Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

An Introduction to Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Sections 1.1–1.2

Neural Networks

- Neural networks have seen an explosion in popularity in recent years.
  - Victories in eye-catching competitions like the *ImageNet* contest have brought them fame.
  - The potential of neural networks is now being realized because of fundamental shifts in hardware paradigms and data availability.
- The new term *deep learning* is a re-birth of the field of neural networks (although it also emphasizes a specific aspect of neural networks).

Overview of the Presented Material

- The videos are based on the book: C. Aggarwal. Neural Networks and Deep learning, *Springer*, 2018.
  - Videos not meant to be exhaustive with respect to book.
  - Helpful in providing a firm grounding of important aspects.
  - Videos can provide the initial background to study more details from the book.
  - Slides available for download at http://www.charuaggarwal.net (with latex source).

Overview of Book

- The book covers both the old and the new in neural networks.
  - The core learning methods like backpropagation, traditional architectures, and specialized architectures for sequence/image applications are covered.
  - The latest methods like variational autoencoders, Generative Adversarial Networks, Neural Turing Machines, attention mechanisms, and reinforcement learning are covered.
    - Reinforcement learning is presented from the view of deep learning applications.
  - The "forgotten architectures" like Radial Basis Function networks and Kohonen self organizing maps are also covered.

Neural Networks: Two Views

- A way to simulate biological learning by simulating the nervous system.

- A way to increase the power of known models in machine learning by stacking them in careful ways as *computational graphs*.
  - The number of nodes in the computational graph controls learning capacity with increasing data.
  - The specific architecture of the computational graph incorporates domain-specific insights (e.g., images, speech).
  - The success of deep computational graphs has led to the coining of the term "*deep learning*."

Historical Origins

- The first model of a computational unit was the *perceptron* (1958).
  - Was roughly inspired by the biological model of a neuron.
  - Was implemented using a large piece of hardware.
  - Generated great excitement but failed to live up to inflated expectations.

- Was not any more powerful than a simple linear model that can be implemented in a few lines of code today.

The Perceptron [Image Courtesy: Smithsonian Institute]

The First Neural Winter: Minsky and Papert's Book

- Minsky and Papert's Book "*Perceptrons*" (1969) showed that the perceptron only had limited expressive power.
  - Essential to put together multiple computational units.

- The book also provided a pessimistic outlook on training multilayer neural networks.
  - Minsky and Papert's book led to the first winter of neural networks.
  - Minsky is often blamed for setting back the field (fairly or unfairly).

- Were Minsky/Papert justified in their pessimism?

Did We Really Not Know How to Train Multiple Units?

- It depends on who you ask.
  - AI researchers didn't know (and didn't believe it possible).
  - Training computational graphs with dynamic programming had already been done in control theory (1960s).

- Paul Werbos proposed backpropagation in his 1974 thesis (and was promptly ignored–formal publication was delayed).
  - Werbos (2006): "*In the early 1970s, I did in fact visit Minsky at MIT. I proposed that we do a joint paper showing that MLPs can in fact overcome the earlier problems if (1) the neuron model is slightly modified to be differentiable; and (2) the training is done in a way that uses the reverse method, which we now call backpropagation in the ANN field. But Minsky was not interested. In fact, no one at MIT or Harvard or any place I could find was interested at the time.*"

General View of Artificial Intelligence (Seventies/Eighties)

- It was the era or work on logic and reasoning (discrete mathematics).
  - Viewed as the panacea of AI.
  - This view had influential proponents like Patrick Henry Winston.

- Work on continuous optimization had few believers.
  - Researchers like Hinton were certainly not from the mainstream.
  - This view has been completely reversed today.
  - The early favorites have little to show in spite of the effort.

Backpropagation: The Second Coming

- Rumelhart, Hinton, and Williams wrote two papers on back-propagation in 1986 (independent from prior work).
  - Paul Werbos's work had been forgotten and buried at the time.
- Rumelhart *et al*'s work is presented beautifully.
- It was able to at least partially resurrect the field.

The Nineties

- Acceptance of backpropagation encouraged more research in multilayer networks.

- By the year 2000, most of the modern architectures had already been set up in some form.
  - They just didn't work very well!
  - The winter continued after a brief period of excitement.

- It was the era of the support vector machine.
  - The new view: SVM was the panacea (at least for supervised learning).

What Changed?

- Modern neural architectures are similar to those available in the year 2000 (with some optimization tweaks).

- **Main difference:** Lots of data and computational power.

- Possible to train large and deep neural networks with millions of neurons.

- Significant Events: Crushing victories of deep learning methods in *ImageNet* competitions after 2010.

- *Anticipatory Excitement:* In a few years, we will have the power to train neural networks with as many computational units as the human brain.

  – Your guess is as good as mine about what happens then.

A Cautionary Note

- Deep learning has now assumed the mantle of the AI panacea.
  - We have heard this story before.
  - Why should it be different this time?
  - Excellent performance on richly structured data (images, speech), but what about others?
- There are indeed settings where you are better off using a conventional machine learning technique like a random forest.

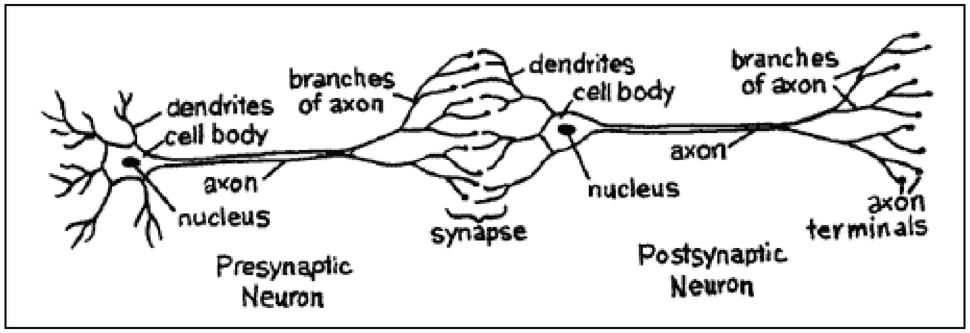
How it All Started: The Biological Inspiration

- Neural networks were originally designed to simulate the learning process in biological organisms.

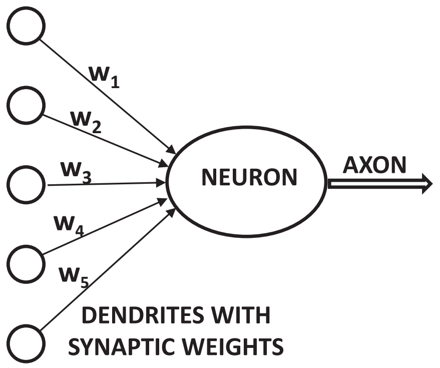
- The human nervous system contains cells called *neurons*.

- The neurons are connected to one another with the use of *synapses*.
  - The strengths of synaptic connections often change in response to external stimuli.
  - This change causes learning in living organisms.

Neural Networks: The Biological Inspiration



(a) Biological neural network



(b) Artificial neural network

- Neural networks contain computation units $\Rightarrow$ Neurons.

- The computational units are connected to one another through weights $\Rightarrow$ Strengths of synaptic connections in biological organisms.

- Each input to a neuron is scaled with a weight, which affects the function computed at that unit.

Learning in Biological vs Artificial Networks

- In living organisms, synaptic weights change in response to external stimuli.
  - An unpleasant experience will change the synaptic weights of an organism, which will train the organism to behave differently.

- In artificial neural networks, the weights are learned with the use of training data, which are input-output pairs (e.g., images and their labels).
  - An error made in predicting the label of an image is the unpleasant "stimulus" that changes the weights of the neural network.
  - When trained over many images, the network learns to classify images correctly.

Comments on the Biological Paradigm

- The biological paradigm is often criticized as a very inexact caricature.
  - The functions computed in a neural network are very different from those in the brain. $\Rightarrow$ No one exactly knows how the brain works.
- Nevertheless, there are several examples, where the principles of neuroscience have been successfully applied in designing neural networks.
  - Convolutional neural networks are based on architectural principles drawn from the cat's visual cortex.
- There has been a renewed focus in recent years in leveraging the principles of neuroscience in neural network design $\Rightarrow$ The "caricature" view is not fully justified.

An Alternative View: The Computational Graph Extension of Traditional Machine Learning

- The elementary units compute similar functions to traditional machine learning models like linear or logistic regression.

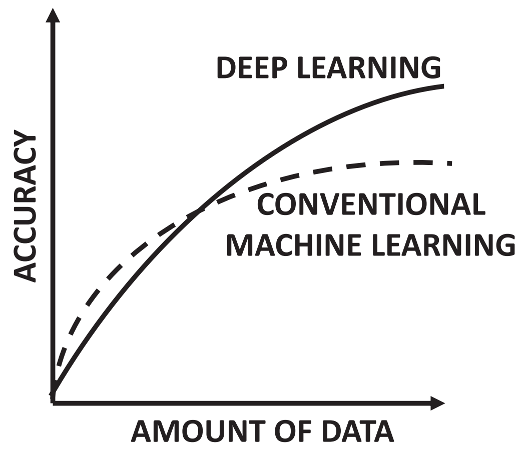
- *Much of what you know about optimization-based machine learning can be recast as shallow neural models*.

  - When large amounts of data are available, these models are unable to learn all the structure.

- Neural networks provide a way to increase the capacity of these models by connecting multiple units as a computational graph (with an increased number of parameters).

- At the same time, connecting the units in a particular way can incorporate domain-specific insights.

Machine Learning versus Deep Learning



- For smaller data sets, traditional machine learning methods often provide slightly better performance.

- Traditional models often provide more choices, interpretable insights, and ways to handcraft features.

- For larger data sets, deep learning methods tend to dominate.

Reasons for Recent Popularity

- The recent success of neural networks has been caused by an increase in data and computational power.
  - Increased computational power has reduced the cycle times for experimentation.
  - If it requires a month to train a network, one cannot try more than 12 variations in an year on a single platform.
  - Reduced cycle times have also led to a larger number of successful tweaks of neural networks in recent years.
  - Most of the models have not changed dramatically from an era where neural networks were seen as impractical.
- We are now operating in a data and computational regime where deep learning has become attractive compared to traditional machine learning.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

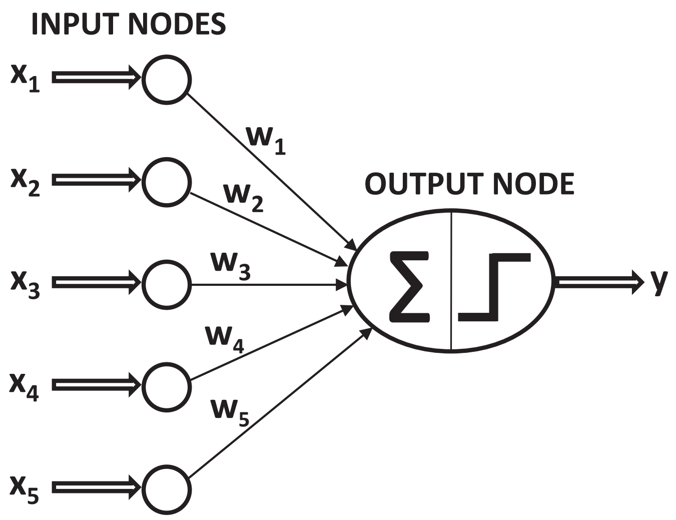
Single Layer Networks: The Perceptron

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Section 1.3

Binary Classification and Linear Regression Problems

- In the binary classification problem, each training pair $(\overline{X}, y)$ contains feature variables $\overline{X} = (x_1, \ldots x_d)$, and label y drawn from $\{-1, +1\}$.
  - Example: Feature variables might be frequencies of words in an email, and the class variable might be an indicator of spam.
  - Given labeled emails, recognize incoming spam.
- In linear regression, the *dependent* variable y is real-valued.
  - Feature variables are frequencies of words in a Web page, and the dependent variable is a prediction of the number of accesses in a fixed period.
- Perceptron is designed for the binary setting.

The Perceptron: Earliest Historical Architecture



- The d nodes in the input layer only transmit the d features $\overline{X} = [x_1 \ldots x_d]$ without performing any computation.

- Output node multiplies input with weights $\overline{W} = [w_1 \ldots w_d]$ on incoming edges, aggregates them, and applies *sign activation*:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\}$$

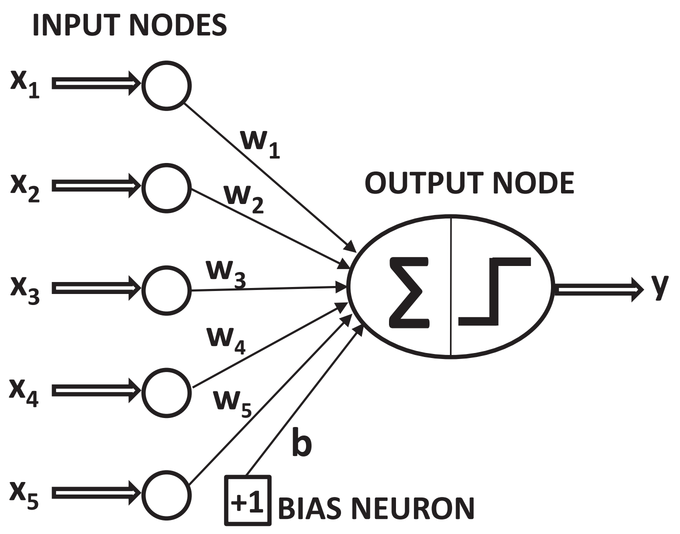
What is the Perceptron Doing?

- Tries to find a *linear separator* $\overline{W} \cdot \overline{X} = 0$ between the two classes.

- Ideally, all positive instances ($y = 1$) should be on the side of the separator satisfying $\overline{W} \cdot \overline{X} > 0$.

- All negative instances ($y = -1$) should be on the side of the separator satisfying $\overline{W} \cdot \overline{X} < 0$.

Bias Neurons



- In many settings (e.g., skewed class distribution) we need an invariant part of the prediction with bias variable b:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X} + b\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j + b\} = \text{sign}\{\sum_{j=1}^{d+1} w_j x_j\}$$

- On setting $w_{d+1} = b$ and x_{d+1} as the input from the bias neuron, it makes little difference to learning procedures $\Rightarrow$ Often implicit in architectural diagrams

Training a Perceptron

- Go through the input-output pairs $(\overline{X}, y)$ one by one and make updates, if predicted value $\hat{y}$ is different from observed value $y \Rightarrow$ Biological readjustment of synaptic weights.

$$\overline{W} \Leftarrow \overline{W} + \alpha \underbrace{(y - \hat{y})}_{\text{Error}} \overline{X}$$

$$\overline{W} \Leftarrow \overline{W} + (2\alpha) y \overline{X} \text{ [For misclassified instances } y - \hat{y} = 2y]$$

- Parameter α is the learning rate $\Rightarrow$ Turns out to be irrelevant in the special case of the perceptron

- One cycle through the entire training data set is referred to as an *epoch* $\Rightarrow$ Multiple epochs required

- How did we derive these updates?

What Objective Function is the Perceptron Optimizing?

- At the time, the perceptron was proposed, the notion of loss function was not popular $\Rightarrow$ Updates were heuristic

- Perceptron optimizes the perceptron criterion for ith training instance:

$$L_i = \max\{-y_i(\overline{W} \cdot \overline{X_i}), 0\}$$

  - Loss function tells us how far we are from a desired solution $\Rightarrow$ Perceptron criterion is 0 when $\overline{W} \cdot \overline{X_i}$ has same sign as y_i.

- Perceptron updates use *stochastic gradient descent* to optimize the loss function and reach the desired outcome.

  - Updates are equivalent to $\overline{W} \Leftarrow \overline{W} - \alpha \left(\frac{\partial L_i}{\partial w_1} \dots \frac{\partial L_i}{\partial w_d} \right)$

Perceptron vs Linear SVMs

- Perceptron criterion is a shifted version of hinge-loss in SVM:

$$L_i^{svm} = \max\{1 - y_i(\overline{W} \cdot \overline{X_i}), 0\}$$

  – The pre-condition for updates in perceptron and SVMs is different:

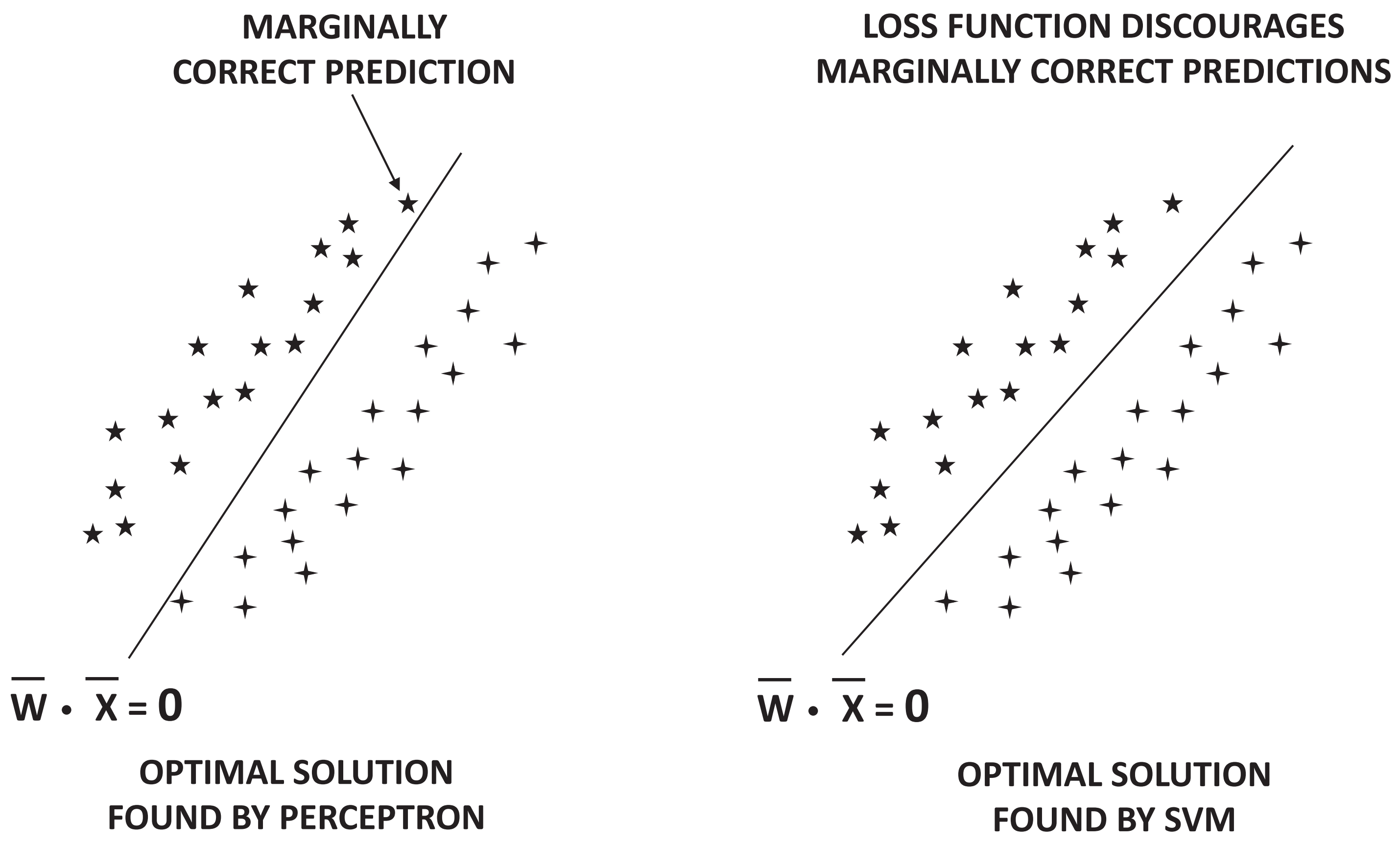
  – In a perceptron, we update when a misclassification occurs: $-y_i(\overline{W} \cdot \overline{X_i}) > 0$

  – In a linear SVM, we update when a misclassification occurs or a classification is "barely correct": $1 - y_i(\overline{W} \cdot \overline{X_i}) > 0$

- Otherwise, *primal* updates of linear SVM are *identical* to perceptron:

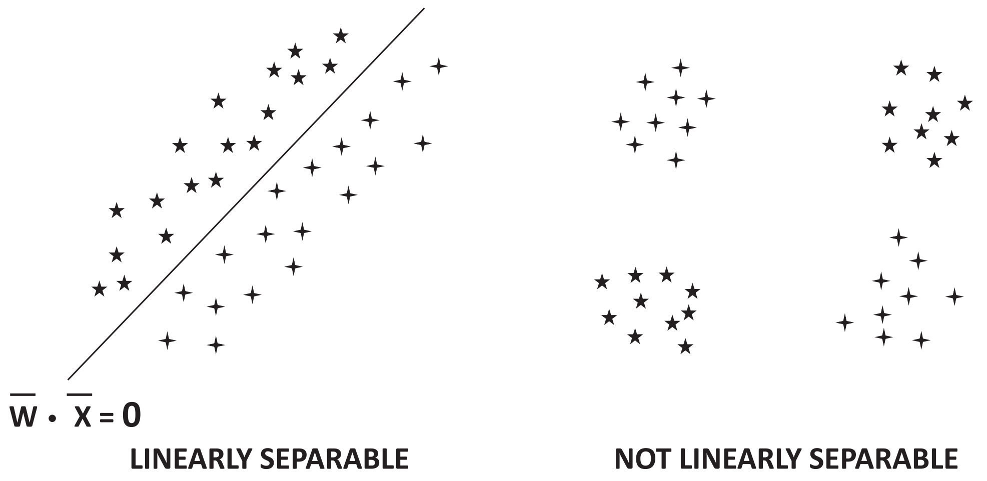
$$\overline{W} \Leftarrow \overline{W} + \alpha y \overline{X}$$

Perceptron vs Linear SVMs


MARGINALLY CORRECT PREDICTION
LOSS FUNCTION DISCOURAGES MARGINALLY CORRECT PREDICTIONS
$\overline{W} \cdot \overline{X} = 0$
$\overline{W} \cdot \overline{X} = 0$
OPTIMAL SOLUTION FOUND BY PERCEPTRON
OPTIMAL SOLUTION FOUND BY SVM

- The more rigorous condition for the update in a linear SVM ensures that points near the decision boundary *generalize* better to the test data.

Where does the Perceptron Fail?



- The perceptron fails at similar problems as a linear SVM
  - **Classical solution:** Feature engineering with Radial Basis Function network $\Rightarrow$ Similar to kernel SVM and good for noisy data
  - **Deep learning solution:** Multilayer networks with non-linear activations $\Rightarrow$ Good for data with a lot of structure

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Activation and Loss Functions

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Section 1.2

Why Do We Need Activation Functions?

- An activation function $\Phi(v)$ in the output layer can control the nature of the output (e.g., probability value in $[0, 1]$)

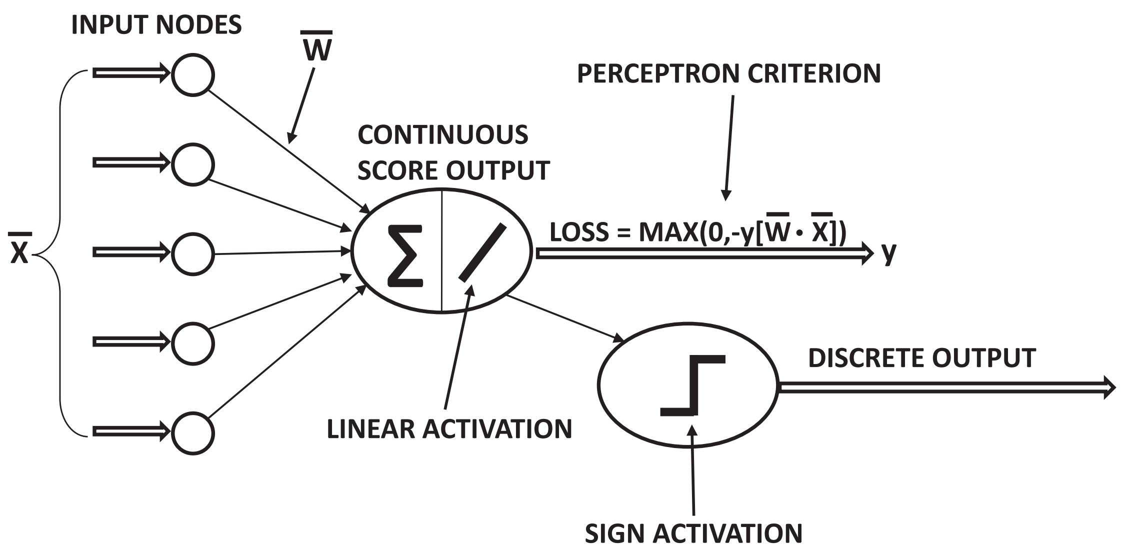
- In *multilayer* neural networks, activation functions bring non-linearity into hidden layers, which increases the complexity of the model.

  - A neural network with any number of layers but only linear activations can be shown to be equivalent to a single-layer network.

- Activation functions required for inference may be different from those used in loss functions in training.

  - Perceptron uses sign function $\Phi(v) = \text{sign}(v)$ for prediction but does not use any activation for computing the perceptron criterion (during training).

Perceptron: Activation Functions



- It is not quite correct to suggest that the perceptron uses sign activation $\Rightarrow$ Only for inference.
  - The perceptron uses *identity activation* (i.e., no activation function) for computing the loss function.

- Typically, a smooth activation function (unlike the sign function) is used to enable a differentiable loss function.

Why Do We Need Loss Functions?

- The loss function is typically paired with the activation function to quantify how far we are from a desired result.
- An example is the perceptron criterion.

$$L_i = \max\{-y(\overline{W} \cdot \overline{X}), 0\}$$

- Note that loss is 0, if the instance $(\overline{X}, y)$ is classified correctly.
- Even though many machine learning problems have discrete outputs, a smooth and continuous loss function is required to enable *gradient-descent* procedures.
- Gradient descent is at the heart of neural network parameter learning.

Identity Activation

- Identity activation $\Phi(v) = v$ is often used in the output layer, when the outputs are real values.

- For a single-layer network, if the training pair is $(\overline{X}, y)$, the output is as follows:

$$\hat{y} = \Phi(\overline{W} \cdot \overline{X}) = \overline{W} \cdot \overline{X}$$

- Use of the squared loss function $(y - \hat{y})^2$ leads to the *linear regression* model with numeric outputs and *Widrow-Hoff learning* with binary outputs.

- Identity activation can be combined with various types of loss functions (e.g., perceptron criterion) even for discrete outputs.

Sigmoid Activation

- Sigmoid activation is defined as $\Phi(v) = 1/(1 + \exp(-v))$.

- For a training pair $(\overline{X}, y)$, one obtains the following prediction in a single-layer network:

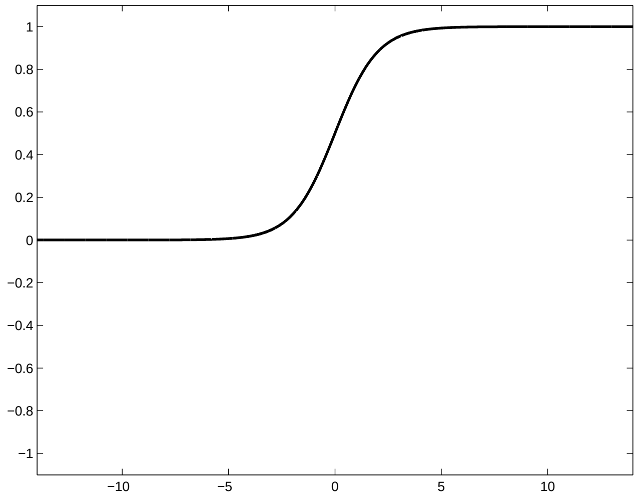
$$\hat{y} = 1/(1 + \exp(-\overline{W} \cdot \overline{X}))$$

- Prediction is the *probability* that class label is $+1$.

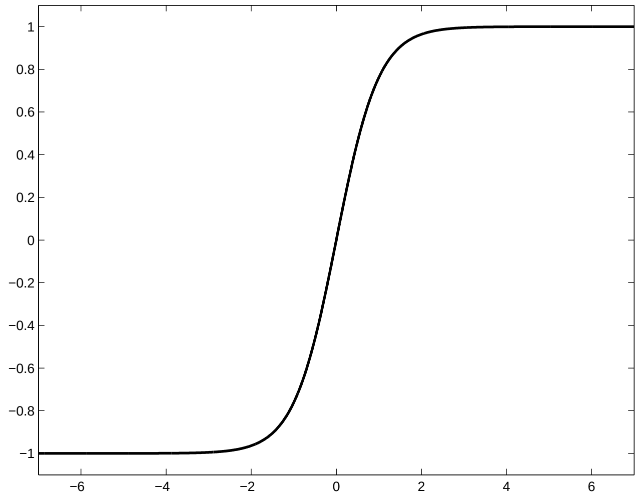
- Paired with *logarithmic loss*, which $-\log(\hat{y})$ for positive instances and $-\log(1 - \hat{y})$ for negative instances.

- Resulting model is *logistic regression*.

Tanh Activation



(a) Sigmoid



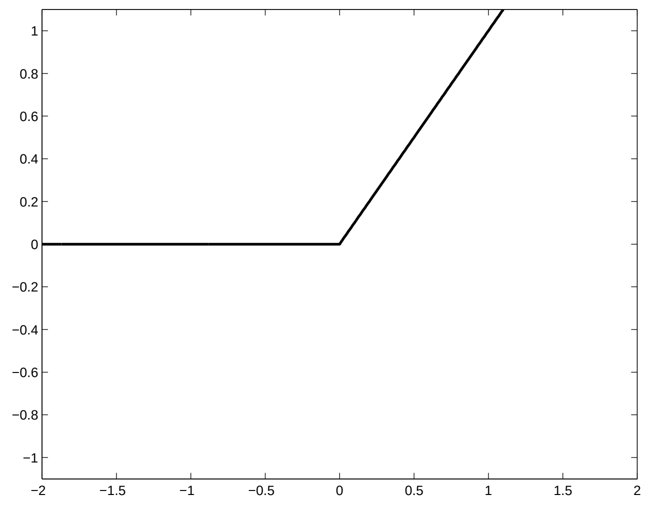
(b) Tanh

- The tanh activation is a scaled and translated version of sigmoid activation.

$$\text{tanh}(v) = \frac{e^{2v} - 1}{e^{2v} + 1} = 2 \cdot \text{sigmoid}(2v) - 1$$

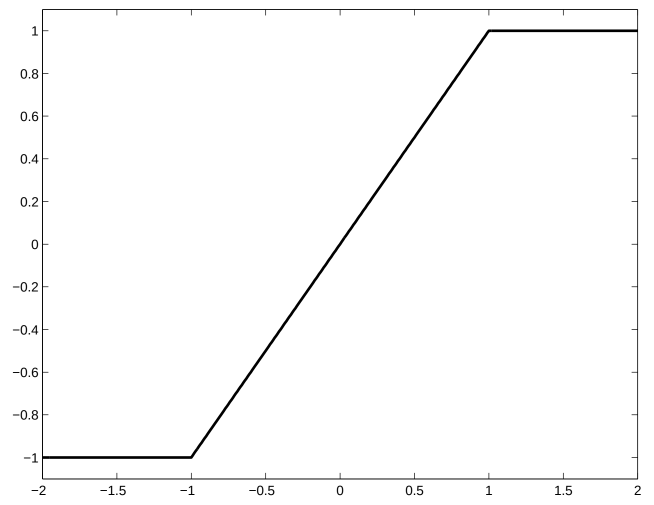
- Often used in hidden layers of multilayer networks

Piecewise Linear Activation Functions



(a) ReLU

$\Phi(v) = \max\{v, 0\}$



(b) Hard Tanh

$\Phi(v) = \max\{\min[v, 1], -1\}$

- Piecewise linear activation functions are easier to train than their continuous counterparts.

Softmax Activation Function

- All activation functions discussed so far map scalars to scalars.

- The softmax activation function maps vectors to vectors.

- Useful in mapping a set of real values to probabilities.

  – Generalization of sigmoid activation, which is used in *multiway* logistic regression.

- Discussed in detail in later lectures.

Derivatives of Activation Functions

- Neural network learning requires gradient descent of the loss.

- Loss is often a function of the output o, which is itself obtained by using the activation function:

$$o = \Phi(v) \tag{1}$$

- Therefore, we often need to compute the partial derivative of o with respect to v during neural network parameter learning.

- Many derivatives are more easily expressed in terms of the output o rather than input v.

Useful Derivatives

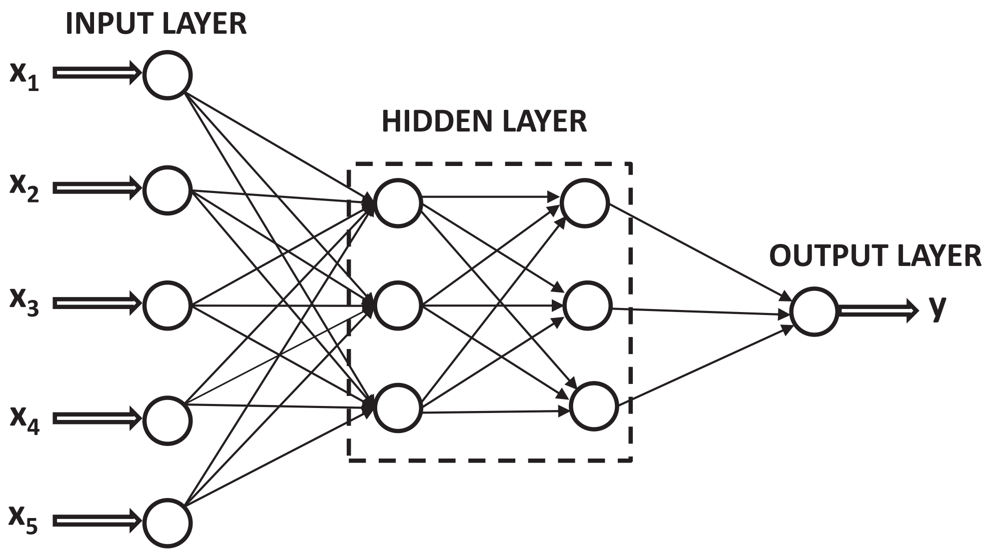
- Sigmoid: $\frac{\partial o}{\partial v} = o(1 - o)$
- Tanh: $\frac{\partial o}{\partial v} = 1 - o^2$
- ReLU: Derivative is 1 for positive values of v and 0 otherwise.
- Hard Tanh: Derivative is 1 for $v \in (-1, 1)$ and 0 otherwise.
- Best to commit to memory, since they are used repeatedly in neural network learning.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Multilayer Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 1, Section 1.3–1.6

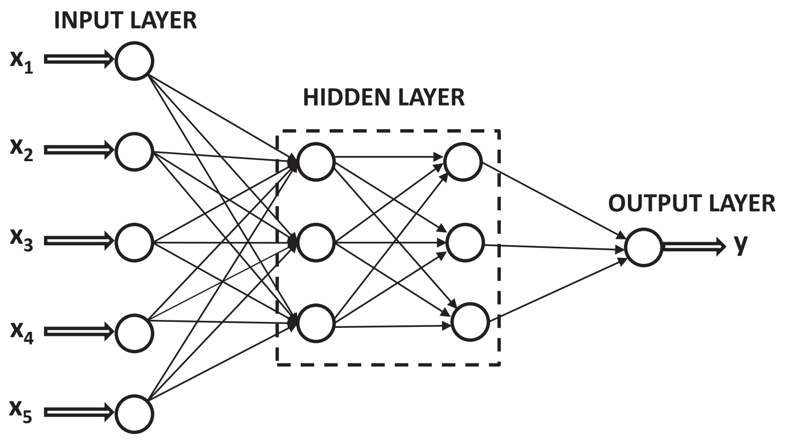
Multilayer Neural Networks



- In multilayer networks, the output of a node can feed into other *hidden* nodes, which in turn can feed into other hidden or output nodes.

- Multilayer neural networks are *always* directed acyclic graphs and *usually* arranged in layerwise fashion.

Multilayer Neural Networks

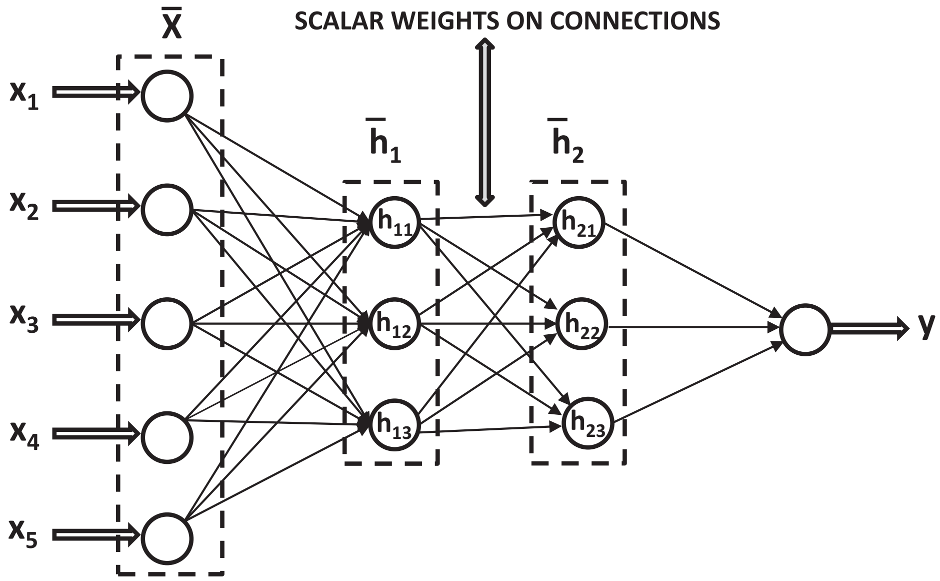


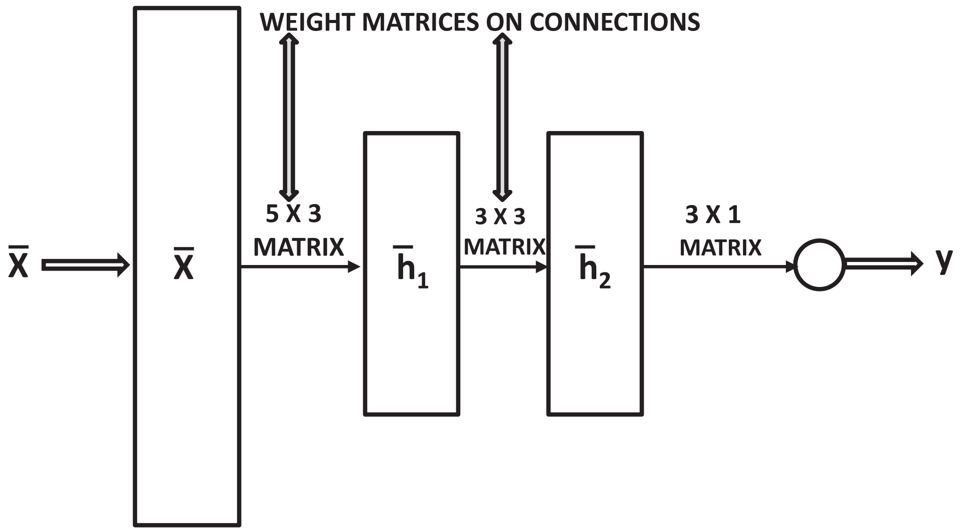
- The layers between the input and output are referred to as *hidden* because they perform intermediate computations.

- Each hidden node uses a combination of a linear transformation and an activation function $\Phi(\cdot)$ (like the output node of the perceptron).

- The use of *nonlinear* activation functions in the hidden layer is crucial in increasing learning capacity.

Scalar versus Vector Diagrams





- Hidden vectors $\overline{h}_1 \ldots \overline{h}_k$ and weight matrices $W_1 \ldots W_{k+1}$.

$$\overline{h}_1 = \Phi(W_1^T \overline{x}) \qquad \text{[Input to Hidden Layer]}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1}^T \overline{h}_p) \quad \forall p \in \{1 \ldots k-1\} \qquad \text{[Hidden to Hidden Layer]}$$
$$\overline{o} = \Phi(W_{k+1}^T \overline{h}_k) \qquad \text{[Hidden to Output Layer]}$$

- $\Phi(\cdot)$ is typically applied element-wise (other than softmax).

Using Activation Functions

- The nature of the activation in output layers is often controlled by the nature of output
  - Identity activation for real-valued outputs, and sigmoid/softmax for binary/categorical outputs.
  - Softmax almost exclusively for output layer and is paired with a particular type of *cross-entropy* loss.

- Hidden layer activations are almost always nonlinear and often use the same activation function over the entire network.
  - Tanh often (but not always) preferable to sigmoid.
  - ReLU has largely replaced tanh and sigmoid in many applications.

Why are Hidden Layers Nonlinear?

- A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network that performs linear regression.

$$\overline{h}_1 = \Phi(W_1^T \overline{x}) = W_1^T \overline{x}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1}^T \overline{h}_p) = W_{p+1}^T \overline{h}_p \quad \forall p \in \{1 \ldots k-1\}$$
$$\overline{o} = \Phi(W_{k+1}^T \overline{h}_k) = W_{k+1}^T \overline{h}_k$$

- We can eliminate the hidden variable to get a simple linear relationship:

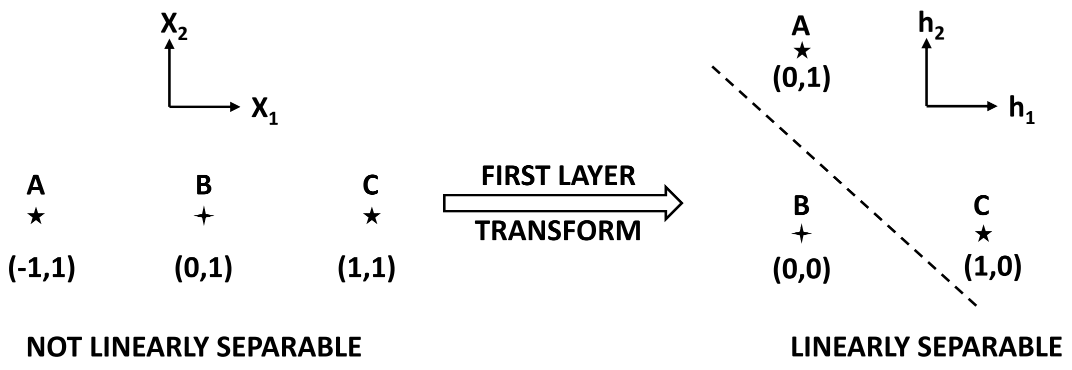
$$\begin{aligned}\overline{o} &= W_{k+1}^T W_k^T \ldots W_1^T \overline{x} \\ &= \underbrace{(W_1 W_2 \ldots W_{k+1})^T}_{W_{xo}^T} \overline{x}\end{aligned}$$

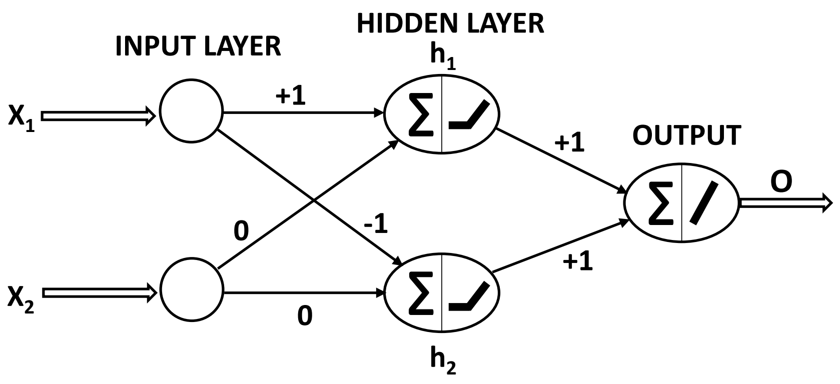
- We get a *single-layer* network with matrix W_{xo}.

Role of Hidden Layers

- Nonlinear hidden layers perform the role of hierarchical feature engineering.
  - Early layers learn primitive features and later layers learn more complex features
  - Image data: Early layers learn elementary edges, the middle layers contain complex features like honeycombs, and later layers contain complex features like a part of a face.
  - *Deep learners are masters of feature engineering*.
- The final output layer is often able to perform inference with transformed features in penultimate layer relatively easily.
- **Perceptron:** Cannot classify linearly inseparable data but can do so with *nonlinear* hidden layers.

Example of Classifying Inseparable Data

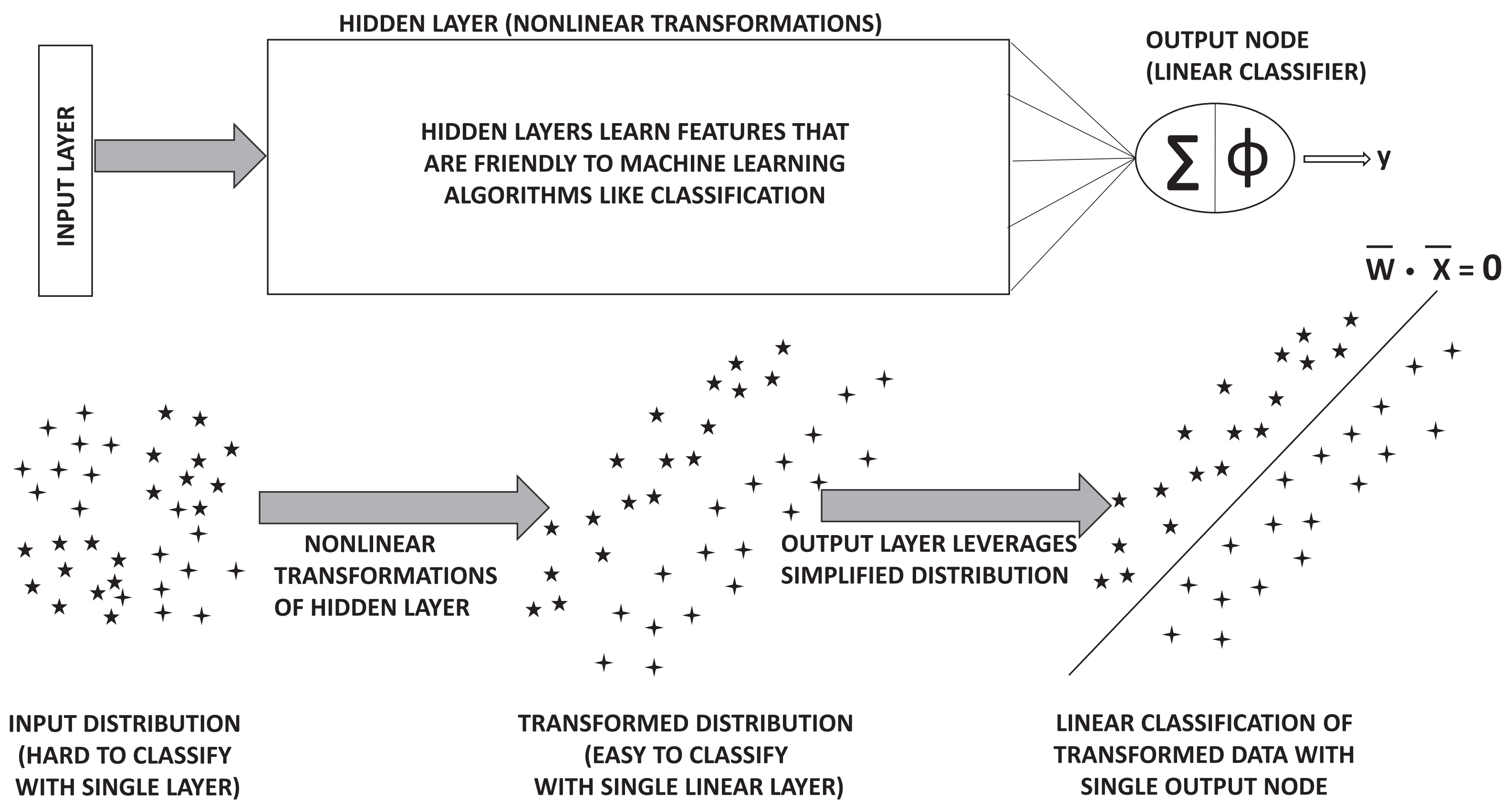




- The hidden units have ReLU activation, and they learn the two new features h_1 and h_2 with linear separator $h_1 + h_2 = 0.5$:

$$h_1 = \max\{x_1, 0\} \quad h_2 = \max\{-x_1, 0\}$$

The Feature Engineering View of Hidden Layers



HIDDEN LAYER (NONLINEAR TRANSFORMATIONS)
INPUT LAYER
HIDDEN LAYERS LEARN FEATURES THAT ARE FRIENDLY TO MACHINE LEARNING ALGORITHMS LIKE CLASSIFICATION
OUTPUT NODE (LINEAR CLASSIFIER)
Σ Φ
y
W · X = 0
NONLINEAR TRANSFORMATIONS OF HIDDEN LAYER
OUTPUT LAYER LEVERAGES SIMPLIFIED DISTRIBUTION
INPUT DISTRIBUTION (HARD TO CLASSIFY WITH SINGLE LAYER)
TRANSFORMED DISTRIBUTION (EASY TO CLASSIFY WITH SINGLE LINEAR LAYER)
LINEAR CLASSIFICATION OF TRANSFORMED DATA WITH SINGLE OUTPUT NODE

- Early layers play the role of feature engineering for later layers.

Multilayer Networks as Computational Graphs

- Consider the case in which each layer computes the vector-to-vector function f_i.

- The overall *composition* function is $f_k o f_{k-1} o \ldots o f_1$

- Function of a function with k levels of recursive nesting!

- It is a complex and ugly-looking function, which is powerful and typically cannot be expressed in closed form.

- The neural network architecture is a directed acyclic computational graph in which each layer is often fully connected.

How Do We Train?

- We want to compute the derivatives with respect to the parameters in all layers to perform gradient descent.

- The complex nature of the composition function makes this difficult.

- The key idea to achieve this goal is *backpropagation*.

- Use the *chain rule of differential calculus* as a dynamic programming update on a directed acyclic graph.

- Details in later lectures.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Connecting Machine Learning with Shallow Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.1

Neural Networks and Machine Learning

- Neural networks are optimization-based learning models.

- Many classical machine learning models use continuous optimization:

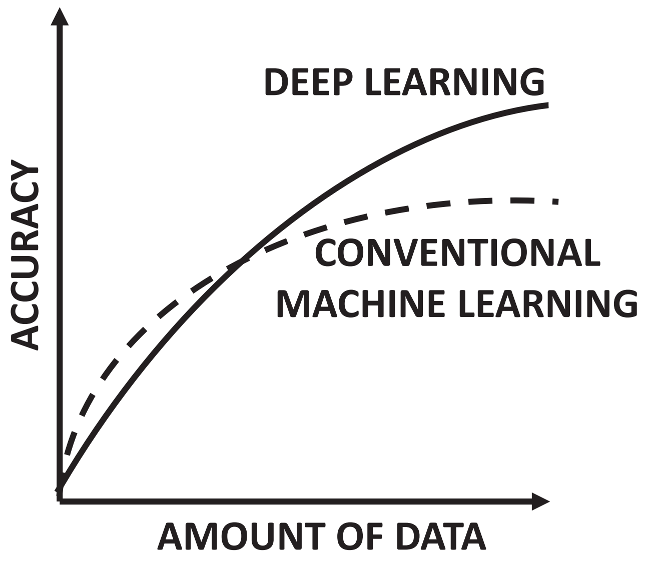
  - SVMs, Linear Regression, and Logistic Regression

  - Singular Value Decomposition

  - (Incomplete) Matrix factorization for Recommender Systems

- All these models can be represented as special cases of shallow neural networks!

The Continuum Between Machine Learning and Deep Learning



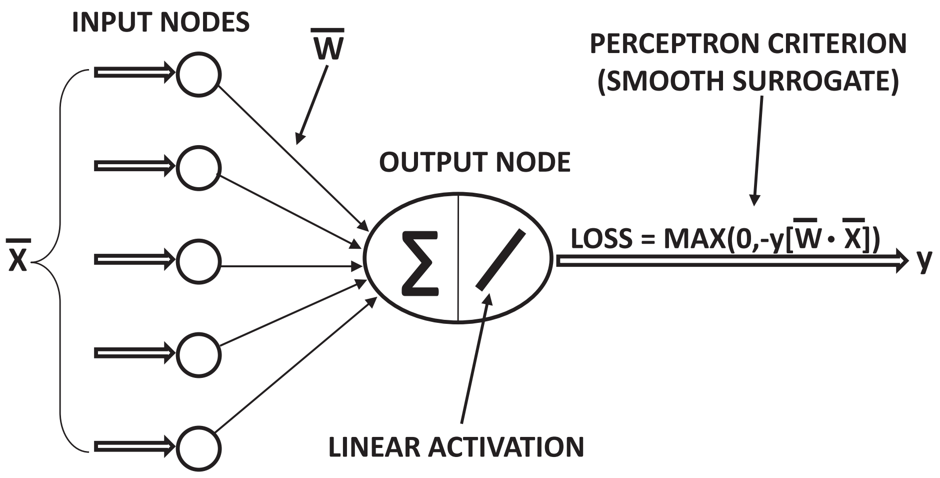
- Classical machine learning models reach their learning capacity early because they are simple neural networks.

