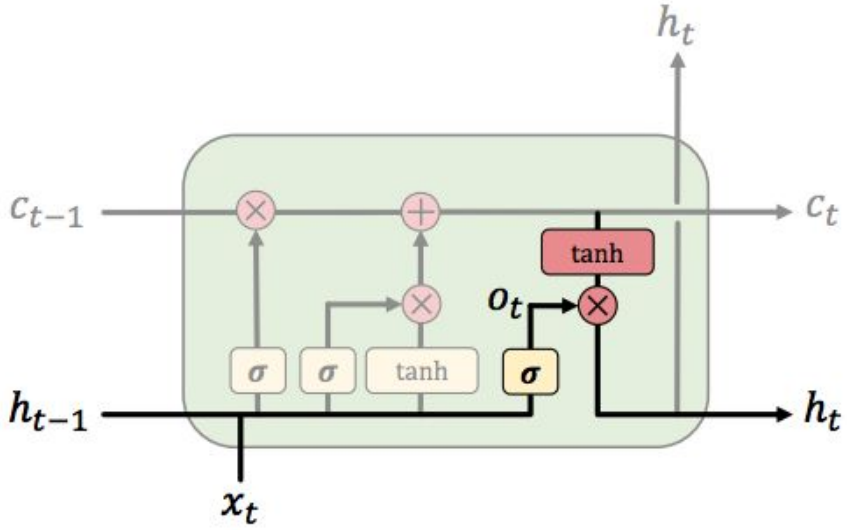


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- Apply forget operation to previous internal cell state: $f_t * C_{t-1}$
- Add new candidate values, scaled by how much we decided to update: $i_t * \tilde{C}_t$

ex: Actually drop old information and add new information about subject's gender.

LSTMs: output filtered version of cell state



$$o_t = \sigma(W_o[\, h_{t-1}, x_t \,] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

- Sigmoid layer: decide what parts of state to output
- Tanh layer: squash values between -1 and 1
- $o_t * \tanh(C_t)$: output filtered version of cell state

ex: Having seen a subject, may output information relating to a verb.

LSTMs: key concepts

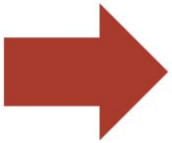
1. Maintain a **separate cell state** from what is outputted
2. Use **gates** to control the **flow of information**
   - Forget gate gets rid of irrelevant information
   - Selectively update cell state
   - Output gate returns a filtered version of the cell state
3. Backpropagation from c_t to c_{t-1} doesn't require matrix multiplication: **uninterrupted gradient flow**

Recurrent neural networks (RNNs)

1. RNNs are well suited for **sequence modeling** tasks
2. Model sequences via a **recurrence relation**
3. Training RNNs with **backpropagation through time**
4. Gated cells like **LSTMs** let us model **long-term dependencies**

One-hot encoding of Words

Vocabulary:
Man, woman, boy, girl, prince, princess, queen, king, monarch



	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

Each word gets a 1x9 vector representation

Word Embedding

Try to build a lower dimensional embedding

Vocabulary:
Man, woman, boy, girl, prince, princess, queen, king, monarch

	Femininity	Youth	Royalty
Man	0	0	0
Woman	1	0	0
Boy	0	1	0
Girl	1	1	0
Prince	0	1	1
Princess	1	1	1
Queen	1	0	1
King	0	0	1
Monarch	0.5	0.5	1

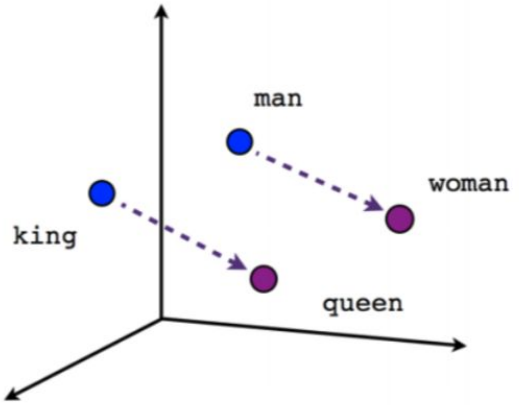
Each word gets a 1x3 vector

Similar words... similar vectors

Word2Vec Word Embedding

Here's a list of words associated with "Sweden" using Word2vec, in order of proximity:

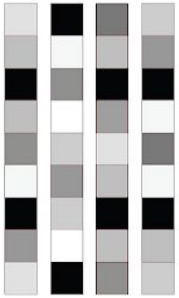
Word	Cosine distance
norway	0.760124
denmark	0.715460
finland	0.620022
switzerland	0.588132
belgium	0.585835
netherlands	0.574631
iceland	0.562368
estonia	0.547621
slovenia	0.531408



[[king]] – [[man]] + [[woman]] = [[queen]]

One-hot encoding vs. Embedding

One-hot word vectors:
- Sparse
- High-dimensional
- Hard-coded



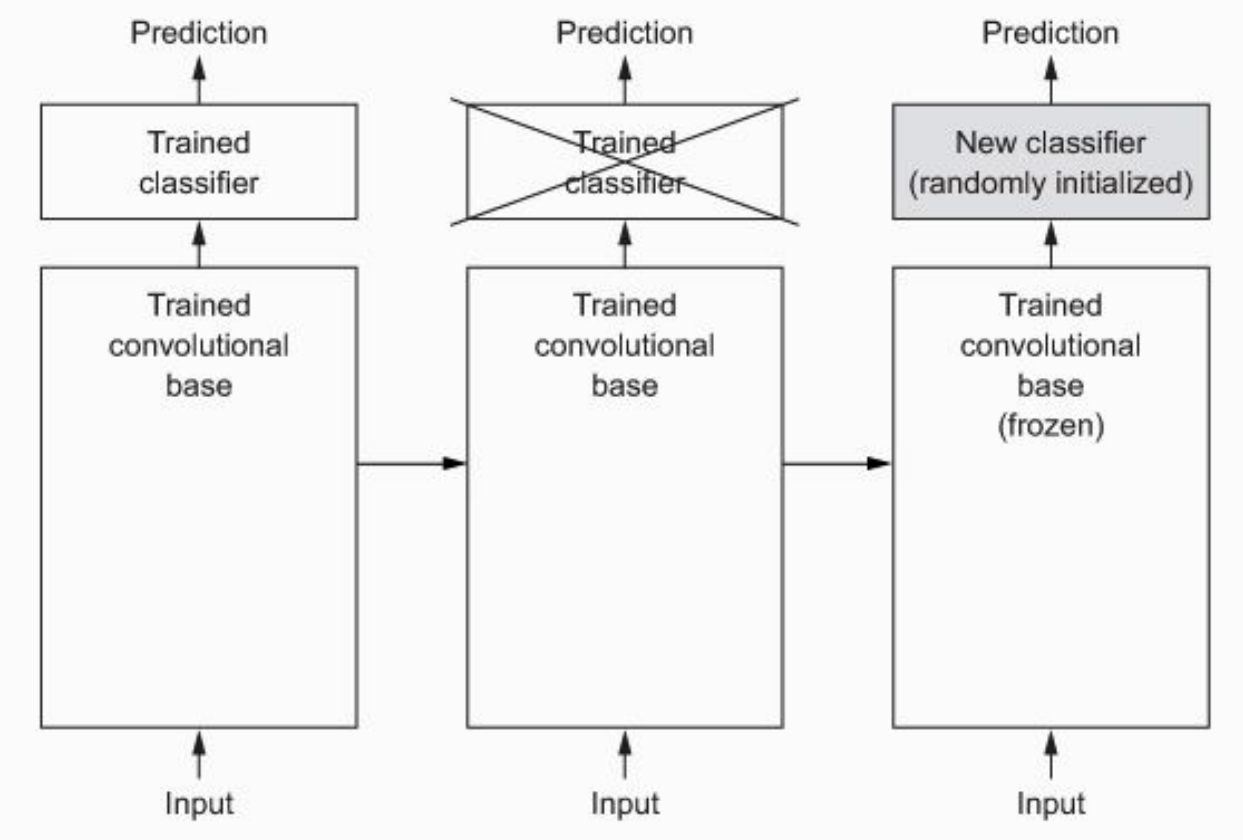
Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Transfer Learning

Transfer learning is the process of taking a pre-trained model (the weights and parameters of a network that has been trained on a large dataset by somebody else) and “fine-tuning” the model with your own dataset.

- Pre-trained model will act as a feature extractor.
- Freeze the weights of all the other layers .
- Remove the last layer of the network and replace it with your own classifier.
- Train the network normally.

Transfer Learning



Lab