- When we have more data, we can add more computational units to improve performance.

The Deep Learning Advantage

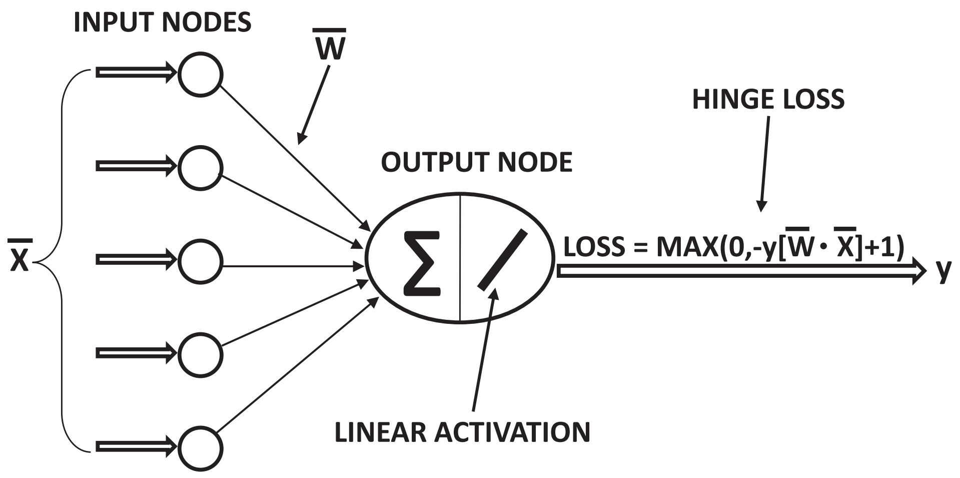
- Exploring the neural models for traditional machine learning is useful because it exposes the cases in which deep learning has an advantage.
  - Add capacity with more nodes for more data.
  - Controlling the structure of the architecture provides a way to incorporate domain-specific insights (e.g., recurrent networks and convolutional networks).
- In some cases, making minor changes to the architecture leads to interesting models:
  - Adding a sigmoid/softmax layer in the output of a neural model for (linear) matrix factorization can result in logistic/multinomial matrix factorization (e.g., word2vec).

Recap: Perceptron versus Linear Support Vector Machine



(a) Perceptron

Loss $= \max\{0, -y(\overline{W} \cdot \overline{X})\}$



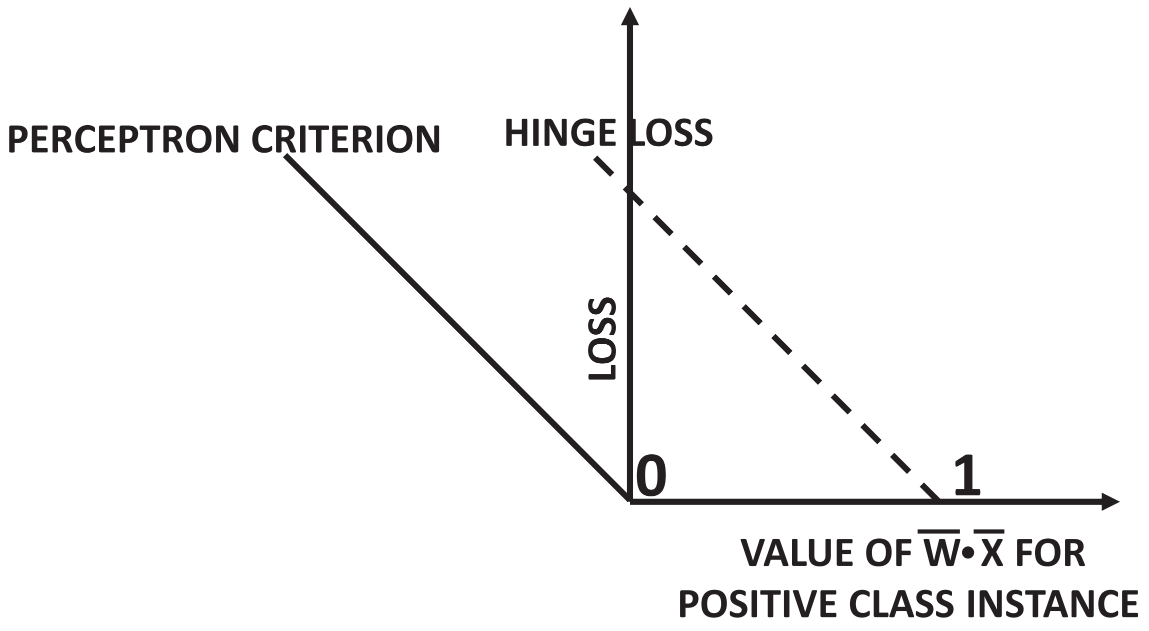
(b) SVM

Loss $= \max\{0, 1 - y(\overline{W} \cdot \overline{X})\}$

- The Perceptron criterion is a minor variation of hinge loss with identical update of $\overline{W} \Leftarrow \overline{W} + \alpha y \overline{X}$ in both cases.

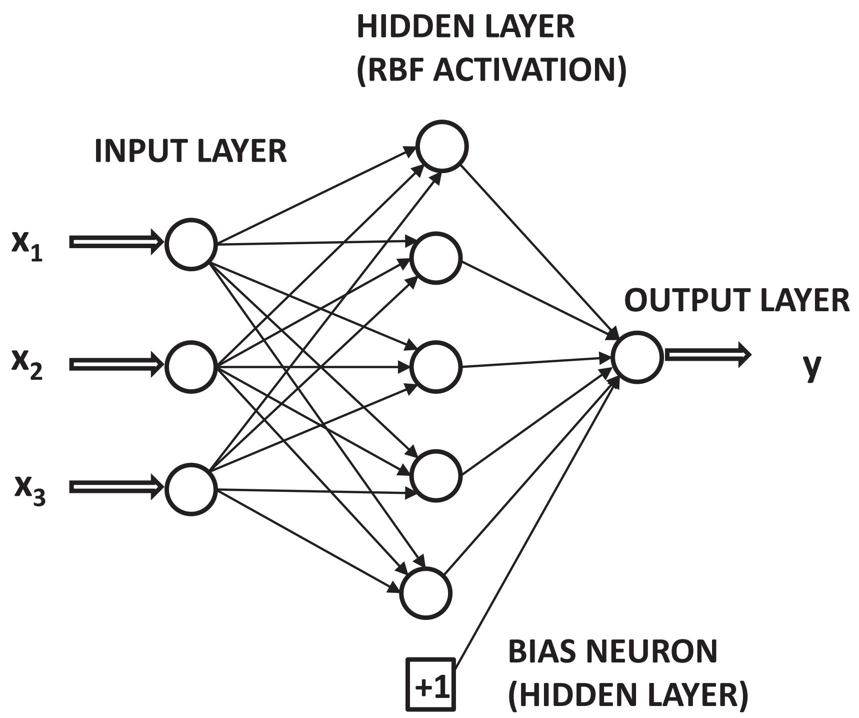
- We update only for misclassified instances in perceptron, but update *also* for "marginally correct" instances in SVM.

Perceptron Criterion versus Hinge Loss



- Loss for positive class training instance at varying values of $\overline{W} \cdot \overline{X}$.

What About the Kernel SVM?



- RBF Network for unsupervised feature engineering.
  - Unsupervised feature engineering is good for noisy data.
  - Supervised feature engineering (with deep learning) is good for learning rich structure.

Much of Machine Learning is a Shallow Neural Model

- By minor changes to the architecture of perceptron we can get:
  - Linear regression, Fisher discriminant, and Widrow-Hoff learning $\Rightarrow$ Linear activation in output node
  - Logistic regression $\Rightarrow$ Sigmoid activation in output node
- Multinomial logistic regression $\Rightarrow$ Softmax Activation in Final Layer
- Singular value decomposition $\Rightarrow$ Linear autoencoder
- Incomplete matrix factorization for Recommender Systems $\Rightarrow$ Autoencoder-like architecture with single hidden layer (also used in *word2vec*)

Why do We Care about Connections?

- Connections tell us about the cases that it makes sense to use conventional machine learning:
  - If you have less data with noise, you want to use conventional machine learning.
  - If you have a lot of data with rich structure, you want to use neural networks.
  - Structure is often learned by using deep neural architectures.

- Architectures like convolutional neural networks can use domain-specific insights.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Neural Models for Linear Regression, Classification, and the Fisher Discriminant [Connections with Widrow-Hoff Learning]

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.2

Widrow-Hoff Rule: The Neural Avatar of Linear Regression

- The perceptron (1958) was historically followed by Widrow-Hoff Learning (1960).

- Identical to linear regression when applied to numerical targets.

  - Originally proposed by Widrow and Hoff for binary targets (not natural for regression).

- The Widrow-Hoff method, when applied to mean-centered features and mean-centered binary class encoding, learns the Fisher discriminant.

- We first discuss linear regression for numeric classes and then visit the case of binary classes.

Linear Regression: An Introduction

- In linear regression, we have training pairs $(\overline{X}_i, y_i)$ for $i \in \{1 \ldots n\}$, so that $\overline{X_i}$ contains d-dimensional features and y_i contains a numerical target.

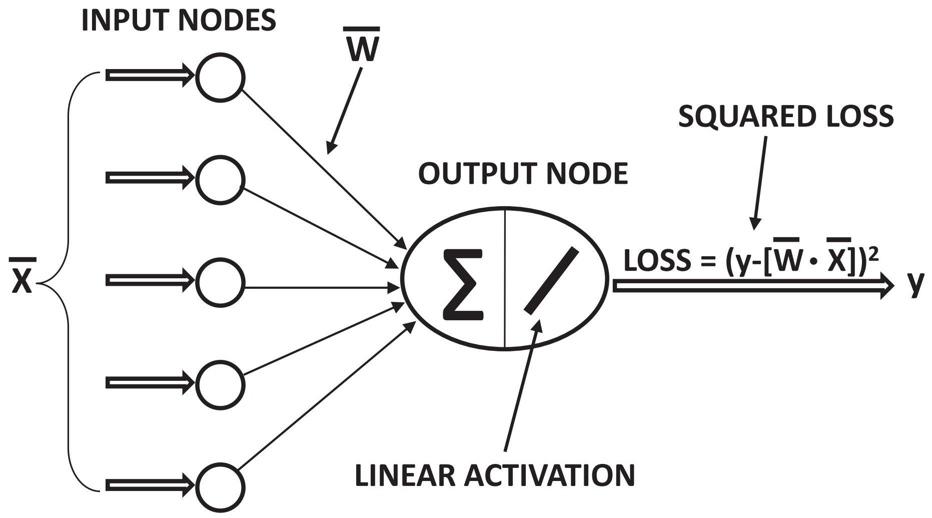
- We use a linear parameterized function to predict $\hat{y}_i = \overline{W} \cdot \overline{X}_i$.

- Goal is to learn $\overline{W}$, so that the sum-of-squared differences between observed y_i and predicted $\hat{y}_i$ is minimized over the entire training data.

- Solution exists in closed form, but requires the inversion of a potentially large matrix.

- Gradient-descent is typically used anyway.

Linear Regression with Numerical Targets:Neural Model



- Predicted output is $\hat{y}_i = \overline{W} \cdot \overline{X}_i$ and loss is $L_i = (y_i - \hat{y}_i)^2$.
- Gradient-descent update is $\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} = \overline{W} + \alpha (y_i - \hat{y}_i) \overline{X}_i$.

Widrow-Hoff: Linear Regression with Binary Targets

- For $y_i \in \{-1, +1\}$, we use same loss of $(y_i - \hat{y}_i)^2$, and update of $\overline{W} \Leftarrow \overline{W} + \alpha \underbrace{(y_i - \hat{y}_i)}_{\text{delta}} \overline{X}_i$.
  - When applied to binary targets, it is referred to as delta rule.
  - Perceptron uses the same update with $\hat{y}_i = \text{sign}\{\overline{W} \cdot \overline{X}_i\}$, whereas Widrow-Hoff uses $\hat{y}_i = \overline{W} \cdot \overline{X}_i$.

- **Potential drawback:** Retrogressive treatment of well-separated points caused by the pretension that binary targets are real-valued.
  - If $y_i = +1$, and $\overline{W} \cdot \overline{X}_i = 10^6$, the point will be heavily penalized for strongly correct classification!
  - Does not happen in perceptron.

Comparison of Widrow-Hoff with Perceptron and SVM

- Convert the binary loss functions and updates to a form more easily comparable to perceptron using $y_i^2 = 1$:

- Loss of $(\overline{X_i}, y_i)$ is $(y_i - \overline{W} \cdot \overline{X_i})^2 = (1 - y_i[\overline{W} \cdot \overline{X_i}])^2$
Update: $\overline{W} \Leftarrow \overline{W} + \alpha y_i (1 - y_i[\overline{W} \cdot \overline{X_i}])\overline{X_i}$

	Perceptron	L_1-Loss SVM
Loss	$\max\{-y_i(\overline{W} \cdot \overline{X_i}), 0\}$	$\max\{1 - y_i(\overline{W} \cdot \overline{X_i}), 0\}$
Update	$\overline{W} \Leftarrow \overline{W} + \alpha y_i I(-y_i[\overline{W} \cdot \overline{X_i}] > 0)\overline{X_i}$	$\overline{W} \Leftarrow \overline{W} + \alpha y_i I(1 - y_i[\overline{W} \cdot \overline{X_i}] > 0)\overline{X_i}$

	Widrow-Hoff	Hinton's L_2-Loss SVM
Loss	$(1 - y_i(\overline{W} \cdot \overline{X_i}))^2$	$\max\{1 - y_i(\overline{W} \cdot \overline{X_i}), 0\}^2$
Update	$\overline{W} \Leftarrow \overline{W} + \alpha y_i (1 - y_i[\overline{W} \cdot \overline{X_i}])\overline{X_i}$	$\overline{W} \Leftarrow \overline{W} + \alpha y_i \max\{(1 - y_i[\overline{W} \cdot \overline{X_i}]), 0\}\overline{X_i}$

Some Interesting Historical Facts

- Hinton proposed the SVM L_2-loss three years before Cortes and Vapnik's paper on SVMs.
  - G. Hinton. Connectionist learning procedures. *Artificial Intelligence*, 40(1–3), pp. 185–234, 1989.
  - Hinton's L_2-loss was proposed to address some of the weaknesses of loss functions like linear regression on binary targets.
  - When used with L_2-regularization, it behaves identically to an L_2-SVM, but the connection with SVM was overlooked.
- The Widrow-Hoff rule is also referred to as ADALINE, LMS (least mean-square method), delta rule, and least-squares classification.

Connections with Fisher Discriminant

- Consider a binary classification problem with training instances $(\overline{X}_i, y_i)$ and $y_i \in \{-1, +1\}$.
  - Mean-center each feature vector as $\overline{X}_i - \overline{\mu}$.
  - Mean-center the binary class by subtracting $\sum_{i=1}^{n} y_i/n$ from each y_i.
- Use the delta rule $\overline{W} \Leftarrow \overline{W} + \alpha \underbrace{(y_i - \hat{y}_i)}_{\text{delta}} \overline{X}_i$ for learning.
- Learned vector is the Fisher discriminant!
  - Proof in Christopher Bishop's book on machine learning.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Neural Models for Logistic Regression

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.2

Logistic Regression: A Probabilistic Model

- Consider the training pair $(\overline{X}_i, y_i)$ with d-dimensional feature variables in $\overline{X_i}$ and class variable $y_i \in \{-1, +1\}$.

- In logistic regression, the sigmoid function is applied to $\overline{W} \cdot \overline{X}_i$, which predicts the probability that y_i is $+1$.

$$\hat{y}_i = P(y_i = 1) = \frac{1}{1 + \exp(-\overline{W} \cdot \overline{X}_i)}$$

- We want to *maximize* $\hat{y}_i$ for positive class instances and $1 - \hat{y}_i$ for negative class instances.

  - Same as *minimizing* $-\log(\hat{y}_i)$ for positive class instances and $-\log(1 - \hat{y}_i)$ for negative instances.

  - Same as minimizing loss $L_i = -\log(|y_i/2 - 0.5 + \hat{y}_i|)$.

  - Alternative form of loss $L_i = \log(1 + \exp[-y_i(\overline{W} \cdot \overline{X}_i)])$

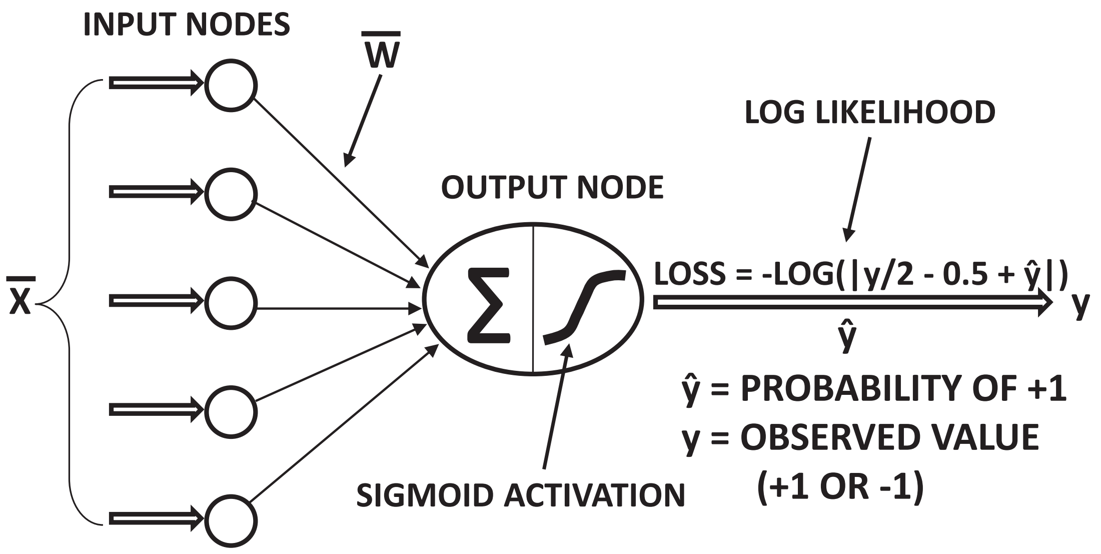
Maximum-Likelihood Objective Functions

- Why did we use the negative logarithms?

- Logistic regression is an example of a maximum-likelihood objective function.

- Our goal is to maximize the *product* of the probabilities of correct classification over all training instances.
  - Same as minimizing the *sum* of the negative log probabilities.
  - Loss functions are always *additive* over training instances.
  - So we are really minimizing $\sum_i -\log(|y_i/2 - 0.5 + \hat{y}_i|)$ which can be shown to be $\sum_i \log(1 + \exp[-y_i(\overline{W} \cdot \overline{X}_i)])$.

Logistic Regression: Neural Model



- Predicted output is $\hat{y}_i = 1/(1 + \exp(-\overline{W} \cdot \overline{X}_i))$ and loss is $L_i = -\log(|y_i/2 - 0.5 + \hat{y}_i|) = \log(1 + \exp[-y_i(\overline{W} \cdot \overline{X}_i)])$.
  - Gradient-descent update is $\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}}$.

$$\overline{W} \Leftarrow \overline{W} + \alpha \frac{y_i \overline{X}_i}{1 + \exp[y_i(\overline{W} \cdot \overline{X}_i)]}$$

Interpreting the Logistic Update

- An important multiplicative factor in the update increment is $1/(1 + \exp[y_i(\overline{W} \cdot \overline{X}_i)])$.

- This factor is $1 - \hat{y}_i$ for positive instances and $\hat{y}_i$ for negative instances $\Rightarrow$ Probability of mistake!

- Interpret as: $\overline{W} \Leftarrow \overline{W} + \alpha \left[\text{Probability of mistake on } (\overline{X_i}, y_i) \right] (y_i \overline{X_i})$

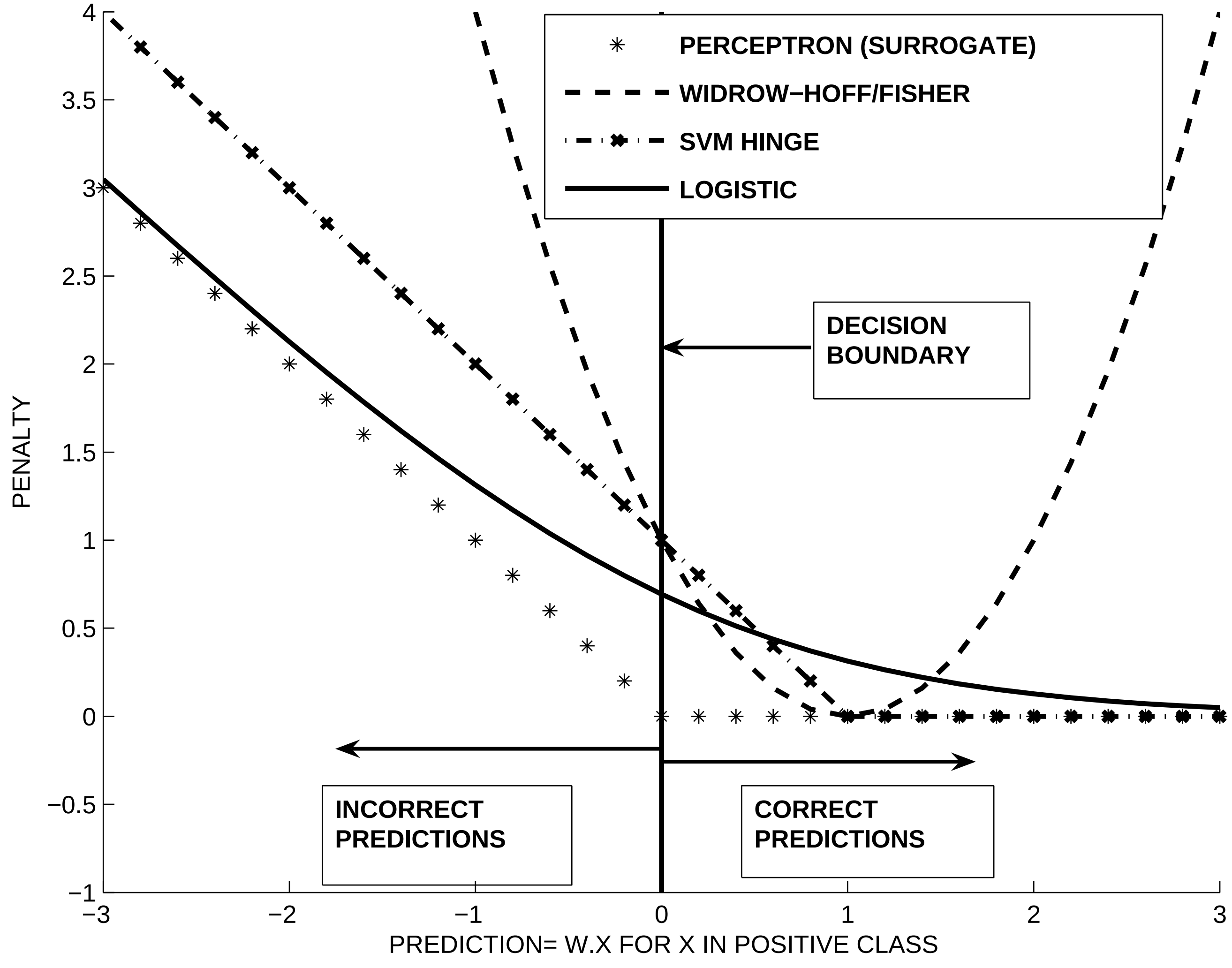
Comparing Updates of Different Models

- The unregularized updates of the perceptron, SVM, Widrow-Hoff, and logistic regression can all be written in the following form:

$$\overline{W} \Leftarrow \overline{W} + \alpha y_i \delta(\overline{X}_i, y_i)\overline{X}_i$$

- The quantity $\delta(\overline{X}_i, y_i)$ is a *mistake function*, which is:
  - Raw mistake value $(1 - y_i(\overline{W} \cdot \overline{X}_i))$ for Widrow-Hoff
  - Mistake indicator whether $(0 - y_i(\overline{W} \cdot \overline{X}_i)) > 0$ for perceptron.
  - Margin/mistake indicator whether $(1 - y_i(\overline{W} \cdot \overline{X}_i)) > 0$ for SVM.
  - Probability of mistake on $(\overline{X}_i, y_i)$ for logistic regression.

Comparing Loss Functions of Different Models



- Loss functions are similar (note Widrow-Hoff retrogression).

Other Comments on Logistic Regression

- Many classical neural models use repeated computational units with logistic and tanh activation functions in hidden layers.

- One can view these methods as feature engineering models that stack multiple logistic regression models.

- The stacking of multiple models creates inherently more powerful models than their individual components.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

The Softmax Activation Function and Multinomial Logistic Regression

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.3

Binary Classes versus Multiple Classes

- All the models discussed so far discuss only the binary class setting in which the class label is drawn from $\{-1, +1\}$.

- Many natural applications contain multiple classes without a natural ordering among them:
  - Predicting the category of an image (e.g., *truck*, *carrot*).
  - *Language models:* Predict the next word in a sentence.

- Models like logistic regression are naturally designed to predict two classes.

Generalizing Logistic Regression

- Logistic regression produces probabilities of the two outcomes of a binary class.

- *Multinomial* logistic regression produces probabilities of multiple outcomes.

  - In order to produce probabilities of multiple classes, we need an activation function with a vector output of probabilities.

  - The *softmax activation function* is a vector-based generalization of the sigmoid activation used in logistic regression.

- Multinomial logistic regression is also referred to as softmax classifier.

The Softmax Activation Function

- The softmax activation function is a natural vector-centric generalization of the scalar-to-scalar sigmoid activation $\Rightarrow$ vector-to-vector function.

- Logistic sigmoid activation: $\Phi(v) = 1/(1 + \exp(-v))$.

- Softmax activation: $\Phi(v_1 \ldots v_k) = \frac{1}{\sum_{i=1}^{k} \exp(v_i)} [\exp(v_1) \ldots \exp(v_k)]$
  - The k outputs (probabilities) sum to 1.

- Binary case of using sigmoid(v) is identical to using 2-element softmax activation with arguments $(v, 0)$.
  - Multinomial logistic regression with 2-element softmax is equivalent to binary logistic regression.

Loss Functions for Softmax

- Recall that we use the negative logarithm of the probability of observed class in binary logistic regression.
  - Natural generalization to multiple classes.
  - Cross-entropy loss: Negative logarithm of the probability of correct class.
  - Probability distribution among incorrect classes has no effect.
- Softmax activation is used almost exclusively in output layer and (almost) always paired with cross-entropy loss.

Cross-Entropy Loss of Softmax

- Like the binary logistic case, the loss L is a negative log probability.

$$\text{Softmax Probability Vector} \Rightarrow [\widehat{y}_1, \widehat{y}_2, \ldots \widehat{y}_k]$$

$$[\widehat{y}_1 \ldots \widehat{y}_k] = \frac{1}{\sum_{i=1}^{k} \exp(v_i)} [\exp(v_1) \ldots \exp(v_k)]$$

- The loss is $-\log(\widehat{y}_c)$, where $c \in \{1 \ldots k\}$ is the correct class of that training instance.

- Cross entropy loss is $-v_{c)} + \log[\sum_{j=1}^{k} \exp(v_j)]$

Loss Derivative of Softmax

- Since softmax is almost always paired with cross-entropy loss L, we can directly estimate $\frac{\partial L}{\partial v_r}$ for each pre-activation value from $v_1 \ldots v_k$.

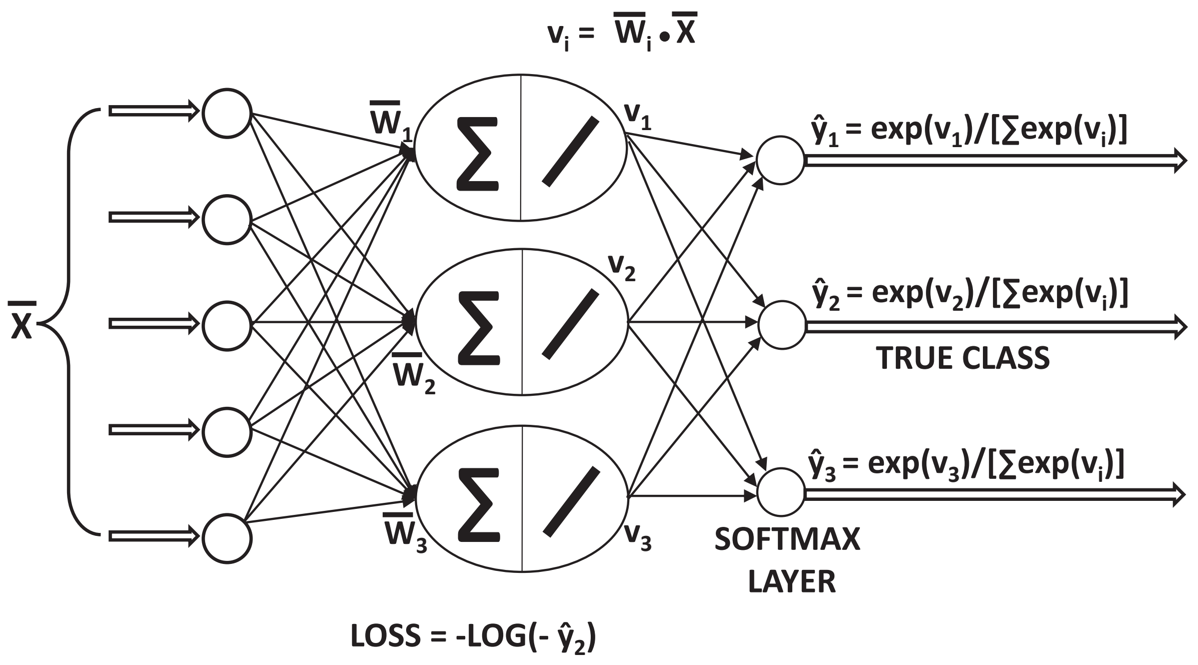
- Differentiate loss value of $-v_c + \log[\sum_{j=1}^{k} \exp(v_j)]$

- Like the sigmoid derivative, the result is best expressed in terms of the post-activation values $\widehat{y}_1 \ldots \widehat{y}_k$.

- The loss derivative of the softmax is as follows:

$$\frac{\partial L}{\partial v_r} = \begin{cases} \widehat{y}_r - 1 & \text{If } r \text{ is correct class} \\ \widehat{y}_r & \text{If } r \text{ is not correct class} \end{cases}$$

Multinomial Logistic Regression



- The ith training instance is $(\overline{X}_i, c(i))$, where $c(i) \in \{1 \ldots k\}$ is class index $\Rightarrow$ Learn k parameter vectors $\overline{W}_1 \ldots \overline{W}_k$.
  - Define real-valued score $v_r = \overline{W}_r \cdot \overline{X}_i$ for rth class.
  - Convert scores to probabilities $\widehat{y}_1 \ldots \widehat{y}_k$ with softmax activation on $v_1 \ldots v_k \Rightarrow$ Hard or soft prediction

Computing the Derivative of the Loss

- The cross-entropy loss for the ith training instance is $L_i = -\log(\hat{y}_{c(i)})$.

- For gradient-descent, we need to compute $\frac{\partial L_i}{\partial \overline{W}_r}$.

- Using chain rule of differential calculus, we get:

$$\frac{\partial L_i}{\partial \overline{W}_r} = \sum_j \left(\frac{\partial L_i}{\partial v_j}\right)\left(\frac{\partial v_j}{\partial \overline{W}_r}\right) = \frac{\partial L_i}{\partial v_r} \underbrace{\frac{\partial v_r}{\partial \overline{W}_r}}_{\overline{X}_i} + \text{Zero-terms}$$

$$= \begin{cases} -\overline{X}_i(1 - \hat{y}_r) & \text{if } r = c(i) \\ \overline{X}_i \, \hat{y}_r & \text{if } r \neq c(i) \end{cases}$$

Gradient Descent Update

- Each separator $\overline{W_r}$ is updated using the gradient:

$$\overline{W_r} \Leftarrow \overline{W_r} - \alpha \frac{\partial L_i}{\partial \overline{W_r}}$$

- Substituting the gradient from the previous slide, we obtain:

$$\overline{W_r} \Leftarrow \overline{W_r} + \alpha \begin{cases} \overline{X_i} \cdot (1 - \widehat{y}_r) & \text{if } r = c(i) \\ -\overline{X_i} \cdot \widehat{y}_r & \text{if } r \neq c(i) \end{cases}$$

Summary

- The book also contains details of the multiclass Perceptron and Weston-Watkins SVM.

- Multinomial logistic regression is a direct generalization of logistic regression.

- If we apply the softmax classifier with two classes, we will obtain $\overline{W_1} = -\overline{W_2}$ to be the same separator as obtained in logistic regression.

- Cross-entropy loss and softmax are almost always paired in output layer (for all types of architectures).

  – Many of the calculus derivations in previous slides are repeatedly used in different settings.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

The Autoencoder for Unsupervised Representation Learning

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.5

Unsupervised Learning

- The models we have discussed so far use training pairs of the form $(\overline{X}, y)$ in which the feature variables $\overline{X}$ and target y are clearly separated.
  - The target variable y provides the *supervision* for the learning process.

- What happens when we do not have a target variable?
  - We want to capture a model of the training data without the guidance of the target.
  - This is an *unsupervised* learning problem.

Example

- Consider a 2-dimensional data set in which all points are distributed on the circumference of an origin-centered circle.

- All points in the first and third quadrant belong to class $+1$ and remaining points are -1.

  - The class variable provides focus to the learning process of the supervised model.

  - An unsupervised model needs to recognize the circular manifold without being told up front.

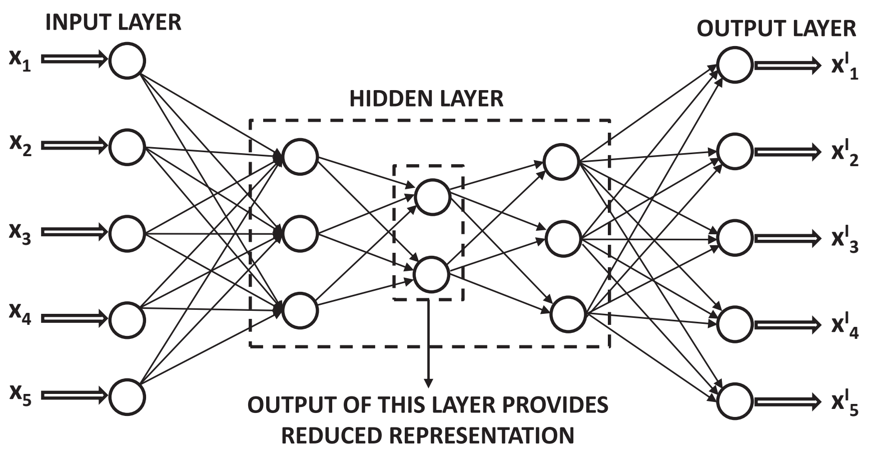
  - The unsupervised model can represent the data in only 1 dimension (angular position).

- Best way of modeling is data-set dependent $\Rightarrow$ Lack of supervision causes problems

Unsupervised Models and Compression

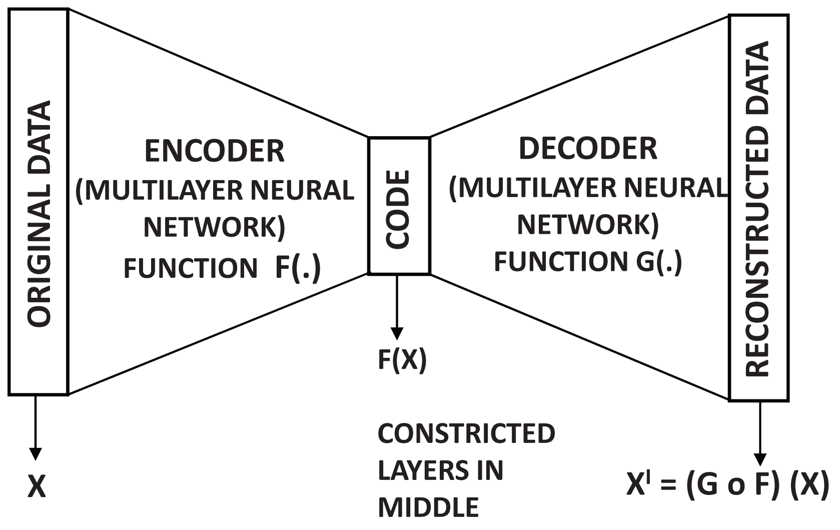
- Unsupervised models are closely related to compression because compression captures a model of regularities in the data.
  - Generative models represent the data in terms of a compressed parameter set.
  - Clustering models represent the data in terms of cluster statistics.
  - Matrix factorization represents data in terms of low-rank approximations (compressed matrices).
- An autoencoder also provides a compressed representation of the data.

Defining the Input and Output of an Autoencoder



- All neural networks work with input-output pairs.
  - In a supervised problem, the output is the label.
- In the autoencoder, the output values are the same as inputs: *replicator neural network*.
  - The loss function penalizes a training instance depending on how far it is from the input (e.g., squared loss).

Encoder and Decoder



- Reconstructing the data might seem like a trivial matter by simply copying the data forward from one layer to another.
  - Not possible when the number of units in the middle are *constricted*.
  - Autoencoder is divided into *encoder* and *decoder*.

Basic Structure of Autoencoder

- It is common (but not necessary) for an M-layer autoencoder to have a symmetric architecture between the input and output.
  - The number of units in the kth layer is the same as that in the $(M-k+1)$th layer.
- The value of M is often odd, as a result of which the $(M+1)/2$th layer is often the most constricted layer.
  - We are counting the (non-computational) input layer as the first layer.
  - The minimum number of layers in an autoencoder would be three, corresponding to the input layer, constricted layer, and the output layer.

Undercomplete Autoencoders and Dimensionality Reduction

- The number of units in each middle layer is typically fewer than that in the input (or output).
  - These units hold a reduced representation of the data, and the final layer can no longer reconstruct the data exactly.
- This type of reconstruction is inherently *lossy*.
- The activations of hidden layers provide an alternative to linear and nonlinear dimensionality reduction techniques.

Overcomplete Autoencoders and Representation Learning

- What happens if the number of units in hidden layer is equal to or larger than input/output layers?
  - There are infinitely many hidden representations with zero error.
  - The middle layers often do not learn the identity function.
  - We can enforce specific properties on the redundant representations by adding constraints/regularization to hidden layer.
    - Training with stochastic gradient descent is itself a form of regularization.
    - One can learn sparse features by adding sparsity constraints to hidden layer.

Applications

- Dimensionality reduction $\Rightarrow$ Use activations of constricted hidden layer

- Sparse feature learning $\Rightarrow$ Use activations of constrained/regularized hidden layer

- Outlier detection: Find data points with larger reconstruction error

  – Related to denoising applications

- Generative models with probabilistic hidden layers (variational autoencoders)

- Representation learning $\Rightarrow$ Pretraining

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Singular Value Decomposition with Autoencoders

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.5

Singular Value Decomposition

- Truncated SVD is the *approximate* decomposition of an $n \times d$ matrix D into $D \approx Q \Sigma P^T$, where Q, Σ, and P are $n \times k$, $k \times k$, and $d \times k$ matrices, respectively.
  - Orthonormal columns of each of P, Q, and nonnegative diagonal matrix Σ.
  - Minimize the squared sum of residual entries in $D - Q \Sigma P^T$.
  - The value of k is typically much smaller than $\min\{n, d\}$.
  - Setting k to $\min\{n, d\}$ results in a zero-error decomposition.

Relaxed and Unnormalized Definition of SVD

- **Two-way Decomposition:** Find and $n \times k$ matrix U, and $d \times k$ matrix V so that $||D - UV^T||^2$ is minimized.
  - Property: At least one optimal pair U and V will have mutually orthogonal columns (but non-orthogonal alternatives will exist).
  - The orthogonal solution can be converted into the 3-way factorization of SVD.
  - Exercise: Given U and V with orthogonal columns, find Q, Σ and P.

- In the event that U and V have non-orthogonal columns at optimality, these columns will span the same subspace as the orthogonal solution at optimality.

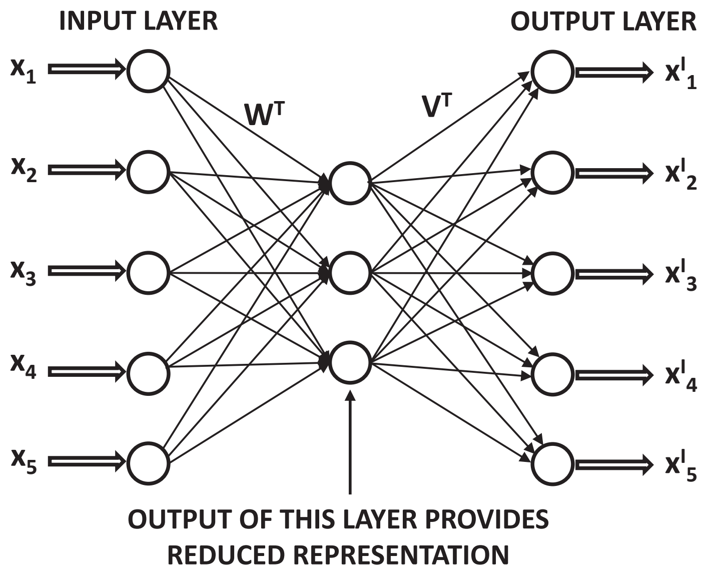
Dimensionality Reduction and Matrix Factorization

- Singular value decomposition is a dimensionality reduction method (like any matrix factorization technique).

$$D \approx UV^T$$

- The n rows of D contain the n training points.
- The n rows of U provide the reduced representations of the training points.
- The k columns of V contain the orthogonal basis vectors.

The Autoencoder Architecture for SVD



- The rows of the matrix D are input to encoder.

- The activations of hidden layer are rows of U and the weights of the decoder contain V.

- The reconstructed data contain the rows of UV^T.

Why is this SVD?

- If we use the mean-squared error as the loss function, we are optimizing $||D - UV^T||^2$ over the entire training data.
  - This is the same objective function as SVD!

- It is possible for gradient-descent to arrive at an optimal solution in which the columns of each of U and V might not be mutually orthogonal.

- Nevertheless, the subspace spanned by the columns of each of U and V will always be the same as that found by the optimal solution of SVD.

Some Interesting Facts

- The optimal encoder weight matrix W will be the pseudo-inverse of the decoder weight matrix V if the training data spans the full dimensionality.

$$W = (V^T V)^{-1} V^T$$

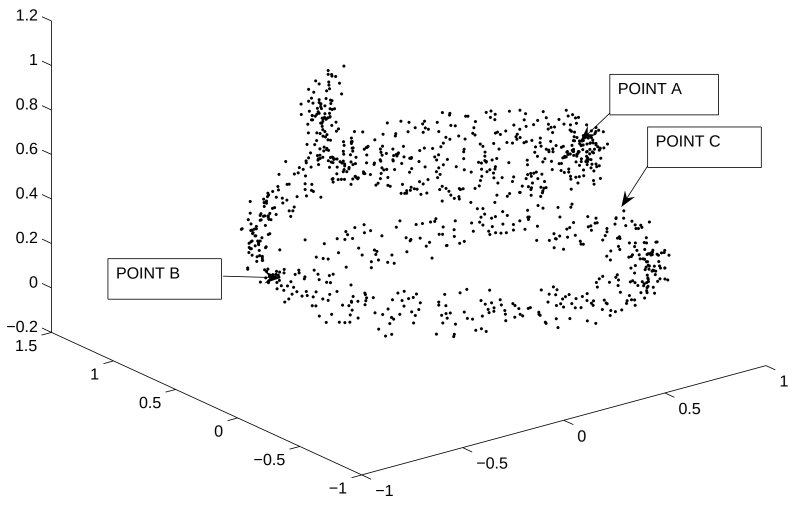
  - If the encoder and decoder weights are tied $W = V^T$, the columns of the weight matrix V will become mutually orthogonal.

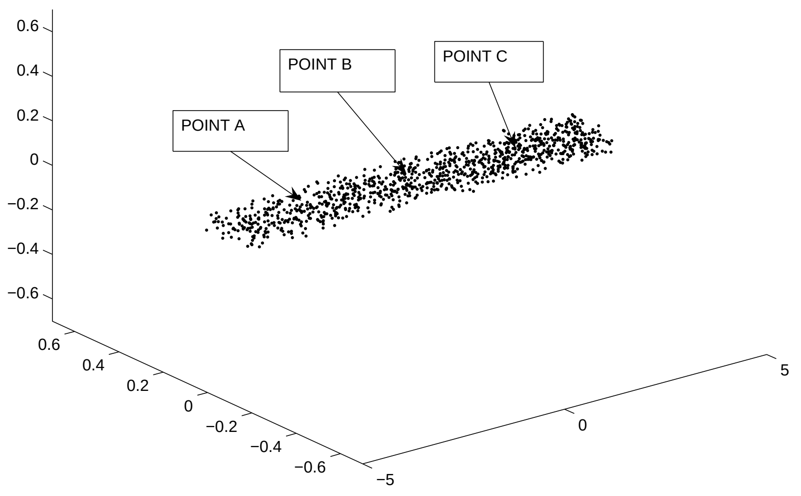
  - Easily shown by substituting $W = V^T$ above and postmultiplying with V to obtain $V^T V = I$.

  - This is exactly SVD!

- Tying encoder-decoder weights does not lead to orthogonality for other architectures, but is a common practice anyway.

Deep Autoencoders





- Better reductions are obtained by using increased depth and nonlinearity.

- Crucial to use nonlinear activations with deep autoencoders.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Row-Index to Row-Value Autoencoders: Incomplete Matrix Factorization for Recommender Systems

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.6

Recommender Systems

- Recap of SVD: Factorizes $D \approx UV^T$ so that the sum-of-squares of residuals $||D - UV^T||^2$ is minimized.
  - Helpful to watch previous lecture on SVD

- In recommender systems (RS), we have an $n \times d$ ratings matrix D with n users and d items.
  - Most of the entries in the matrix are unobserved
  - Want to minimize $||D - UV^T||^2$ only over the observed entries
  - Can reconstruct the entire ratings matrix using $UV^T \Rightarrow$ Most popular method in traditional machine learning.

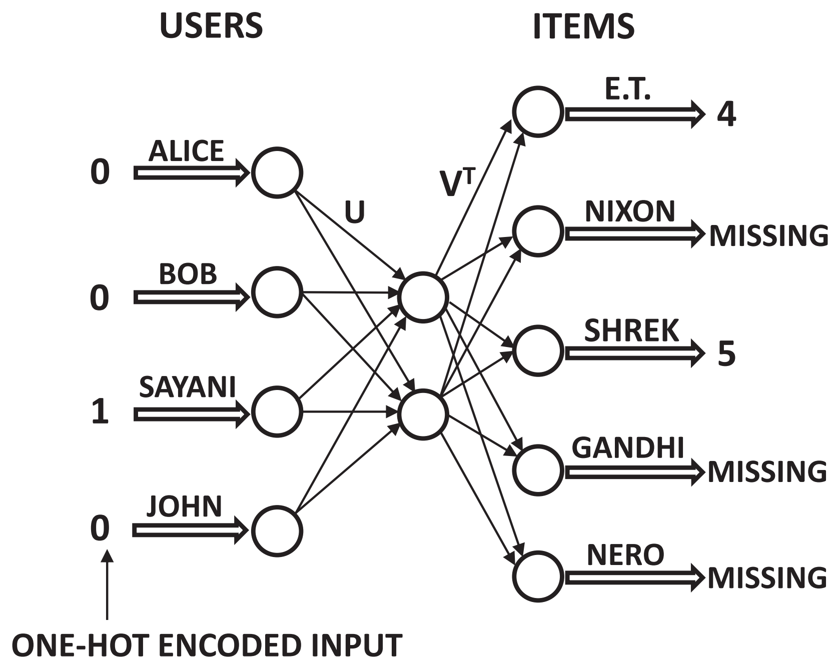
Difficulties with Autoencoder

- If some of the inputs are missing, then using an autoencoder architecture will implicitly assume default values for some inputs (like zero).
  - This is a solution used in some recent methods like *AutoRec*.
  - Does not exactly simulate classical MF used in recommender systems because it implicitly makes assumptions about unobserved entries.
- None of the proposed architectures for recommender systems in the deep learning literature exactly map to the classical factorization method of recommender systems.

Row-Index-to-Row-Value Autoencoder

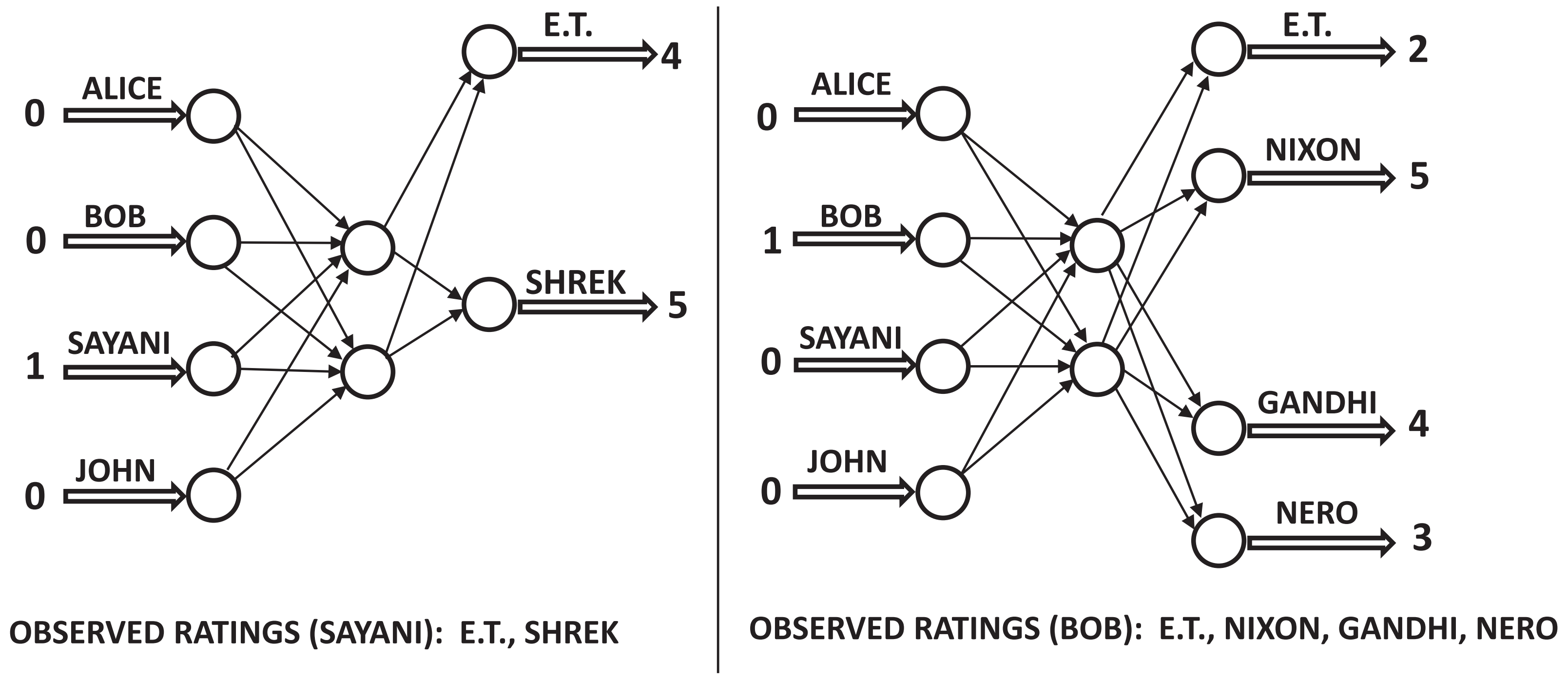
- Autoencoders map row values to row values.
  - Discuss an autoencoder architecture to map the one-hot encoded row *index* to the row values.
  - Not standard definition of autoencoder.
  - Can handle incomplete values but cannot handle out-of-sample data.
  - Also useful for representation learning (e.g., node representation of graph adjacency matrix).
- The row-index-to-row-value architecture is not recognized as a separate class of architectures for MF (but used often enough to deserve recognition as a class of MF methods).

Row-Index-to-Row-Value Autoencoder for RS



- Encoder and decoder weight matrices are U and V^T.
  - Input is one-hot encoded row index (only in-sample)
  - Number of nodes in hidden layer is factorization rank.
  - Outputs contain the ratings for that row index.

How to Handle Incompletely Specified Entries?



0
ALICE
0
BOB
1
SAYANI
0
JOHN
E.T.
4
SHREK
5
OBSERVED RATINGS (SAYANI): E.T., SHREK
0
ALICE
1
BOB
0
SAYANI
0
JOHN
E.T.
2
NIXON
5
GANDHI
4
NERO
3
OBSERVED RATINGS (BOB): E.T., NIXON, GANDHI, NERO

- Each user has his/her own neural architecture with missing outputs.

- Weights across different user architectures are shared.

Equivalence to Classical Matrix Factorization for RS

- Since the two weight matrices are U and V^T, the one-hot input encoding will pull out the relevant row from UV^T.

- Since the outputs only contain the observed values, we are optimizing the sum-of-square errors over observed values.

- Objective functions in the two cases are equivalent!

Training Equivalence

- For k hidden nodes, there are k paths between each user and each item identifier.

- Backpropagation updates weights along all k paths from each observed item rating to the user identifier.

  – Backpropagation in a later lecture.

- These k updates can be shown to be *identical* to classical matrix factorization updates with stochastic gradient descent.

- Backpropagation on neural architecture is identical to classical MF stochastic gradient descent.

Advantage of Neural View over Classical MF View

- The neural view provides natural ways to add power to the architecture with nonlinearity and depth.
  - Much like a child playing with a LEGO toy.
  - You are shielded from the ugly details of training by an inherent modularity in neural architectures.
  - The name of this magical modularity is backpropagation.
- If you have binary data, you can add logistic outputs for logistic matrix factorization.
- *Word2vec* belongs to this class of architectures (but *direct* relationship to nonlinear matrix factorization is not recognized).

Importance of Row-Index-to-Row-Value Autoencoders

- Several MF methods in machine learning can be expressed as row-index-to-row-value autoencoders (but not widely recognized–RS matrix factorization a notable example).

- Several row-index-to-row-value architectures in NN literature are also not fully recognized as matrix factorization methods.
  - The full relationship of *word2vec* to matrix factorization is often not recognized.
  - *Indirect* relationship to *linear* PPMI matrix factorization was shown by Levy and Goldberg.
  - In a later lecture, we show that *word2vec* is *directly* a form of *nonlinear* matrix factorization because of its row-index-to-row-value architecture and nonlinear activation.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Word2vec: The Skipgram Model

Neural Networks and Deep Learning, Springer, 2018
Chapter 2, Section 2.7

Word2Vec: An Overview

- *Word2vec* computes embeddings of words using sequential proximity in sentences.
  - If *Paris* is closely related to *France*, then *Paris* and *France* must occur together in small windows of sentences.
    - Their embeddings should also be somewhat similar.
  - Continuous bag-of-words predicts central word from context window.
  - Skipgram model predicts context window from central word.

Words and Context

- A window of size t on either side is predicted using a word.

- This model tries to predict the context $w_{i-t}w_{i-t+1}\dots w_{i-1}$ $w_{i+1}\dots w_{i+t-1}w_{i+t}$ around word w_i, given the ith word in the sentence, denoted by w_i.

- The total number of words in the context window is $m = 2t$.

- One can also create a $d \times d$ word-context matrix C with frequencies c_{ij}.

- We want to find an embedding of each word.

Where have We Seen this Setup Before?

- Similar to recommender systems with *implicit feedback*.

- Instead of user-item matrices, we have square word-context matrices.

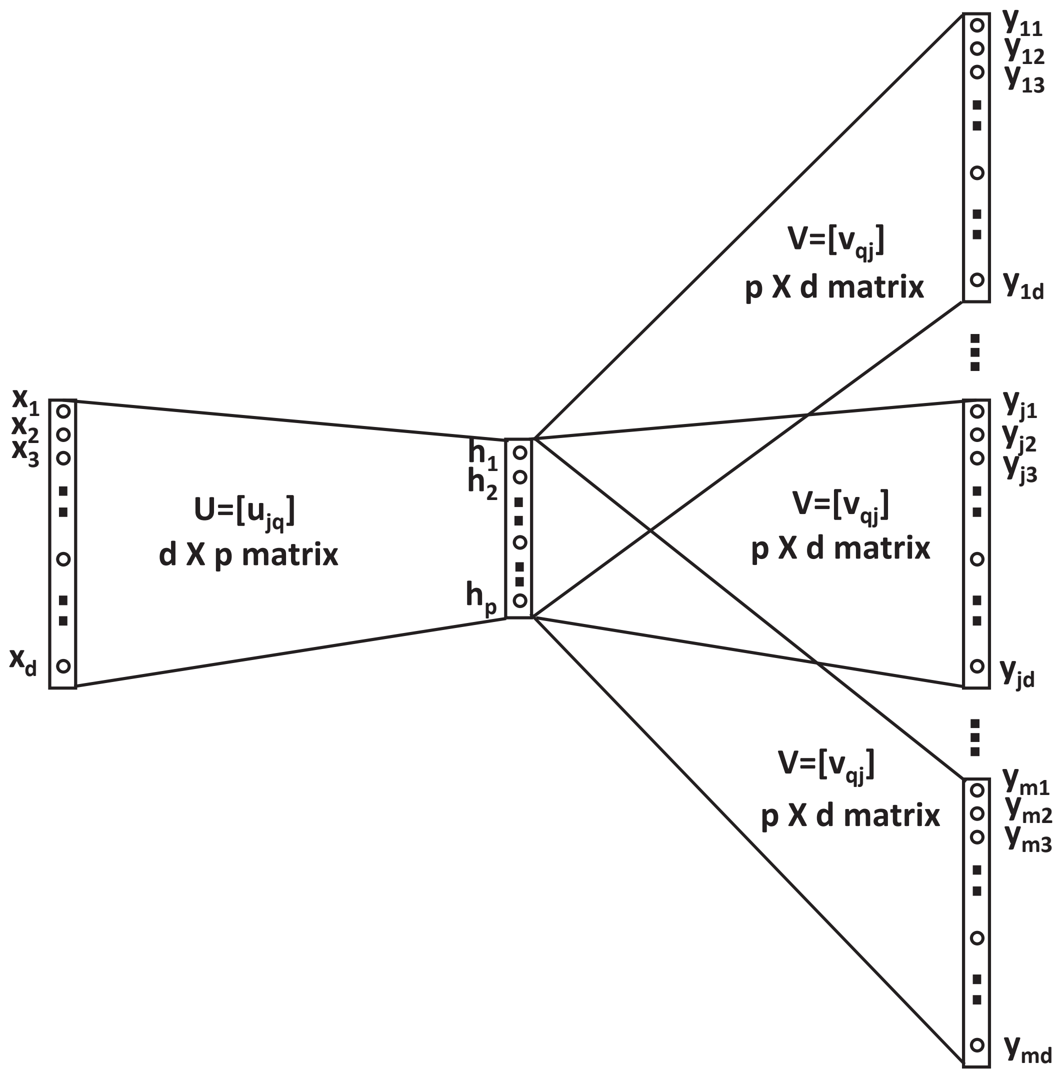
  - The frequencies correspond to the number of times a contextual word (column id) appears for a target word (row id).

  - Analogous to the number of units bought by a user (row id) of an item (column id).

  - An unrecognized fact is that skipgram *word2vec* uses an almost identical model to current recommender systems.

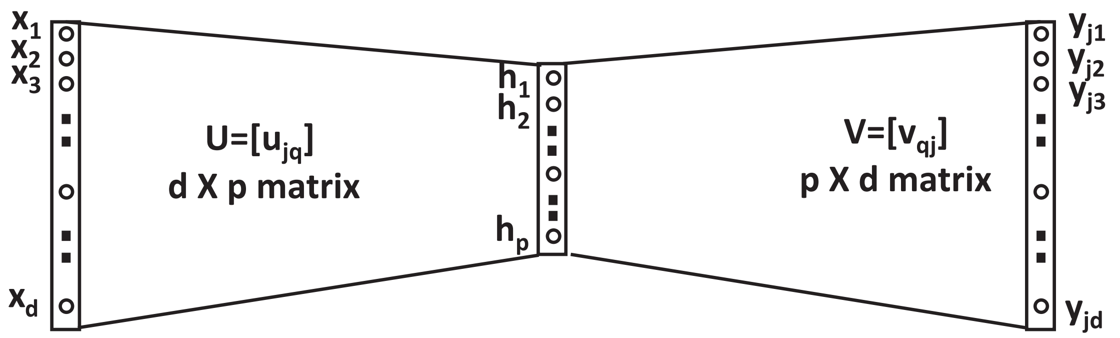
- Helpful to watch previous lecture on recommender systems with row-index-to-value autoencoders.

Word2Vec: Skipgram Model



- Input is the one-hot encoded word identifier and output contains m *identical* softmax probability sets.

Word2Vec: Skipgram Model



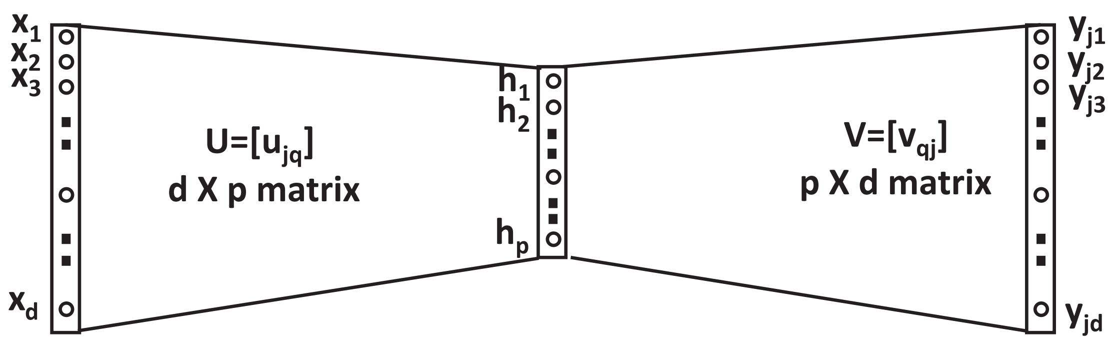
MINIBATCH THE m d-DIMENSIONAL OUTPUT VECTORS IN EACH CONTEXT WINDOW DURING STOCHASTIC GRADIENT DESCENT. THE SHOWN OUTPUTS y_{jk} CORRESPOND TO THE jth OF m OUTPUTS.

- Since the m outputs are identical, we can collapse the m outputs into a single output.

- Mini-batch the words in a context window to achieve the same effect.

- Gradient descent steps for each instance are proportional to $d \Rightarrow$ Expensive.

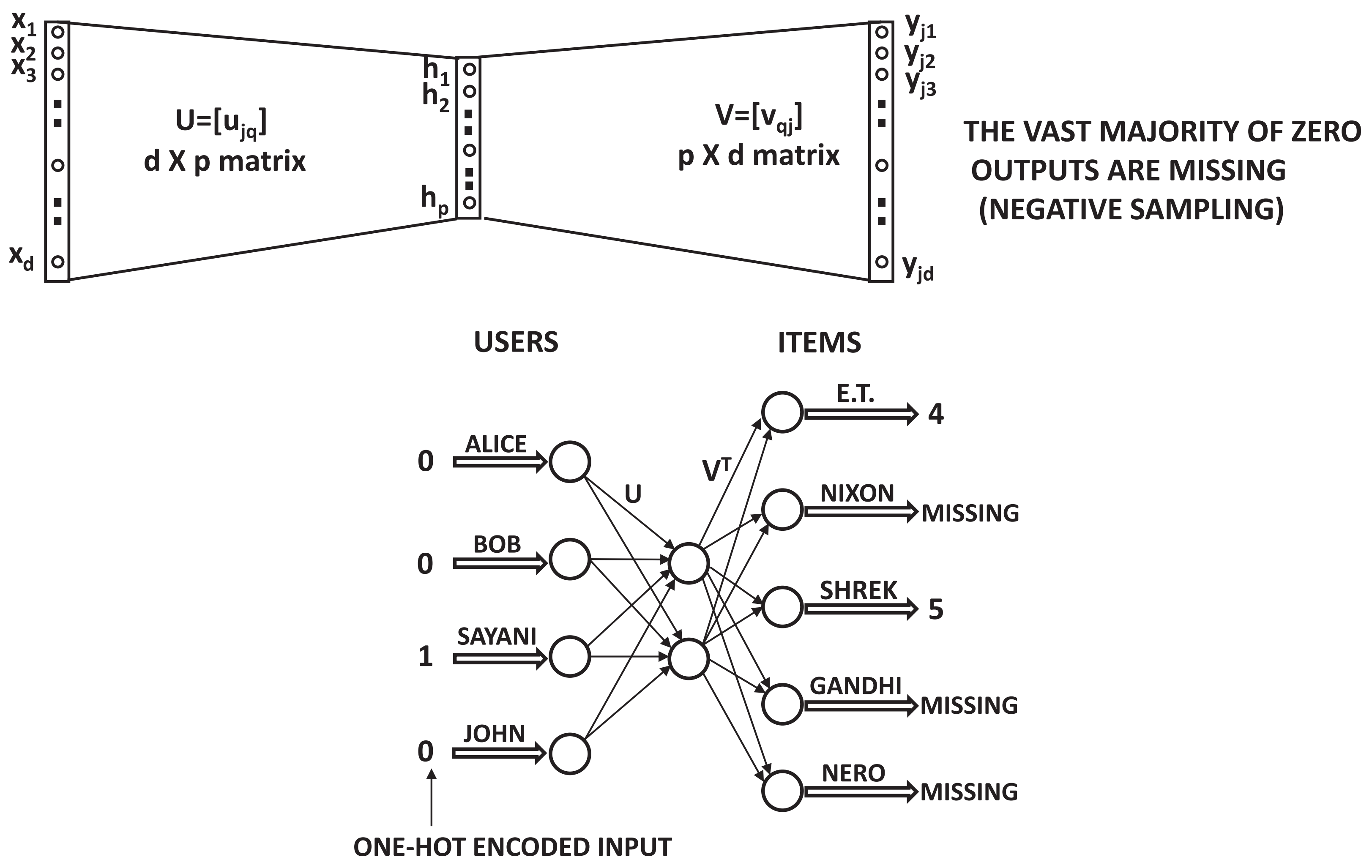
Word2Vec: Skipgram Model with Negative Sampling



MINIBATCH THE m d-DIMENSIONAL OUTPUT VECTORS IN EACH CONTEXT WINDOW DURING STOCHASTIC GRADIENT DESCENT. THE SHOWN OUTPUTS y_{jk} CORRESPOND TO THE jth OF m OUTPUTS.

- Change the softmax layer into sigmoid layer.
- Of the d outputs, keep the positive output and sample k out of the remaining $d - 1$ (with log loss).
- Where have we seen missing outputs before?

Can You See the Similarity?



- Main difference: Sigmoid output layer with log loss.

Word2Vec is Nonlinear Matrix Factorization

- Levy and Goldberg showed an *indirect* relationship between *word2vec* SGNS and PPMI matrix factorization.

- We provide a much more direct result in the book.
  - Word2vec is (weighted) logistic matrix factorization.
  - Not surprising because of the similarity with the recommender architecture.
  - Logistic matrix factorization is already used in recommender systems!
  - Neither the *word2vec* authors nor the community have pointed out this *direct* connection.

Other Extensions

- We can apply a row-index-to-value autoencoder to any type of matrix to learn embeddings of either rows or columns.

- Applying to graph adjacency matrix leads to node embeddings.

  – Idea has been used by *DeepWalk* and *node2vec* after (indirectly) enhancing the matrix entries with random-walk methods.

  – Details of graph embedding methods in book.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation I: Computing Derivatives in Computational Graphs [without Backpropagation] in Exponential Time

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.2

Why Do We Need Backpropagation?

- To perform any kind of learning, we need to compute the partial derivative of the loss function with respect to each intermediate weight.
  - Simple with single-layer architectures like the perceptron.
  - Not a simple matter with multi-layer architectures.

The Complexity of Computational Graphs

- A computational graph is a directed acyclic graph in which each node computes a function of its incoming node variables.

- A neural network is a special case of a computational graph.
  - Each node computes a combination of a linear vector multiplication and a (possibly nonlinear) activation function.

- The output is a very complicated *composition* function of each intermediate weight in the network.
  - The complex composition function might be hard to express neatly in closed form.
    - Difficult to differentiate!

Recursive Nesting is Ugly!

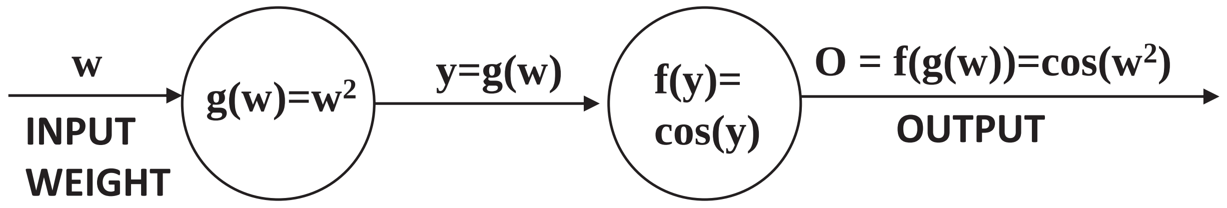
- Consider a computational graph containing two nodes in a path and input w.

- The first node computes $y = g(w)$ and the second node computes the output $o = f(y)$.
  - Overall composition function is $f(g(w))$.
  - Setting $f()$ and $g()$ to the sigmoid function results in the following:

$$f(g(w)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-w)}\right]} \tag{1}$$

  - Increasing path length increases recursive nesting.

Backpropagation along Single Path (Univariate Chain Rule)



- Consider a two-node path with $f(g(w)) = \cos(w^2)$

- In the univariate chain rule, we compute product of *local* derivatives.

$$\frac{\partial f(g(w))}{\partial w} = \underbrace{\frac{\partial f(y)}{\partial y}}_{-sin(y)} \cdot \underbrace{\frac{\partial g(w)}{\partial w}}_{2w} = -2w \cdot \sin(y) = -2w \cdot \sin(w^2)$$

- Local derivatives are easy to compute because they care about their own input and output.

Backpropagation along Multiple Paths (Multivariate Chain Rule)

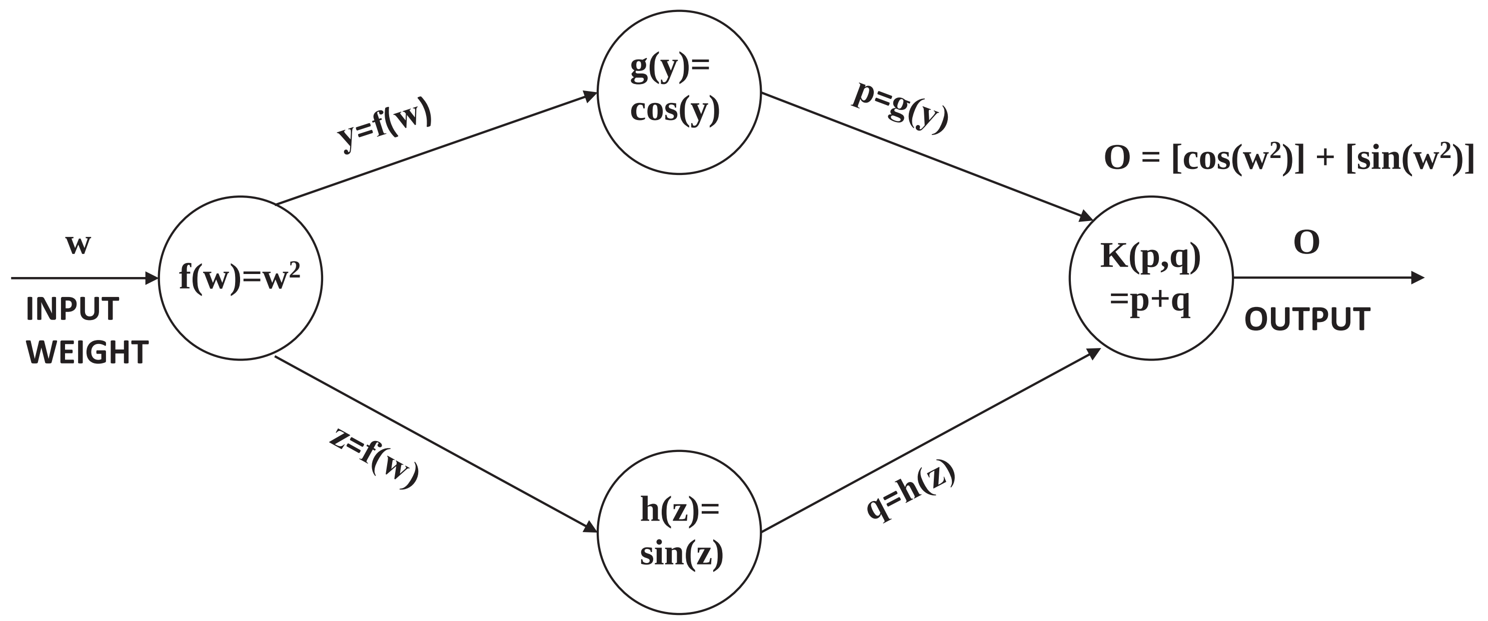
- Neural networks contain multiple nodes in each layer.

- Consider the function $f(g_1(w), \ldots g_k(w))$, in which a unit computing the *multivariate* function $f(\cdot)$ gets its inputs from k units computing $g_1(w) \ldots g_k(w)$.

- The *multivariable chain rule* needs to be used:

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w} \qquad (2)$$

Example of Multivariable Chain Rule



$$\frac{\partial o}{\partial w} = \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{\text{-sin}(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w}$$

$$= -2w \cdot \sin(y) + 2w \cdot \cos(z)$$

$$= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)$$

- Product of local derivatives along *all* paths from w to o.

Pathwise Aggregation Lemma

- Let a non-null set $\mathcal{P}$ of paths exist from a variable w in the computational graph to output o.
  - Local gradient of node with variable $y(j)$ with respect to variable $y(i)$ for directed edge (i, j) is $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$
- The value of $\frac{\partial o}{\partial w}$ is given by computing the product of the local gradients along each path in $\mathcal{P}$, and summing these products over all paths.

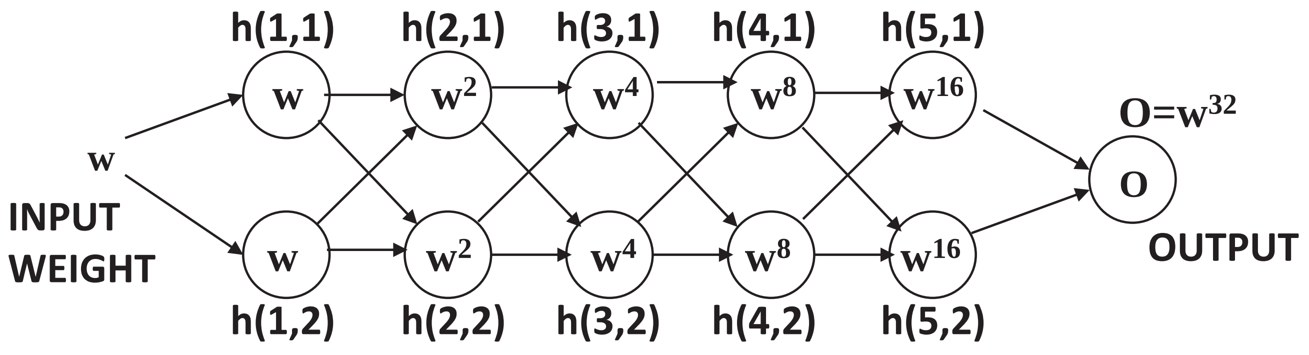
$$\frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j) \tag{3}$$

- Observation: Each $z(i, j)$ easy to compute.

An Exponential Time Algorithm for Computing Partial Derivatives

- The path aggregation lemma provides a simple way to compute the derivative with respect to intermediate variable w
  - Use computational graph to compute each value $y(i)$ of nodes i in a forward phase.
  - Compute local derivative $z(i,j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge (i,j) in the network.
  - Identify the set $\mathcal{P}$ of all paths from the node with variable w to the output o.
  - For each path $P \in \mathcal{P}$ compute the product $M(P) = \prod_{(i,j)\in P} z(i,j)$ of the local derivatives on that path.
  - Add up these values over all paths $P \in \mathcal{P}$.

Example: Deep Computational Graph with Product Nodes



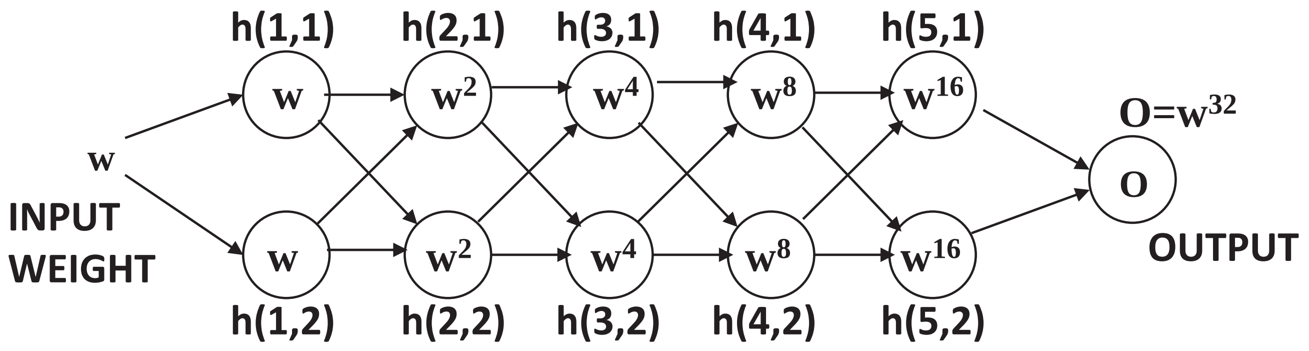
EACH NODE COMPUTES THE PRODUCT OF ITS INPUTS

- Each node computes product of its inputs $\Rightarrow$ Partial derivative of xy with respect to one input x is the other input y.

- Computing product of partial derivatives along a path is equivalent to computing product of values along the only other node disjoint path.

- Aggregative product of partial derivatives (only in this case) equals aggregating products of values.

Example of Increasing Complexity with Depth



EACH NODE COMPUTES THE PRODUCT OF ITS INPUTS

$$\frac{\partial O}{\partial w} = \sum_{j_1, j_2, j_3, j_4, j_5 \in \{1,2\}^5} \underbrace{\prod h(1, j_1)}_{w} \underbrace{h(2, j_2)}_{w^2} \underbrace{h(3, j_3)}_{w^4} \underbrace{h(4, j_4)}_{w^8} \underbrace{h(5, j_5)}_{w^{16}}$$

$$= \sum_{\text{All 32 paths}} w^{31} = 32 w^{31}$$

- Impractical with increasing depth.

Observations on Exponential Time Algorithm

- Not very practical approach $\Rightarrow$ Million paths for a network with 100 nodes in each layer and three layers.

- *This is the approach of traditional machine learning with complex objective functions in closed form*.
  - For a composition function in closed form, manual differentiation explicitly traverses all paths with chain rule.
  - The algebraic expression of the derivative of a complex function might not fit the paper you write on.
  - Explains why most of traditional machine learning is a shallow neural model.

- The beautiful *dynamic programming* idea of backpropagation rescues us from complexity.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation II: Using Dynamic Programming [Backpropagation] to Compute Derivatives in Polynomial Time

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.2

Differentiating Composition Functions

- Neural networks compute composition functions with a lot of *repetitiveness* caused by a node appearing in multiple paths.

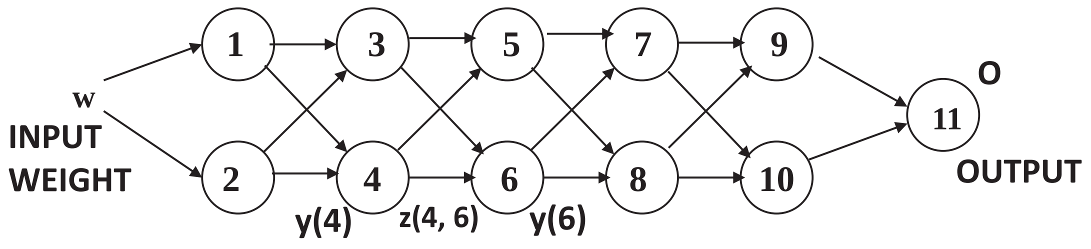
- The most natural and intuitive way to differentiate such a composition function is not the most *efficient* way to do it.

- Natural approach: Top down

$$f(w) = \sin(w^2) + \cos(w^2)$$

- We should not have to differentiate w^2 twice!

- Dynamic programming collapses repetitive computations to reduce exponential complexity into polynomial complexity!



EACH NODE i CONTAINS y(i) AND EACH EDGE BETWEEN i AND j CONTAINS z(i, j)
EXAMPLE: z(4, 6)= PARTIAL DERIVATIVE OF y(6) WITH RESPECT TO y(4)

- We want to compute the derivative of the output with respect to variable w.

- We can easily compute $z(i,j) = \frac{\partial y(j)}{\partial y(i)}$.

- Naive approach computes $S(w,o) = \frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i,j)$ by explicit aggregation over all paths in $\mathcal{P}$.

Dynamic Programming and Directed Acyclic Graphs

- Dynamic programming used extensively in directed acyclic graphs.
  - **Typical:** Exponentially aggregative path-centric functions between source-sink pairs.
  - **Example:** Polynomial solution to longest path problem in directed acyclic graphs (NP-hard in general).
  - **General approach:** Starts at either the source or sink and *recursively* computes the relevant function over paths of increasing length by reusing intermediate computations.

- Our path-centric function: $S(w, o) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j)$.
  - Backwards direction makes more sense here because we have to compute derivative of output (sink) with respect to all variables in early layers.

Dynamic Programming Update

- Let $A(i)$ be the set of nodes at the ends of outgoing edges from node i.

- Let $S(i, o)$ be the *intermediate* variable indicating the same path aggregative function from i to o.

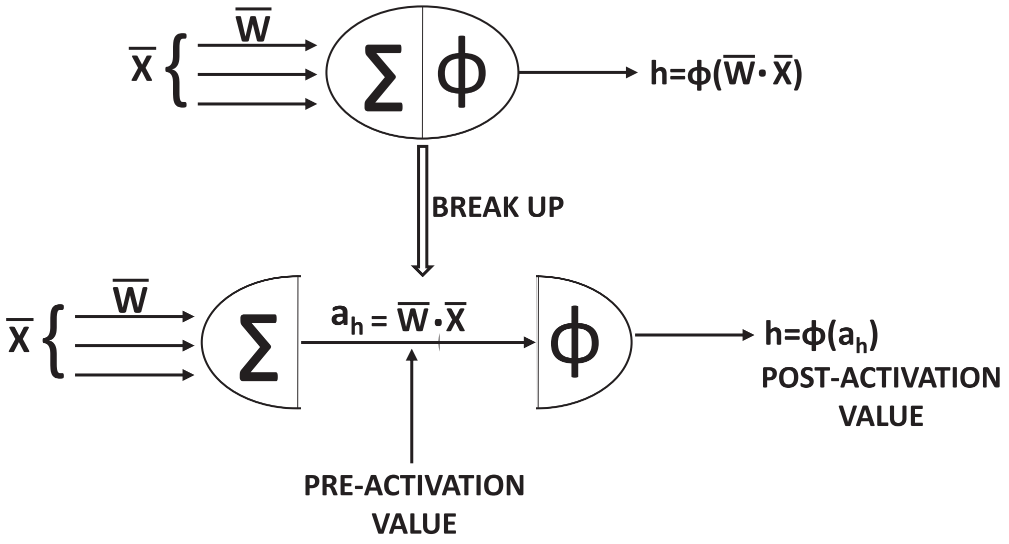
$$S(i, o) \Leftarrow \sum_{j \in A(i)} S(j, o) \cdot z(i, j) \tag{4}$$

- Initialize $S(o, o)$ to 1 and compute backwards to reach $S(w, o)$.
  - Intermediate computations like $S(i, o)$ are also useful for computing derivatives in other layers.

- Do you recognize the multivariate chain rule in Equation 4?

$$\frac{\partial o}{\partial y(i)} = \sum_{j \in A(i)} \frac{\partial o}{\partial y(j)} \cdot \frac{\partial y(j)}{\partial y(i)}$$

How Does it Apply to Neural Networks?



- A neural network is a special case of a computational graph.
  - We can define the computational graph in multiple ways.
  - Pre-activation variables or post-activation variables or both as the node variables of the computation graph?
  - The three lead to different updates but the end result is equivalent.

Pre-Activation Variables to Create Computational Graph

- Compute derivative $\delta(i, o)$ of loss L at o with respect to pre-activation variable at node i.

- We always compute loss derivatives $\delta(i, o)$ with respect to activations in *nodes* during dynamic programming rather than *weights*.

  – Loss derivative with respect to weight w_{ij} from node i to node j is given by the product of $\delta(j, o)$ and hidden variable at i (why?)

- Key points: $z(i, j) = w_{ij} \cdot \Phi'_i$, Initialize $S(o, o) = \delta(o, o) = \frac{\partial L}{\partial o} \Phi'_o$

$$\delta(i, o) = S(i, o) = \Phi'_i \sum_{j \in A(i)} w_{ij} S(j, o) = \Phi'_i \sum_{j \in A(i)} w_{ij} \delta(j, o) \tag{5}$$

Post-Activation Variables to Create Computation Graph

- The variables in the computation graph are hidden values *after* activation function application.

- Compute derivative $\Delta(i, o)$ of loss L at o with respect to post-activation variable at node i.

- Key points: $z(i, j) = w_{ij} \cdot \Phi'_j$, Initialize $S(o, o) = \Delta(o, o) = \frac{\partial L}{\partial o}$

$$\Delta(i, o) = S(i, o) = \sum_{j \in A(i)} w_{ij} S(j, o) \Phi'_j = \sum_{j \in A(i)} w_{ij} \Delta(j, o) \Phi'_j \tag{6}$$

  – Compare with pre-activation approach $\delta(i, o) = \Phi'_i \sum_{j \in A(i)} w_{ij} \delta(j, o)$

  – Pre-activation approach more common in textbooks.

Variables for Both Pre-Activation and Post-Activation Values

- Nice way of decoupling the linear multiplication and activation operations.

- Simplified approach in which each layer is treated as a single node with a vector variable.

  – Update can be computed in vector and matrix multiplications.

- Topic of discussion in next part of the backpropagation series.

Losses at Arbitrary Nodes

- We assume that the loss is incurred at a single output node.
- In case of multiple output nodes, one only has to add up the contributions of different outputs in the backwards phase.
- In some cases, penalties may be applied to hidden nodes.
- For a hidden node i, we add an "initialization value" to $S(i, o)$ just after it has been computed during dynamic programming, which is based on its penalty.
  - Similar treatment as the initialization of an output node, except that we *add* the contribution to existing value of $S(i, o)$.

Handling Shared Weights

- You saw an example in autoencoders where encoder and decoder weights are shared.

- Also happens in specialized architectures like recurrent or convolutional neural networks.

- Can be addressed with a simple application of the chain rule.

- Let $w_1 \ldots w_r$ be r copies of the same weight w in the neural network.

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{r} \frac{\partial L}{\partial w_i} \cdot \frac{\partial w_i}{\partial w} = \sum_{i=1}^{r} \frac{\partial L}{\partial w_i} \tag{7}$$

- Pretend all weights are different and just add!

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation III: A Decoupled View of Vector-Centric Backpropagation

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.2

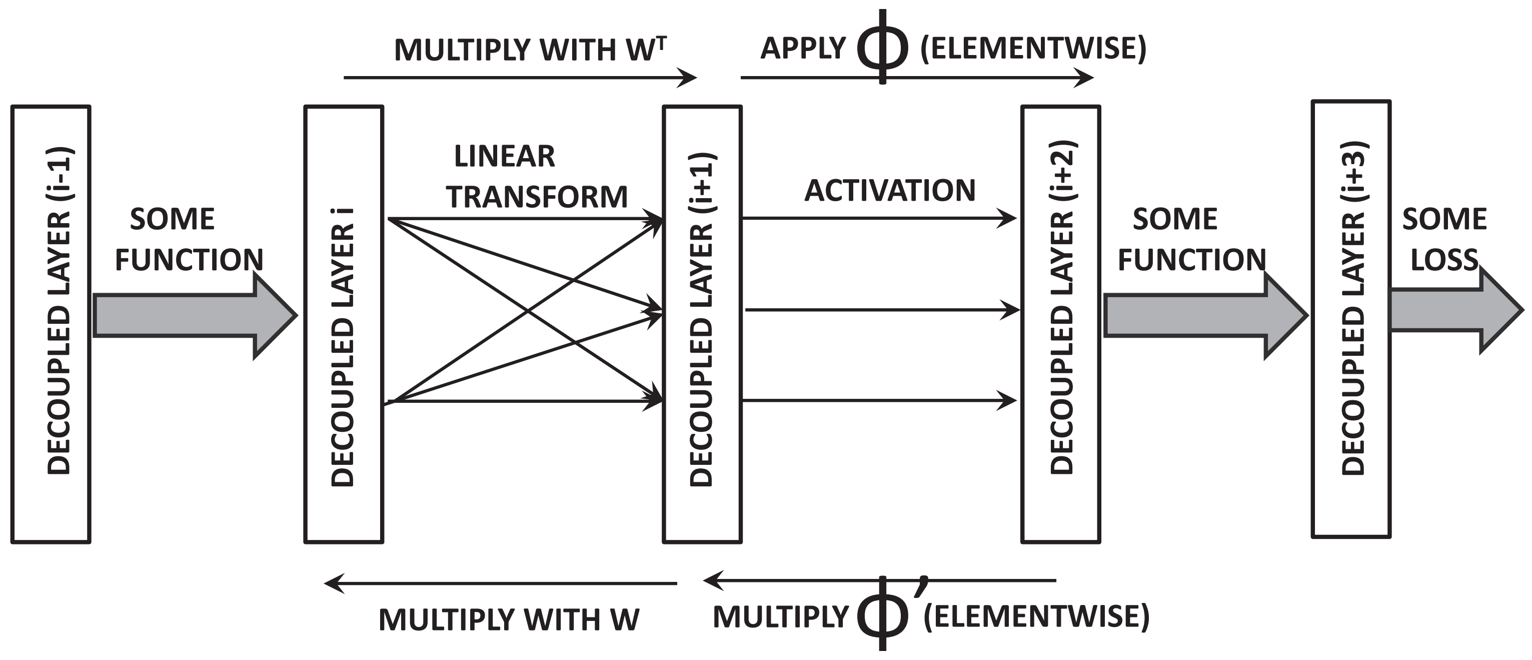
Multiple Computational Graphs from Same Neural Network

- We can create a computational graph in multiple ways from the variables in a neural network.
  - Computational graph of pre-activation variables (part II of lecture)
  - Computational graph of post-activation variables (part II of lecture)
  - Computational graph of both (this part of the lecture)
- Using both pre-activation and post-activation variables creates decoupled backpropagation updates for linear layer and for activation function.

Scalar Versus Vector Computational Graphs

- The backpropagation discussion so far uses scalar operations.
- Neural networks are constructed in layer-wise fashion.
- We can treat an entire layer as a node with a vector variable.
- We want to use layer-wise operations on vectors.
  - Most real implementations use vector and matrix multiplications.
- Want to decouple the operations of linear matrix multiplication and activation function in separate "layers."

Vector-Centric and Decoupled View of Single Layer



- Note that linear matrix multiplication and activation function are separate layers.

- Method 1 (requires knowledge of matrix calculus): You can use the vector-to-vector chain rule to backpropagate on a single path!

Converting Scalar Updates to Vector Form

- **Recap:** When the partial derivative of node q with respect to node p is $z(p, q)$, the dynamic programming update is:

$$S(p, o) = \sum_{q \in \text{Next Layer}} S(q, o) \cdot z(p, q) \tag{8}$$

- We can write the above update in vector form by creating a single column vector $\overline{g}_i$ for layer i $\Rightarrow$ Contains $S(p, o)$ for all values of p.

$$\overline{g}_i = Z\overline{g}_{i+1} \tag{9}$$

- The matrix $Z = [z(p, q)]$ is the transpose of the Jacobian!
  - We will use the notation $J = Z^T$ in further slides.

The Jacobian

- Consider layer i and layer-$(i+1)$ with activations $\overline{z}_i$ and $\overline{z}_{i+1}$.
  - The kth activation in layer-$(i+1)$ is obtained by applying an arbitrary function $f_k(\cdot)$ on the vector of activations in layer-i.

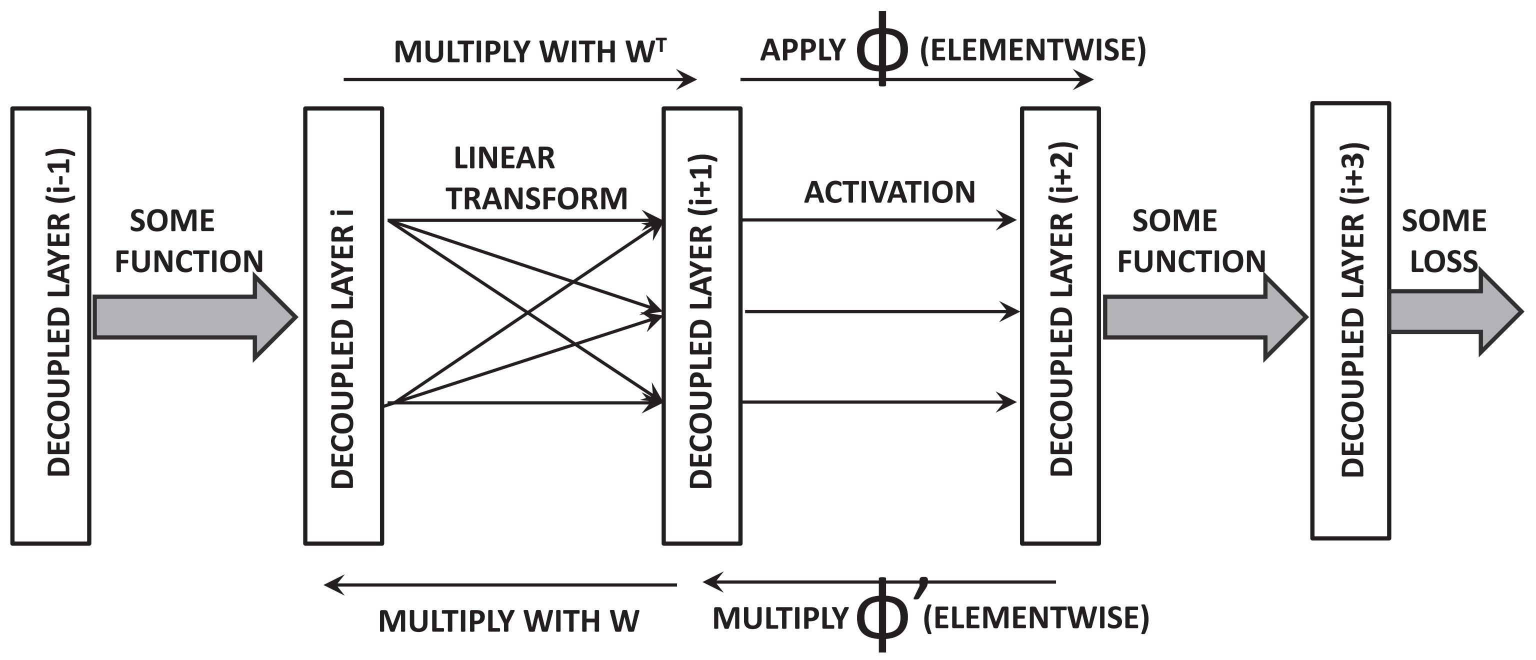
- Definition of Jacobian matrix entries:

$$J_{kr} = \frac{\partial f_k(\overline{z}_i)}{\partial \overline{z}_i^{(r)}} \tag{10}$$

- Backpropagation updates:

$$\overline{g}_i = J^T \overline{g}_{i+1} \tag{11}$$

Effect on Linear Layer and Activation Functions



- Backpropagation is multiplication with transposed weight matrix for linear layer.

- Elementwise multiplication with derivative for activation layer.

Table of Forward Propagation and Backward Propagation

Function	Forward	Backward
Linear	$\overline{z}_{i+1} = W^T \overline{z}_i$	$\overline{g}_i = W \overline{g}_{i+1}$
Sigmoid	$\overline{z}_{i+1} =$sigmoid$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (1 - \overline{z}_{i+1})$
Tanh	$\overline{z}_{i+1} =$tanh$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (1 - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	Set to ± 1 ($\not\in [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\not\in [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T \overline{g}_{i+1}$ J is Jacobian (Equation 10)

- Two types of Jacobians: Linear layers are dense and activation layers are sparse.

- Maximization function used in max-pooling.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Neural Network Training [Initialization, Preprocessing, Mini-Batching, Tuning, and Other Black Art]

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.3

How to Check Correctness of Backpropagation

- Consider a particular weight w of a randomly selected edge in the network.

- Let $L(w)$ be the current value of the loss.

- The weight of this edge is perturbed by adding a small amount $\epsilon > 0$ to it.

- Estimate of derivative:
$$\frac{\partial L(w)}{\partial w} \approx \frac{L(w+\epsilon) - L(w)}{\epsilon} \tag{12}$$

- When the partial derivatives do not match closely enough, it might be indicative of an incorrectness in implementation.

What Does “Closely Enough” Mean?

- Algorithm-determined derivative is G_e and the approximate derivative is G_a.

$$\rho = \frac{|G_e - G_a|}{|G_e + G_a|} \tag{13}$$

- The ratio should be less than 10^{-6}.

- If ReLU is used, the ratio should be less than 10^{-3}.

- Should perform the checks for a sample of the weights a few times during training.

Stochastic Gradient Descent

- We have always worked with *point-wise* loss functions so far.
  - Corresponds to stochastic gradient descent.
  - In practice, stochastic gradient descent is only a randomized approximation of the true loss function.
- True loss function is typically additive over points.
  - **Example:** Sum-of-squared errors in regression.
  - Computing gradient over a single point is like sampled gradient estimate.

Mini-batch Stochastic Gradient Descent

- One can improve accuracy of gradient computation by using a batch of instances.
  - Instead of holding a vector of activations, we hold a matrix of activations in each layer.
  - Matrix-to-matrix multiplications required for forward and backward propagation.
  - Increases the memory requirements.
- Typical sizes are powers of 2 like 32, 64, 128, 256

Why Does Mini-Batching Work?

- At early learning stages, the weight vectors are very poor.
  - Training data is highly redundant in terms of important patterns.
  - Small batch sizes gives the correct direction of gradient.
- At later learning stages, the gradient direction becomes less accurate.
  - But some amount of noise helps avoid overfitting anyway!
- Performance on out-of-sample data does not deteriorate!

Feature Normalization

- **Standardization:** Normalize to zero mean and unit variance.

- **Whitening:** Transform the data to a de-correlated axis system with principal component analysis (mean-centered SVD).
  - Truncate directions with extremely low variance.
  - Standardize the other directions.

- **Basic principle:** Assume that data is generated from Gaussian distribution and give equal importance to all directions.

Weight Initialization

- Initializations are surprisingly important.
  - Poor initializations can lead to bad convergence behavior.
  - Instability across different layers (vanishing and exploding gradients).

- More sophisticated initializations such as pretraining covered in later lecture.

- Even some simple rules in initialization can help in conditioning.

Symmetry Breaking

- Bad idea to initialize weights to the same value.
  - Results in weights being updated in lockstep.
  - Creates redundant features.
- Initializing weights to random values breaks symmetry.
- Average magnitude of the random variables is important for stability.

Sensitivity to Number of Inputs

- More inputs increase output sensitivity to the average weight.
  - Additive effect of multiple inputs: variance linearly increases with number of inputs r.
  - Standard deviation scales with the square-root of number of inputs r.

- Each weight is initialized from Gaussian distribution with standard deviation $\sqrt{1/r}$ ($\sqrt{2/r}$ for ReLU).

- **More sophisticated:** Use standard deviation of $\sqrt{2/(r_{in} + r_{out})}$.

Tuning Hyperparameters

- Hyperparameters represent the parameters like number of layers, nodes per layer, learning rate, and regularization parameter.

- Use separate validation set for tuning.

- Do not use same data set for backpropagation training as tuning.

Grid Search

- Perform grid search over parameter space.
  - Select set of values for each parameter in some “reasonable” range.
  - Test over all combination of values.
- Careful about parameters at borders of selected range.
- **Optimization:** Search over coarse grid first, and then drill down into region of interest with finer grids.

How to Select Values for Each Parameter

- Natural approach is to select uniformly distributed values of parameters.
  - Not the best approach in many cases! $\Rightarrow$ Log-uniform intervals.
  - Search uniformly in reasonable values of log-values and then exponentiate.
  - **Example:** Uniformly sample log-learning rate between -3 and -1, and then raise it to the power of 10.

Sampling versus Grid Search

- With a large number of parameters, grid search is still expensive.

- With 10 parameters, choosing just 3 values for each parameter leads to $3^{10} = 59049$ possibilities.

- Flexible choice is to sample over grid space.

- Used more commonly in large-scale settings with good results.

Large-Scale Settings

- Multiple threads are often run with sampled parameter settings.

- Accuracy tracked on a separate out-of-sample validation set.

- Bad runs are detected and killed after a certain number of epochs.

- New runs may also be started after killing threads (if needed).

- Only a few winners are trained to completion and the predictions combined in an ensemble.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Gradient Ratios, Vanishing and Exploding Gradient Problems

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.4

Effect of Varying Slopes in Gradient Descent

- Neural network learning is a *multivariable* optimization problem.

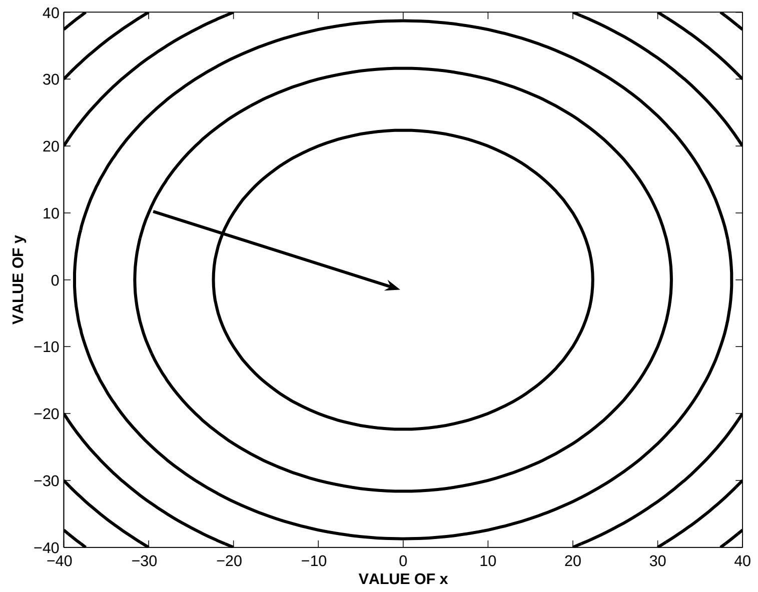
- Different weights have different magnitudes of partial derivatives.

- Widely varying magnitudes of partial derivatives affect the learning.

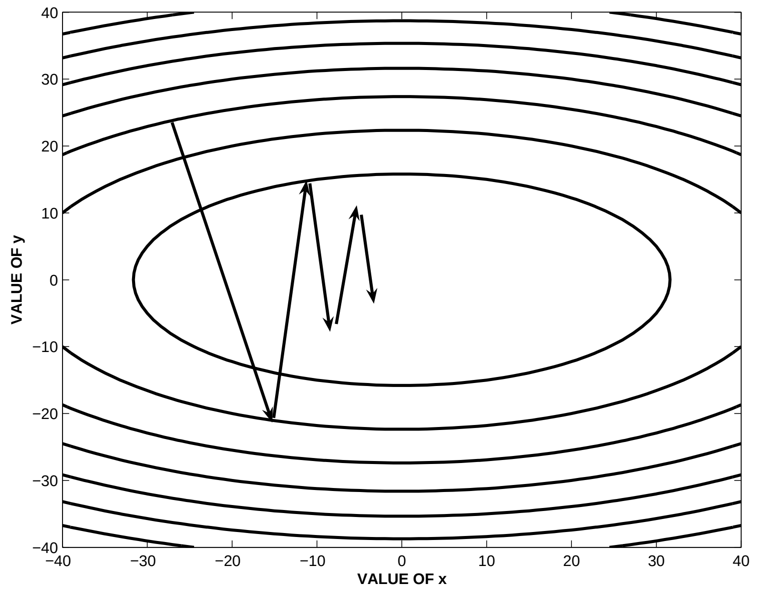
- Gradient descent works best when the different weights have derivatives of similar magnitude.

  – *The path of steepest descent in most loss functions is only an instantaneous direction of best movement, and is not the correct direction of descent in the longer term.*

Example



(a) Loss function is circular bowl $L = x^2 + y^2$



(b) Loss function is elliptical bowl $L = x^2 + 4y^2$

- Loss functions with varying sensitivity to different attributes

Revisiting Feature Normalization

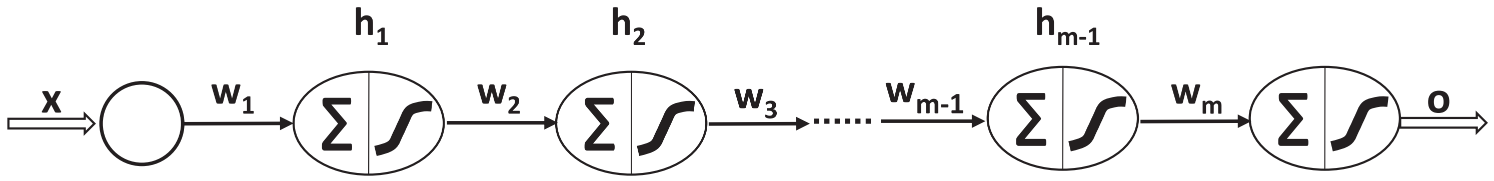
- In the previous lecture, we discussed feature normalization.
- When features have very different magnitudes, gradient ratios of different weights are likely very different.
- Feature normalization helps even out gradient ratios to some extent.
  - Exact behavior depends on target variable and loss function.

The Vanishing and Exploding Gradient Problems

- An extreme manifestation of varying sensitivity occurs in deep networks.

- The weights/activation derivatives in different layers affect the backpropagated gradient in a multiplicative way.
  - With increasing depth this effect is magnified.
  - The partial derivatives can either increase or decrease with depth.

Example



- Neural network with one node per layer.
- Forward propagation multiplicatively depends on each weight and activation function evaluation.
- Backpropagated partial derivative get multiplied by weights and activation function derivatives.
- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish).
- Hard to initialize weights exactly right.

Activation Function Propensity to Vanishing Gradients

- Partial derivative of sigmoid with output $o \Rightarrow o(1-o)$.
  - Maximum value at $o = 0.5$ of 0.25.
  - For 10 layers, the activation function alone will multiply by less than $0.25^{10} \approx 10^{-6}$.

- At extremes of output values, the partial derivative is close to 0, which is called *saturation*.

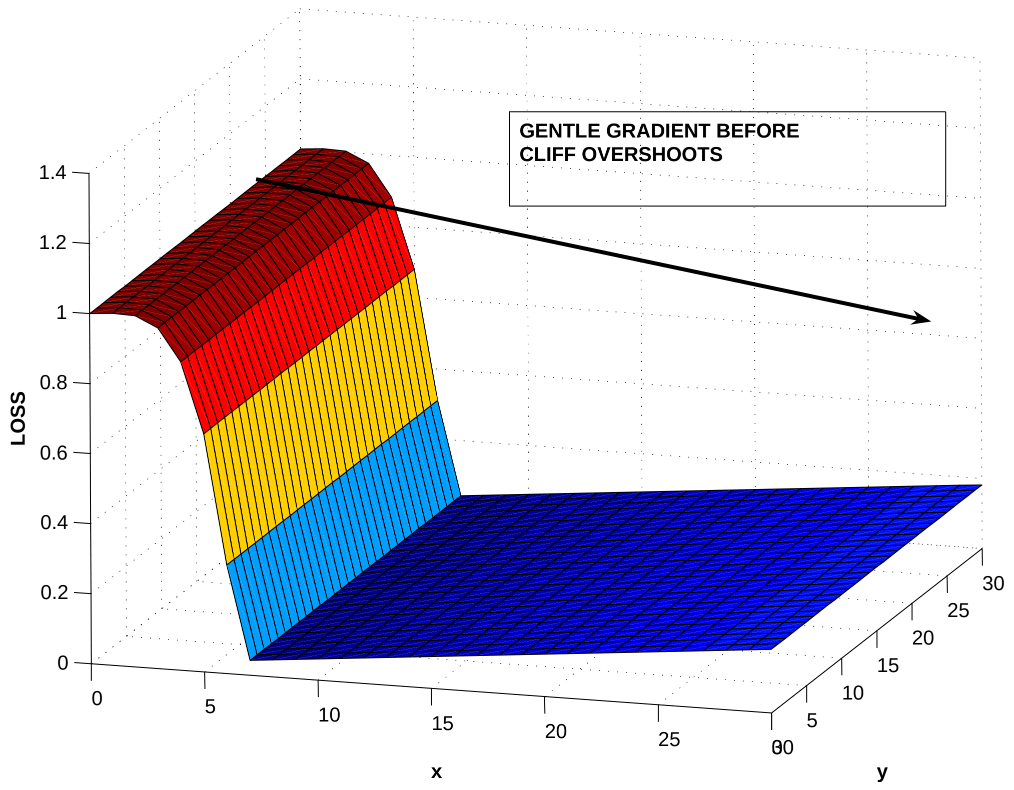
- The tanh activation function with partial derivative $(1-o^2)$ has a maximum value of 1 at $o = 0$, but saturation will still cause problems.

Exploding Gradients

- Initializing weights to very large values to compensate for the activation functions can cause exploding gradients.

- Exploding gradients can also occur when weights across different layers are shared (e.g., recurrent neural networks).
  - The effect of a finite change in weight is extremely unpredictable across different layers.
  - Small *finite* change changes loss negligibly, but a slightly larger value might change loss drastically.

Cliffs



- Often occurs with the exploding gradient problem.

A Partial Fix to Vanishing Gradients

- The ReLU has linear activation for nonnegative values and otherwise sets outputs to 0.
- The ReLU has a partial derivative of 1 for nonnegative inputs.
- However, it can have a partial derivative of 0 in some cases and never get updated.
  - Neuron is permanently dead!

Leaky ReLU

- For negative inputs, the leaky ReLU can still propagate some gradient backwards.
  - At the reduced rate of $\alpha < 1$ times the learning case for nonnegative inputs:

$$\Phi(v) = \begin{cases} \alpha \cdot v & v \leq 0 \\ v & \text{otherwise} \end{cases} \tag{14}$$

- The value of α is a hyperparameter chosen by the user.

- The gains with the leaky ReLU are not guaranteed.

Maxout

- The activation used is $\max\{\overline{W_1} \cdot \overline{X}, \overline{W_2} \cdot \overline{X}\}$ with two coefficient vectors.

- One can view the maxout as a generalization of the ReLU.
  - The ReLU is obtained by setting one of the coefficient vectors to 0.
  - The leaky ReLU can also be simulated by setting the other coefficient vector to $\overline{W_2} = \alpha \overline{W_1}$.

- Main disadvantage is that it doubles the number of parameters.

Gradient Clipping for Exploding Gradients

- Try to make the different components of the partial derivatives more even.
  - *Value-based clipping:* All partial derivatives outside ranges are set to range boundaries.
  - *Norm-based clipping:* The entire gradient vector is normalized by the L_2-norm of the entire vector.
- One can achieve a better conditioning of the values, so that the updates from mini-batch to mini-batch are roughly similar.
- Prevents an anomalous gradient explosion during the course of training.

Other Comments on Vanishing and Exploding Gradients

- The methods discussed above are only partial fixes.

- Other fixes discussed in later lectures:
  - Stronger initializations with pretraining.
  - Second-order learning methods that make use of second-order derivatives (or *curvature* of the loss function).

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

First-Order Gradient Descent Methods

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.5

First-Order Descent

- First-order methods work with steepest-descent directions.

- Modifications to basic form of steepest-descent:

  - Need to reduce step sizes with algorithm progression.

  - Need a way of avoiding local optima.

  - Need to address widely varying slopes with respect to different weight parameters.

Learning Rate Decay

- Initial learning rates should be high but reduce over time.

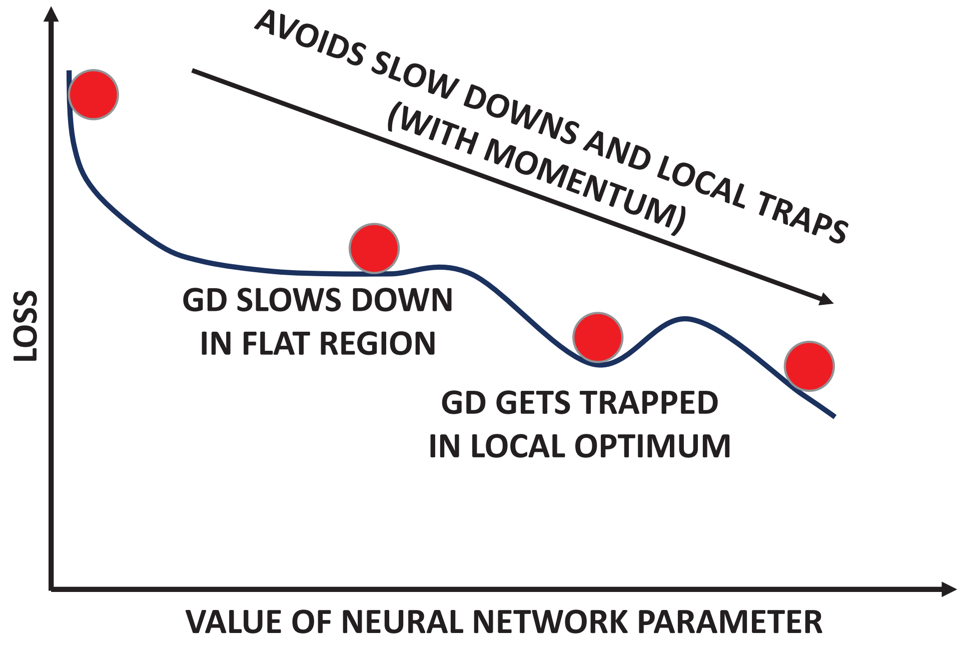
- The two most common decay functions are *exponential decay* and *inverse decay*.

- The learning rate α_t can be expressed in terms of the initial decay rate α_0 and epoch t as follows:

  $$\alpha_t = \alpha_0 \exp(-k \cdot t) \quad \text{[Exponential Decay]}$$
  $$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \quad \text{[Inverse Decay]}$$

  The parameter k controls the rate of the decay.

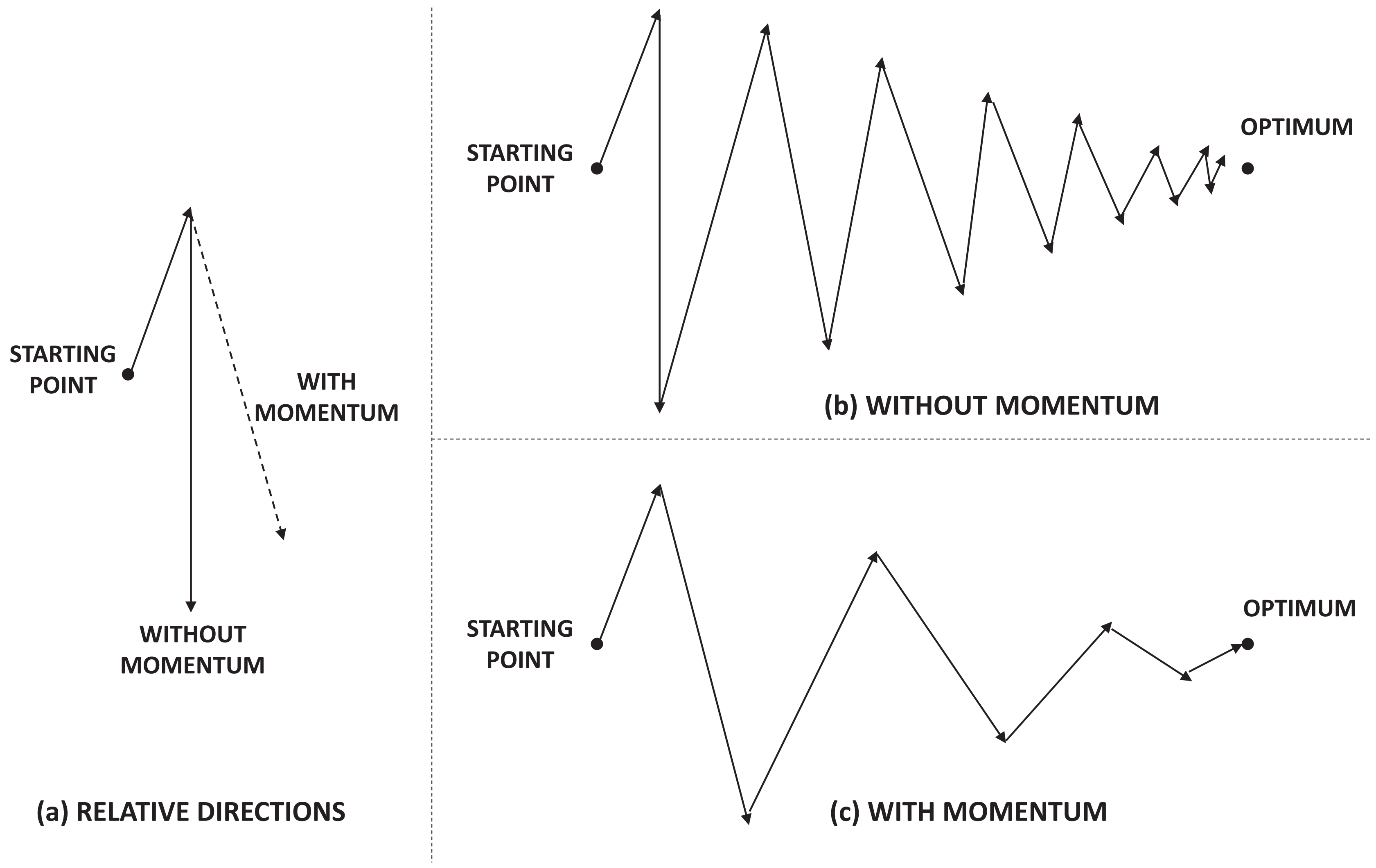
Momentum Methods: Marble Rolling Down Hill



- Use a *friction parameter* $\beta \in (0, 1)$ to gain speed in direction of movement.

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Avoiding Zig-Zagging with Momentum



Nesterov Momentum

- Modification of the traditional momentum method in which *the gradients are computed at a point that would be reached after executing a β-discounted version of the previous step again*.

- Compute at a point reached using only the momentum portion of the current update:

$$\overline{V} \Leftarrow \underbrace{\beta\overline{V}}_{\text{Momentum}} - \alpha\frac{\partial L(\overline{W} + \beta\overline{V})}{\partial\overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

- Put on the brakes as the marble reaches near bottom of hill.

- Nesterov momentum should always be used with mini-batch SGD (rather than SGD).

AdaGrad

- *Aggregate* squared magnitude of ith partial derivative in A_i.

- The square-root of A_i is proportional to the root-mean-square slope.

  – The absolute value will increase over time.

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{15}$$

- The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i}\right); \quad \forall i \tag{16}$$

- Use $\sqrt{A_i + \epsilon}$ in the denominator to avoid ill-conditioning.

AdaGrad Intuition

- Scaling the derivative inversely with $\sqrt{A_i}$ encourages faster *relative* movements along gently sloping directions.
  - Absolute movements tend to slow down prematurely.
  - Scaling parameters use stale values.

RMSProp

- The RMSProp algorithm uses *exponential smoothing* with parameter $\rho \in (0, 1)$ in the relative estimations of the gradients.
  - Absolute magnitudes of scaling factors do not grow with time.
  - Problem of staleness is ameliorated.

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{17}$$

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

- Use $\sqrt{A_i + \epsilon}$ to avoid ill-conditioning.

RMSProp with Nesterov Momentum

- Possible to combine RMSProp with Nesterov Momentum

$$v_i \Leftarrow \beta v_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i}\right); \quad w_i \Leftarrow w_i + v_i \quad \forall i$$

- Maintenance of A_i is done with shifted gradients as well.

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i}\right)^2 \quad \forall i \tag{18}$$

AdaDelta and Adam

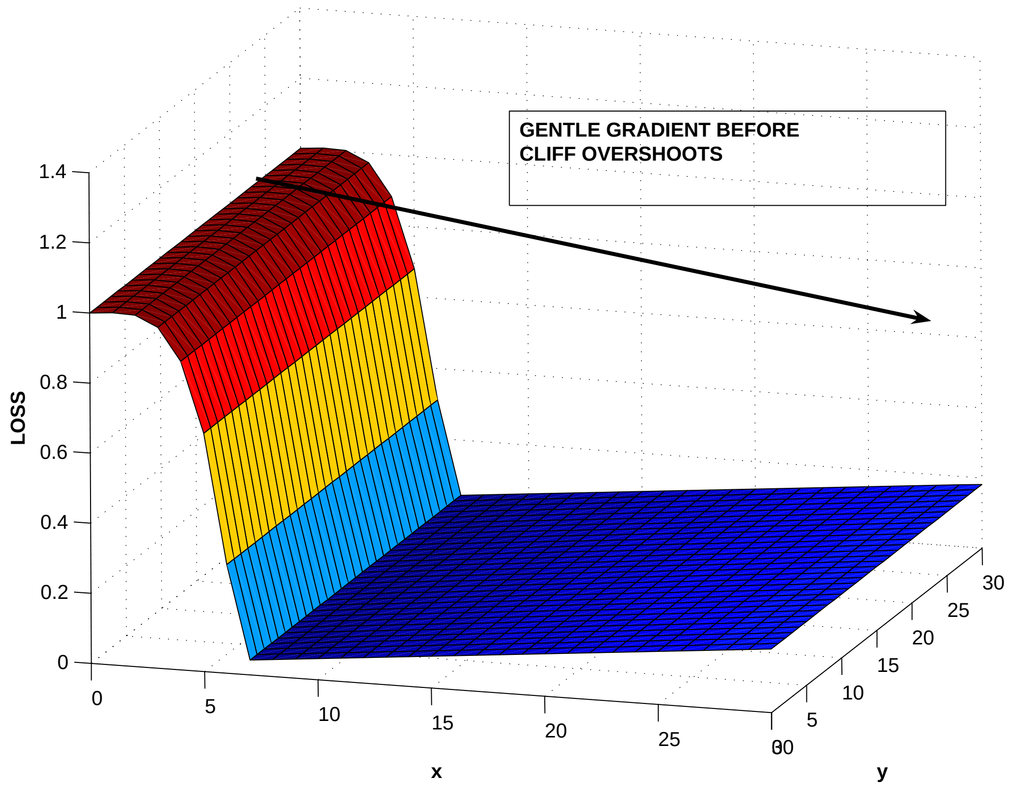
- Both methods derive intuition from RMSProp
  - AdaDelta track of an exponentially smoothed value of the *incremental changes* of weights Δw_i in previous iterations to decide parameter-specific learning rate.
  - Adam keeps track of exponentially smoothed gradients from previous iterations (in addition to normalizing like RMSProp).
- Adam is extremely popular method.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Second-Order Gradient Descent Methods

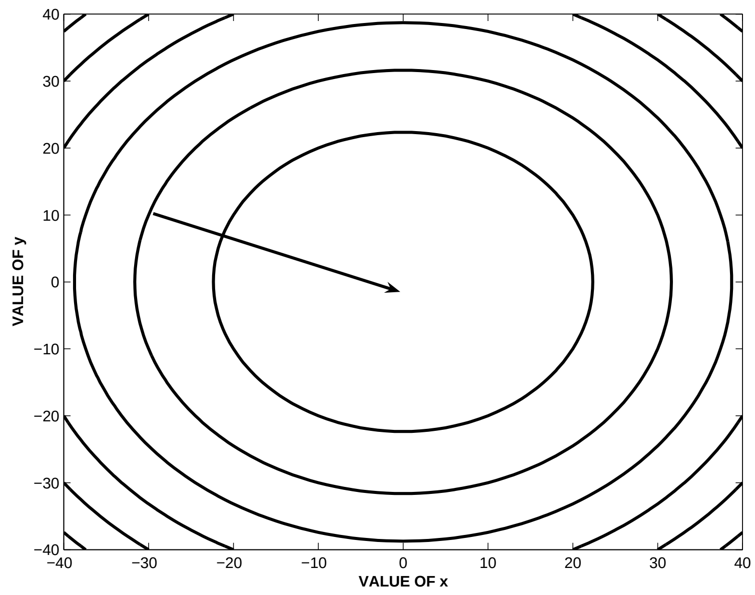
Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.5.5

Why Second-Order Methods?

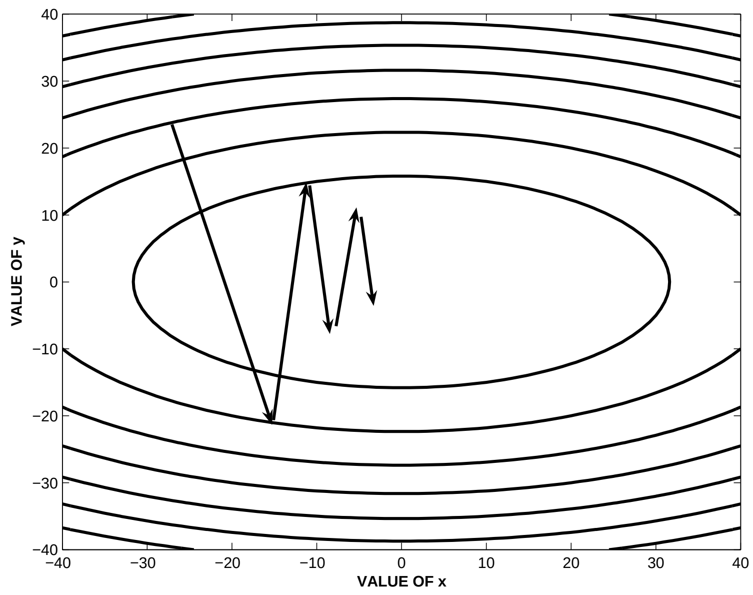


- First-order methods are not enough when there is curvature.

Revisiting the Bowl



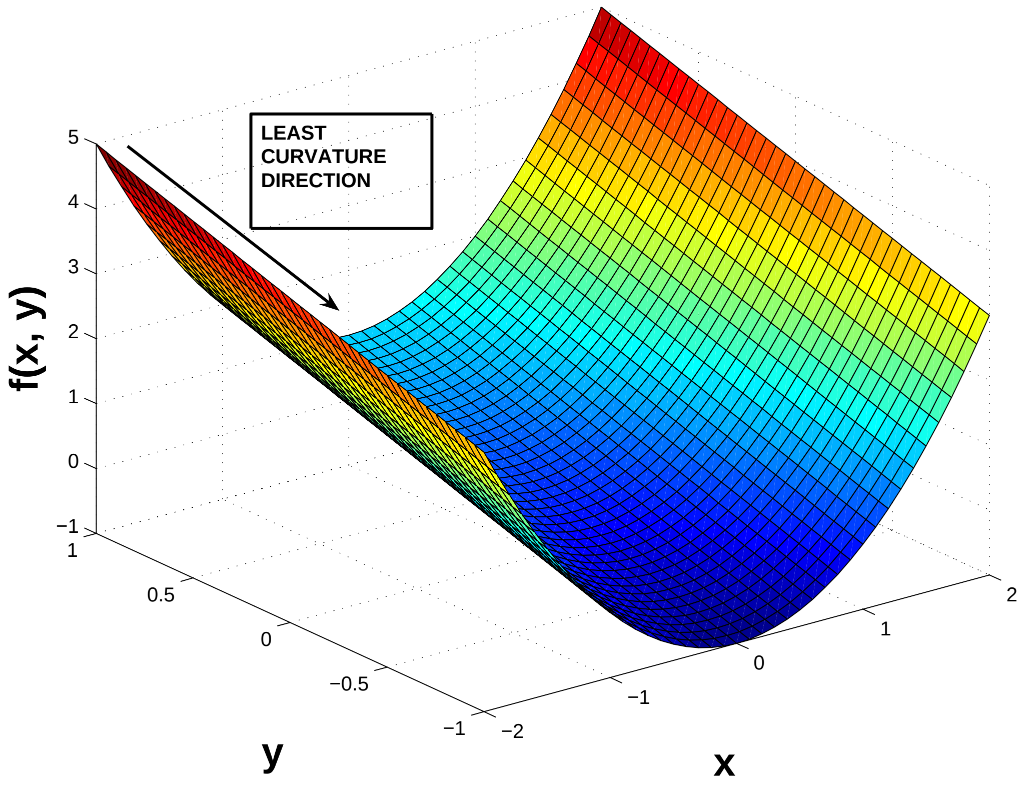
(a) Loss function is circular bowl
$L = x^2 + y^2$



(b) Loss function is elliptical bowl
$L = x^2 + 4y^2$

- High curvature directions cause bouncing in spite of higher gradient $\Rightarrow$ Need second-derivative for more information.

A Valley



- Gently sloping directions are better with less curvature!

The Hessian

- The second-order derivatives of the loss function $L(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 L(\overline{W})}{\partial w_i \partial w_j}$$

- The partial derivatives use all pairwise parameters in the denominator.

- For a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i, j)th entry is H_{ij}.

Quadratic Approximation of Loss Function

- One can write a quadratic approximation of the loss function with Taylor expansion about $\overline{W_0}$:

$$L(\overline{W}) \approx L(\overline{W}_0)+(\overline{W}-\overline{W}_0)^T[\nabla L(\overline{W}_0)]+\frac{1}{2}(\overline{W}-\overline{W}_0)^T H(\overline{W}-\overline{W}_0) \tag{19}$$

- One can derive a single-step optimality condition from initial point $\overline{W_0}$ by setting the gradient to 0.

Newton's Update

- Can solve quadratic approximation in one step from initial point $\overline{W}_0$.

  $\nabla L(\overline{W}) = 0$ [Gradient of Loss Function]
  $\nabla L(\overline{W}_0) + H(\overline{W} - \overline{W}_0) = 0$ [Gradient of Taylor approximation]

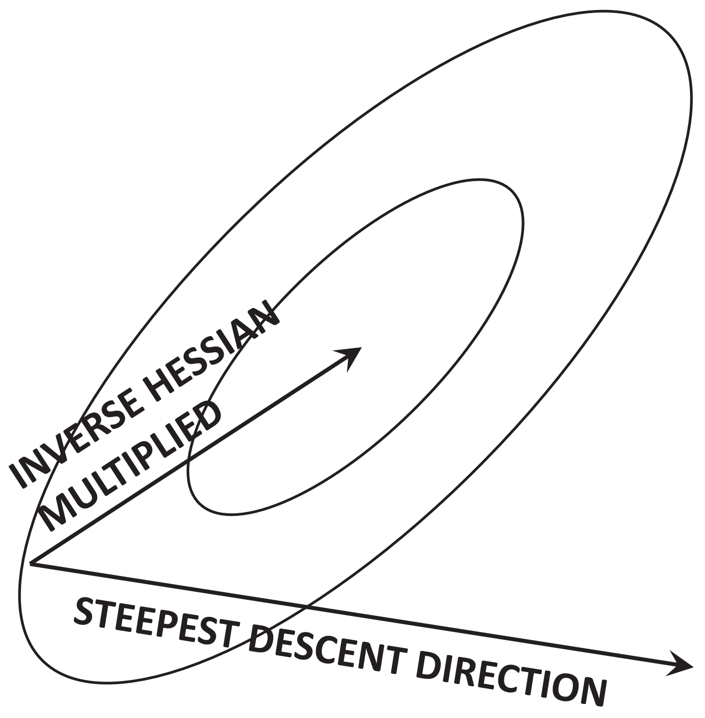
- Rearrange optimality condition to obtain Newton update:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla L(\overline{W}_0)] \tag{20}$$

- Note the ratio of first-order to second-order $\Rightarrow$ Trade-off between speed and curvature

- Step-size not needed!

Why Second-Order Methods?



- Pre-multiplying with the inverse Hessian finds a trade-off between speed of descent and curvature.

Basic Second-Order Algorithm and Approximations

- Keep making Newton's updates to convergence (single step needed for quadratic function)
  - Even computing the Hessian is difficult!
  - Inverting it is even more difficult
- Solutions:
  - Approximate the Hessian.
  - Find an algorithm that works with projection $H\overline{v}$ for some direction $\overline{v}$.

Conjugate Gradient Method

- Get to optimal in d steps (instead of single Newton step) where d is number of parameters.

- Use optimal step-sizes to get best point along a direction.

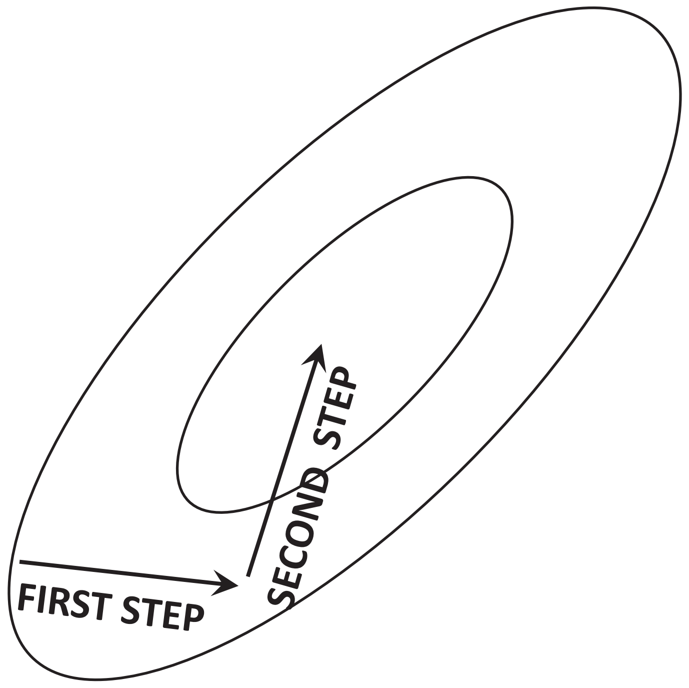
- *Thou shalt not worsen with respect to previous directions!*

- **Conjugate direction:** The gradient of the loss function on *any* point on an update direction is always orthogonal to the previous update directions.

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H [\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t \tag{21}$$

- For quadratic function, it requires d updates instead of single update of Newton method.

Conjugate Gradients on 2-Dimensional Quadratic



- Two conjugate directions are required to reach optimality

Conjugate Gradient Algorithm

- For quadratic functions only.
  - Update $\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$. Here, the step size α_t is computed using line search.
  - Set $\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t}\right) \overline{q}_t$. Increment t by 1.

- For non-quadratic functions approximate loss function with Taylor expansion and perform $\ll d$ of the above steps. Then repeat.

Efficiently Computing Projection of Hessian

- The update requires computation of the *projection* of the Hessian rather than inversion of Hessian.

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H [\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t \tag{22}$$

- Easy to perform numerically!

$$H\overline{v} \approx \frac{\nabla L(\overline{W}_0 + \delta \overline{v}) - \nabla L(\overline{W}_0)}{\delta} \tag{23}$$

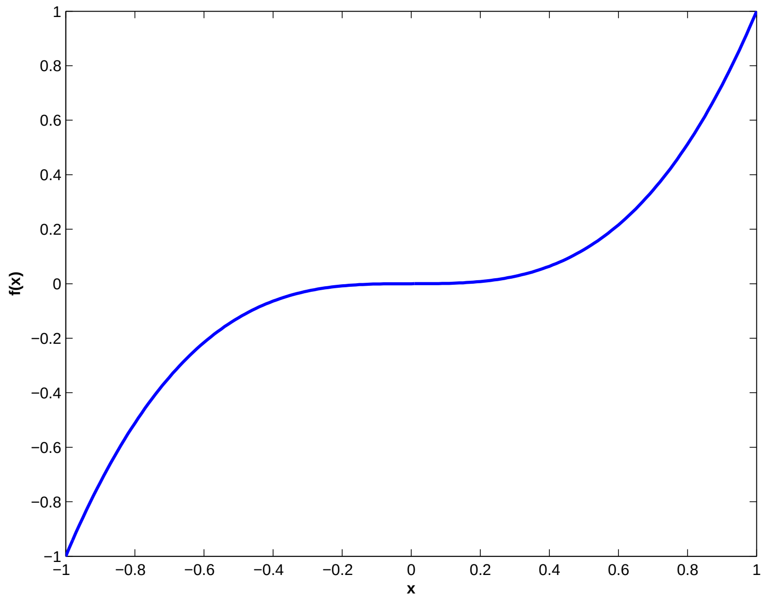
Other Second-Order Methods

- *Quasi-Newton Method:* A sequence of increasingly accurate approximations of the inverse Hessian matrix are used in various steps.

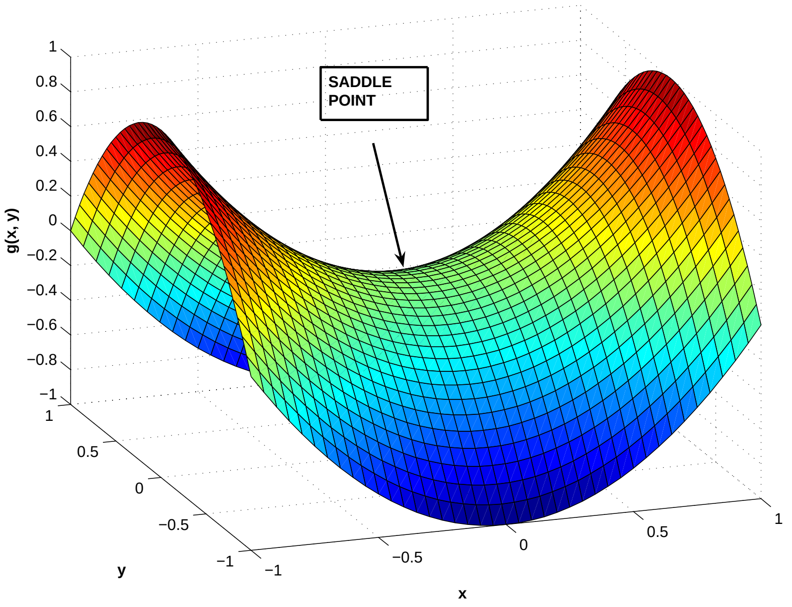
- Many variations of this approach.

- Commonly-used update is BFGS, which stands for the Broyden–Fletcher–Goldfarb–Shanno algorithm and its limited memory variant L-BFGS.

Problems with Second-Order Methods



(a) $f(x) = x^3$
Degenerate



(b) $f(x) = x^2 - y^2$
Stationary

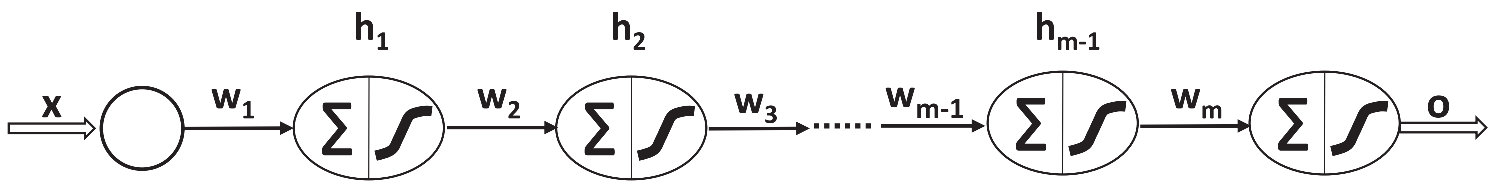
- Saddle points: Whether it is maximum or minimum depends on which direction we approach it from.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Batch Normalization

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.6

Revisiting the Vanishing and Exploding Gradient Problems



- Neural network with one node per layer.

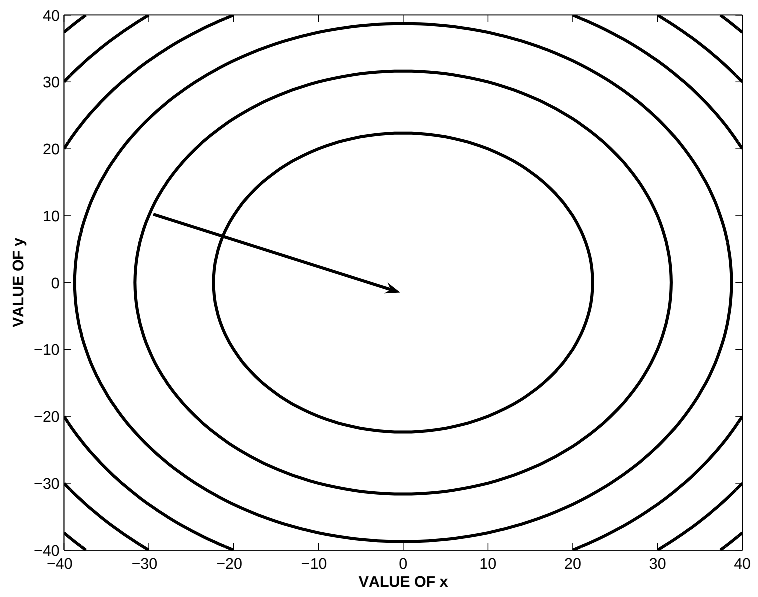
- Forward propagation multiplicatively depends on each weight and activation function evaluation.

- Backpropagated partial derivative get multiplied by weights and activation function derivatives.

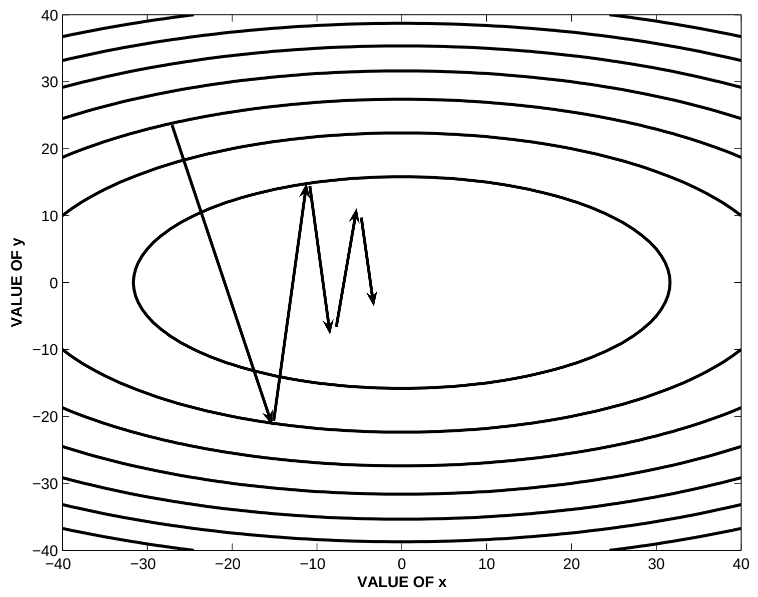
- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish).

- Hard to initialize weights exactly right.

Revisiting the Bowl



(a) Loss function is circular bowl
$L = x^2 + y^2$



(b) Loss function is elliptical bowl
$L = x^2 + 4y^2$

- Varying scale of different parameters will cause bouncing
- Varying scale of features causes varying scale of parameters

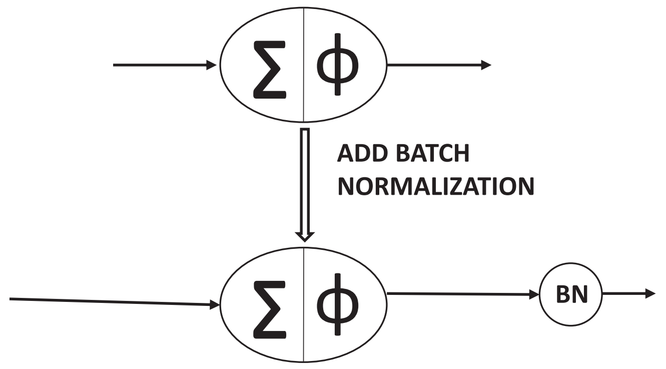
Input Shift

- One can view the input to each layer as a shifting data set of hidden activations during training.

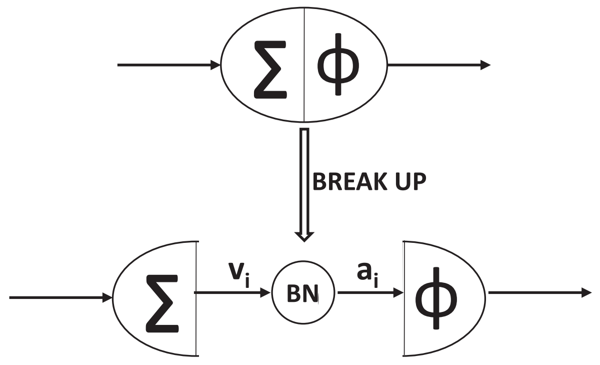
- A shifting input causes problems during learning.
  - Convergence becomes slower.
  - Final result may not generalize well because of unstable inputs.

- Batch normalization ensures (somewhat) more stable inputs to each layer.

Solution: Batch Normalization



(a) Post-activation normalization



(b) Pre-activation normalization

- Add an additional layer than normalizes in *batch-wise* fashion.

- Additional learnable parameters to ensure that optimal level of nonlinearity is used.

- Pre-activation normalization more common than post-activation normalization.

Batch Normalization Node

- The ith unit contains two parameters β_i and γ_i that need to be learned.

- Normalize over *batch* of m instances for ith unit.

$$\mu_i = \frac{\sum_{r=1}^{m} v_i^{(r)}}{m} \quad \forall i \qquad \text{[Batch Mean]}$$

$$\sigma_i^2 = \frac{\sum_{r=1}^{m} (v_i^{(r)} - \mu_i)^2}{m} + \epsilon \quad \forall i \qquad \text{[Batch Variance]}$$

$$\hat{v}_i^{(r)} = \frac{v_i^{(r)} - \mu_i}{\sigma_i} \quad \forall i, r \qquad \text{[Normalize Batch Instances]}$$

$$a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i \quad \forall i, r \qquad \text{[Scale with Learnable Parameters]}$$

- Why do we need β_i and γ_i?
  - Most activations will be near zero (near-linear regime).

Changes to Backpropagation

- We need to backpropagate through the newly added layer of normalization nodes.
    - The BN node can be treated like any other node.

- We want to optimize the parameters β_i and γ_i.
    - The gradients with respect to these parameters are computed during backpropagation.

- Detailed derivations in book.

Issues in Inference

- The transformation parameters μ_i and σ_i depend on the batch.

- How should one compute them during testing when a *single* test instance is available?

- The values of μ_i and σ_i are computed up front using the *entire* population (of training data), and then treated as constants during testing time.

  – One can also maintain exponentially weighted averages during training.

- The normalization is a simple linear transformation during inference.

Batch Normalization as Regularizer

- Batch normalization also acts as a regularizer.

- Same data point can cause somewhat different updates depending on which batch it is included in.

- One can view this effect as a kind of noise added to the update process.

- Regularization is can be shown to be equivalent to adding a small amount of noise to the training data.

- The regularization is relatively mild.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Model Generalization and the Bias-Variance Trade-Off

Neural Networks and Deep Learning, Springer, 2018
Chapter 4, Section 4.1-4.2

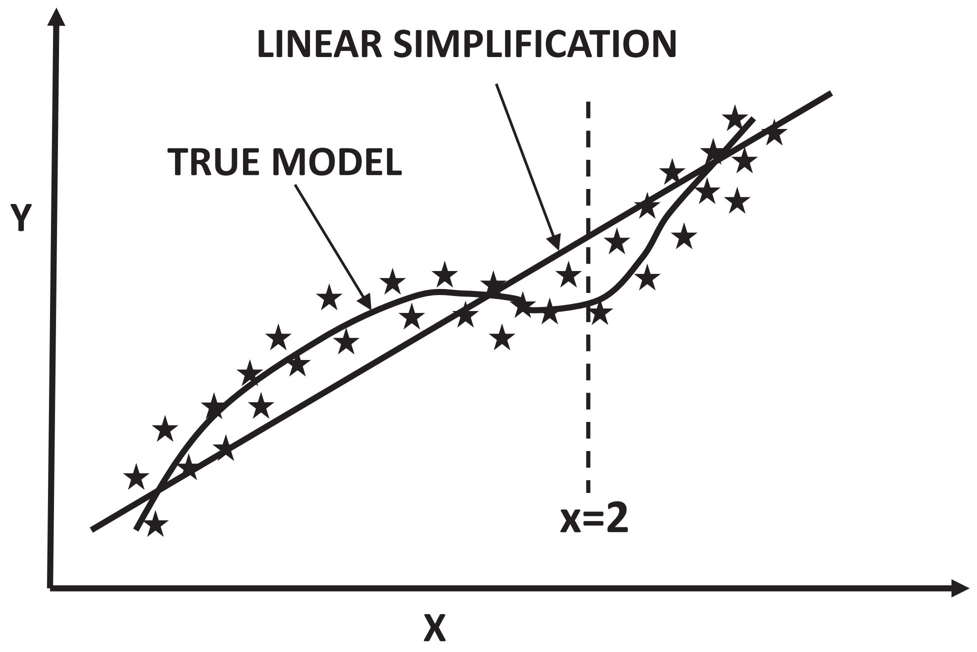
What is Model Generalization?

- In a machine learning problem, we try to generalize the known dependent variable on seen instances to unseen instances.
  - Unseen $\Rightarrow$ The model did not see it during training.
  - Given training images with seen labels, try to label an unseen image.
  - Given training emails labeled as spam or nonspam, try to label an unseen email.
- The classification accuracy on instances used to train a model is usually higher than on unseen instances.
  - We only care about the accuracy on unseen data.

Memorization vs Generalization

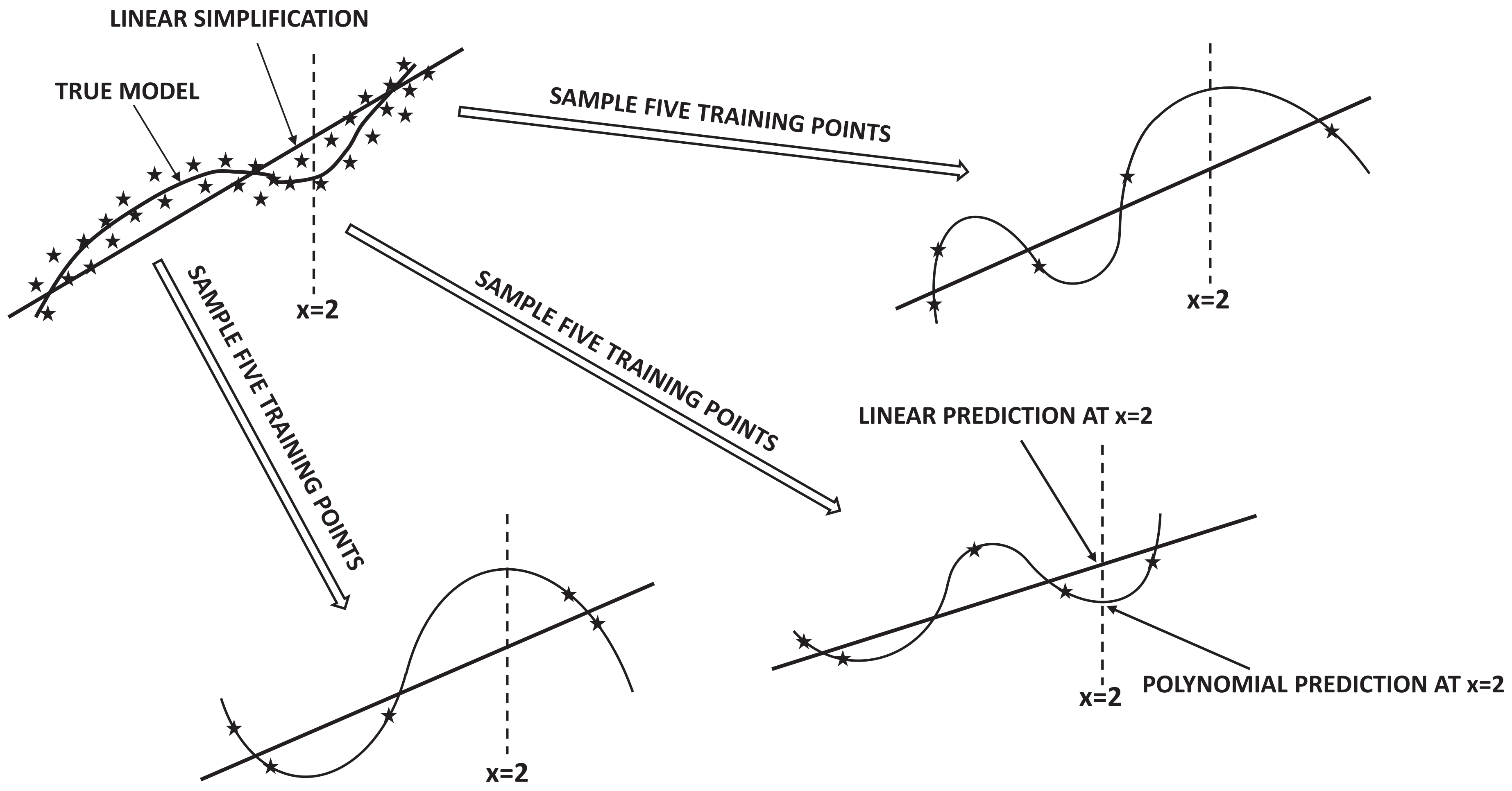
- Why is the accuracy on seen data higher?
  - Trained model remembers some of the irrelevant nuances.
- When is the gap between seen and unseen accuracy likely to be high?
  - When the amount of data is limited.
  - When the model is complex (which has higher *capacity* to remember nuances).
  - The combination of the two is a deadly cocktail.
- A high accuracy gap between the predictions on seen and unseen data is referred to as *overfitting*.

Example: Predict y from x



- **First impression:** Polynomial model such as $y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$ is "better" than linear model $y = w_0 + w_1 x$.
  - Bias-variance trade-off says: "Not necessarily! How much data do you have?"

Different Training Data Sets with Five Points


LINEAR SIMPLIFICATION
TRUE MODEL
x=2
SAMPLE FIVE TRAINING POINTS
x=2
SAMPLE FIVE TRAINING POINTS
SAMPLE FIVE TRAINING POINTS
LINEAR PREDICTION AT x=2
x=2
POLYNOMIAL PREDICTION AT x=2
x=2

- Zero error on training data but wildly varying predictions of $x = 2$

Observations

- The higher-order model is more complex than the linear model and has less *bias*.
  - But it has more parameters.
  - For a small training data set, the learned parameters will be more sensitive to the nuances of that data set.
  - Different training data sets will provide different predictions for y at a particular x.
  - This variation is referred to as model *variance*.
- Neural networks are inherently low-bias and high-variance learners $\Rightarrow$ Need ways of handling complexity.

Noise Component

- Unlike bias and variance, noise is a property of the *data* rather than the model.

- Noise refers to unexplained variations ϵ_i of data from true model $y_i = f(x_i) + \epsilon_i$.

- Real-world examples:

  – Human mislabeling of test instance $\Rightarrow$ Ideal model will never predict it accurately.

  – Error during collection of temperature due to sensor malfunctioning.

- Cannot do anything about it even if seeded with knowledge about true model.

Bias-Variance Trade-off: Setup

- Imagine you are given the true distribution $\mathcal{B}$ of training data (including labels).

- You have a principled way of sampling data sets $\mathcal{D} \sim \mathcal{B}$ from the training distribution.

- Imagine you create an infinite number of training data sets (and trained models) by repeated sampling.

- You have a *fixed* set $\mathcal{T}$ of unlabeled test instances.
  - The test set $\mathcal{T}$ does not change over different training data sets.
  - Compute prediction of each instance in $\mathcal{T}$ for each trained model.

Informal Definition of Bias

- Compute averaged prediction of each test instance x over different training models $g(x, \mathcal{D})$.

- Averaged prediction of test instance will be different from true (unknown) model $f(x)$.

- Difference between (averaged) $g(x, \mathcal{D})$ and $f(x)$ caused by erroneous assumptions/simplifications in modeling $\Rightarrow$ Bias
  - **Example:** Linear simplification to polynomial model causes bias.
  - If the true (unknown) model $f(x)$ were an order-4 polynomial, and we used any polynomial of order-4 or greater in $g(x, \mathcal{D})$, bias would be 0.

Informal Definition of Variance

- The value $g(x, \mathcal{D})$ will vary with $\mathcal{D}$ for fixed x.
  - The prediction of the same test instance will be different over different trained models.
- All these predictions cannot be simultaneously correct $\Rightarrow$ Variation contributes to error
- Variance of $g(x, \mathcal{D})$ over different training data sets $\Rightarrow$ Model Variance
  - **Example:** Linear model will have low variance.
  - Higher-order model will have high variance.

Bias-Variance Equation

- Let $E[MSE]$ be the expected mean-squared error of the fixed set of test instances over different samples of training data sets.

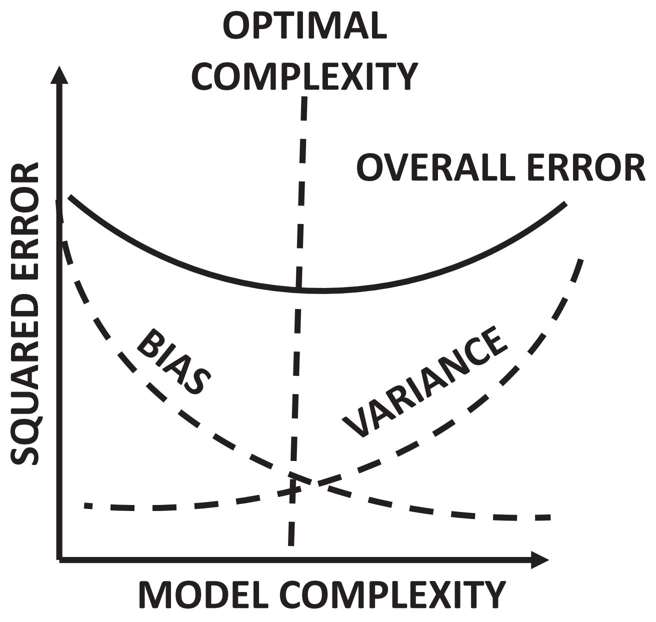
$$E[MSE] = \text{Bias}^2 + \text{Variance} + \text{Noise} \tag{1}$$

  - In linear models, the bias component will contribute more to $E[MSE]$.

  - In polynomial models, the variance component will contribute more to $E[MSE]$.

- We have a trade-off, when it comes to choosing model complexity!

The Bias-Variance Trade-Off



- Optimal point of model complexity is somewhere in middle.

Key Takeaway of Bias-Variance Trade-Off

- A model with greater complexity might be *theoretically* more accurate (i.e., low bias).
  - But you have less control on what it might predict on a tiny training data set.
  - Different training data sets will result in widely *varying* predictions of same test instance.
  - Some of these must be wrong $\Rightarrow$ Contribution of model variance.
- *A more accurate model for infinite data is not a more accurate model for finite data.*
  - Do not use a sledgehammer to swat a fly!

Model Generalization in Neural Networks

- The recent success of neural networks is made possible by increased data.
  - Large data sets help in generalization.

- In a neural network, increasing the number of hidden units in intermediate layers tends to increase complexity.

- Increasing depth often helps in reducing the number of units in hidden layers.

- Proper design choices can reduce overfitting in complex models $\Rightarrow$ Better to use complex models with appropriate design choices

How to Detect Overfitting

- The error on test data might be caused by several reasons.
  - Other reasons might be bias (underfitting), noise, and poor convergence.

- Overfitting shows up as a large gap between in-sample and out-of-sample accuracy.

- First solution is to collect more data.
  - More data might not always be available!

Improving Generalization in Neural Networks

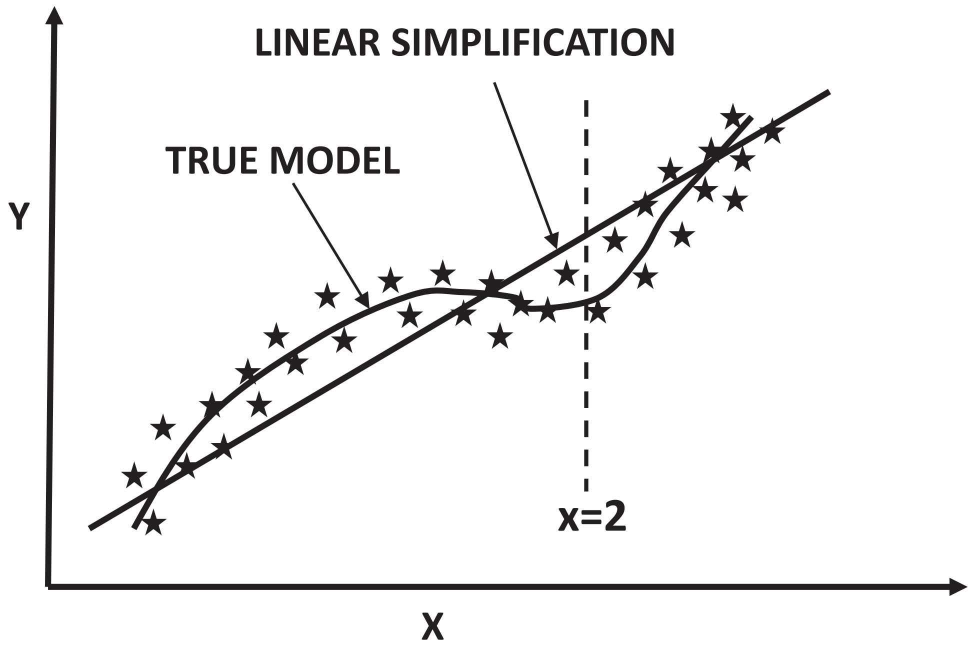
- Key techniques to improve generalization:
  - Penalty-based regularization.
  - Constraints like shared parameters.
  - Using ensemble methods like *Dropout*.
  - Adding noise and stochasticity to input or hidden units.
- Discussion in upcoming lectures.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Penalty-Based Regularization

Neural Networks and Deep Learning, Springer, 2018
Chapter 4, Section 4.4

Revisiting Example: Predict y from x



- **First impression:** Polynomial model such as $y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$ is "better" than linear model $y = w_0 + w_1 x$.
  - However, with less data, using the linear model is better.

Economy in Parameters

- A lower-order model has economy in parameters.
  - A linear model uses two parameters, whereas an order-4 model uses five parameters.
  - Economy in parameters discourages overfitting.
- Choosing a neural network with fewer units per layer enforces economy.

Soft Economy vs Hard Economy

- Fixing the architecture up front is an inflexible solution.

- A softer solution uses a larger model but imposes a (tunable) penalty on parameter use.

$$\hat{y} = \sum_{i=0}^{d} w_i x^i \tag{2}$$

- Loss function: $L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2 + \underbrace{\lambda \cdot \sum_{i=0}^{d} w_i^2}_{L_2-\text{Regularization}}$

- The (tuned) value of λ decides the level of regularization.

- Softer approach with a complex model performs better!

Effect on Updates

- For learning rate α, effect on update is to multiply parameter with $(1 - \alpha\lambda) \in (0, 1)$.

$$w_i \Leftarrow w_i(1 - \alpha\lambda) - \alpha \frac{\partial L}{\partial w_i}$$

  - **Interpretation:** Decay-based forgetting!

- Unless a parameter is important, it will have small absolute value.
  - Model decides what is important.
  - Better than inflexibly deciding up front.

L_1-Regularization

- In L_1-regularization, an L_1-penalty is imposed on the loss function.

$$L = \sum_{(x,y)\in\mathcal{D}} (y - \hat{y})^2 + \lambda \cdot \sum_{i=0}^{d} |w_i|_1$$

- Update has slightly different form:

$$w_i \Leftarrow w_i - \alpha\lambda s_i - \alpha\frac{\partial L}{\partial w_i}$$

- The value of s_i is the partial derivative of $|w_i|$ w.r.t. w_i:

$$s_i = \begin{cases} -1 & w_i < 0 \\ +1 & w_i > 0 \end{cases}$$

L_1- or L_2-Regularization?

- L_1-regularization leads to sparse parameter learning.
  - Zero values of w_i can be dropped.
  - Equivalent to dropping edges from neural network.

- L_2-regularization generally provides better performance.

Connections with Noise Injection

- *L_2-regularization with parameter λ is equivalent to adding Gaussian noise with variance λ to input.*
  - **Intuition:** Bad effect of noise will be minimized with simpler models (smaller parameters).
  - Proof in book.
- Result is only true for single layer network (linear regression).
  - Main value of result is in providing general intuition.
- Similar results can be shown for denoising autoencoders.

Penalizing Hidden Units

- One can also penalize hidden units.
- Applying L_1-penalty leads to sparse activations.
- More common in unsupervised applications for *sparse feature learning*.
- Straightforward modification of backpropagation.
  - Penalty contributions from hidden units are picked up in backward phase.

Hard and Soft Weight Sharing

- Fix particular weights to be the same based on domain-specific insights.
  - Discussed in lecture on backpropagation.

- **Soft Weight Sharing:** Add the penalty $\lambda(w_i - w_j)^2/2$ to loss function.
  - Update to w_i includes the extra term $\alpha\lambda(w_j - w_i)$.
  - Update to w_j includes the extra term $\alpha\lambda(w_i - w_j)$.
  - Pulls weights closer to one another.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Dropout

Neural Networks and Deep Learning, Springer, 2018
Chapter 4, Section 4.5

Feature Co-Adaptation

- The process of training a neural network often leads to a high level of dependence among features.

- Different parts of the network train at different rates:
  - Causes some parts of the network to adapt to others.

- This is referred to as feature co-adaptation.

- Uninformative dependencies are sensitive to nuances of specific training data $\Rightarrow$ Overfitting.

One-Way Adaptation

- Consider a single-hidden layer neural network.
  - All edges into and out of half the hidden nodes are fixed to random values.
  - Only the other half are updated during backpropagation.
- Half the features will adapt to the other half (random features).
- Feature co-adaptation is natural in neural networks where rate of training varies across different parts of network over time.
  - Partially a manifestation of training inefficiency (over and above true synergy).

Why is Feature Co-Adaptation Bad?

- We want features working together only when essential for prediction.
  - We do not want features adjusting to each other because of inefficiencies in training.
  - Does not generalize well to new test data.
- We want *many* groups of minimally essential features for robust prediction $\Rightarrow$ Better redundancies.
- We do not want a *few* large and inefficiently created groups of co-adapted features.

Basic Dropout Training Procedure

- For each training instance do:
  - Sample each node in the network in each layer (except output layer) with probability p.
  - Keep only edges for which both ends are included in network.
  - Perform forward propagation and backpropagation only on sampled network.
- Note that weights are shared between different sampled networks.

Basic Dropout Testing Procedures

- First procedure:
  - Perform repeated sampling (like training) and average results.
  - Geometric averaging for probabilistic outputs (averaging log-likelihood)

- Second procedure with *weight scaling inference rule* (more common):
  - Multiply weight of each outgoing edge of a sampled node i with its sampling probability p_i.
  - Perform single inference on full network with down-scaled weights.

Why Does Dropout Help?

- By dropping nodes, we are forcing the network to learn without the presence of some inputs (in each layer).
- Will resist co-adaptation, unless the features are truly synergistic.
- Will create many (smaller) groups of self-sufficient predictors.
- Many groups of self-sufficient predictors will have a model-averaging effect.

The Regularization Perspective

- One can view the dropping of a node as the same process as adding masking noise.
  - Noise is added to both input and hidden layers.

- Adding noise is equivalent to regularization.

- Forces the weights to become more spread out.
  - Updates are distributed across weights based on sampling.

Practical Aspects of Dropout

- Typical dropout rate (i.e., probability of exclusion) is somewhere between 20% to 50%.
- Better to use a larger network with Dropout to enable learning of independent representations.
- Dropout is applied to both input layers and hidden layers.
- Large learning rate with decay and large momentum.
- Impose a max-norm constraint on the size of network weights.
  - Norm of input weights to a node upper bounded by constant c.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Unsupervised Pretraining

Neural Networks and Deep Learning, Springer, 2018
Chapter 4, Section 4.7

Importance of Initialization

- Bad initializations can lead to unstable convergence.

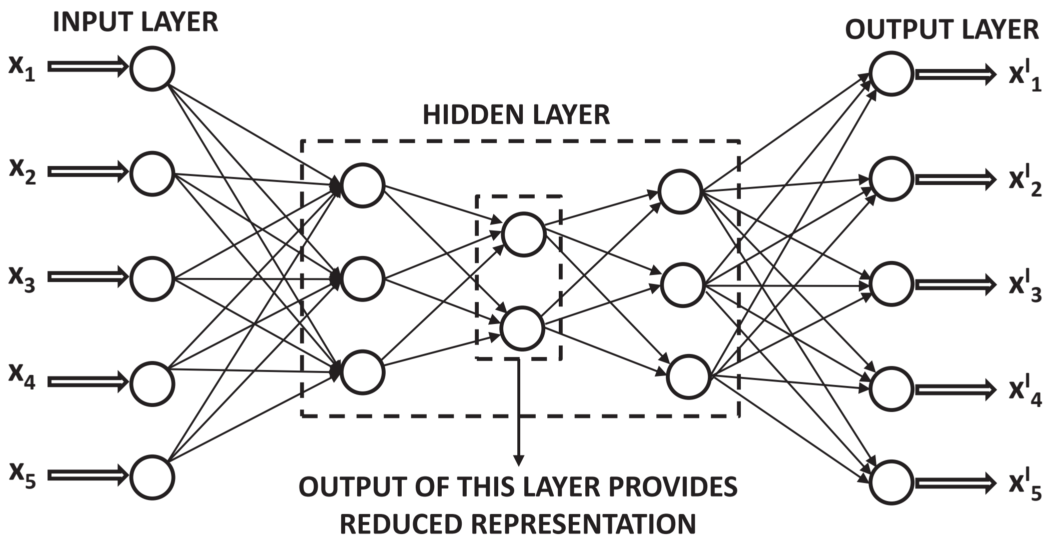
- Typical approach is to initialize to a Gaussian with variance $1/r$, where r is the indegree of the neuron.
  - Xavier initialization uses both indegree and outdegree.

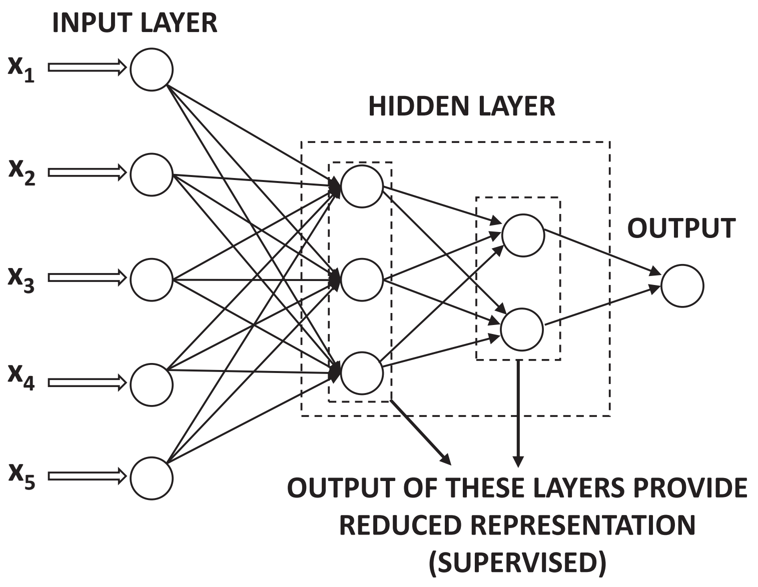
- Pretraining goes beyond these simple initializations *by using the training data*.

Types of Pretraining

- *Unsupervised pretraining:* Use training data without labels for initialization.
  - Improves convergence behavior.
  - Regularization effect.
- *Supervised pretraining:* Use training data with labels for initialization.
  - Improves convergence but might overfit.
- Focus on unsupervised pretraining.

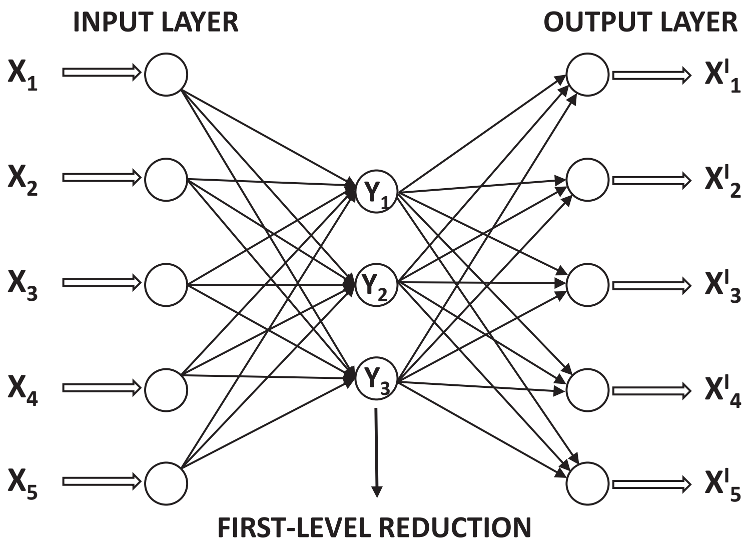
Types of Base Applications



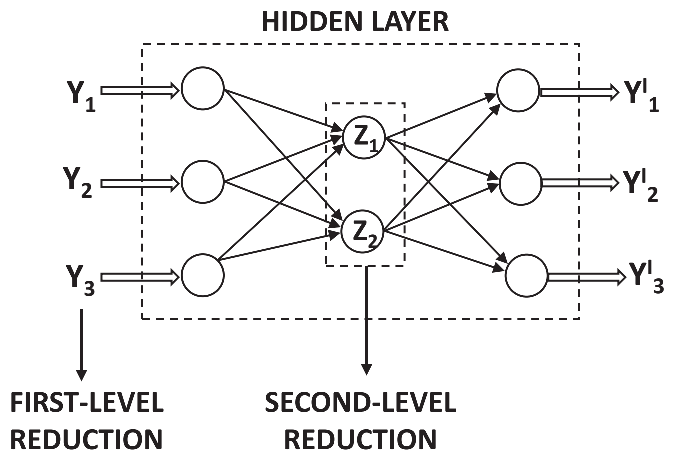

INPUT LAYER
HIDDEN LAYER
OUTPUT
x_1
x_2
x_3
x_4
x_5
OUTPUT OF THESE LAYERS PROVIDE REDUCED REPRESENTATION (SUPERVISED)

- Both the two neural architectures use almost the same pre-training procedure

Layer-Wise Pretraining a Deep Autoencoder



(a) Pretraining first-level reduction and outer weights



(b) Pretraining second-level reduction and inner weights

- Pretraining deep autoencoder helps in convergence issues

Pretraining a Supervised Learner

- For a supervised learner with k hidden layers:
  - Remove output layer and create an autoencoder with $(2k-1)$ hidden layers [Refer two slides back].
  - Pretrain autoencoder as discussed in previous slide.
  - Keep only weights from encoder portion and cap with output layer.
  - Pretrain only output layer.
  - Fine-tune all layers.

Some Observations

- For unsupervised pretraining, other methods may be used.

- Historically, restricted Boltzmann machines were used before autoencoders.

- One does not need to pretrain layer-by-layer.

  – We can group multiple layers together for pretraining (e.g., VGGNet).

  – Trade-off between component-wise learning and global quality.

Why Does Pretraining Work?

- Pretraining already brings the activations of the neural network to the manifold of the data distribution.

- Features correspond to repeated patterns in the data.

- Fine-tuning learns to combine/modify relevant ones for inference.
  - Pretraining initializes the problem closer to the basin of global optima.
  - Hinton: "*To recognize shapes, first learn to generate images.*"

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Regularization in Unsupervised Applications [Denoising, Contractive, Variational Autoencoders]

Neural Networks and Deep Learning, Springer, 2018
Chapter 4, Section 4.10

Supervised vs Unsupervised Applications

- There is always greater tendency to overfit in supervised applications.
  - In supervised applications, we are trying to learn a single bit of target data.
  - In unsupervised applications, a lot more target data is available.

- The goal of regularization is often to provide specific properties to the reduced representation.

- Regularized autoencoders often use a larger number of hidden units than inputs (overcomplete).

Sparse Feature Learning

- Use a larger number of hidden units than input units.

- Add L_1-penalties to the hidden layer.
  - Backpropagation picks up the flow from penalties in hidden layer.

- Use only top activations in hidden layer.
  - Backpropagate only through top activations.
  - Behaves like adaptive ReLU.

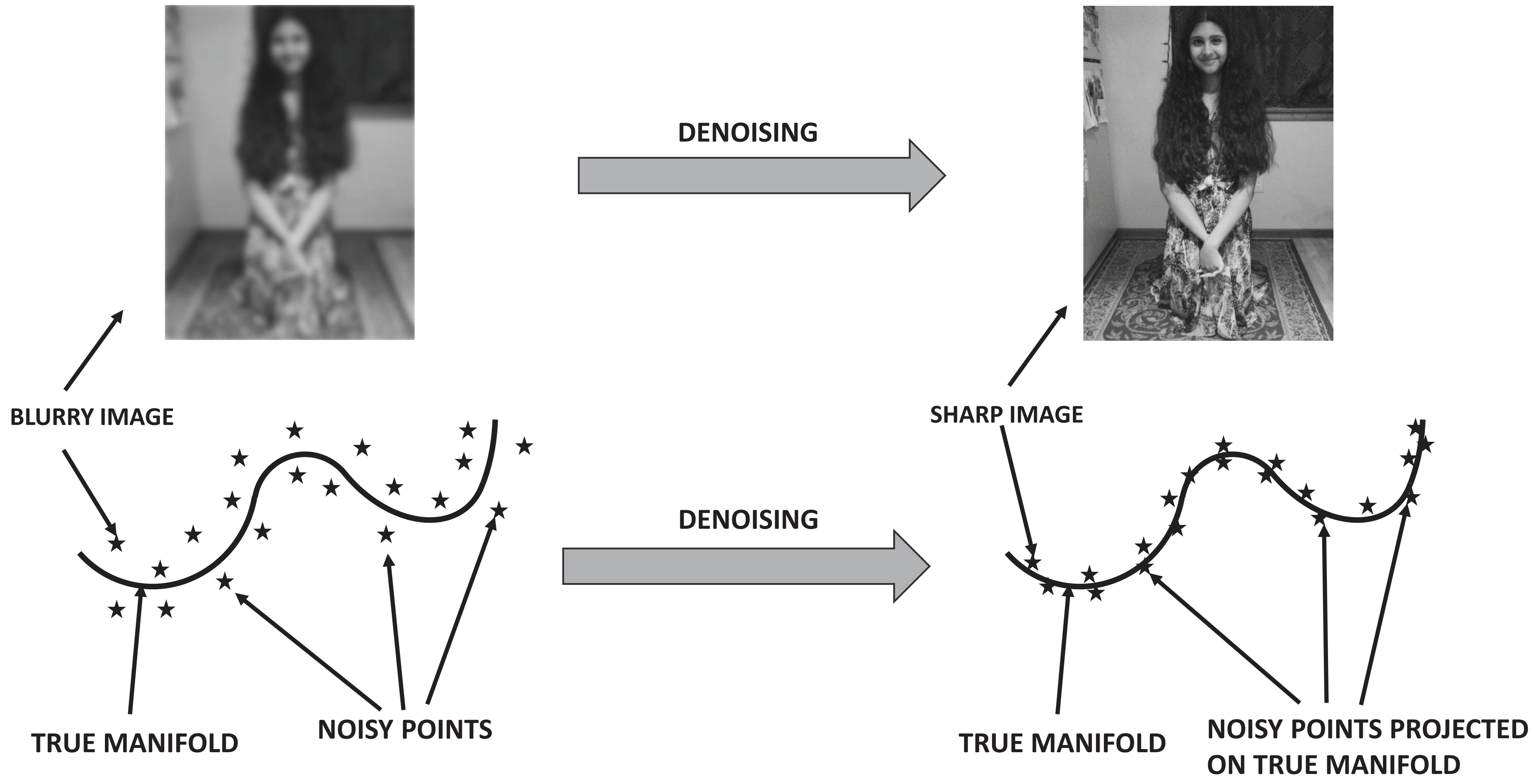
Denoising Autoencoder

- Add noise to the input representation.
  - Gaussian noise for real-valued data and masking noise for binary data.

- Output remains unchanged.

- For single-layer autoencoder with linear activations, Gaussian noise results in L_2-regularized SVD.

Illustration of Denoising Autoencoder



Gradient-Based Penalization: Contractive Autoencoders

- We do not want the hidden representation to change very significantly with small *random* changes in input values.
  - Key point: Most random changes in full-dimensional space are roughly perpendicular to a low-dimensional manifold containing the training data.

- Use a regularization term which tends to selectively damp the component of the movement perpendicular to manifold.
  - Regularizer damps in all directions, but faces no resistance in orthogonal direction to manifold.

Loss Function

- The loss function adds up the reconstruction error and uses penalties on the gradients of the hidden layer.

$$L = \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 \tag{3}$$

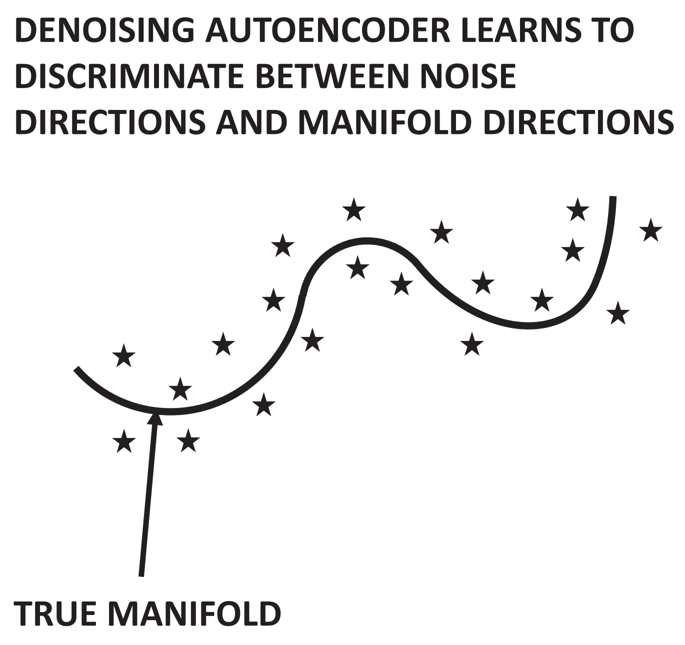
- Regularizer = Sum of squares of the partial derivatives of all hidden variables with respect to all input dimensions.

- Problem with k hidden units denoted by $h_1 \ldots h_k$:

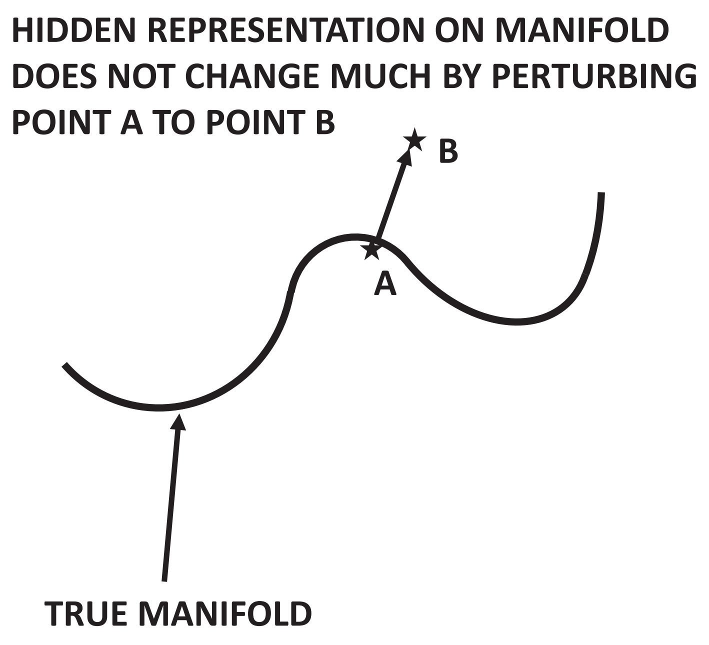
$$R = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{k} \left(\frac{\partial h_j}{\partial x_i} \right)^2 \tag{4}$$

- We want to optimize $L + \lambda R \Rightarrow$ Using single linear layer leads to L_2-regularized SVD!

Contractive Autoencoder vs Denoising Autoencoder


DENOISING AUTOENCODER LEARNS TO DISCRIMINATE BETWEEN NOISE DIRECTIONS AND MANIFOLD DIRECTIONS
TRUE MANIFOLD

DENOISING AUTOENCODER


HIDDEN REPRESENTATION ON MANIFOLD DOES NOT CHANGE MUCH BY PERTURBING POINT A TO POINT B
B
A
TRUE MANIFOLD

CONTRACTIVE AUTOENCODER

- Movements inconsistent with data distribution are damped.
- New data point will be projected to manifold (like denoising autoencoder)

Variational Autoencoder

- All the autoencoders discussed so far create a deterministic hidden representation.

- The variational autoencoder creates a *stochastic* hidden representation.

- The output is a *sample* from the stochastic representation.

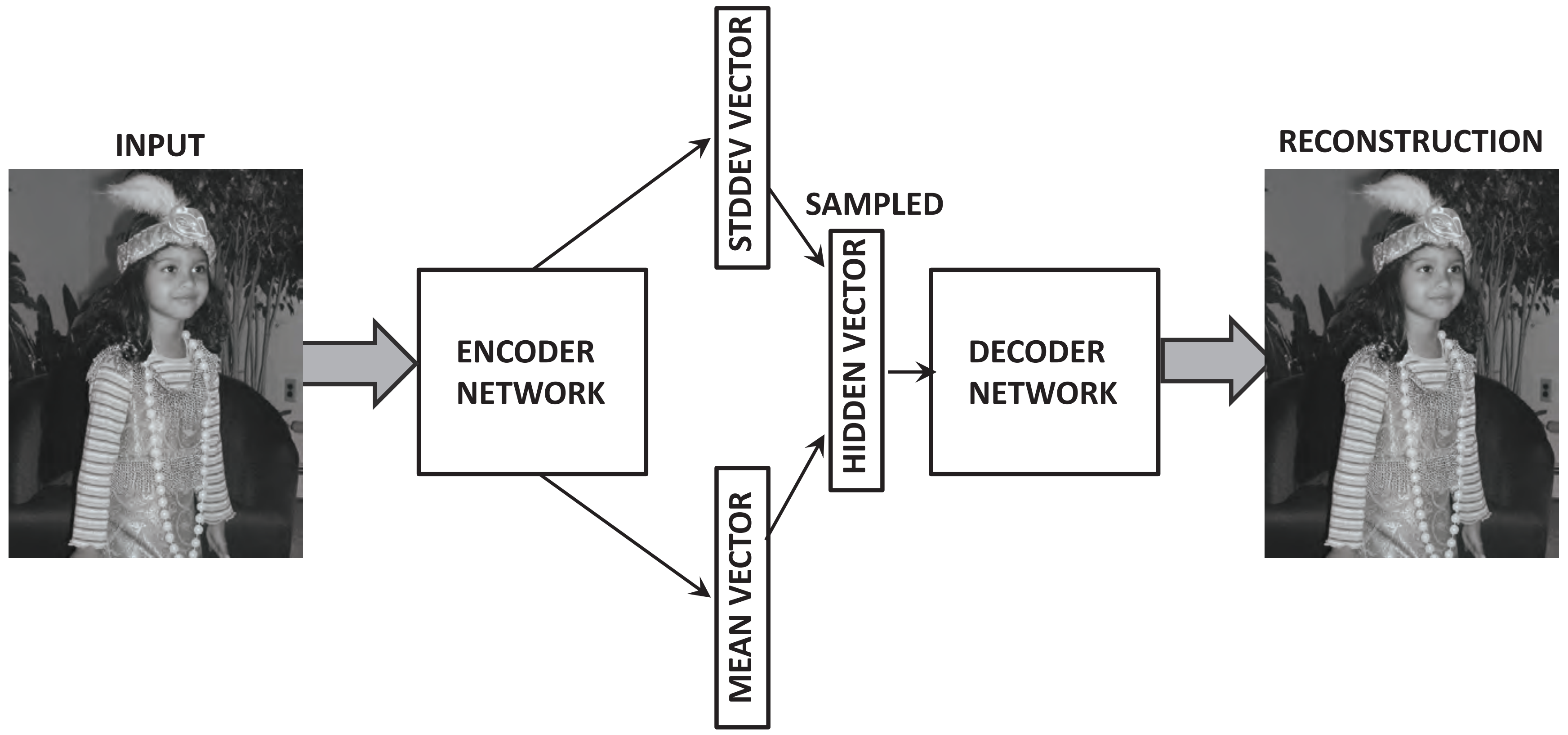
- Objective contains (i) reconstruction error of sample, and (ii) regularization terms pushing the parameters of distribution to unit Gaussian.

Regularization of Hidden Distribution

- The hidden distribution is pushed towards Gaussian with zero mean and unit variance in k dimensions *over the full training data*.
  - However, the *conditional* distribution on a specific input point will be a Gaussian with its own mean vector $\overline{\mu}(\overline{X})$ and standard deviation vector $\overline{\sigma}(\overline{X})$.
  - The encoder outputs $\overline{\mu}(\overline{X})$ and $\overline{\sigma}(\overline{X})$ to create samples for decoder.

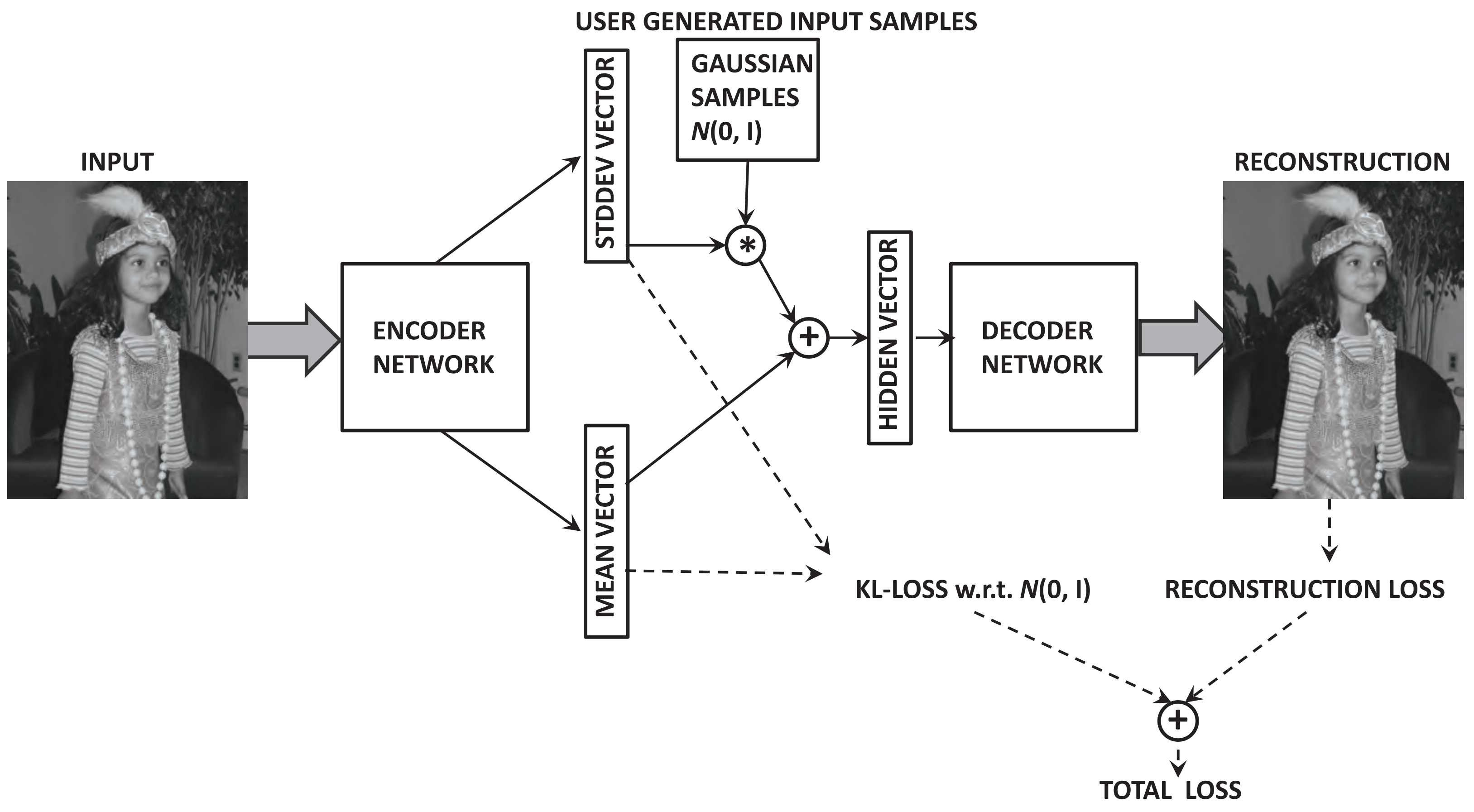
- Regularizer computes KL-divergence between $\mathcal{N}(0, I)$ and $\mathcal{N}(\overline{\mu(\overline{X})}, \overline{\sigma}(\overline{X}))$.

Stochastic Architecture with Deterministic Inputs



- One of the operations is sampling from hidden layer ⇒ Cannot backpropagate!

Conversion to Deterministic Architecture with Stochastic Inputs



- Sampling is accomplished by using pre-generated input samples $\Rightarrow$ Can backpropagate!

Objective Function

- Reconstruction loss same as other models:

$$L = \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 \tag{5}$$

- Regularizer is KL-divergence between unit Gaussian and conditional Gaussian:

$$R = \frac{1}{2}\left(\underbrace{||\overline{\mu}(\overline{X})||^2}_{\overline{\mu}(\overline{X})_i \Rightarrow 0} + \underbrace{||\overline{\sigma}(\overline{X})||^2 - 2\sum_{i=1}^{k} \ln(\overline{\sigma}(\overline{X})_i)}_{\overline{\sigma}(\overline{X})_i \Rightarrow 1} - k \right) \tag{6}$$

- Overall objective is $L + \lambda R \Rightarrow$ Backpropagate with deterministic architecture!

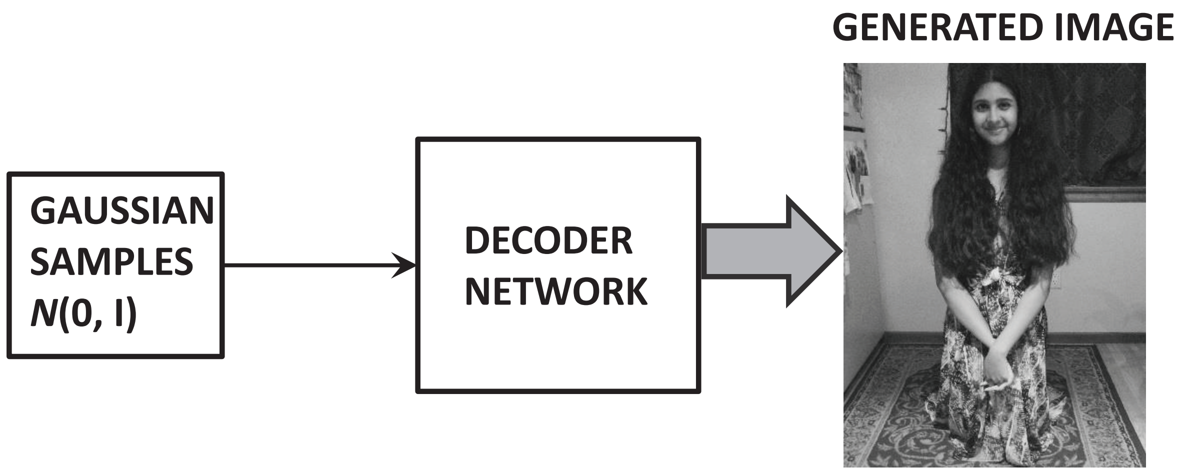
Connections

- A variational autoencoder will regularize because stochastic (noisy) hidden representation needs to reconstruct.
  - One can interpret the mean as the representation and the standard deviation as the noise robustness of hidden representation.
  - In a denoising autoencoder, we add noise to the inputs.
- Contractive autoencoder is also resistant to noise in inputs (by penalizing hidden-to-input *derivative*).
  - Ensures that hidden representation makes muted changes with small input noise.

Comparisons

- In denoising autoencoder, noise resistance is shared by encoder and decoder.
  - Often use both in denoising applications.
- In contractive autoencoder, encoder is responsible for noise resistance.
  - Often use only encoder for dimensionality reduction.
- In variational autoencoder, decoder is responsible for noise resistance.
  - Often use only decoder [next slide].

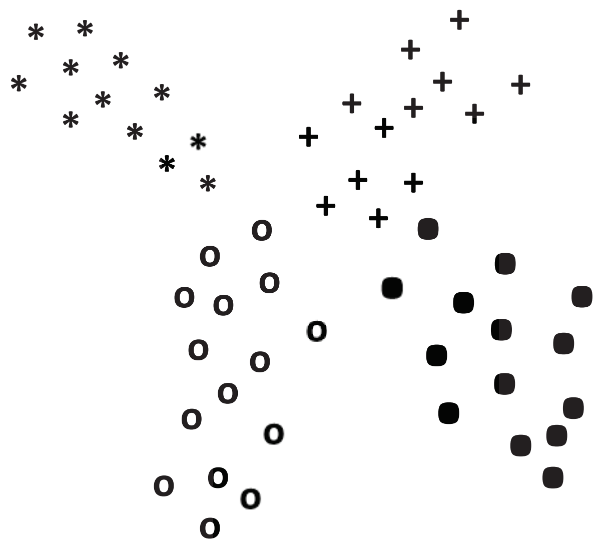
Variational Autoencoder is Useful as Generative Model



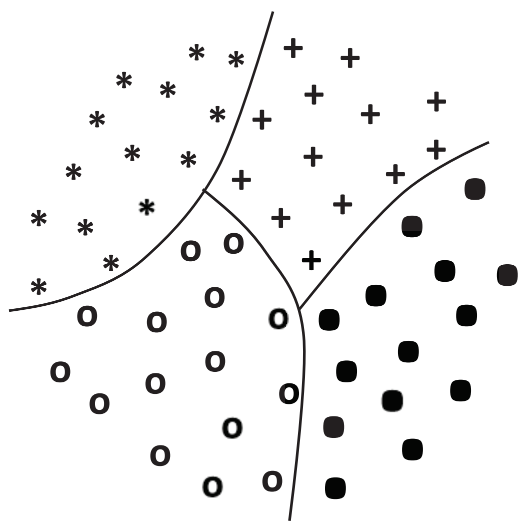
- Throw away encoder and feed samples from $\mathcal{N}(0, I)$ to decoder.

- Why is this possible for variational autoencoders and not other types of models?

Effect of the Variational Regularization



2-D LATENT EMBEDDING
(NO REGULARIZATION)



2-D LATENT EMBEDDING
(VAE)

- Most autoencoders will create representations with large discontinuities in the hidden space.

- Discontinuous regions will not generate meaningful points.

Applications of Variational Autoencoder

- Variational autoencoders have similar applications as Generative Adversarial Networks (GANs).
  - Can also develop conditional variants to fill in missing information (like cGANs).
  - More details in book.
- Quality of generated data is often not as sharp as GANs.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Radial Basis Function Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 5

Radial Basis Function Networks

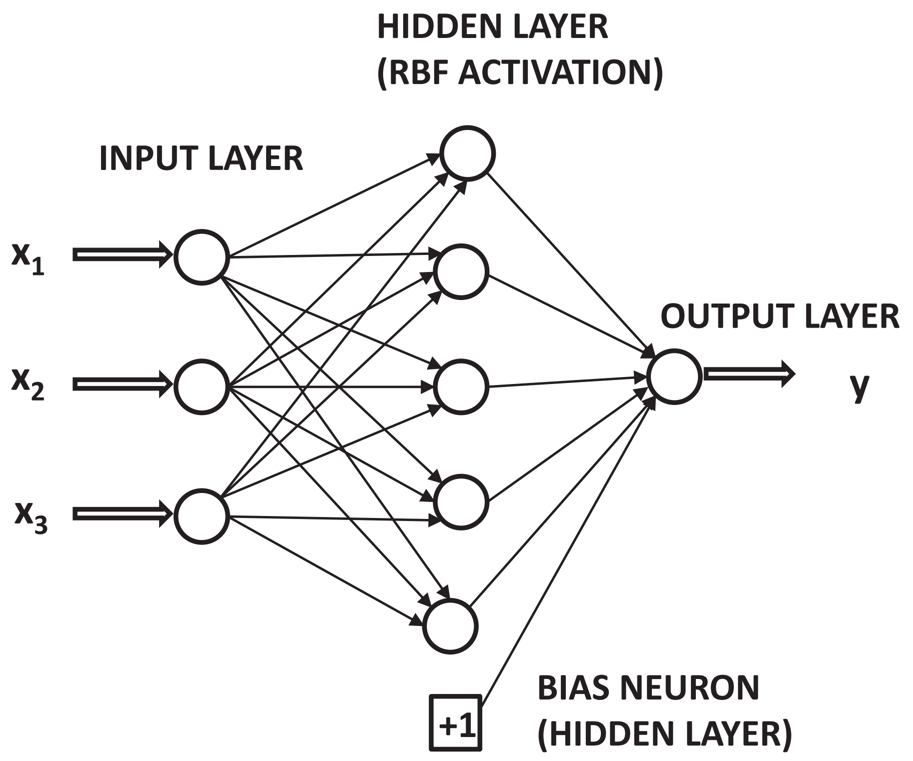
- Radial basis function (RBF) networks represent a fundamentally different paradigm in neural networks.
  - Not deep learners $\Rightarrow$ Often a single *unsupervised* hidden layer is used.
  - Deep learners represent an exercise in supervised feature engineering.
- RBF networks are closely related to SVMs.
  - SVMs represent a special case of RBF networks.
  - Like SVMs, RBF networks are universal function approximators.

When to Use RBF Networks

- Deep networks work best when the data has rich structure (e.g., images).
  - Property of hierarchical and supervised feature engineering.

- RBF networks are best when the data is noisy (but structure is less intricate).
  - Unsupervised feature engineering is robust to noise.

RBF Network



- Single (unsupervised) hidden layer with high dimensionality $m \gg d$ and linear output layer.

- Each hidden unit contains a prototype vector and activation depends on similarity of input to prototype (kernel similarity!).

Workings of the RBF Network

- Each of m hidden units has its own prototype vector $\overline{\mu}_i$ and bandwidth σ_i.
  - Common to set each $\sigma_i = \sigma$.

- For input vector $\overline{X}$, activation h_i of ith hidden unit (no weights!):

$$h_i = \Phi_i(\overline{X}) = \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right) \quad \forall i \in \{1, \ldots, m\} \qquad (1)$$

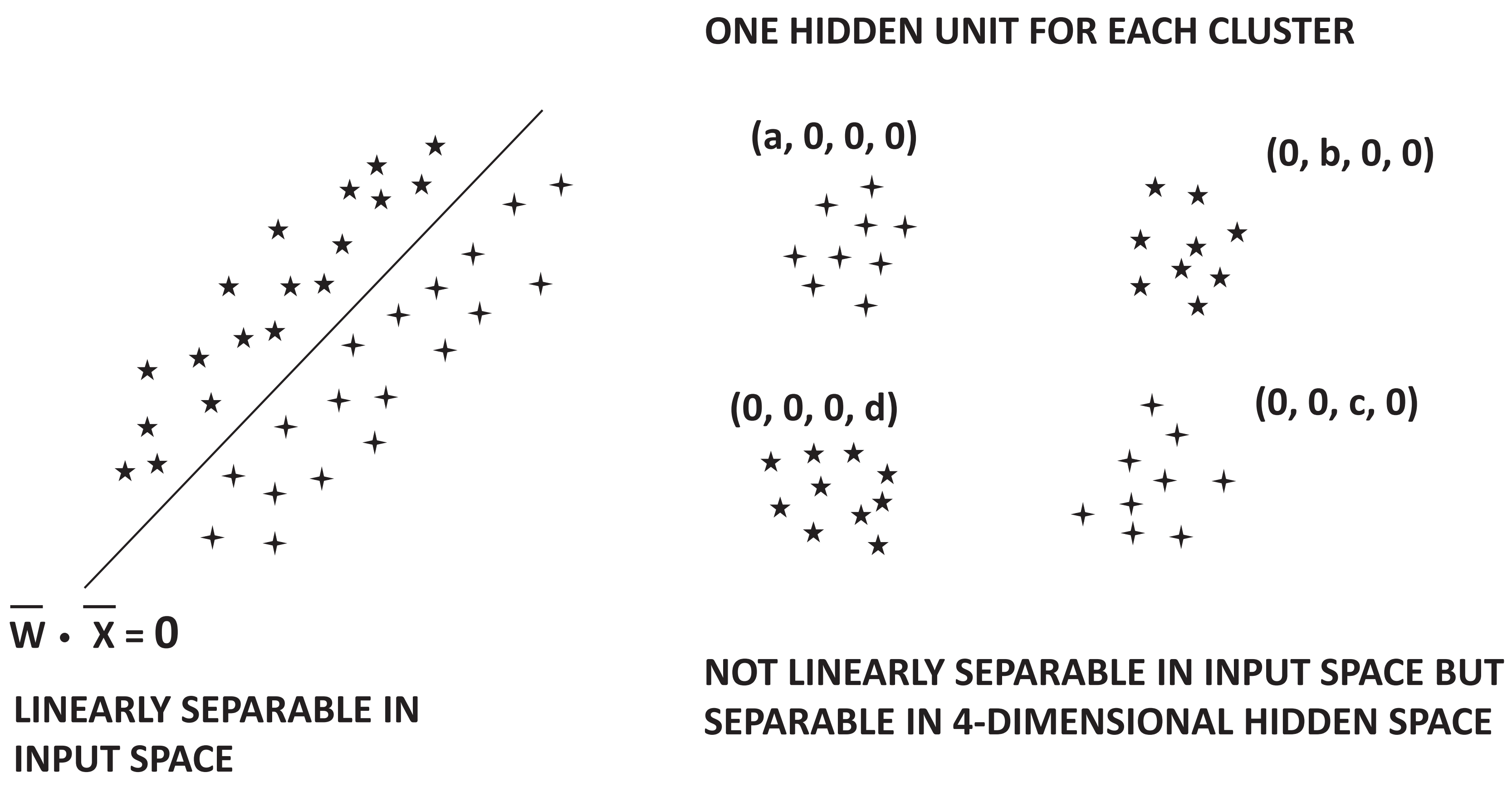
- Output layer is linear classifier/regressor with weights w_i.

$$\hat{y} = \sum_{i=1}^{m} w_i h_i \text{ [Real-valued outputs]}$$

How do RBF Networks Classify Nonlinearly Separable Classes?

- Work on Cover's principle of separability of patterns.
- Transforming low-dimensional data to high-dimensional space leads to greater ease in linear separation.
- The prototypes define local influence regions of the space.
  - Each feature corresponds to a local region.
- The final layer puts each region on the appropriate side of the separator.

Illustration of Separation Process


ONE HIDDEN UNIT FOR EACH CLUSTER
(a, 0, 0, 0)
(0, b, 0, 0)
(0, 0, 0, d)
(0, 0, c, 0)
$\overline{W} \cdot \overline{X} = 0$
LINEARLY SEPARABLE IN INPUT SPACE
NOT LINEARLY SEPARABLE IN INPUT SPACE BUT SEPARABLE IN 4-DIMENSIONAL HIDDEN SPACE

- One prototype from each cluster.
- Each local region is mapped to its own feature with a possible linear separator as $\overline{W} = [1, -1, 1, -1]$.

Training an RBF Network

- Training works in two phases:
  - Learn the prototype vectors $\overline{\mu}_i$ and bandwidth σ in an unsupervised manner.
  - Learn the weights of the output layer in supervised manner.
    - Straightforward training of single-layer network with engineered features.

Training the Hidden Layer

- Only need to find the prototype vectors $\overline{\mu}_i$ and bandwidth σ.
  - The prototypes can be sampled from data or can be centroids of clusters.

- Let d_{max} be maximum distance between pairs of prototypes and d_{ave} be average distance.
  - Two heuristic choices of σ are $d_{max}/\sqrt{m}$ and $2 \cdot d_{ave}$.
  - The bandwidth can also also be tuned on validation data.

Kernel Methods are Special Cases of RBF Networks

- Set the prototypes to all data points and:
  - Linear output layer (squared loss) for kernel regression/Fisher discriminant.
  - Linear output layer (hinge loss) for SVM
  - Logistic output layer (log loss) for kernel logistic regression
- Proofs in book.

Are Supervised Methods Any Good?

- Supervised training methods for hidden layer discussed in book.

- Generally, supervision of hidden layer leads to overfitting.
  - Supervised feature engineering is generally done by deep networks.
  - RBF networks are too shallow!
  - RBF prototype/bandwidth parameters have too complicated a loss surface to be learned in a supervised manner.

- Only mild forms of supervision desirable (e.g., tuning σ or mildly supervised prototype collection).

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Restricted Boltzmann Machines

Neural Networks and Deep Learning, Springer, 2018
Chapter 6

Restricted Boltzmann Machines

- Most of the neural architectures map inputs to outputs.
  - Ideal for supervised models.
  - Autoencoders can be used for unsupervised models by replicating the output.

- Restricted Boltzmann machines are borrowed from probabilistic graphical models.
  - Graph of probabilistic dependencies between *binary* states that are *outcomes* of distributions.
  - Binary training data provides some examples of states.
  - Ideal for unsupervised models.

Key Differences from Conventional Neural Networks

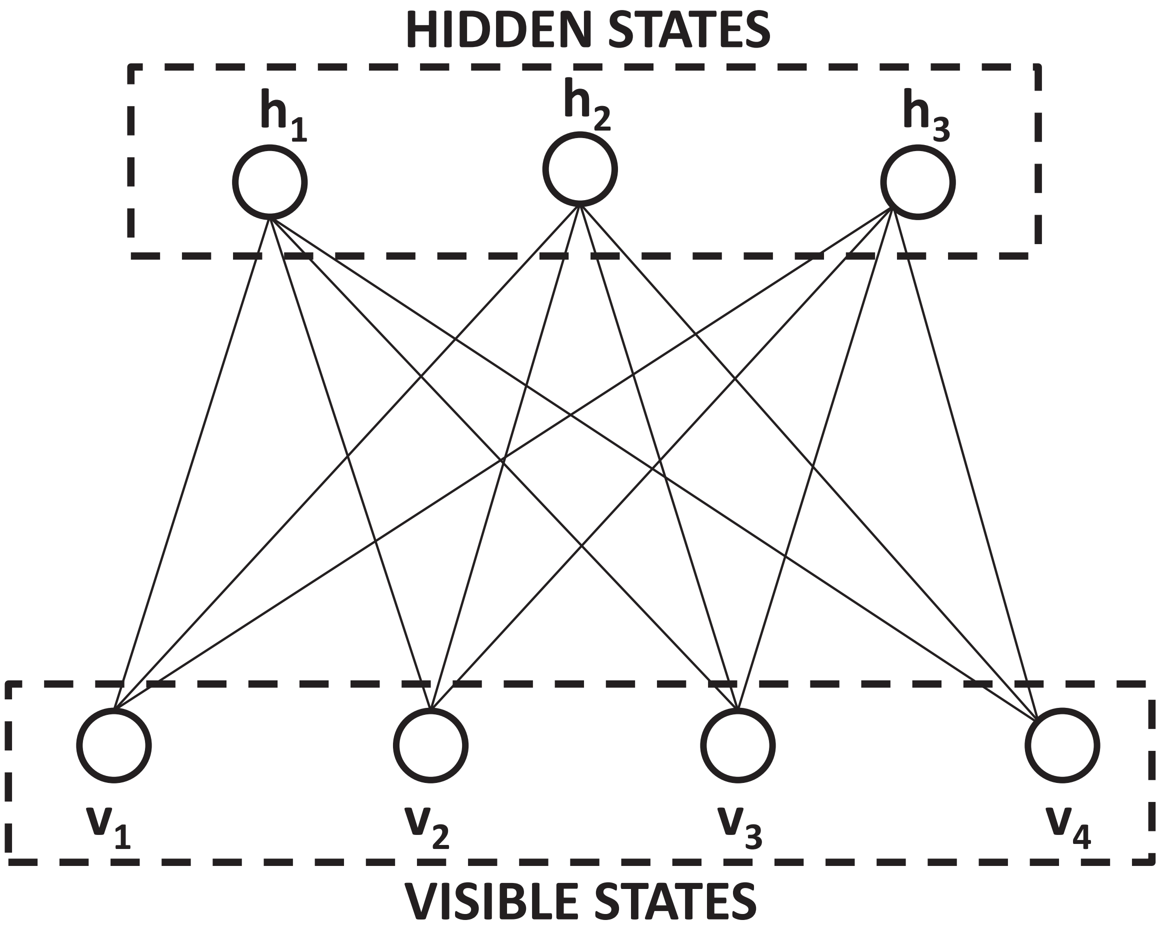
- No input to output mapping
- States are *discrete samples* of probability distributions with interdependencies among samples.
- Training points provide examples of *some* (visible) states.
- **Computational graph abstraction:** The parameterized edges define dependencies among states.
  - The computational graph abstraction is main commonality (can be exploited for pre-training)
  - Can approximately convert a sampling-based dependency to real-valued operation for initialization of a related (conventional) neural network

Historical Significance

- Most of the practical applications in neural networks use supervised learning.

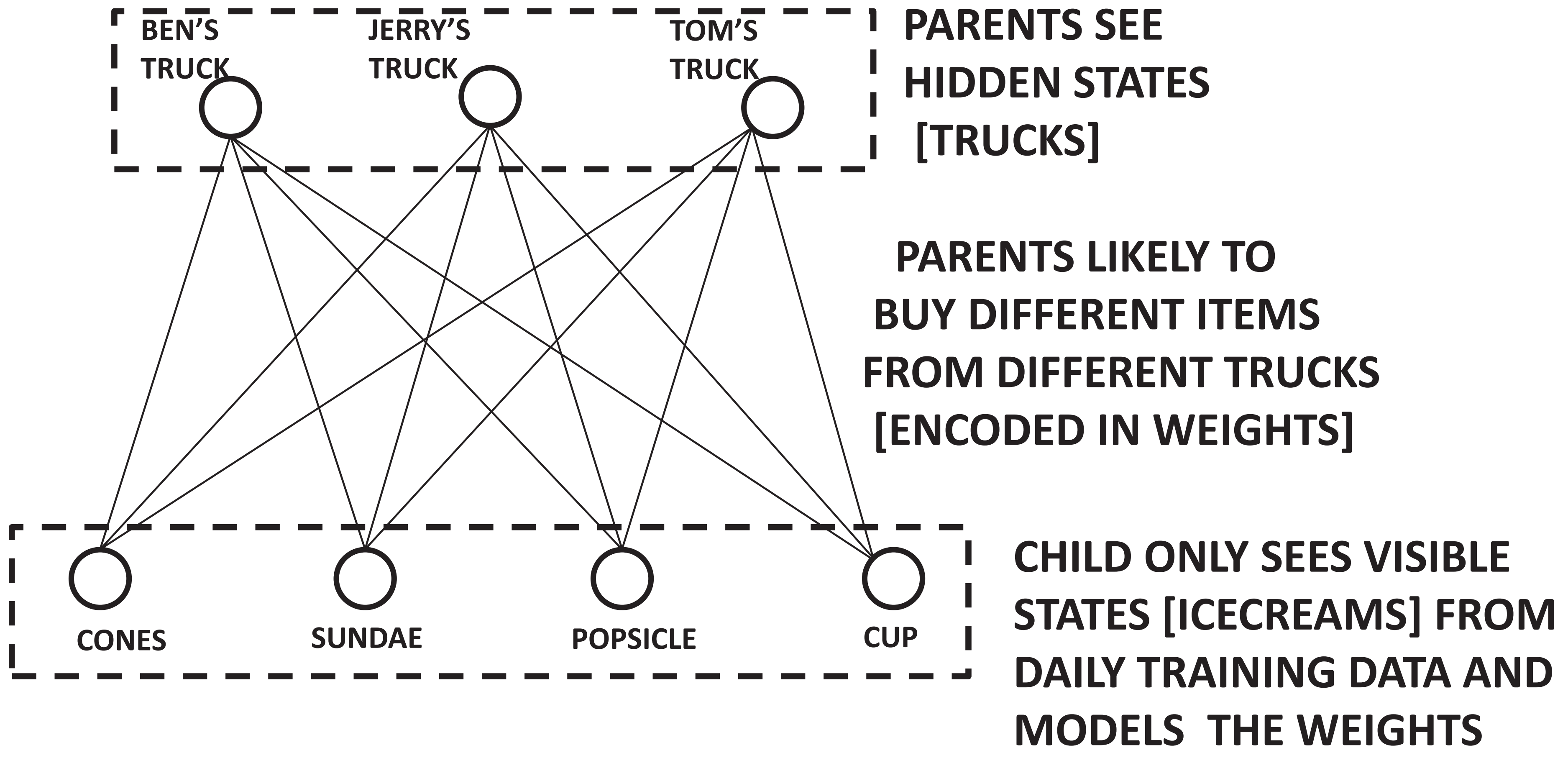
- RBMs can still be used for unsupervised pre-training of conventional neural networks and also extended to supervised learning.
  - Replace binary state outcomes with fractional probabilities
  - Treat fractional values as the activations of a conventional neural network
  - Pretraining owes its historical origins to RBMs

Defining a Restricted Boltzmann Machine



- *Bipartite* graph of *binary* hidden states and visible states connected by undirected edges signifying probabilistic dependencies $\Rightarrow$ Origin of word "*restricted*" from bipartite model

An Interpretable Boltzmann Machine


BEN'S TRUCK
JERRY'S TRUCK
TOM'S TRUCK
PARENTS SEE HIDDEN STATES [TRUCKS]
PARENTS LIKELY TO BUY DIFFERENT ITEMS FROM DIFFERENT TRUCKS [ENCODED IN WEIGHTS]
CONES
SUNDAE
POPSICLE
CUP
CHILD ONLY SEES VISIBLE STATES [ICECREAMS] FROM DAILY TRAINING DATA AND MODELS THE WEIGHTS

- Undirected model $\Rightarrow$ Probability to buy icecreams and pick trucks depend on one another (using weights)

What Kind of Model does a Restricted Boltzmann Machine Build?

- Probability distributions of the *binary* hidden and visible states depend on one another.
  - Weights on edges control probabilistic dependencies.
  - Training data assumed to be samples of visible states.
- We want to learn weights that are “consistent” with training samples.
- Use energy function to force “consistency” $\Rightarrow$ Unsupervised model.
- The model can use learned weights to output samples that are consistent with training data $\Rightarrow$ Generative model.

Notations

- We assume that the *binary* hidden units are $h_1 \dots h_m$ and the visible units are $v_1 \dots v_d$.

- The bias associated with the visible node v_i be denoted by $b_i^{(v)}$.

- The bias associated with hidden node h_j is denoted by $b_j^{(h)}$.

- The weight of the edge between visible node v_i and hidden node h_j is denoted by w_{ij}.

- Can be generalized to non-binary data with some work.

Probabilistic Relationships

- Want to learn weights w_{ij} so that samples of the training data are most "consistent" with the following relationships:

$$P(h_j = 1|\overline{v}) = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{1}$$

$$P(v_i = 1|\overline{h}) = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_{j=1}^{m} h_j w_{ij})} \tag{2}$$

- Use energy function to force consistency by minimizing expected value of $E = -\sum_i b_i^{(v)} v_i - \sum_j b_j^{(h)} h_j - \sum_{i,j:i<j} w_{ij} v_i h_j$

How Data is Generated from a Boltzmann Machine

- Data is generated by using *Gibb's sampling*.
- Randomly initialize visible states and then sample hidden states using Equation 1 (previous slide).
- Alternately sample hidden states and visible states using Equations 1 and 2 until thermal equilibrium is reached.
- A particular set of visible states at thermal equilibrium provides a sample of a binary training vector.
- The weights implicitly encode the distribution by defining probabilistic dependencies.

Intuition for Weights

- Consider weights like affinities $\Rightarrow$ Large positive values of w_{ij} imply that states will be "on " together.

- We already have samples showing which visible states are "on" together.

- Weights will be learned in such a way that hidden states will be connected to correlated visible states with large weights.
  - **Biological motivation:** In Hebbian learning, a synapse between two neurons is strengthened when the neurons on either side of the synapse have highly correlated outputs.
  - Contrastive divergence algorithm learns weights.

Overview of Contrastive Divergence

- **Positive phase:** Draw b instances of hidden states based on visible states fixed to each of a mini-batch of b training points $\Rightarrow$ Yields $\langle v_i, h_j \rangle_{pos}$

- **Negative phase:** For each of the b instances in positive phase *continue to* alternately sample visible states and hidden states from one another for r iterations $\Rightarrow$ Yields $\langle v_i, h_j \rangle_{neg}$

$$w_{ij} \Leftarrow w_{ij} + \alpha \left(\langle v_i, h_j \rangle_{pos} - \langle v_i, h_j \rangle_{neg} \right)$$
$$b_i^{(v)} \Leftarrow b_i^{(v)} + \alpha \left(\langle v_i, 1 \rangle_{pos} - \langle v_i, 1 \rangle_{neg} \right)$$
$$b_j^{(h)} \Leftarrow b_j^{(h)} + \alpha \left(\langle 1, h_j \rangle_{pos} - \langle 1, h_j \rangle_{neg} \right)$$

Remarks on Contrastive Divergence

- Strictly speaking, the negative phase needs a very large number of iterations to reach *thermal equilibrium* in negative phase.

- Positive phase requires only one iteration because visible states are fixed to training points.

- Contrastive divergence says that only a small number of iterations of negative phase are sufficient for "good" update of weight vector (even without thermal equilibrium).

- In the early phases of training, one iteration is enough for "good" update.

- Can increase number of iterations in later phases.

Utility of Unsupervised Learning

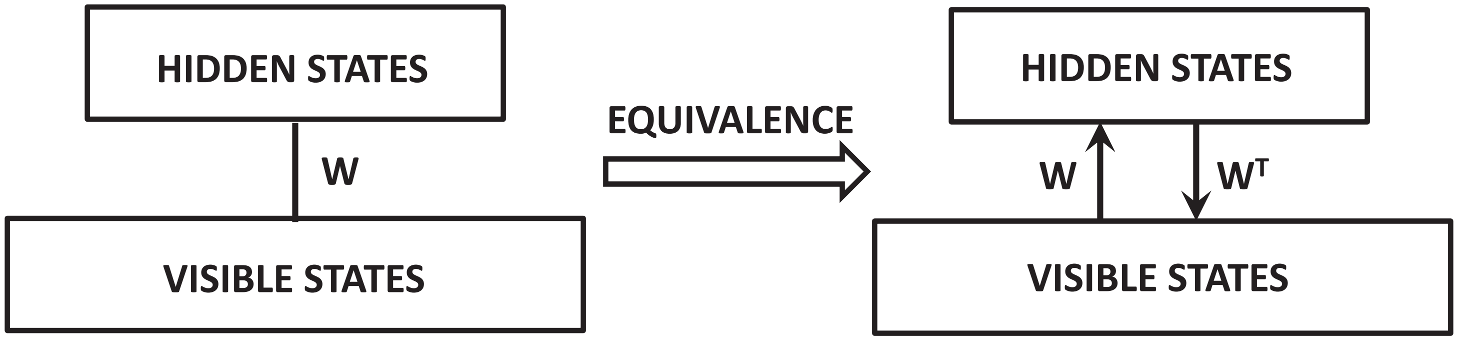
- One can use an RBM to initialize an autoencoder for binary data (later slides).

- Treat the sigmoid-based sampling as an a sigmoid activation.

- Basic idea can be extended to multilayer neural networks by using *stacked RBMs*.

  - One of the earliest methods for pretraining.

Equivalence of Directed and Undirected Models

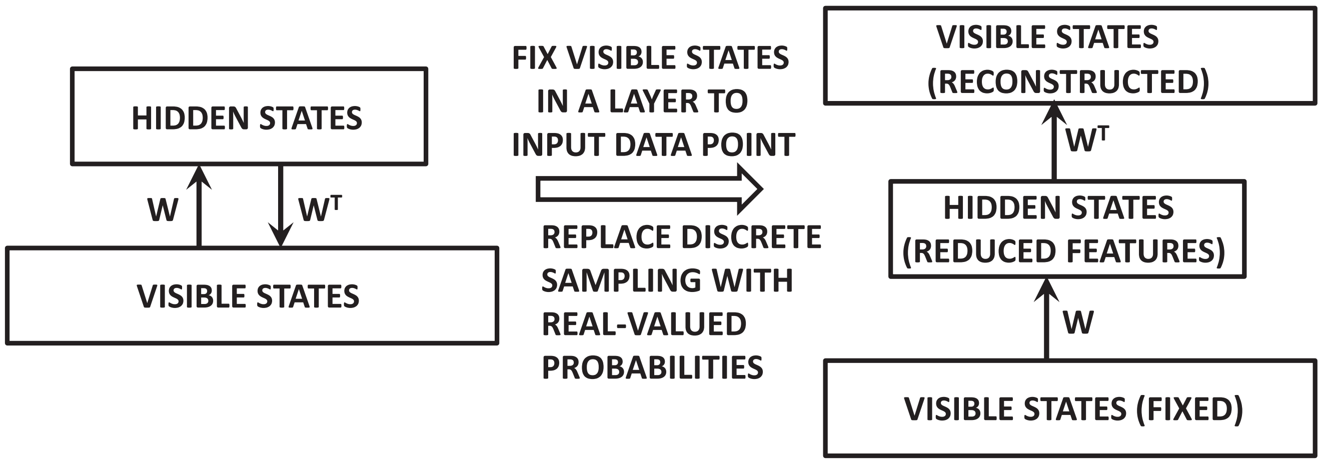


- Replace undirected edges with directed edges

$$\overline{h} \sim \text{Sigmoid}(\overline{v}, \overline{b}^{(h)}, W)$$
$$\overline{v} \sim \text{Sigmoid}(\overline{h}, \overline{b}^{(v)}, W^T)$$

- Replace sampling with real-valued operations

Using a Trained RBM to Initialize a Conventional Autoencoder


HIDDEN STATES
W
W^T
VISIBLE STATES
FIX VISIBLE STATES IN A LAYER TO INPUT DATA POINT
REPLACE DISCRETE SAMPLING WITH REAL-VALUED PROBABILITIES
VISIBLE STATES (RECONSTRUCTED)
W^T
HIDDEN STATES (REDUCED FEATURES)
W
VISIBLE STATES (FIXED)

- Architecture on right uses *real-valued* sigmoid operations rather than discrete operations $\Rightarrow$ Conventional autoencoder!.

$$\hat{h}_j = \frac{1}{1 + \exp(-b_j^{(h)} - \sum_{i=1}^{d} v_i w_{ij})} \tag{3}$$

$$\hat{v}_i = \frac{1}{1 + \exp(-b_i^{(v)} - \sum_{j=1}^{m} \hat{h}_j w_{ij})} \tag{4}$$

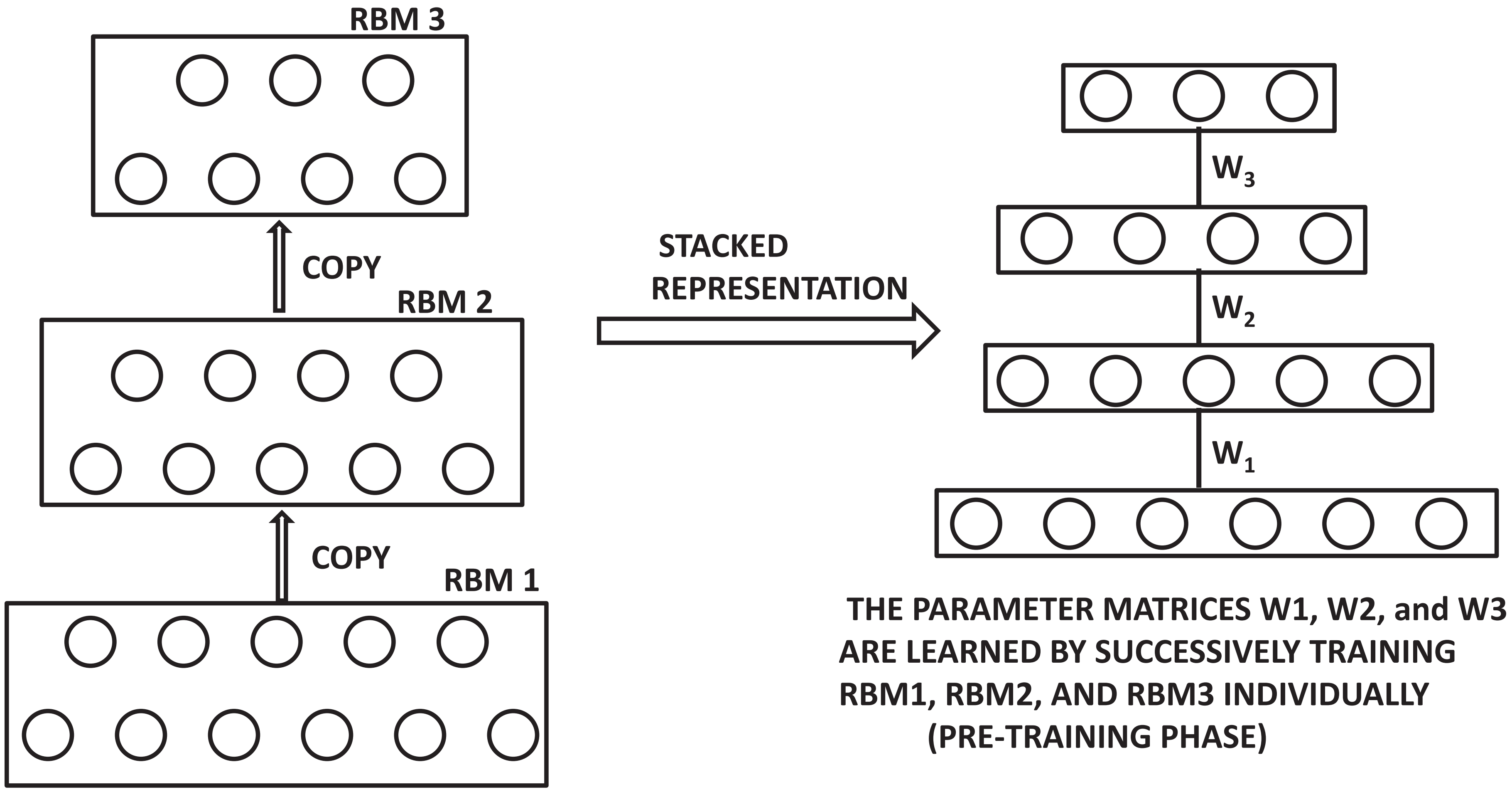
Why Use an RBM to Initialize a Conventional Neural Network?

- In the early years, conventional neural networks did not train well (especially with increased depth).
  - Vanishing and exploding gradient problems
  - RBM trains with contrastive divergence (no vanishing and exploding gradient)

- Real-valued approximation was used with stacked RBMs to initialize deep networks

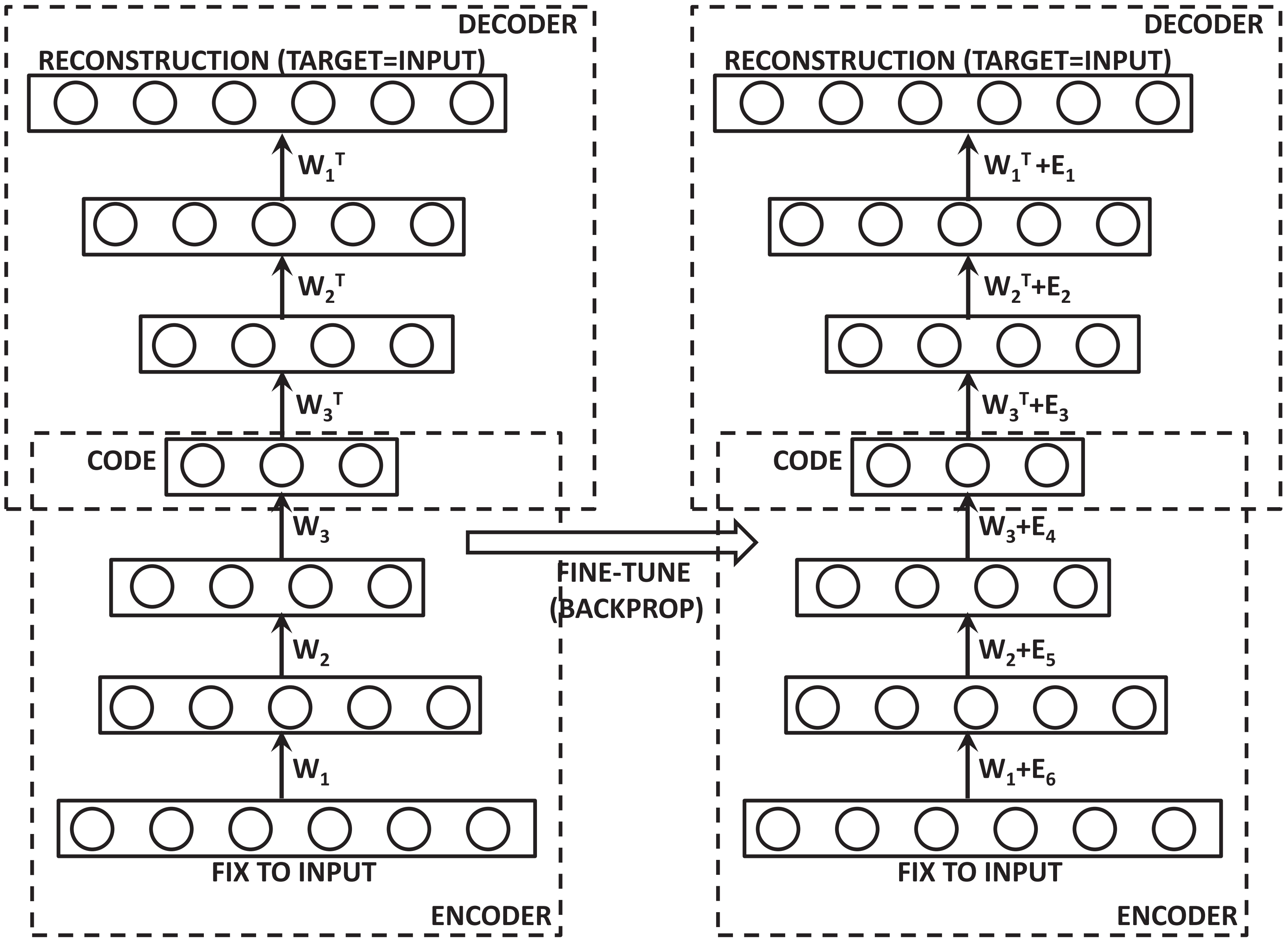
- Approach was generalized to conventional autoencoders later

Stacked RBM


RBM 3
COPY
RBM 2
COPY
RBM 1
STACKED REPRESENTATION
W_3
W_2
W_1
THE PARAMETER MATRICES W1, W2, and W3 ARE LEARNED BY SUCCESSIVELY TRAINING RBM1, RBM2, AND RBM3 INDIVIDUALLY (PRE-TRAINING PHASE)

- Train different layers sequentially

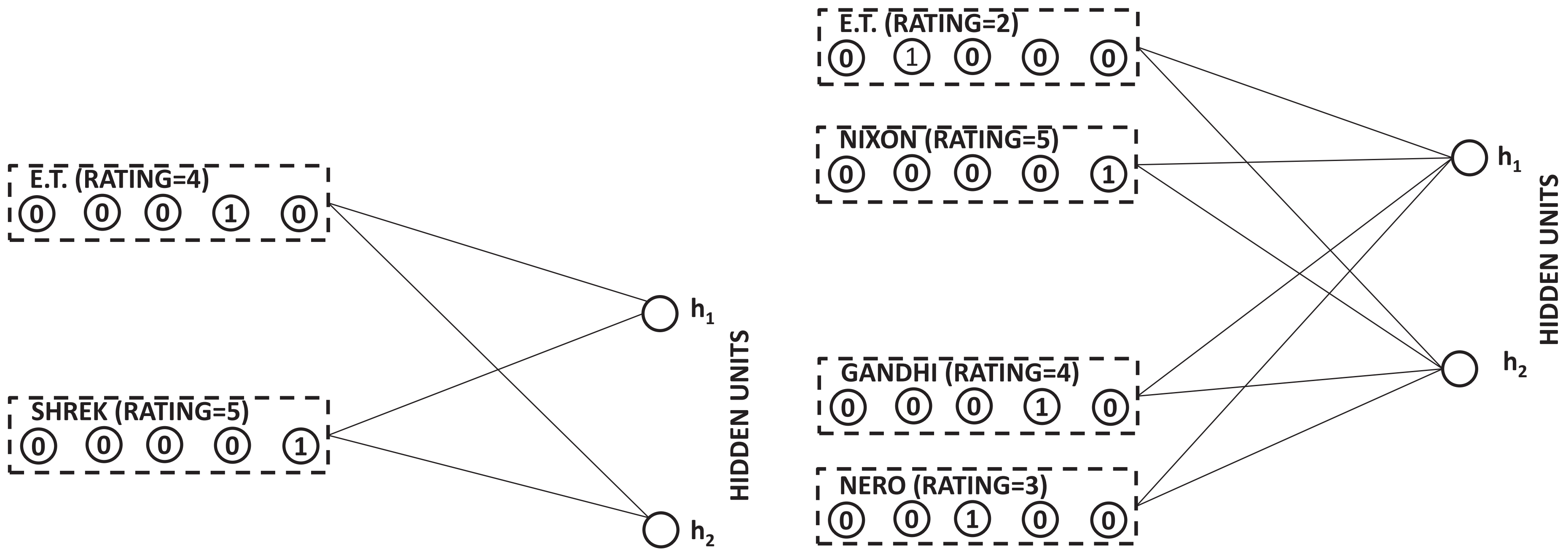
Stacked RBM to Conventional Neural Network



Applications

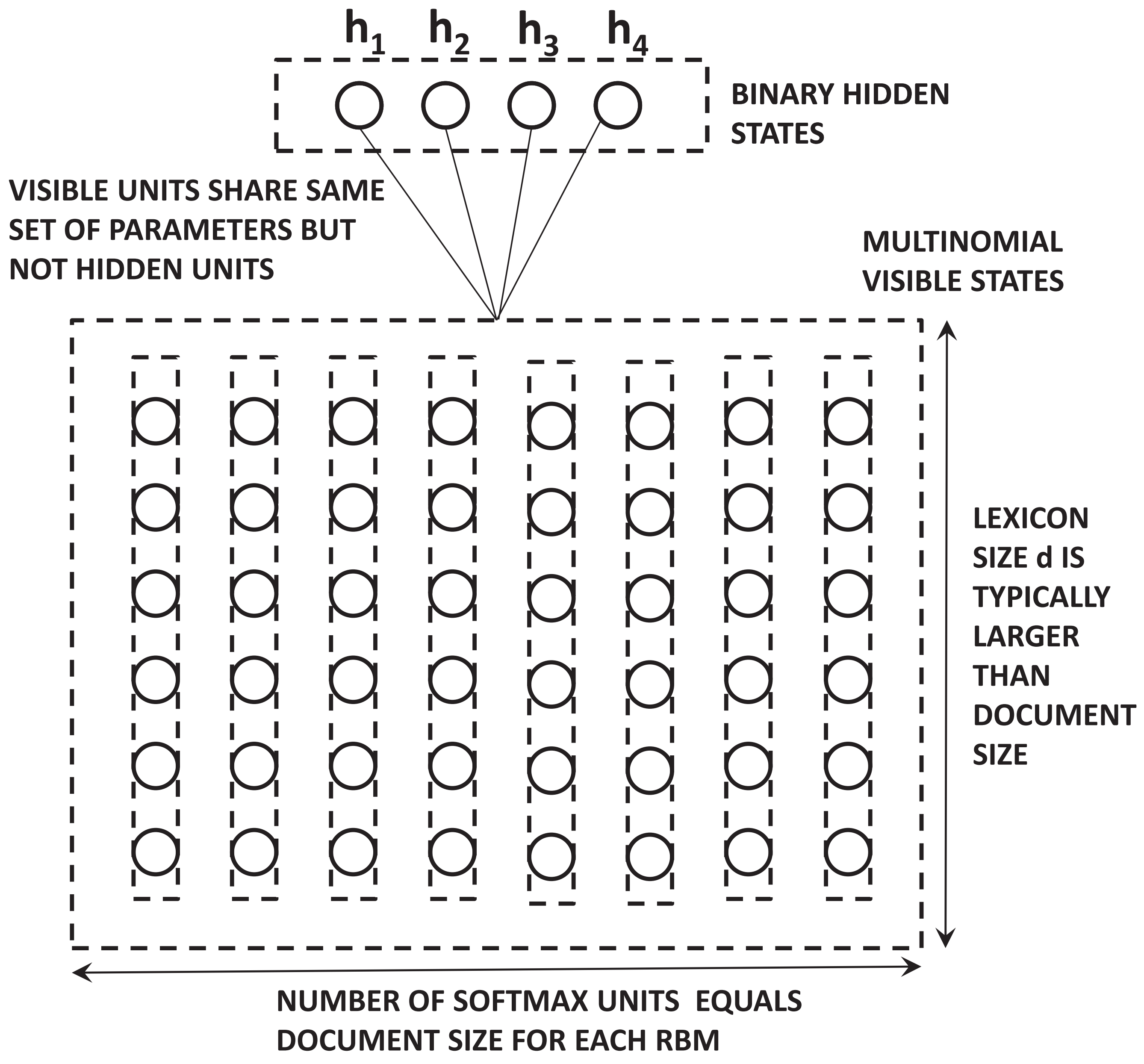
- Pretraining can be used for supervised and unsupervised applications
  - Collaborative filtering: Was a component of Netflix prize contest
    - Gives different results from an autoencoder-like architecture in an earlier lecture
  - Topic models
  - Classification

Collaborative Filtering



- Changes for softmax activations and shared weights across RBMs

Topic Models

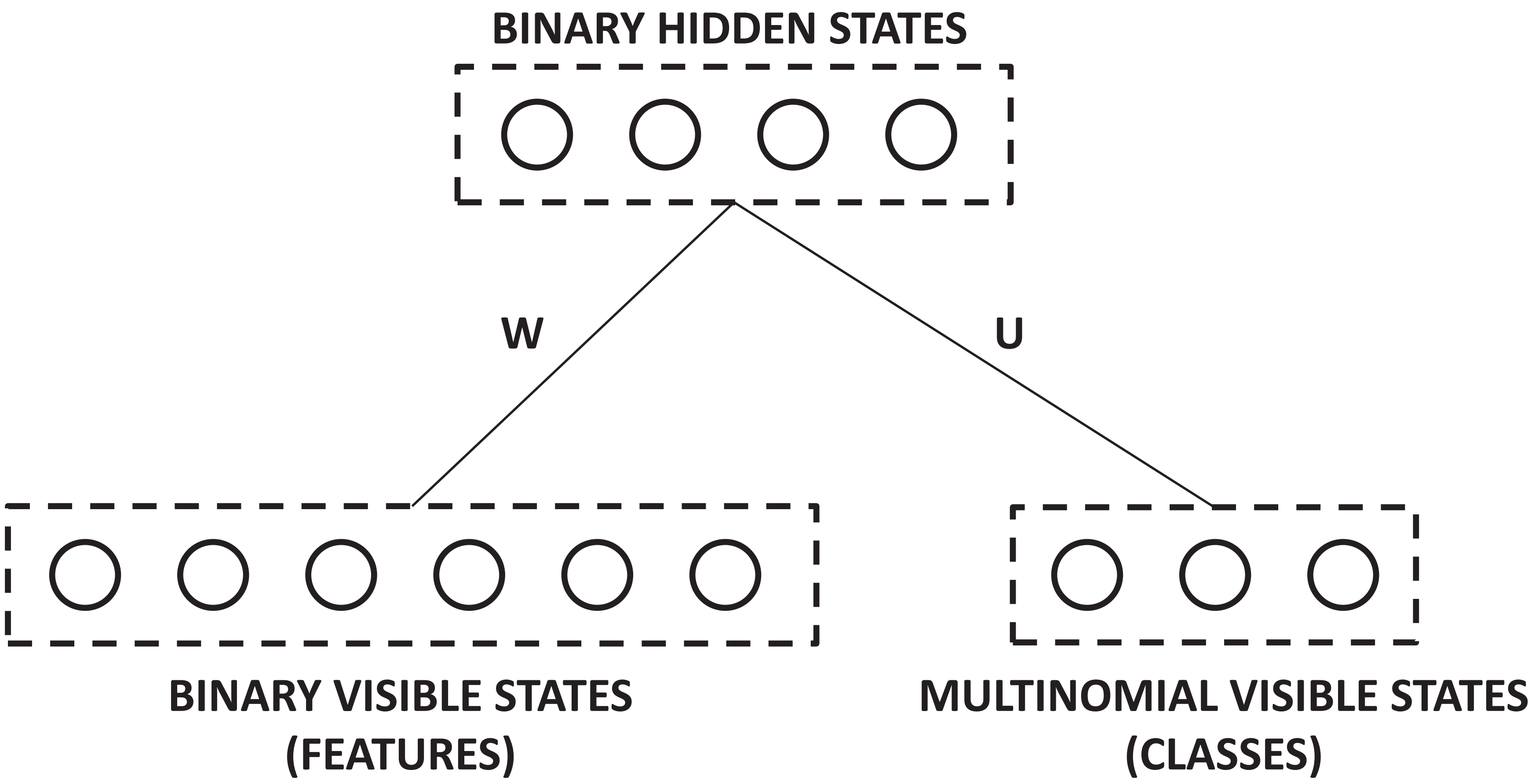

h_1 h_2 h_3 h_4
BINARY HIDDEN STATES
VISIBLE UNITS SHARE SAME SET OF PARAMETERS BUT NOT HIDDEN UNITS
MULTINOMIAL VISIBLE STATES
LEXICON SIZE d IS TYPICALLY LARGER THAN DOCUMENT SIZE
NUMBER OF SOFTMAX UNITS EQUALS DOCUMENT SIZE FOR EACH RBM

Classification

- Can be used for unsupervised pretraining for classification
  - Goal of RBM is only to learn features in unsupervised way
  - Class label does not get a state in RBM

- Can also be used to train by treating a class label as a state.
  - Hidden features are connected to both class variables and feature variables.
  - *Generative* approach of RBMs does not fully optimize for classification accuracy $\Rightarrow$ Need *discriminative* Boltzmann Machines (Larochelle *et al*).

Classification Architecture



Comments

- RBMs represent a special case of probabilistic graphical models.

- Provides an alternative to the autoencoder.

- Can be extended to non-binary data.

- These models are not quite as popular anymore.

- Significant historical significance in starting the idea of pre-training for deep models.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Recurrent Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 7.1–7.2

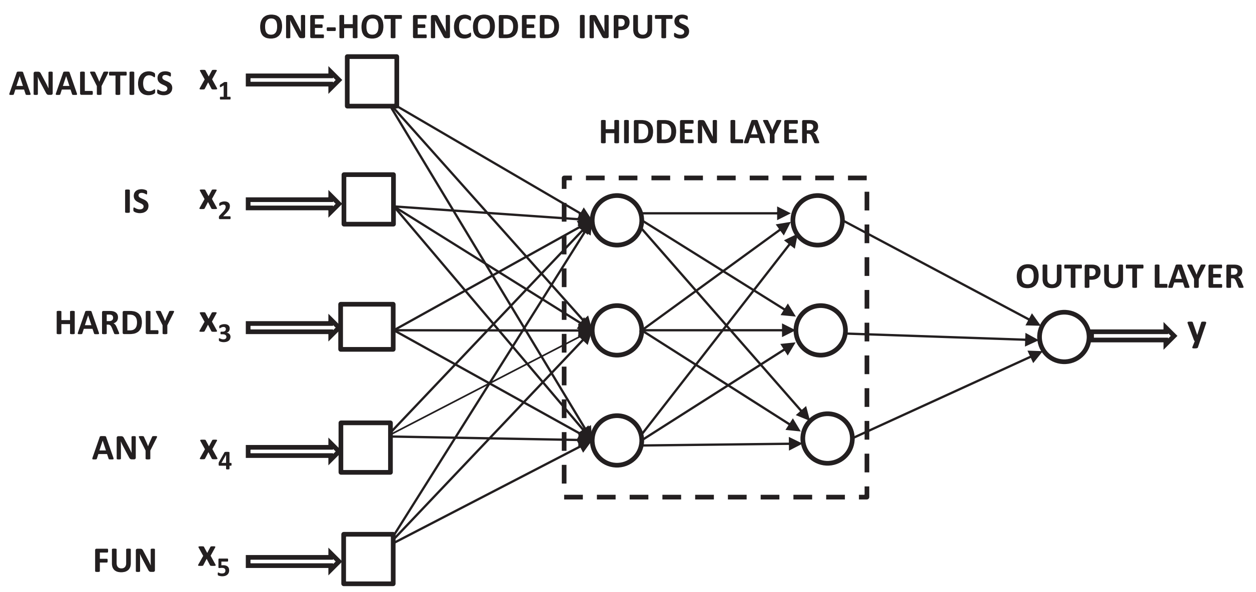
The Challenges of Processing Sequences

- Conventional neural networks have a fixed number of (possibly independent) input dimensions and outputs.

- Sequences often have variable length in which the different items (dimensions) of the sequence are related to one another:
  - Words in a sentence.
  - Values in a time-series
  - Biological sequences

- Recurrent neural networks address such domains.

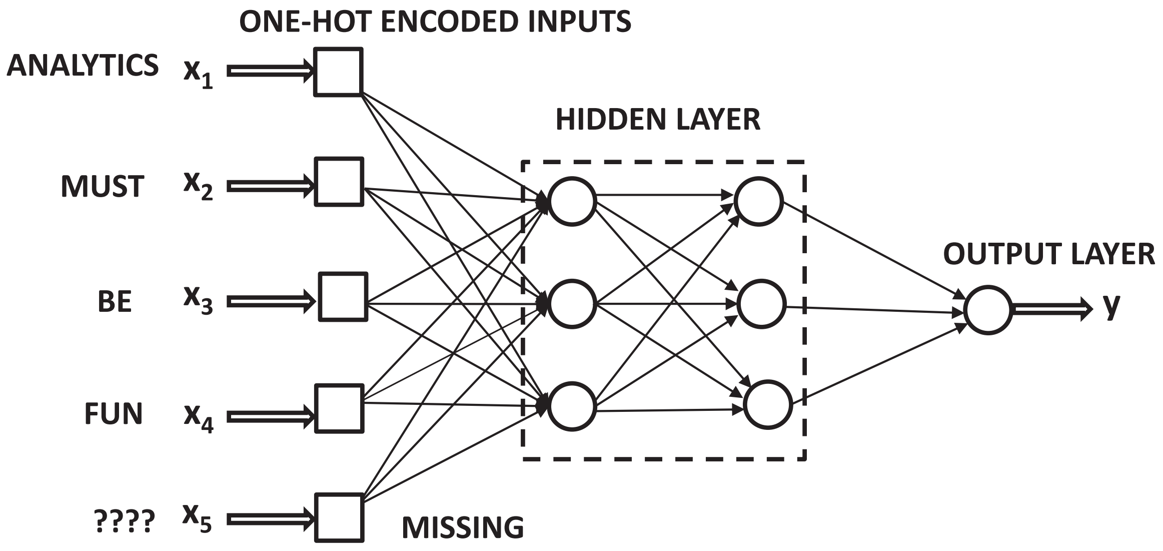
Problems with Conventional Architecture [Sentiment Analysis]



- The architecture above cannot handle sequences of length larger than 5.

- The sequential relationship among input elements is not encoded well.
  - Distinguish "*cat chased mouse*" and "*mouse chased cat*"

Problems with Conventional Architecture

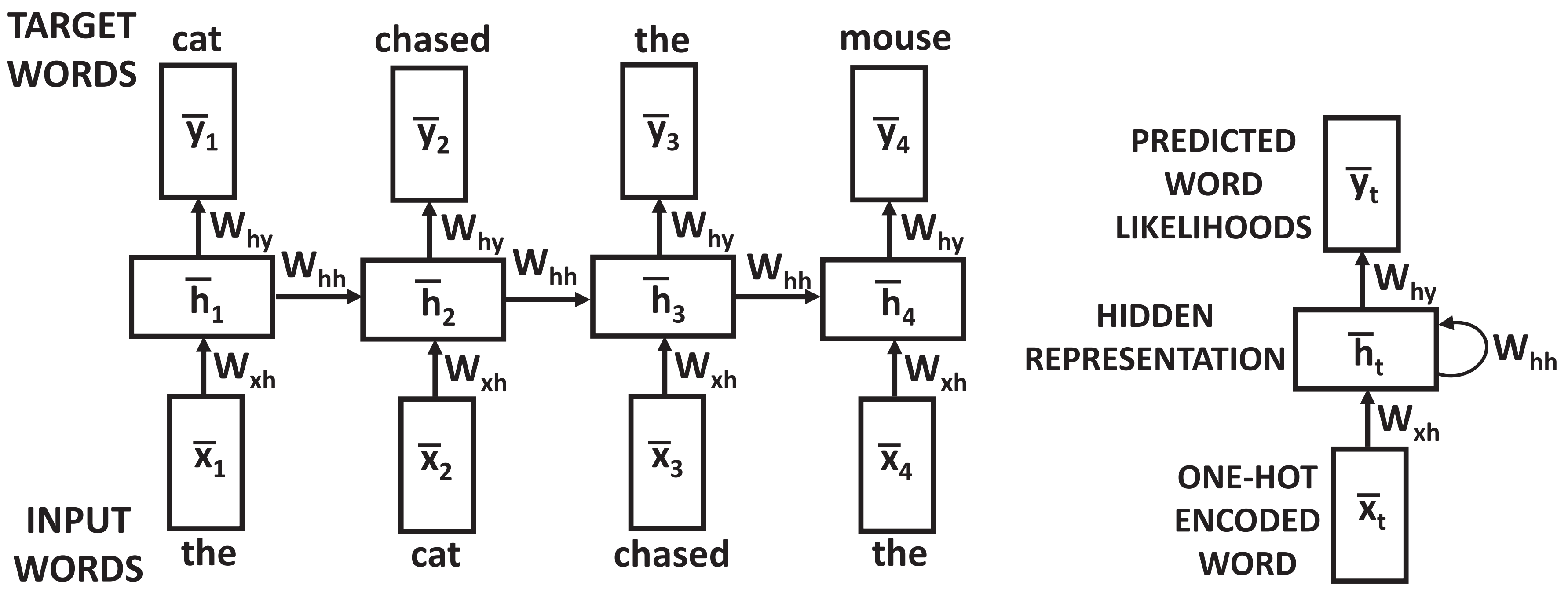


- Small changes in word ordering change sentiment.
- Missing inputs if sequence is too small.
- How do you create an architecture with variable number of inputs but fixed number of parameters?
  - Machine translation has variable number of *outputs*!

Desiderata for Architecture Processing Sequences

- The ith element of the sequence should be fed into the neural network after the network has had a chance to see what has come before it.
  - Can be enforced by feeding ith element of sequence into layer i.
  - *Each layer associated with a time-stamp* (variable number of layers).
- **Markov Property:** Any element of the sequence can be predicted from its preceding elements using a fixed model.
  - Can be (approximately) enforced by using shared weights across layers.
  - Number of weight parameters independent of number of layers!

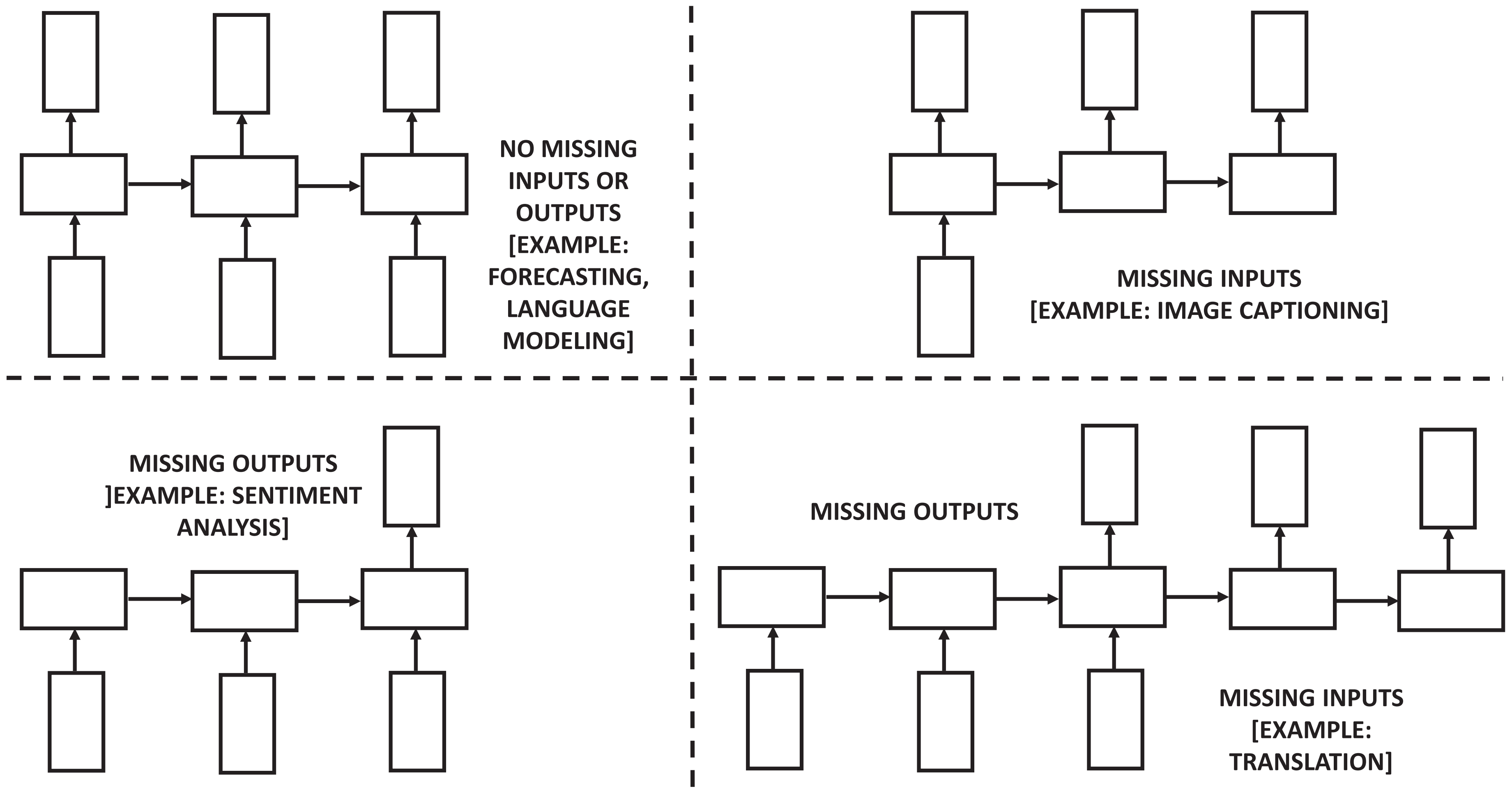
A Time-Layered Recurrent Network [Predicting Next Word]



- Note that the weight matrices W_{xh}, W_{hh}, and W_{hy} are shared across layers (vector architecture).

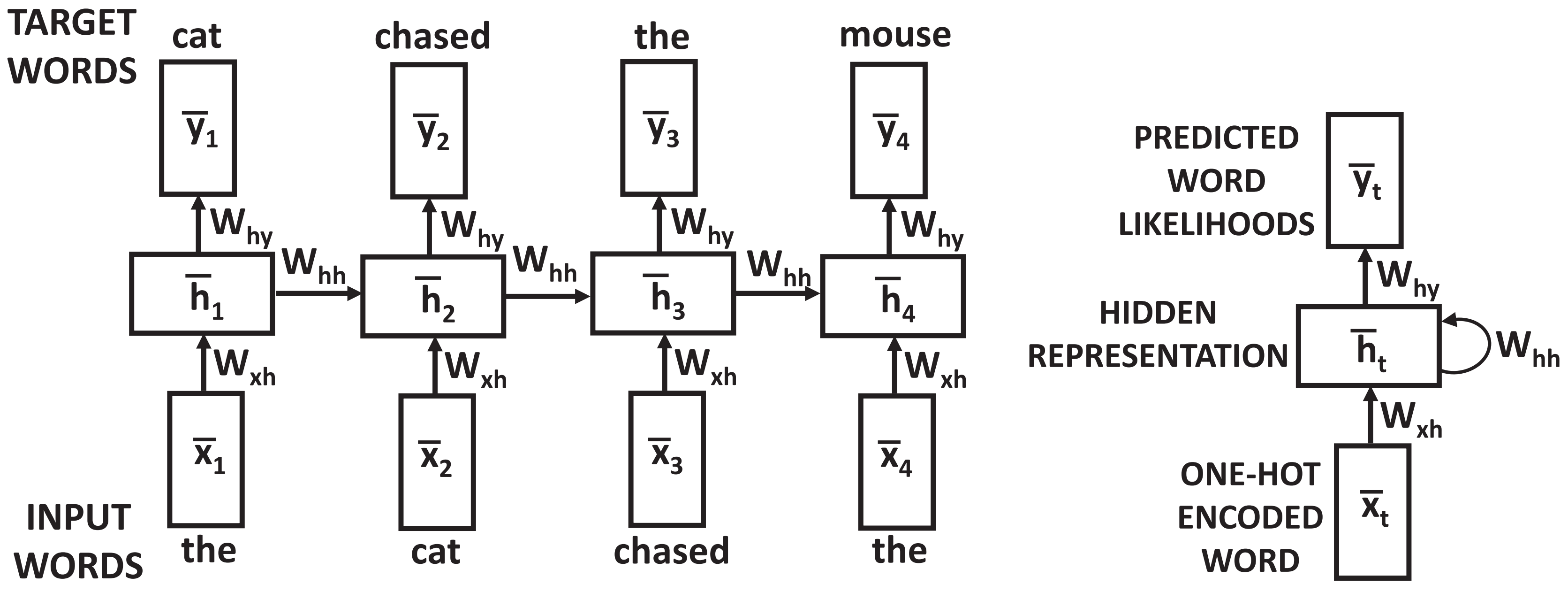
- An input $\overline{x}_t$ directly encounters the hidden state constructed from all inputs before it.

Variations


NO MISSING INPUTS OR OUTPUTS [EXAMPLE: FORECASTING, LANGUAGE MODELING]
MISSING INPUTS [EXAMPLE: IMAGE CAPTIONING]
MISSING OUTPUTS]EXAMPLE: SENTIMENT ANALYSIS]
MISSING OUTPUTS
MISSING INPUTS [EXAMPLE: TRANSLATION]

- One does not need to have inputs and outputs at each time stamp!

Recurrent Network: Basic Computations



- $\overline{h}_t = f(\overline{h}_{t-1}, \overline{x}_t) = \tanh(W_{xh}\overline{x}_t + W_{hh}\overline{h}_{t-1})$ [Typical hidden-to-hidden]

- $\overline{y}_t = W_{hy}\overline{h}_t$ [Real-valued hidden-to-output $\Rightarrow$ Optional and depends on output layer]

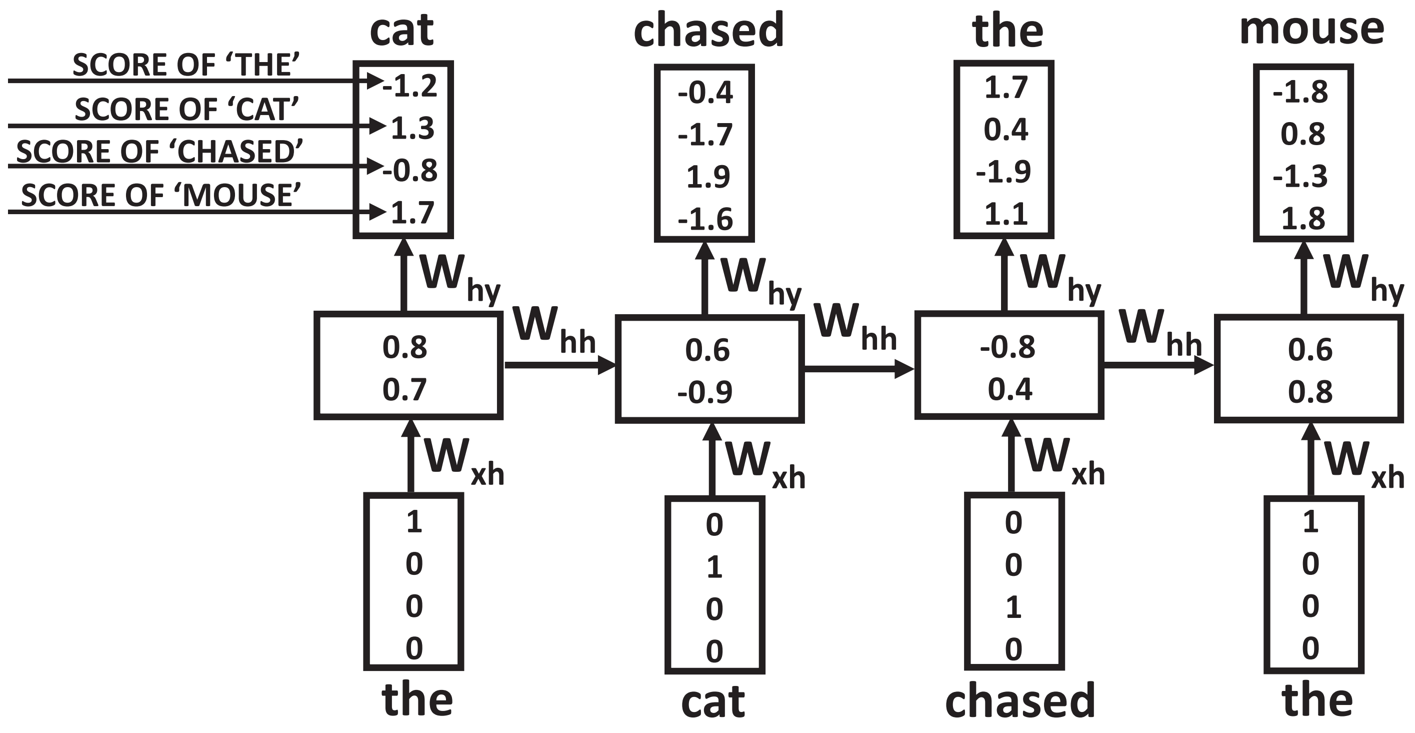
Flexibility for Variable-Length Inputs

- We define the function for $\overline{h}_t$ in terms of t inputs.

- We have $\overline{h}_1 = f(\overline{h}_0, \overline{x}_1)$ and $\overline{h}_2 = f(f(\overline{h}_0, \overline{x}_1), \overline{x}_2)$.
  - The vector $\overline{h}_1$ is a function of only $\overline{x}_1$, whereas $\overline{h}_2$ is a function of both $\overline{x}_1$ and $\overline{x}_2$.

- The vector $\overline{h}_t$ is a function of $\overline{x}_1 \ldots \overline{x}_t$.
  - Can provide an output based on entire history.

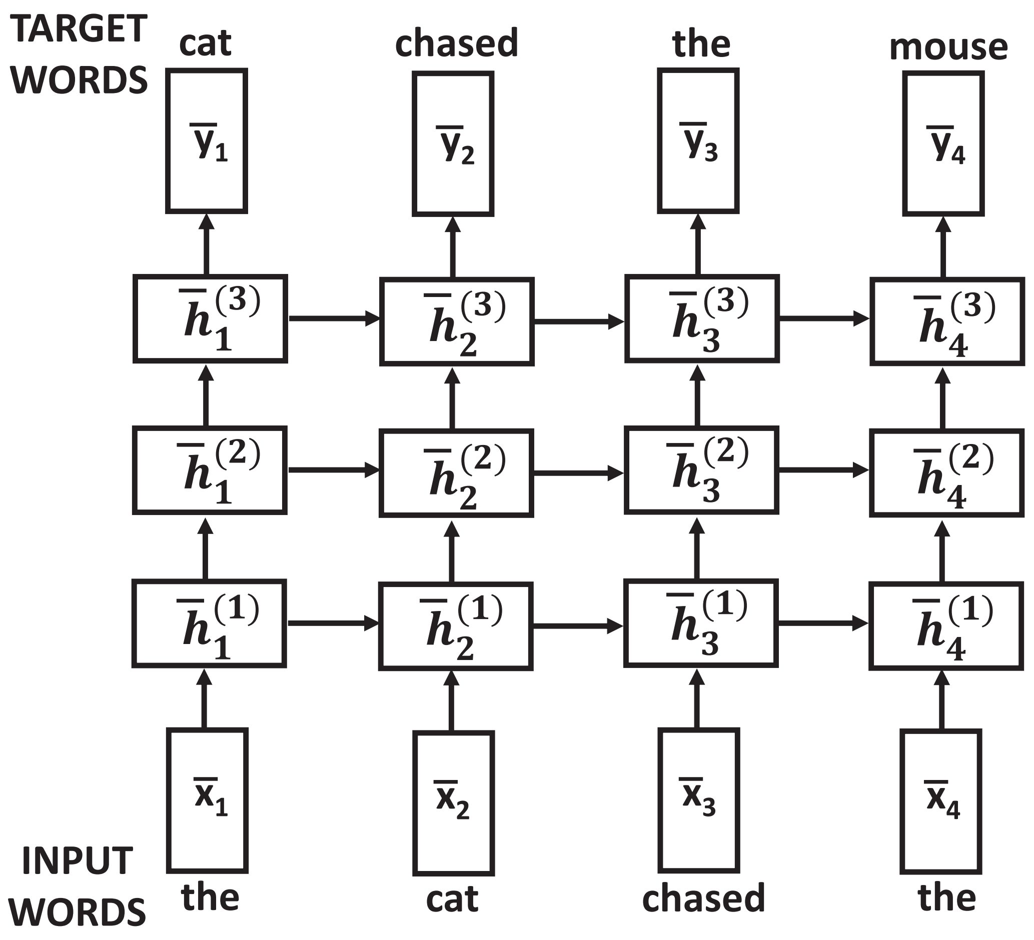
Language Modeling Example: Predicting the Next Word



- Predicting the next word for the sentence "*The cat chased the mouse.*"

- Lexicon of size four.

Multilayer Recurrent Networks



$$\overline{h}_t^{(k)} = \tanh\, W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix}$$

Training a Recurrent Network

- Main difference from traditional backpropagation is the issue of shared parameters:
  - Pretend that the weights are not shared and apply normal backpropagation to compute the gradients with respect to each copy of the weights.
  - Add up the gradients over the various copies of each weight.
  - Perform the gradient-descent update.
- Algorithm is referred to as "*backpropagation through time.*"

Truncated Backpropagation through Time

- The number of layers in a recurrent network depends on the the length of the sequence.

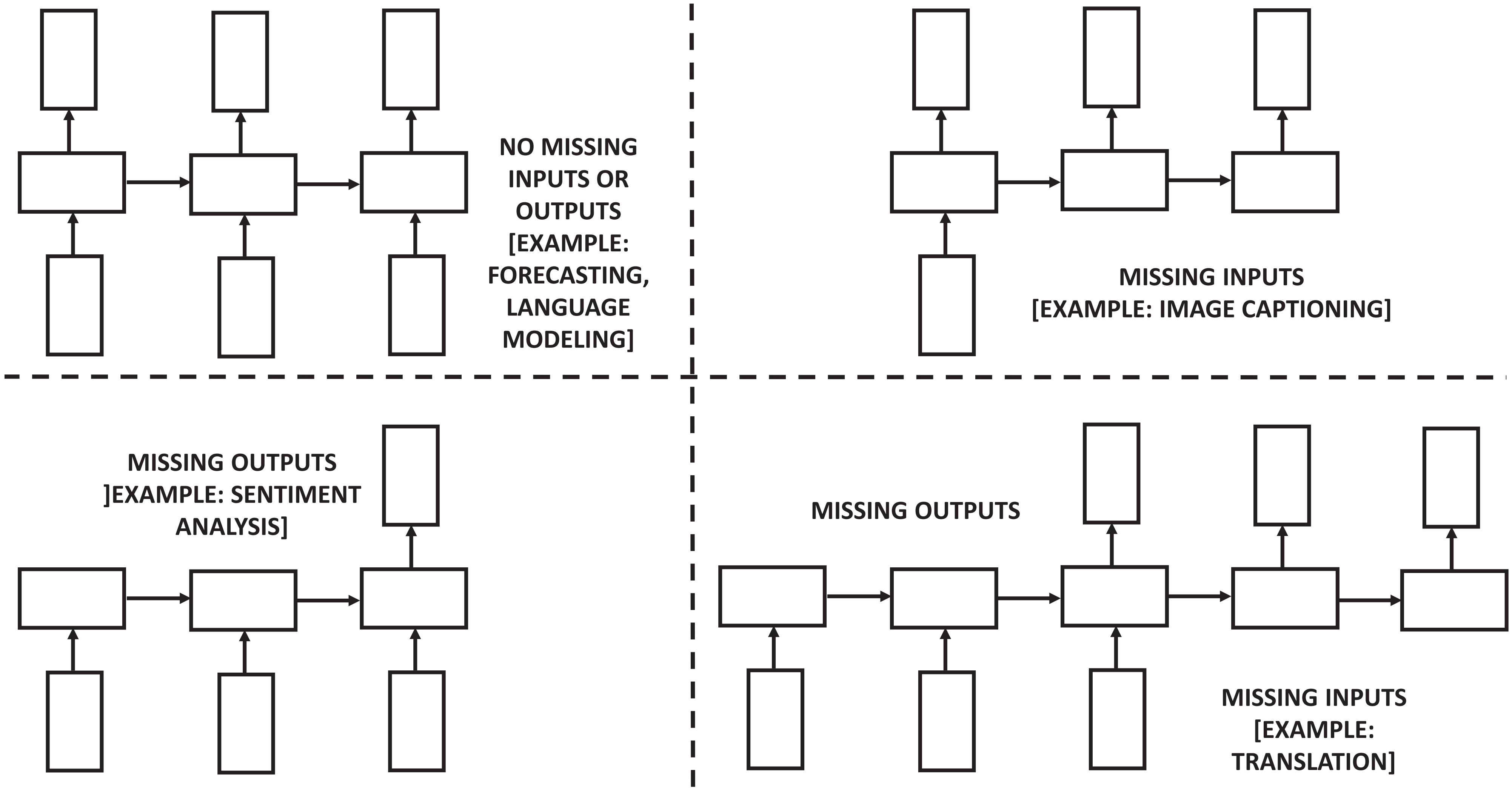
- Causes problems in memory, speed, and convergence.
  - Process chunk by chunk (around 100 sequence elements/layers).
  - Compute forward states exactly using the final state of previous chunk.
  - Compute loss backpropagation only over current chunk and update parameters $\Rightarrow$ Analogous to stochastic gradient descent.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Applications of Recurrent Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 7.7

Recurrent Neural Network Applications are Architecture Sensitive!


NO MISSING INPUTS OR OUTPUTS [EXAMPLE: FORECASTING, LANGUAGE MODELING]
MISSING INPUTS [EXAMPLE: IMAGE CAPTIONING]
MISSING OUTPUTS]EXAMPLE: SENTIMENT ANALYSIS]
MISSING OUTPUTS
MISSING INPUTS [EXAMPLE: TRANSLATION]

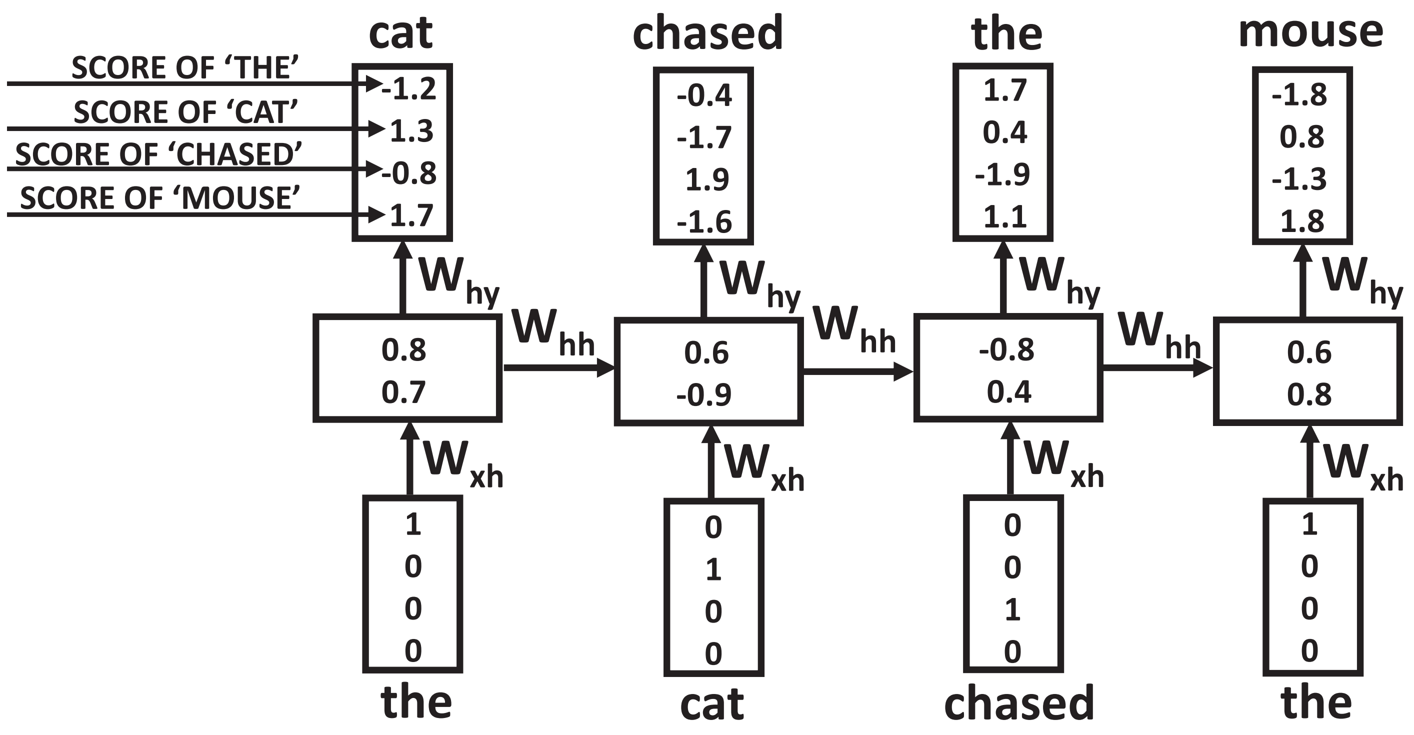
Missing Inputs

- Missing inputs represent a bigger challenge than missing outputs.
  - Missing inputs often occur at inference when output is sequence
  - Language modeling/Image captioning/machine translation
  - How to generate the second word in caption without knowing the first?
- Simply setting missing inputs to random values will result in bias
- Missing inputs are sequentially sampled from outputs at inference time [discussed a few slides later]

Observations

- Most of the applications use advanced variants like LSTMs and bidirectional recurrent networks.
  - Advanced variants covered in later lectures.
- Describe general principles using a simple recurrent network.
- Principles generalize easily to any type of architecture.

Language Modeling: Predicting the Next Word



- Predicting the next word for the sentence "*The cat chased the mouse*."

- Can be used for time-series prediction.

Generating a Language Sample

- Predicting next word is straightforward (all inputs available).

- Predicting a language sample runs into the problem of missing inputs.
  - Sample a token generated at each time-stamp, and input to next time-stamp.
  - To improve accuracy, use beam search to expand on the most likely possibilities by always keeping track of the b best sequence prefixes of any particular length.
  - One can generate an arbitrary sequence of text that reflects the training data set.

Tiny Shakespeare Character-Level RNN: Karpathy, Johnson, Fei-Fei

- Executed code from `https://github.com/karpathy/char-rnn`

- After 5 iterations:

  KING RICHARD II:
  Do cantant,-'for neight here be with hand her,-
  Eptar the home that Valy is thee.

  NORONCES:
  Most ma-wrow, let himself my hispeasures;
  An exmorbackion, gault, do we to do you comforr,
  Laughter's leave: mire sucintracce shall have theref-Helt.

Tiny Shakespeare Character-Level RNN: Karpathy, Johnson, Fei-Fei

- After 50 iterations:

  KING RICHARD II:
  Though they good extremit if you damed;
  Made it all their fripts and look of love;
  Prince of forces to uncertained in conserve
  To thou his power kindless. A brives my knees
  In penitence and till away with redoom.

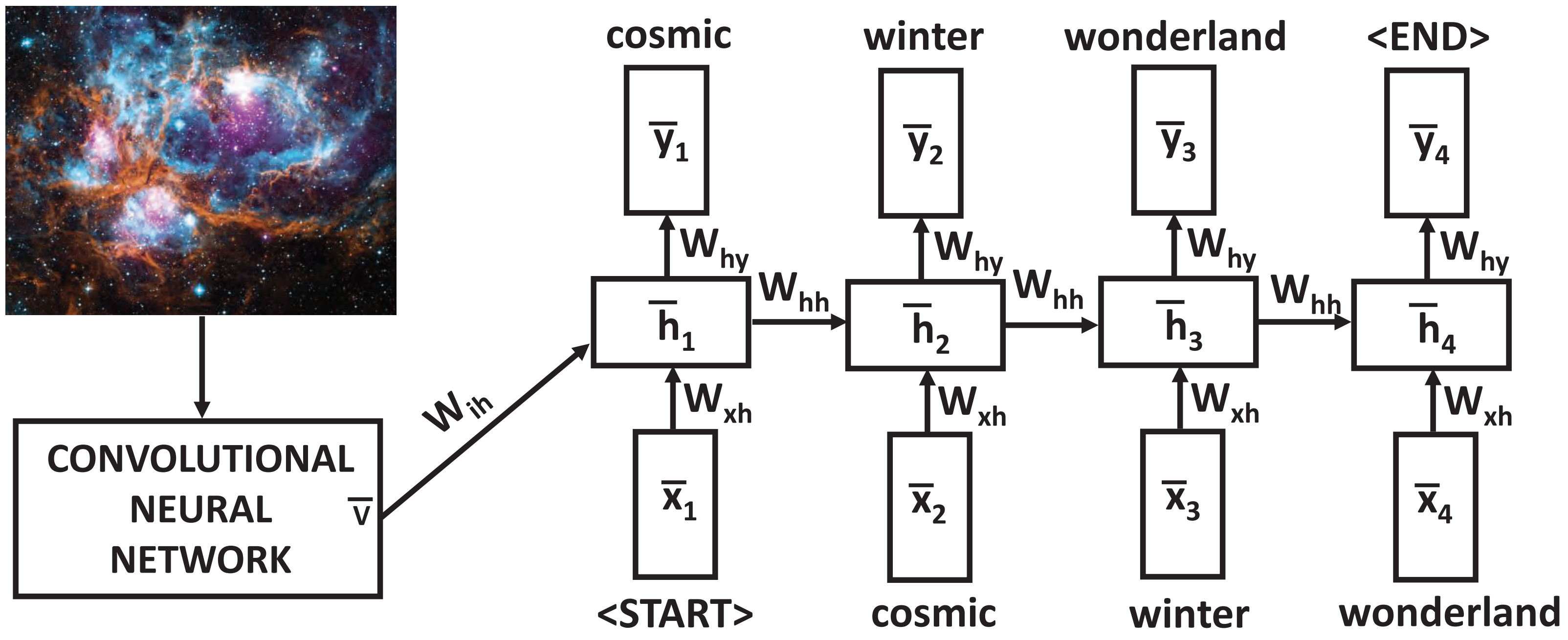
  GLOUCESTER:
  Between I must abide.

What Good is Language Modeling?

- Generating a language sample might seem like an exercise in futility.
  - Samples are syntactically correct but not semantically meaningful.

- Samples are often useful when conditioned on some other data:
  - **Conditioning on an image:** Image captioning
  - **Conditioning on a sequence:** Machine translation and sequence-to-sequence learning

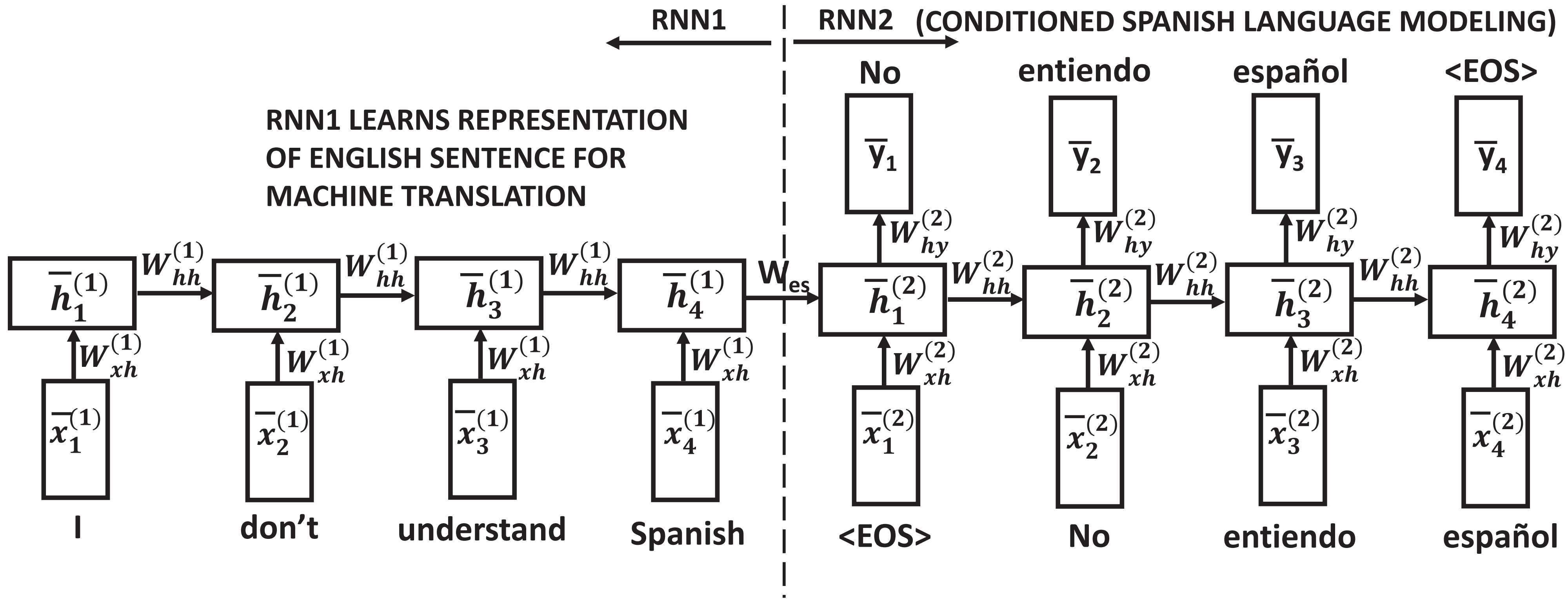
Image Captioning



- Language sample conditioned on an image $\Rightarrow$ Provides meaningful descriptions of images

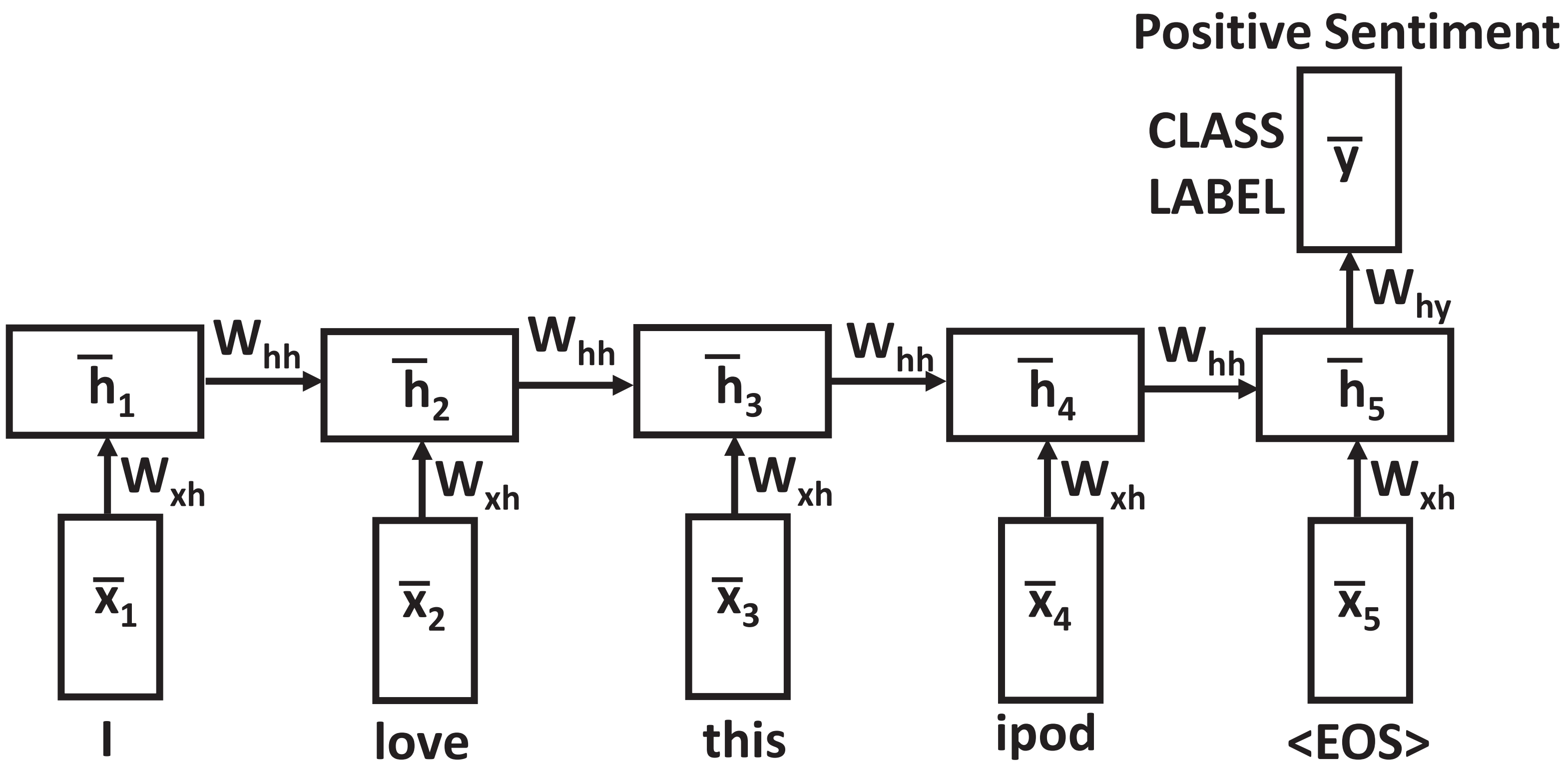
- Need START and END tokens

Machine Translation and Sequence-to-Sequence Learning


RNN1
RNN2 (CONDITIONED SPANISH LANGUAGE MODELING)
RNN1 LEARNS REPRESENTATION OF ENGLISH SENTENCE FOR MACHINE TRANSLATION
No
entiendo
español
<EOS>
$\overline{y}_1$ $\overline{y}_2$ $\overline{y}_3$ $\overline{y}_4$
$W_{hy}^{(2)}$
$\overline{h}_1^{(1)}$ $\overline{h}_2^{(1)}$ $\overline{h}_3^{(1)}$ $\overline{h}_4^{(1)}$
$W_{hh}^{(1)}$
W_{es}
$\overline{h}_1^{(2)}$ $\overline{h}_2^{(2)}$ $\overline{h}_3^{(2)}$ $\overline{h}_4^{(2)}$
$W_{hh}^{(2)}$
$W_{xh}^{(1)}$
$W_{xh}^{(2)}$
$\overline{x}_1^{(1)}$ $\overline{x}_2^{(1)}$ $\overline{x}_3^{(1)}$ $\overline{x}_4^{(1)}$
$\overline{x}_1^{(2)}$ $\overline{x}_2^{(2)}$ $\overline{x}_3^{(2)}$ $\overline{x}_4^{(2)}$
I
don't
understand
Spanish
<EOS>
No
entiendo
español

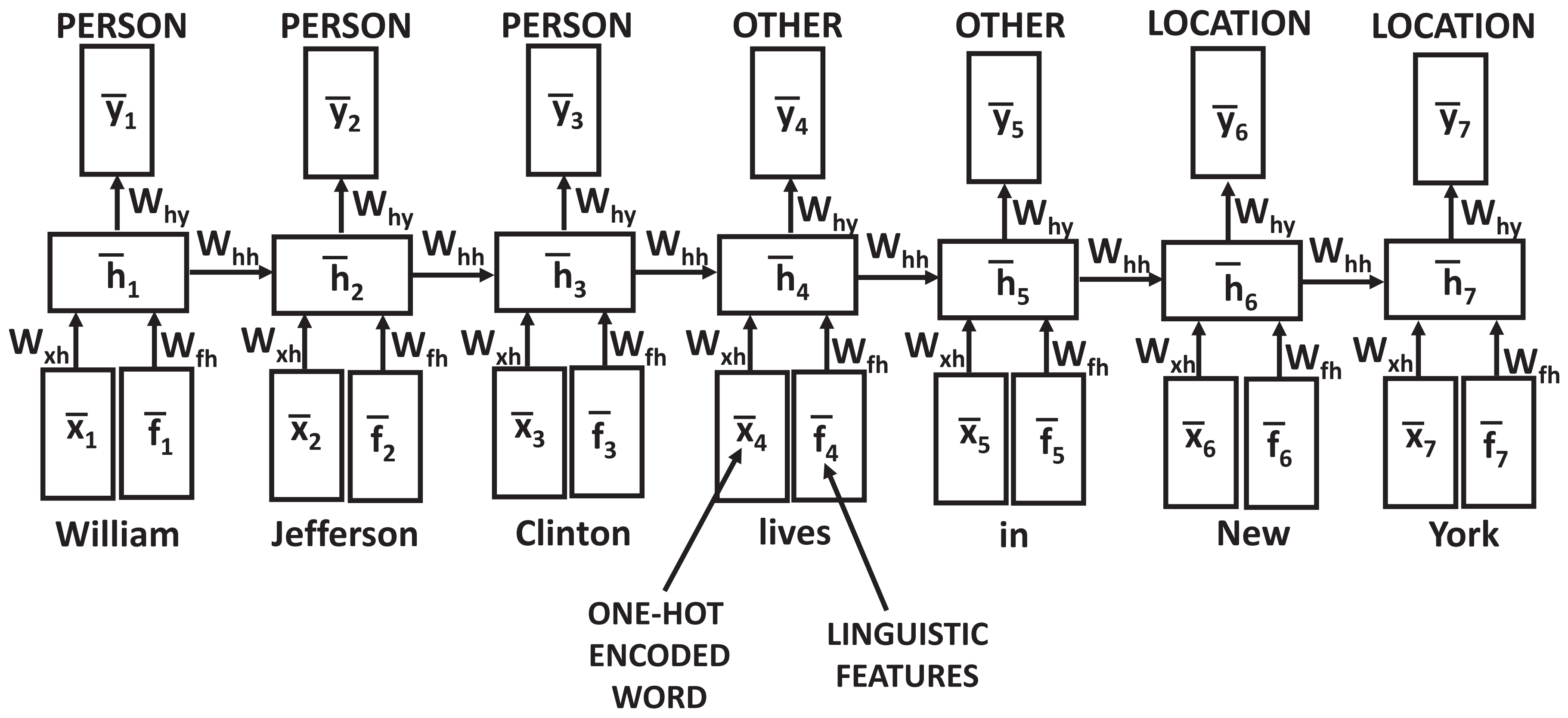
- Can be used for any sequence-to-sequence application including self-copies (recurrent autoencoders)

Sentence-Level Classification



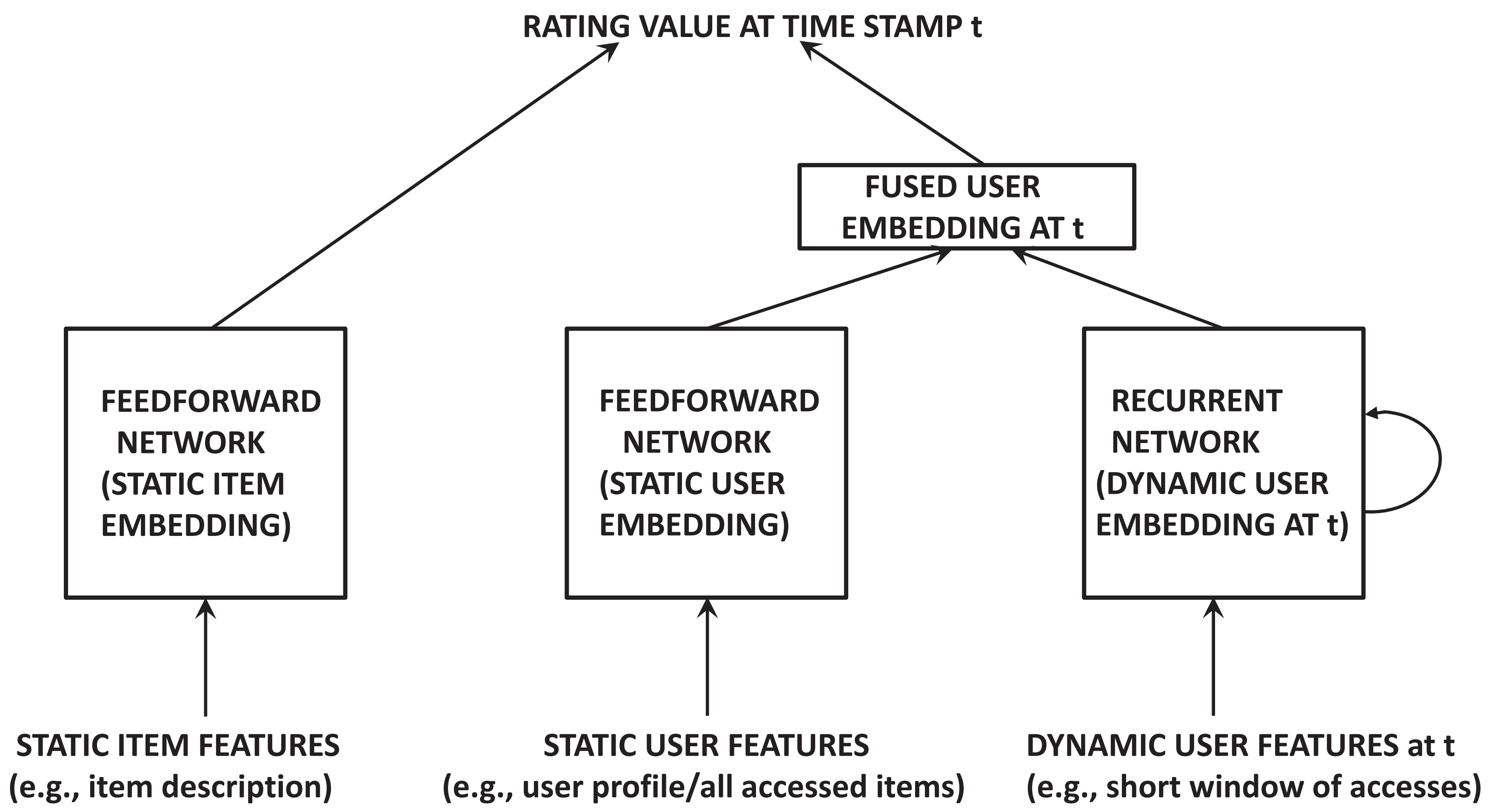
- Single target at the very end of the sentence.

Token-Level Classification


PERSON
PERSON
PERSON
OTHER
OTHER
LOCATION
LOCATION
$\overline{y}_1$ $\overline{y}_2$ $\overline{y}_3$ $\overline{y}_4$ $\overline{y}_5$ $\overline{y}_6$ $\overline{y}_7$
W_{hy}
$\overline{h}_1$ $\overline{h}_2$ $\overline{h}_3$ $\overline{h}_4$ $\overline{h}_5$ $\overline{h}_6$ $\overline{h}_7$
W_{hh}
W_{xh}
W_{fh}
$\overline{x}_1$ $\overline{f}_1$ $\overline{x}_2$ $\overline{f}_2$ $\overline{x}_3$ $\overline{f}_3$ $\overline{x}_4$ $\overline{f}_4$ $\overline{x}_5$ $\overline{f}_5$ $\overline{x}_6$ $\overline{f}_6$ $\overline{x}_7$ $\overline{f}_7$
William
Jefferson
Clinton
lives
in
New
York
ONE-HOT ENCODED WORD
LINGUISTIC FEATURES

- Label output for each token.

Temporal Recommender Systems


RATING VALUE AT TIME STAMP t
FUSED USER EMBEDDING AT t
FEEDFORWARD NETWORK (STATIC ITEM EMBEDDING)
FEEDFORWARD NETWORK (STATIC USER EMBEDDING)
RECURRENT NETWORK (DYNAMIC USER EMBEDDING AT t)
STATIC ITEM FEATURES (e.g., item description)
STATIC USER FEATURES (e.g., user profile/all accessed items)
DYNAMIC USER FEATURES at t (e.g., short window of accesses)

- Label output for each token.

Protein Structure Prediction

- The elements of the sequence are the symbols representing one of the 20 amino acids.

- The 20 possible amino acids are akin to the vocabulary used in the text setting.

- Each position is associated with a class label: *alpha-helix*, *beta-sheet*, or *coil*.
  - The problem can be reduced to token-level classification.

Speech and Handwriting Recognition

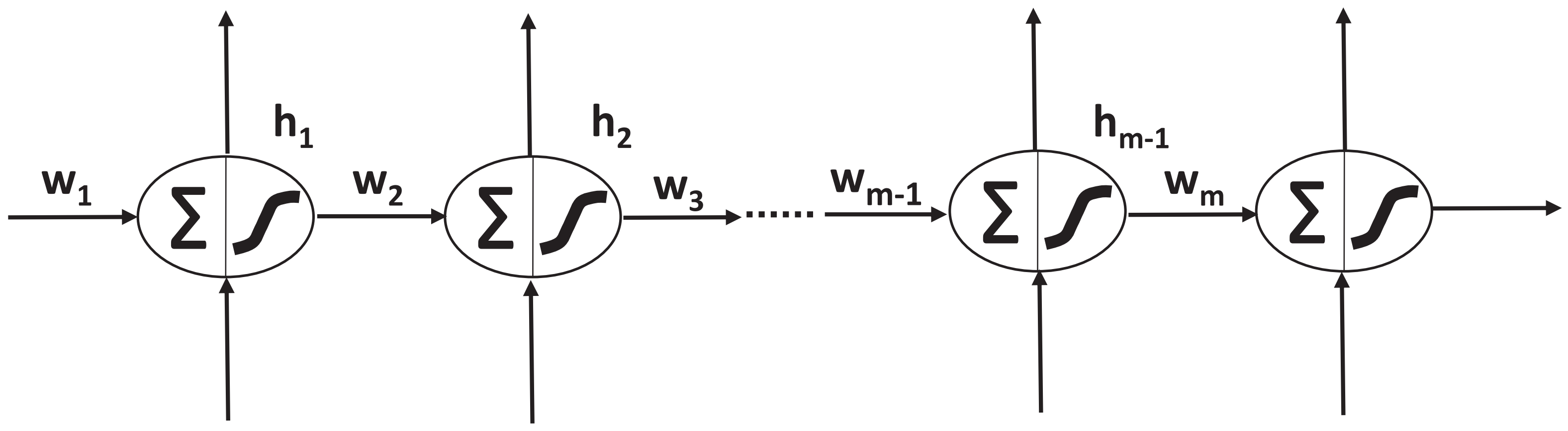
- Speech and handwriting recognition are sequential applications.
  - Frame representation of audios is transcribed into character sequence.
  - Convert sequence of strokes into character sequence.
  - Details and references in book.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

LSTMs and GRUs

Neural Networks and Deep Learning, Springer, 2018
Chapters 7.5 and 7.6

Vanishing and Exploding Gradient Problems



- Neural network with one node per layer.

- Backpropagated partial derivative get multiplied by weights and activation function derivatives.

- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish).

Generalization to Multi-Node Layers

- Vectorwise back-propagation is done by using the Jacobian J.

$$\overline{g}_t = J^T \overline{g}_{t+1} \tag{1}$$

- The (i, j)th entry of J is the (i, j)th entry of the square hidden-to-hidden matrix W_{hh} multiplied with the derivative of the tanh function at the current value in node j.

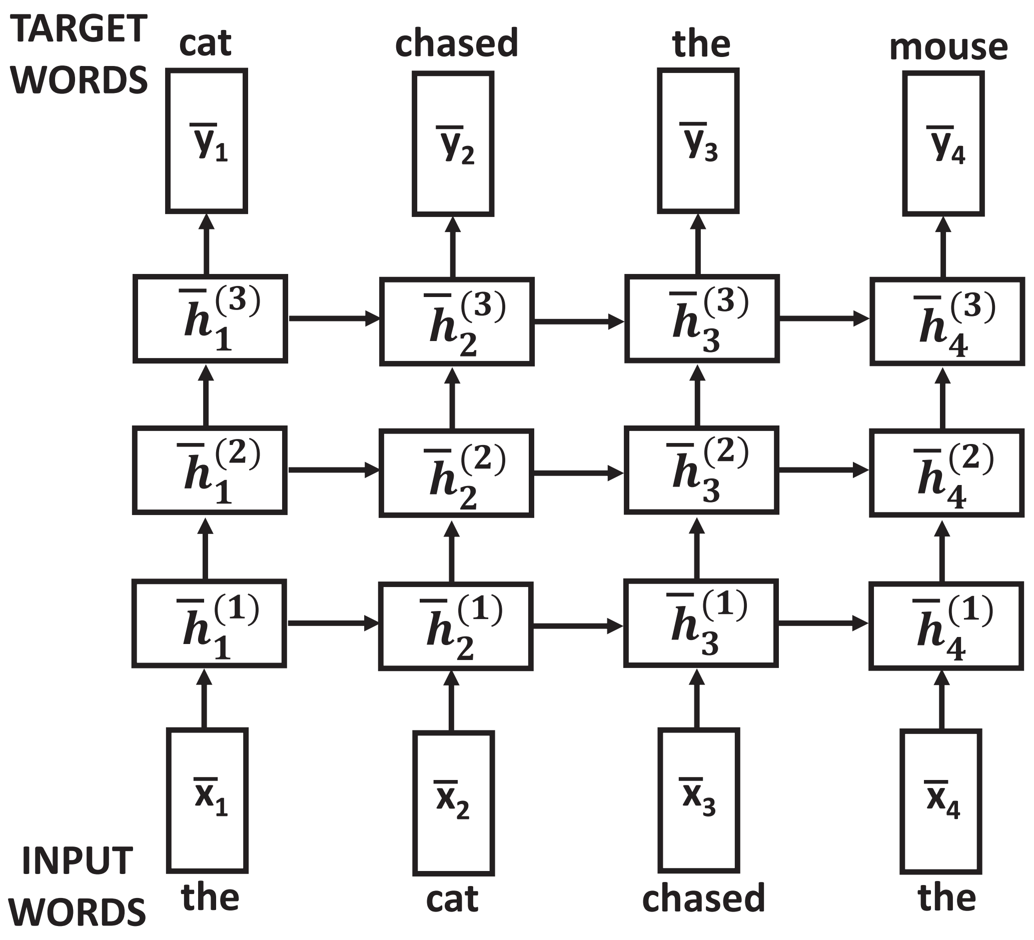
- Gradients cause successive multiplication with transposed Jacobian $J^T = P\Delta P^{-1}$.

- Multiplying m times leads to $(J^T)^m = P\Delta^m P^{-1} \Rightarrow$ Largest eigenvalue decides everything!

Other Issues with Recurrent Networks

- Hard to retain the information in a hidden state with successive matrix multiplications.
  - Hidden states of recurrent networks are inherently short-term.
  - No mechanism for fine-grained control of what information to retain from hidden states.
- The LSTM uses *analog gates* to control the flow of information.

Recap: Multilayer Recurrent Networks



$$\overline{h}_t^{(k)} = \tanh W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix}$$

Long-Term vs Short Term Memory

- A recurrent neural network only carries forward a hidden state $\overline{h}_t^{(k)}$ across time layers.

- An LSTM carries forward both a hidden state $\overline{h}_t^{(k)}$ and a cell state $\overline{c}_t^{(k)}$.
  - The hidden state is like short-term memory (updated aggressively).
  - The cell state is like long-term memory (updated gently).
  - Gates used to control updates from layer to layer.
  - Leaking between short-term and long-term memory allowed.

Setting Up Intermediate Variables and Gates

- Assume that hidden state and cell state are p-dimensional vectors.

- The matrix $W^{(k)}$ is of size $4p \times 2p$ $[4p \times (p+d)$ for $k=1]$.

- The intermediate variables are p-dimensional vector variables $\overline{i}$, $\overline{f}$, $\overline{o}$, and $\overline{c}$ that can be stacked to create a $4p$-dimensional vector:

$$\begin{array}{l} \text{Input Gate:} \\ \text{Forget Gate:} \\ \text{Output Gate:} \\ \text{Cell Increment:} \end{array} \quad \begin{bmatrix} \overline{i} \\ \overline{f} \\ \overline{o} \\ \overline{c} \end{bmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix} \qquad (2)$$

- Right-hand side of update equation looks similar to that of a multilayer recurrent network, except that we are setting up 4 intermediate variable vectors.

Updating Cell States and Hidden States

- Selectively forget and/or add to long-term memory

$$\overline{c}_t^{(k)} = \underbrace{\overline{f} \odot \overline{c}_{t-1}^{(k)}}_{\text{Reset?}} + \underbrace{\overline{i} \odot \overline{c}}_{\text{Increment?}} \tag{3}$$

  – Long-term memory can be thought of as *iterative feature engineering* rather than *hierarchical feature engineering* $\Rightarrow$ Principle used in ResNet

- Selectively leak long-term memory to hidden state

$$\overline{h}_t^{(k)} = \overline{o} \odot \tanh(\overline{c}_t^{(k)}) \tag{4}$$

  – In some variations, the tanh function is not used in Equation 4, and the raw cell state is used.

Intuition for LSTM

- Examine the LSTM with single dimension:

$$c_t = c_{t-1} * f + i * c \tag{5}$$

- Partial derivative of c_t with respect to c_{t-1} is $f \Rightarrow$ multiply gradient flow with f!

- Element wise multiplication ensures that the result is also true for p-dimensional states.

- Each time-stamp has a different values of $f \Rightarrow$ less likely to cause vanishing gradients

- Gradient flow for c_t inherited by h_t because $h_t = o * \tanh(c_t)$

Gated Recurrent Unit

- Simplification of LSTM but not a special case.
  - Does not use explicit cell states.
  - Controls updates carefully.

GRU Updates

- Use two matrices $W^{(k)}$ and $V^{(k)}$ of sizes $2p \times 2p$ and $p \times 2p$, respectively.
  - In the first layer, the matrices are of sizes $2p \times (p+d)$ and $p \times (p+d)$.

- intermediate, p-dimensional vector variables $\overline{z}_t$ and $\overline{r}_t$, respectively $\Rightarrow$ Update and reset gates

$$\begin{matrix}\text{Update Gate:} \\ \text{Reset Gate:}\end{matrix} \quad \begin{bmatrix} \overline{z} \\ \overline{r} \end{bmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \end{pmatrix} W^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{h}_{t-1}^{(k)} \end{bmatrix} \tag{6}$$

$$\overline{h}_t^{(k)} = \overline{z} \odot \overline{h}_{t-1}^{(k)} + (1 - \overline{z}) \odot \tanh V^{(k)} \begin{bmatrix} \overline{h}_t^{(k-1)} \\ \overline{r} \odot \overline{h}_{t-1}^{(k)} \end{bmatrix} \tag{7}$$

Explanation of Updates

- The reset gate $\overline{r}$ decides how much of the hidden state to carry over from the previous time-stamp for a matrix-based transformation.

- The update gate $\overline{z}$ decides the *relative* strength of the matrix-based update and a more direct copy from previous time stamp.

- The direct copy from the previous time stamp stabilizes the gradient flows.

Intuition for GRU

- Consider a 1-dimensional and single-layer GRU update:

$$h_t = z \cdot h_{t-1} + (1-z) \cdot \tanh[v_1 \cdot x_t + v_2 \cdot r \cdot h_{t-1}] \qquad (8)$$

- One can compute the derivative as follows:

$$\frac{\partial h_t}{\partial h_{t-1}} = z + \text{(Additive Terms)} \qquad (9)$$

- The extra term $z \in (0,1)$ helps in passing *unimpeded* gradient flow and makes computations more stable.
  - Additive terms heavily depend on $(1-z)$.
  - Gradient factors are *different* for each time stamp.

- These networks are sometimes referred to as *highway networks*.

Comparisons of LSTM and GRU

- K. Greff et al. LSTM: A search space odyssey. *IEEE TNNLS*, 2016.
  - Many variants of LSTM equations.
- J. Chung et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv:1412.3555*, 2014.
- R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. *International Conference on Machine Learning*, pp. 2342–2350, 2015.
- Main advantage of GRU is simplicity.
- None of the LSTM variants could perform better than it in a reliable way.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Convolutional Neural Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.1–8.2

Convolutional Neural Networks

- Like recurrent neural networks, convolutional neural networks are *domain-aware* neural networks.
  - The structure of the neural network encodes domain-specific information.
  - Specifically designed for images.

- Images have length, width, and a *depth* corresponding to the number of color channels (typically 3: RGB).

- All layers are spatially structured with length, width, and depth.

History

- Motivated by Hubel and Wiesel's understanding of the cat's visual cortex.
  - Particular shapes in the visual field excite neurons $\Rightarrow$ Sparse connectivity with shared weights.
  - Hierarchical arrangement of neurons into simple and complex cells.
  - Neocognitron was first variant $\Rightarrow$ Motivated *LeNet-5*
- Success in *ImageNet (ILSVRC)* competitions brought attention to deep learning.

Basic Structure of a Convolutional Neural Network

- Most layers have length, width, and depth.
  - The length and width are almost always the same.
  - The depth is 3 for color images, 1 for grayscale, and an arbitrary value for hidden layers.

- Three operations are convolution, max-pooling, and ReLU.
  - Maxpooling substituted with strided convolution in recent years.
  - The convolution operation is analogous to the matrix multiplication in a conventional network.

Filter for Convolution Operation

- Let the input volume of layer q have dimensions $L_q \times B_q \times d_q$.

- The operation uses a *filter* of size $F_q \times F_q \times d_q$.

  – The filter's spatial dimensions must be no larger than layer's spatial dimensions.

  – The filter depth *must* match input volume.

  – Typically, the filter spatial size F_q is a small odd number like 3 or 5.

Convolution Operation

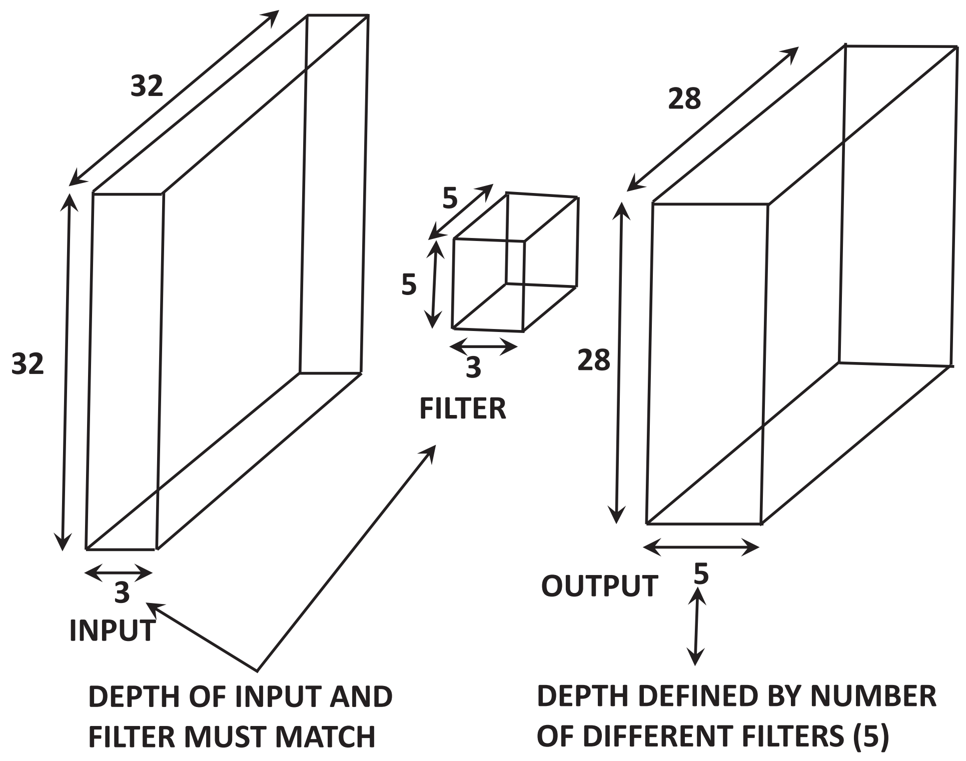
- Spatially align the top-left corner of filter with each of $(L_q - F_q + 1) \times (B_q - F_q + 1)$ spatial positions.
  - Corresponds to number of positions for top left corner in input volume, so that filter fully fits inside layer volume.

- Perform elementwise multiplication between input/filter over all $F_q \times F_q \times d_q$ aligned elements and add.

- Creates a single spatial map in the output of size $(L_q - F_q + 1) \times (B_q - F_q + 1)$.

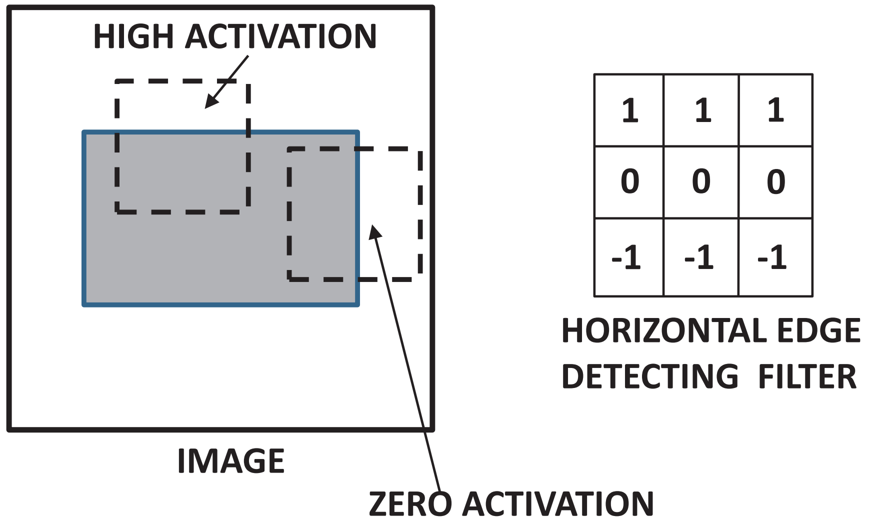
- Multiple filters create depth in output volume.

Convolution Operation: Pictorial Illustration of Dimensions



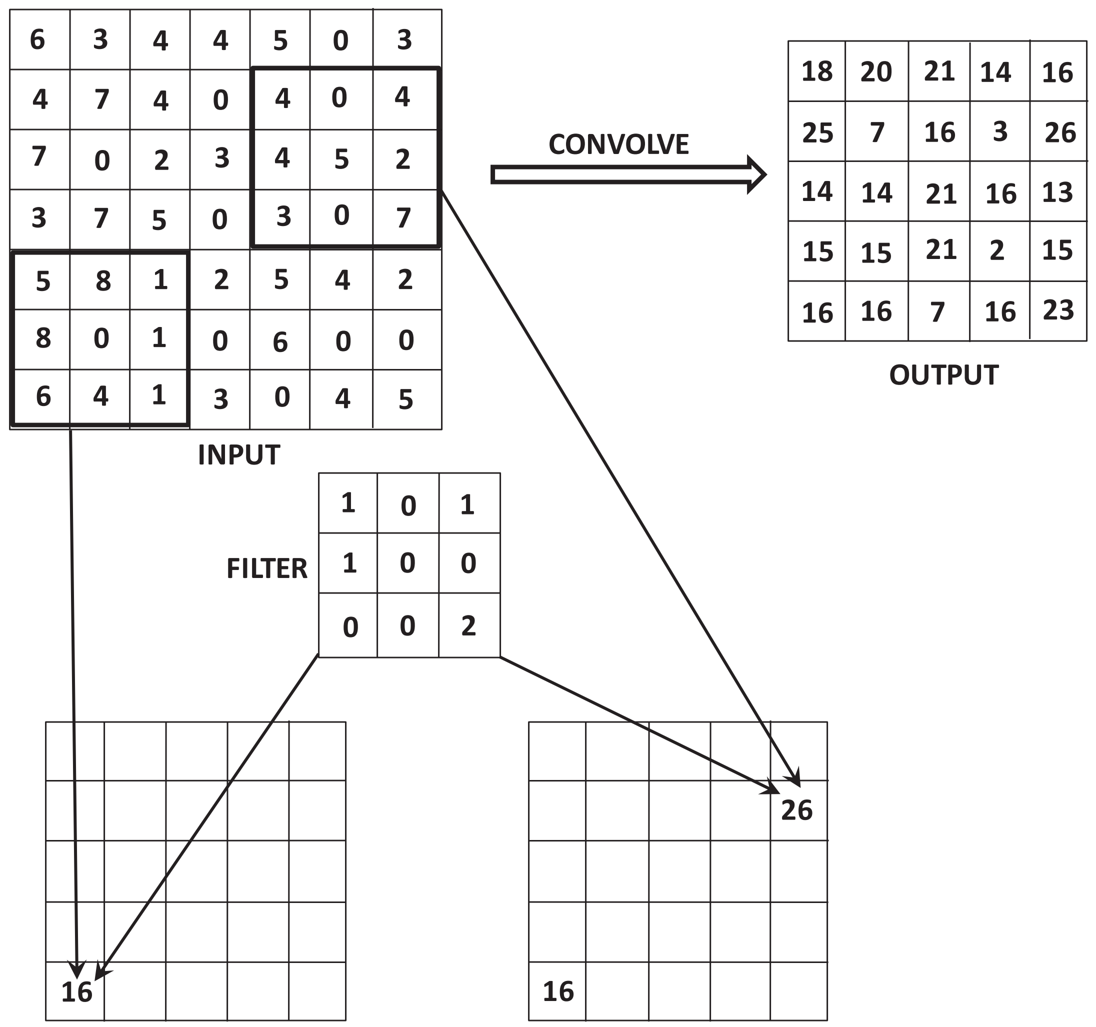
32
32
3
INPUT
5
5
3
FILTER
28
28
5
OUTPUT
DEPTH OF INPUT AND FILTER MUST MATCH
DEPTH DEFINED BY NUMBER OF DIFFERENT FILTERS (5)

(a) Input and output dimensions



(b) Sliding the filter

Convolution Operation: Numerical Example with Depth 1



6	3	4	4	5	0	3
4	7	4	0	4	0	4
7	0	2	3	4	5	2
3	7	5	0	3	0	7
5	8	1	2	5	4	2
8	0	1	0	6	0	0
6	4	1	3	0	4	5

1	0	1
1	0	0
0	0	2

18	20	21	14	16
25	7	16	3	26
14	14	21	16	13
15	15	21	2	15
16	16	7	16	23

- Add up over multiple activations maps from the depth

Understanding Convolution

- Sparse connectivity because we are creating a feature from a region in the input volume of the size of the filter.
  - Trying to explore smaller regions of the image to find shapes.
- Shared weights because we use the same filter across entire spatial volume.
  - Interpret a shape in various parts of the image in the same way.

Effects of Convolution

- Each feature in a hidden layer captures some properties of a region of input image.

- A convolution in the qth layer increases the *receptive field* of a feature from the qth layer to the $(q+1)$th layer.

- Consider using a 3×3 filter successively in three layers:
  - The activations in the first, second, and third hidden layers capture pixel regions of size 3×3, 5×5, and 7×7, respectively, in the *original input image*.

Need for Padding

- The convolution operation reduces the size of the $(q+1)$th layer in comparison with the size of the qth layer.
  - This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image (or of the feature map, in the case of hidden layers).

- This problem can be resolved by using *padding*.

- By adding $(F_q - 1)/2$ "pixels" all around the borders of the feature map, one can maintain the size of the spatial image.

Application of Padding with 2 Zeros

$$
\begin{array}{|c|c|c|c|c|c|c|}
\hline
6 & 3 & 4 & 4 & 5 & 0 & 3 \\ \hline
4 & 7 & 4 & 0 & 4 & 0 & 4 \\ \hline
7 & 0 & 2 & 3 & 4 & 5 & 2 \\ \hline
3 & 7 & 5 & 0 & 3 & 0 & 7 \\ \hline
5 & 8 & 1 & 2 & 5 & 4 & 2 \\ \hline
8 & 0 & 1 & 0 & 6 & 0 & 0 \\ \hline
6 & 4 & 1 & 3 & 0 & 4 & 5 \\ \hline
\end{array}
\xrightarrow{\text{PAD}}
\begin{array}{cc|c|c|c|c|c|c|c|cc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \cline{3-9}
0 & 0 & 6 & 3 & 4 & 4 & 5 & 0 & 3 & 0 & 0 \\ \cline{3-9}
0 & 0 & 4 & 7 & 4 & 0 & 4 & 0 & 4 & 0 & 0 \\ \cline{3-9}
0 & 0 & 7 & 0 & 2 & 3 & 4 & 5 & 2 & 0 & 0 \\ \cline{3-9}
0 & 0 & 3 & 7 & 5 & 0 & 3 & 0 & 7 & 0 & 0 \\ \cline{3-9}
0 & 0 & 5 & 8 & 1 & 2 & 5 & 4 & 2 & 0 & 0 \\ \cline{3-9}
0 & 0 & 8 & 0 & 1 & 0 & 6 & 0 & 0 & 0 & 0 \\ \cline{3-9}
0 & 0 & 6 & 4 & 1 & 3 & 0 & 4 & 5 & 0 & 0 \\ \cline{3-9}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
$$

Types of Padding

- No padding: When no padding is used around the borders of the image $\Rightarrow$ Reduces the size of spatial footprint by $(F_q - 1)$.

- Half padding: Pad with $(F_q - 1)/2$ "pixels" $\Rightarrow$ Maintain size of spatial footprint.

- Full padding: Pad with $(F_q - 1)$ "pixels" $\Rightarrow$ Increase size of spatial footprint with $(F_q - 1)$.

$$2 * \text{Amount Padded} + \text{Size Reduction} = (F_q - 1) \qquad (1)$$

- Note that amount padded is on both sides $\Rightarrow$ Explains factor of 2

Strided Convolution

- When a stride of S_q is used in the qth layer, the convolution is performed at the locations 1, $S_q + 1$, $2\,S_q + 1$, and so on along both spatial dimensions of the layer.

- The spatial size of the output on performing this convolution has height of $(L_q - F_q)/S_q + 1$ and a width of $(B_q - F_q)/S_q + 1$.
  - Exact divisibility is required.

- Strided convolutions are sometimes used in lieu of max-pooling.

Max Pooling

- The pooling operation works on small grid regions of size $P_q \times P_q$ in each layer, and produces another layer *with the same depth*.

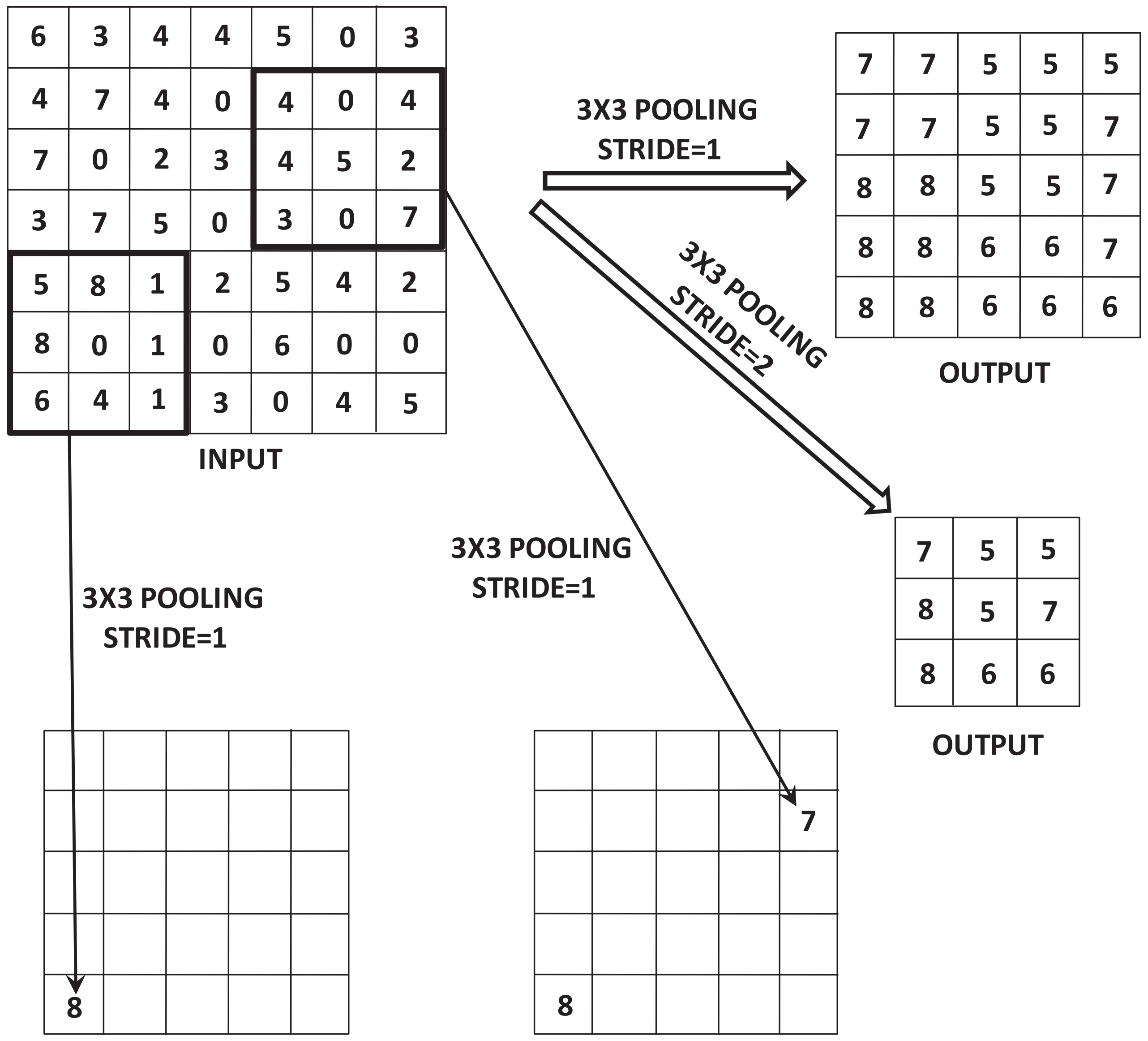
- For each square region of size $P_q \times P_q$ in each of the d_q activation maps, the *maximum* of these values is returned.

- It is common to use a stride $S_q > 1$ in pooling (often we have $P_q = S_q$).

- Length of the new layer will be $(L_q - P_q)/S_q + 1$ and the breadth will be $(B_q - P_q)/S_q + 1$.

- Pooling drastically reduces the spatial dimensions of each activation map.

Pooling Example



INPUT

6	3	4	4	5	0	3
4	7	4	0	4	0	4
7	0	2	3	4	5	2
3	7	5	0	3	0	7
5	8	1	2	5	4	2
8	0	1	0	6	0	0
6	4	1	3	0	4	5

3X3 POOLING STRIDE=1 — OUTPUT

7	7	5	5	5
7	7	5	5	7
8	8	5	5	7
8	8	6	6	7
8	8	6	6	6

3X3 POOLING STRIDE=2 — OUTPUT

7	5	5
8	5	7
8	6	6

ReLU

- Use of ReLU is a straightforward one-to-one operation.

- The number of feature maps and spatial footprint size is retained.

- Often stuck at the end of a convolution operation and not shown in architectural diagrams.

Fully Connected Layers: Stuck at the End

- Each feature in the final spatial layer is connected to each hidden state in the first fully connected layer.

- This layer functions in exactly the same way as a traditional feed-forward network.

- In most cases, one might use more than one fully connected layer to increase the power of the computations towards the end.

- The connections among these layers are exactly structured like a traditional feed-forward network.

- The vast majority of parameters lie in the fully connected layers.

Interleaving Between Layers

- The convolution, pooling, and ReLU layers are typically interleaved in order to increase expressive power.

- The ReLU layers often follow the convolutional layers, just as a nonlinear activation function typically follows the linear dot product in traditional neural networks.

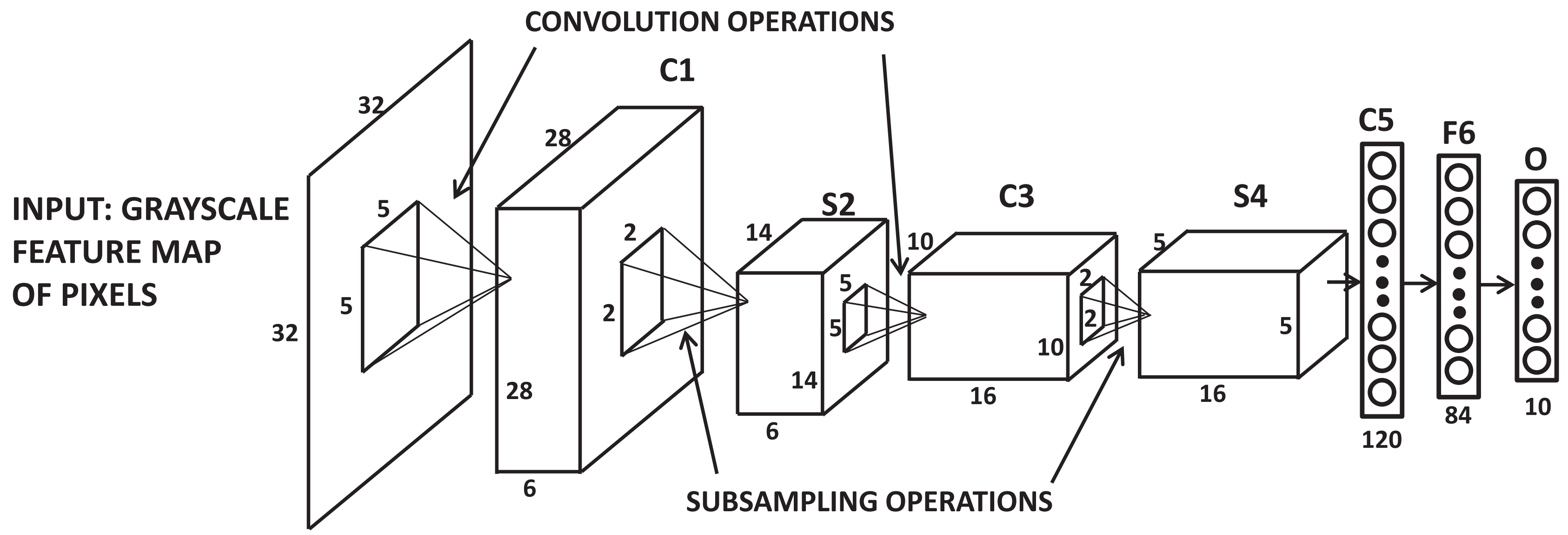
- After two or three sets of convolutional-ReLU combinations, one might have a max-pooling layer.

- Examples:

CRCRP

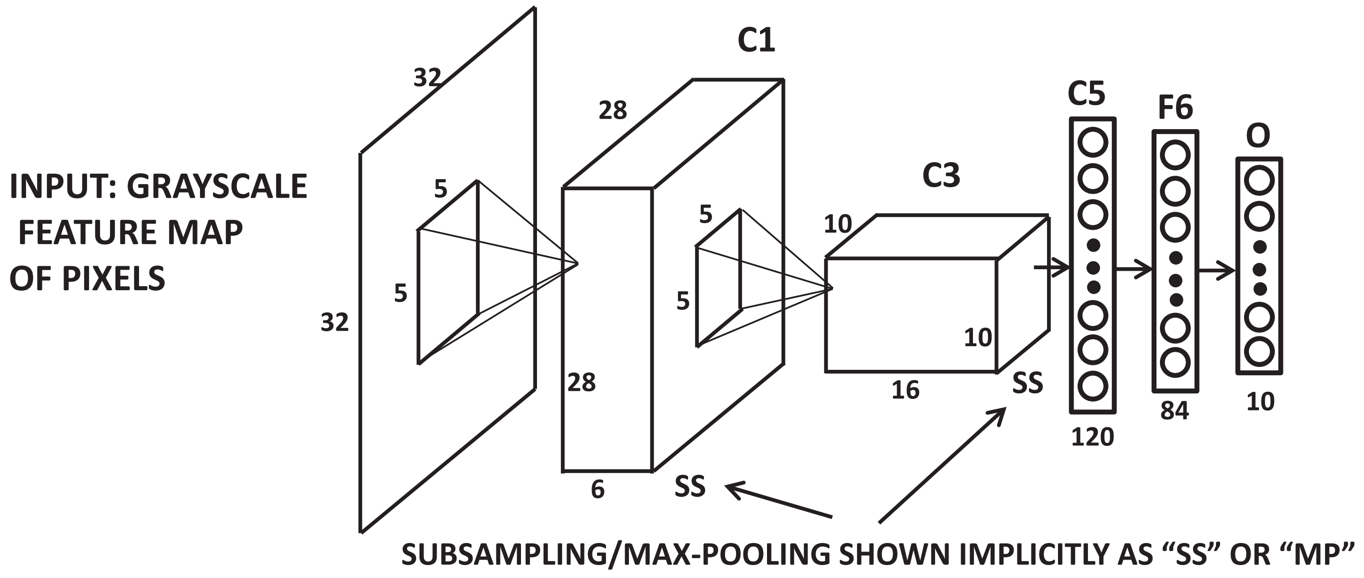
CRCRCRP

Example: LeNet-5: Full Notation



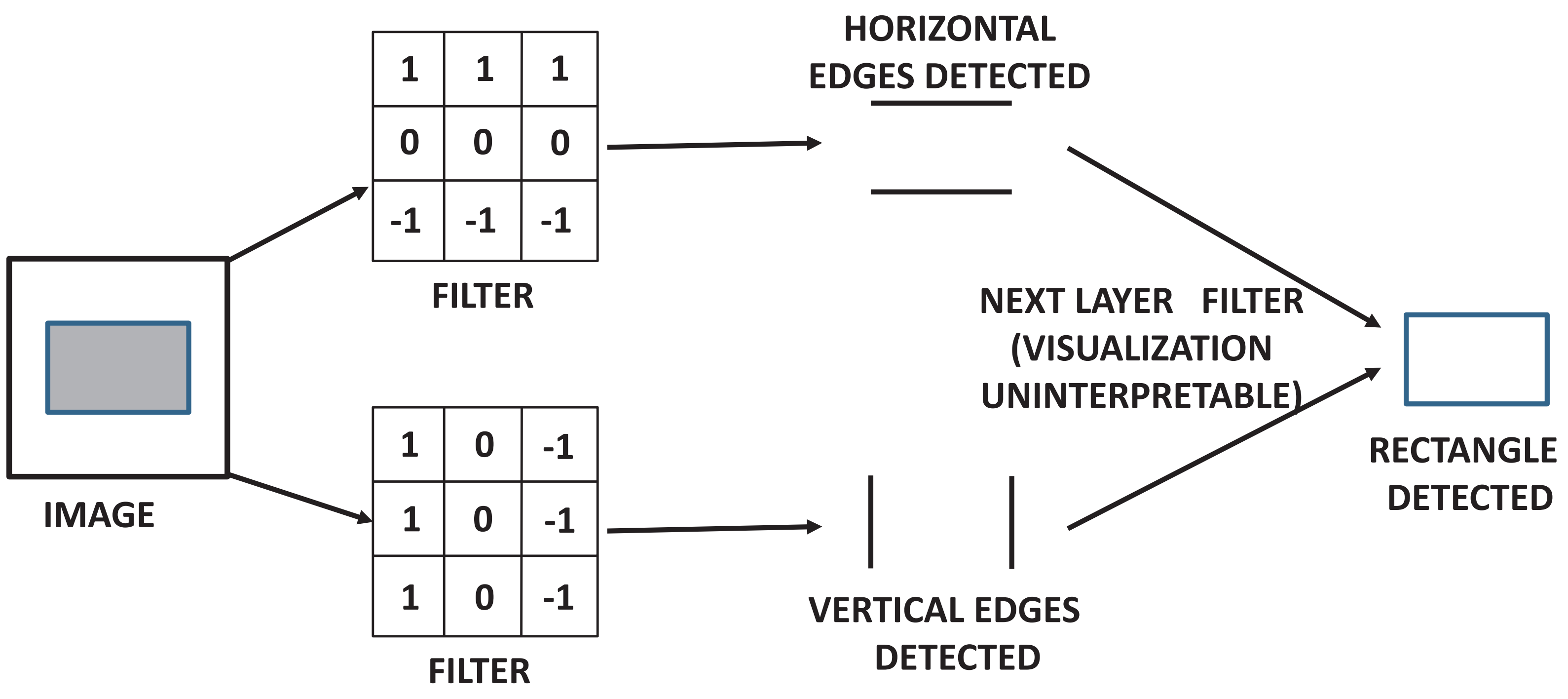
- The earliest convolutional network

Example: LeNet-5: Shorthand Notation


INPUT: GRAYSCALE FEATURE MAP OF PIXELS
32
32
5
5
C1
28
28
6
5
5
SS
C3
10
10
16
SS
C5
120
F6
84
O
10
SUBSAMPLING/MAX-POOLING SHOWN IMPLICITLY AS "SS" OR "MP"

- Subsampling layers are not explicitly shown

Feature Engineering



- The early layers detect primitive features and later layers complex ones.

- During training the filters will be learned to identify relevant shapes.

Hierarchical Feature Engineering

- Successive layers put together primitive features to create more complex features.
- Complex features represent regularities in the data, that are valuable for features.
- Mid-level features might be honey-combs.
- Higher-level features might be a part of a face.
- The network is a master of extracting repeating shapes in data-driven manner.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation in Convolutional Neural Networks and Its Visualization Applications

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.3, 8.5

Backpropagation in Convolutional Neural Networks

- Three operations of convolutions, max-pooling, and ReLU.
- The ReLU backpropagation is the same as any other network.
  - Passes gradient to a previous layer only if the original input value was positive.
- The max-pooling passes the gradient flow through the largest cell in the input volume.
- Main complexity is in backpropagation through convolutions.

Backpropagating through Convolutions

- Traditional backpropagation is transposed matrix multiplication.

- Backpropagation through convolutions is transposed convolution (i.e., with an *inverted filter*).

- Derivative of loss with respect to each cell is backpropagated.

  – Elementwise approach of computing which cell in input contributes to which cell in output.

  – Multiplying with an inverted filter.

- Convert layer-wise derivative to weight-wise derivative and add over shared weights.

Backpropagation with an Inverted Filter [Single Channel]

a	b	c
d	e	f
g	h	i

FILTER DURING CONVOLUTION

i	h	g
f	e	d
c	b	a

FILTER DURING BACKPROPAGATION

- Multichannel case: We have 20 filters for 3 input channels (RGB) $\Rightarrow$ We have $20 \times 3 = 60$ spatial slices.

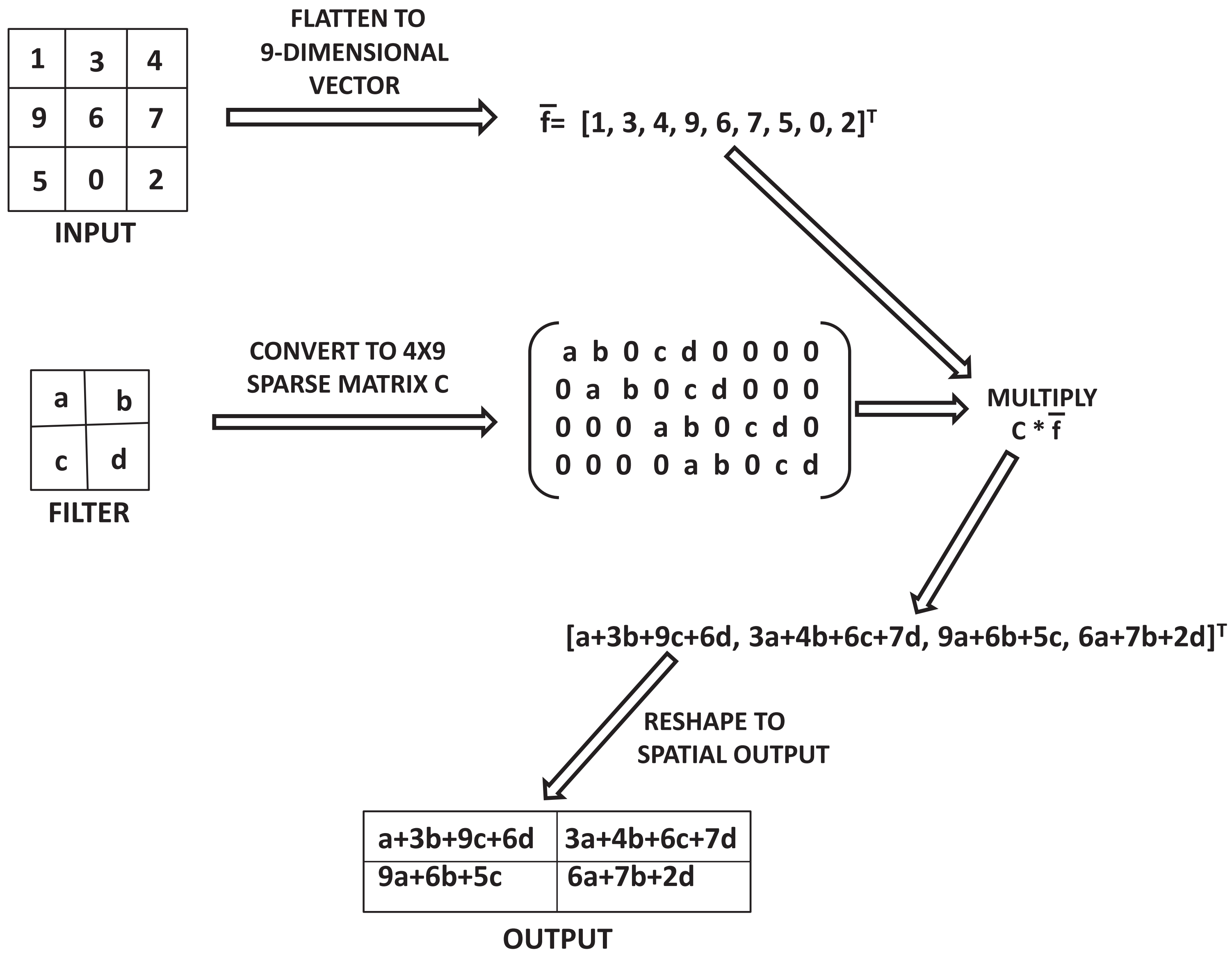
- Each of these 60 spatial slices will be inverted and grouped into 3 sets of filters with depth 20 (one each for RGB).

- Backpropagate with newly grouped filters.

Convolution as a Matrix Multiplication

- Convolution can be presented as a matrix multiplication.
  - Useful during forward and backward propagation.
  - Backward propagation can be presented as transposed matrix multiplication

Convolution as a Matrix Multiplication

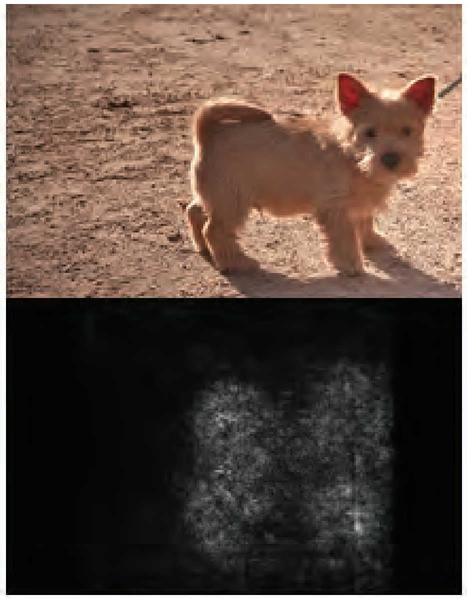


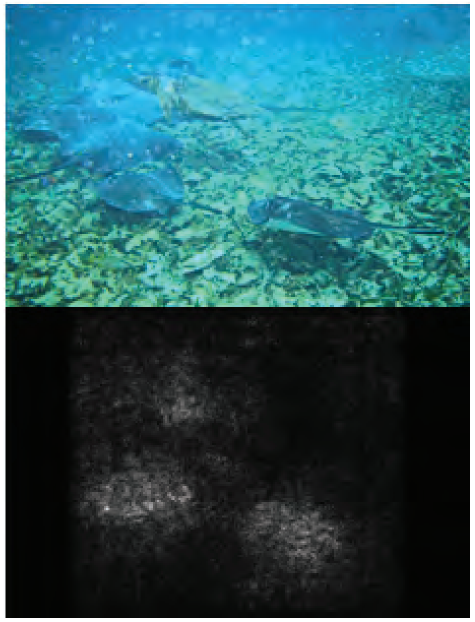
1 3 4
9 6 7
5 0 2
INPUT
FLATTEN TO 9-DIMENSIONAL VECTOR
$\bar{f}$= [1, 3, 4, 9, 6, 7, 5, 0, 2]^T
a b
c d
FILTER
CONVERT TO 4X9 SPARSE MATRIX C
a b 0 c d 0 0 0 0
0 a b 0 c d 0 0 0
0 0 0 a b 0 c d 0
0 0 0 0 a b 0 c d
MULTIPLY C * $\bar{f}$
[a+3b+9c+6d, 3a+4b+6c+7d, 9a+6b+5c, 6a+7b+2d]^T
RESHAPE TO SPATIAL OUTPUT
a+3b+9c+6d
3a+4b+6c+7d
9a+6b+5c
6a+7b+2d
OUTPUT

Gradient-Based Visualization

- Can backpropagate all the way back to the input layer.
- Imagine the output o is the probability of a class label like "dog"
- Compute $\frac{\partial o}{\partial x_i}$ for each pixel x_i of each color.
  - Compute maximum absolute magnitude of gradient over RGB colors and create grayscale image.

Gradient-Based Visualization





- Examples of portions of specific images activated by particular class labels. (©2014 Simonyan, Vedaldi, and Zisserman)

Getting Cleaner Visualizations

- The idea of "deconvnet" is sometimes used for cleaner visualizations.

- Main difference is in terms of how ReLU's are treated.

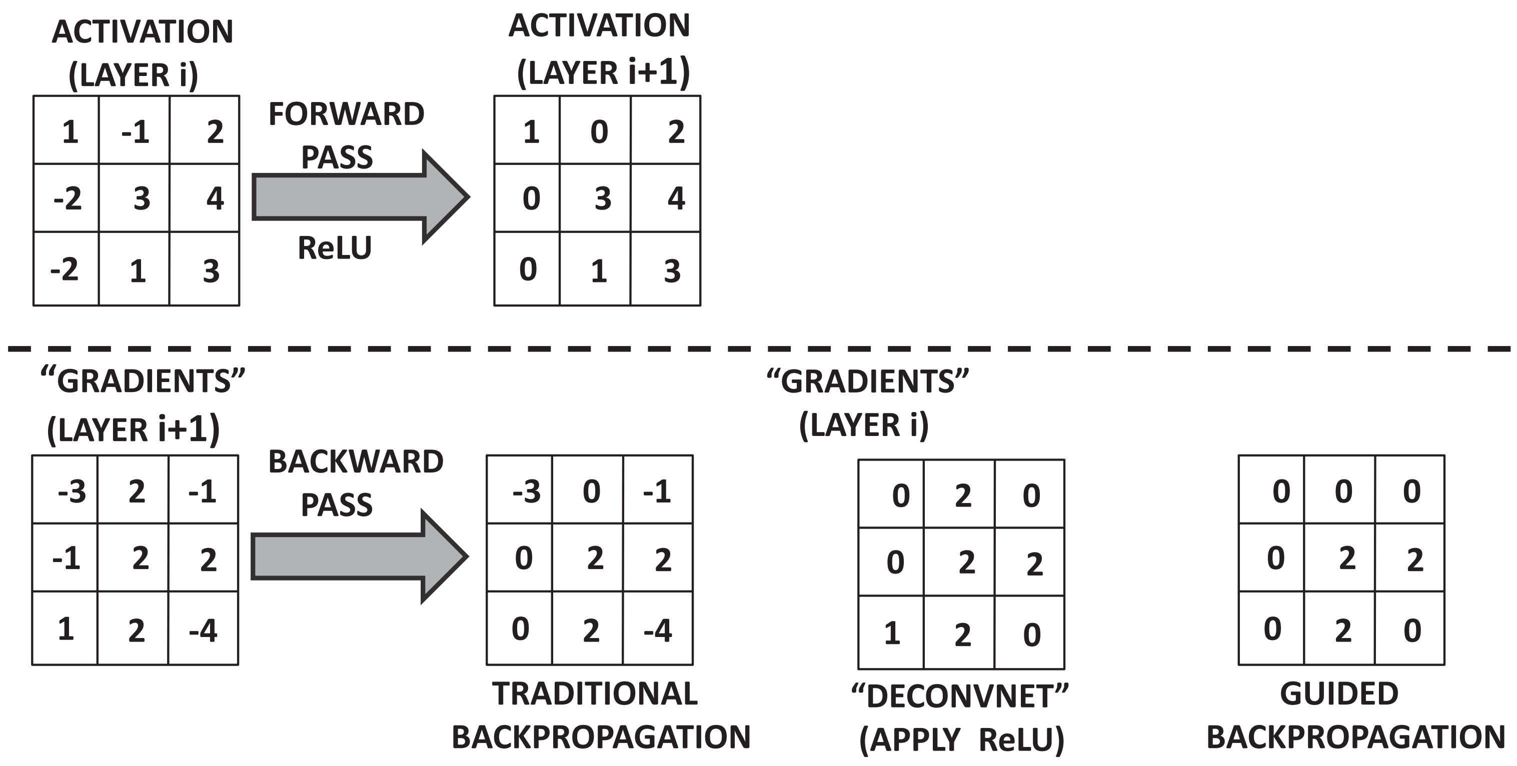
  - Normal backpropagation passes on gradient through ReLU when *input* is positive.

  - "Deconvnet" passes on gradient through ReLU when *backpropagated gradient* is positive.

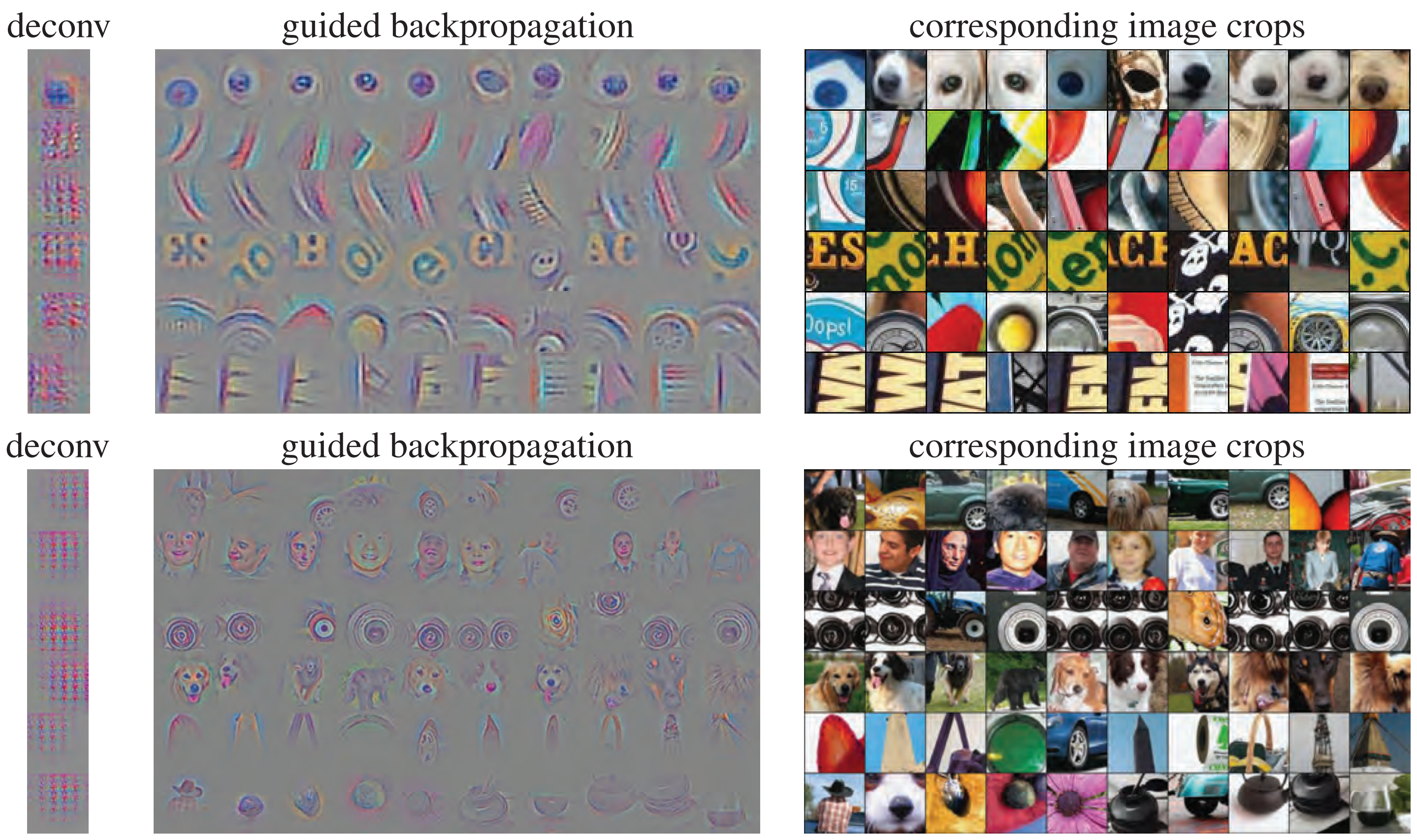
Guided Backpropagation

- A variation of backpropagation, referred to as *guided back-propagation* is more useful for visualization.
  - Guided backpropagation is a combination of gradient-based visualization and “deconvnet.”
  - Set an entry to zero, if either of these rules sets an entry to zero.

Illustration of “Deconvnet” and Guided Backpropagation



Derivatives of Features with Respect to Input Pixels



- ©2015 Springenberg, Dosovitskiy, Brox, Riedmiller

Creating a fantasy image that matches a label

- The value of o might be the unnormalized score for "*banana*."

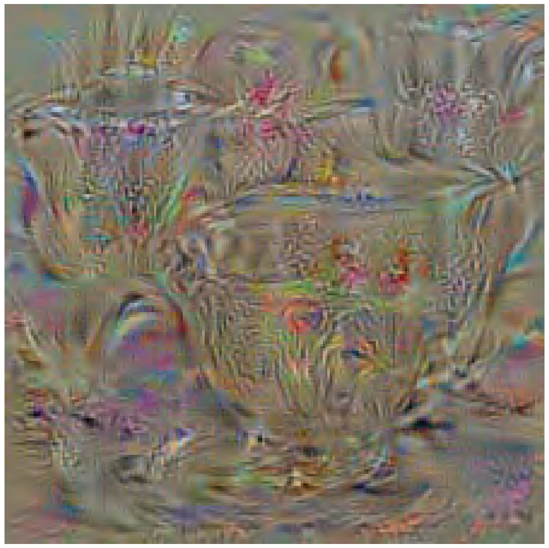
- We would like to learn the input image $\overline{x}$ that maximizes the output o, while applying regularization to $\overline{x}$:

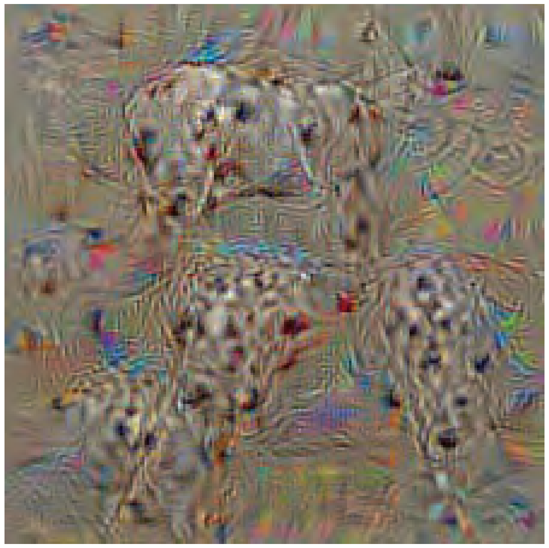
$$\text{Maximize}_{\overline{x}} \; J(\overline{x}) = (o - \lambda ||\overline{x}||^2)$$

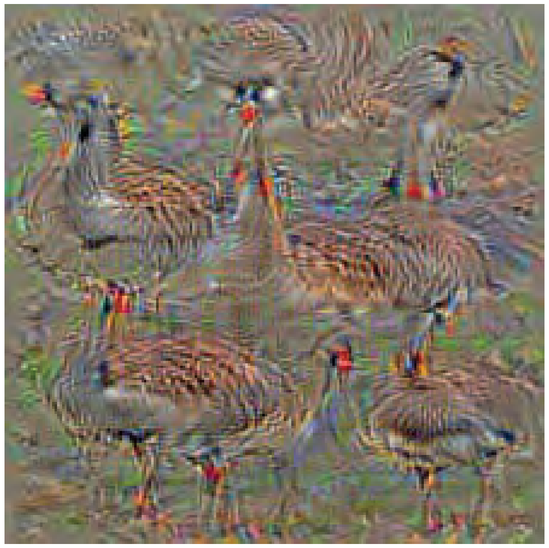
- Here, λ is the regularization parameter.

- Update $\overline{x}$ while keeping weights fixed!

Examples


cup


dalmatian


goose

Generalization to Autoencoders

- Ideas have been generalized to convolutional autoencoders.

- Deconvolution operation similar to backpropagation.

- One can combine pretraining with backpropagation.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

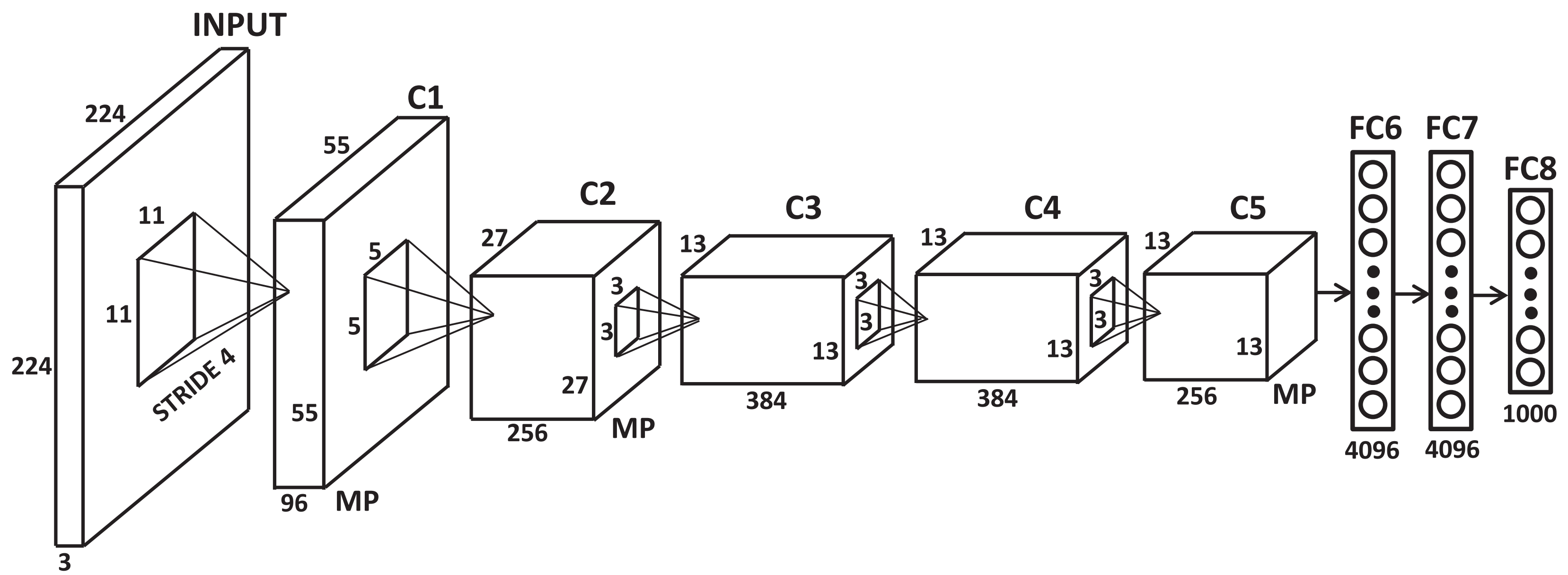
Case Studies of Convolutional Neural Networks

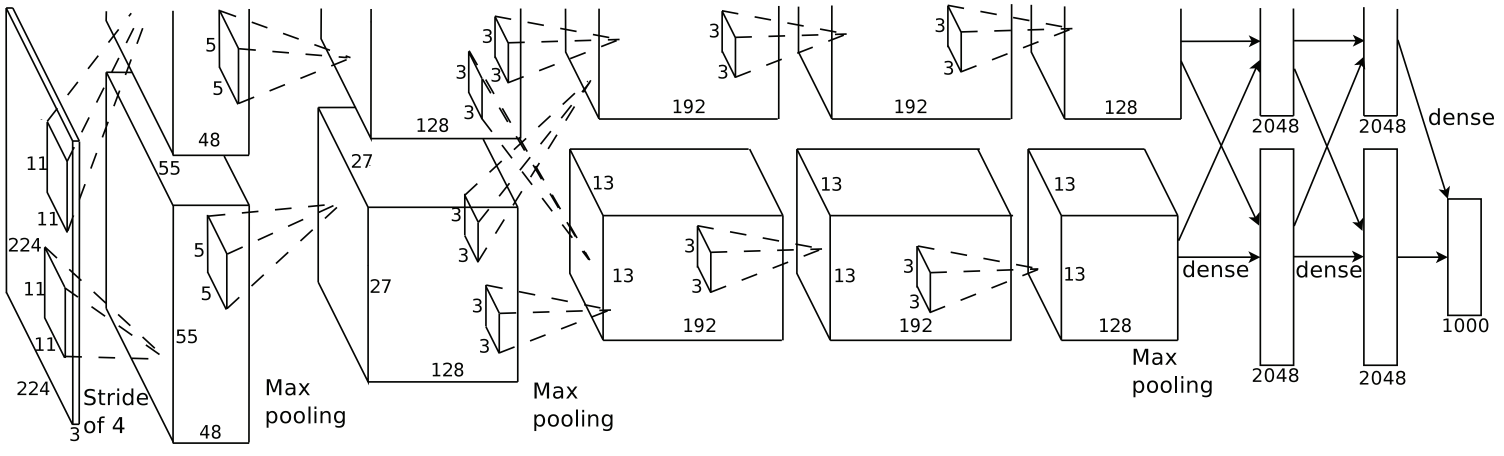
Neural Networks and Deep Learning, Springer, 2018
Chapter 8.4

AlexNet

- Winner of the 2012 ILSVRC contest
  - Brought attention to the area of deep learning
- First use of ReLU
- Use Dropout with probability 0.5
- Use of 3×3 pools at stride 2
- Heavy data augmentation

AlexNet Architecture





Other Comments on AlexNet

- Popularized the notion of FC7 features
- The features in the penultimate layer are often extracted and used for various applications
- Pretrained versions of AlexNet are available in most deep learning frameworks.
- Single network had top-5 error rate of 18.2%
- Ensemble of seven CNN had top-5 error rate of 15.4%

ZfNet

	AlexNet	*ZFNet*
Volume: Operations:	$224 \times 224 \times 3$ Conv 11×11 (stride 4)	$224 \times 224 \times 3$ Conv 7×7 (stride 2), MP
Volume: Operations:	$55 \times 55 \times 96$ Conv 5×5, MP	$55 \times 55 \times 96$ Conv 5×5 (stride 2), MP
Volume: Operations:	$27 \times 27 \times 256$ Conv 3×3, MP	$13 \times 13 \times 256$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 512$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 1024$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 256$ MP, Fully connect	$13 \times 13 \times 512$ MP, Fully connect
FC6: Operations:	4096 Fully connect	4096 Fully connect
FC7: Operations:	4096 Fully connect	4096 Fully connect
FC8: Operations:	1000 Softmax	1000 Softmax

- AlexNet Variation $\Rightarrow$ (Clarifai) won 2013 (14.8%/11.1%)

VGG

- One of the top entries in 2014 (but not winner).

- Notable for its design principle of reduced filter size and increased depth

- All filters had spatial footprint of 3×3 and padding of 1

- Maxpooling was done using 2×2 regions at stride 2

- Experimented with a variety of configurations between 11 and 19 layers

Principle of Reduced Filter Size

- A single 7×7 filter will have 49 parameters over one channel.

- Three 3×3 filters will have a receptive field of size 7×7, but will have only 27 parameters.

- **Regularization advantage:** A single filter will capture only primitive features but three successive filters will capture more complex features.

VGG Configurations

Name:	A	A-LRN	B	C	D	E
# Layers	11	11	13	16	16	19
	C3D64	C3D64	C3D64	C3D64	C3D64	C3D64
		LRN	C3D64	C3D64	C3D64	C3D64
	M	M	M	M	M	M
	C3D128	C3D128	C3D128	C3D128	C3D128	C3D128
			C3D128	C3D128	C3D128	C3D128
	M	M	M	M	M	M
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
				C1D256	C3D256	C3D256
						C3D256
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC1000	FC1000	FC1000	FC1000	FC1000	FC1000
	S	S	S	S	S	S

VGG Design Choices and Performance

- Max-pooling had the responsibility of reducing spatial foot-print.

- Number of filters often increased by 2 after each max-pooling

  – Volume remained roughly constant.

- VGG had top-5 error rate of 7.3%

- Column D was the best architecture [previous slide]

GoogLeNet

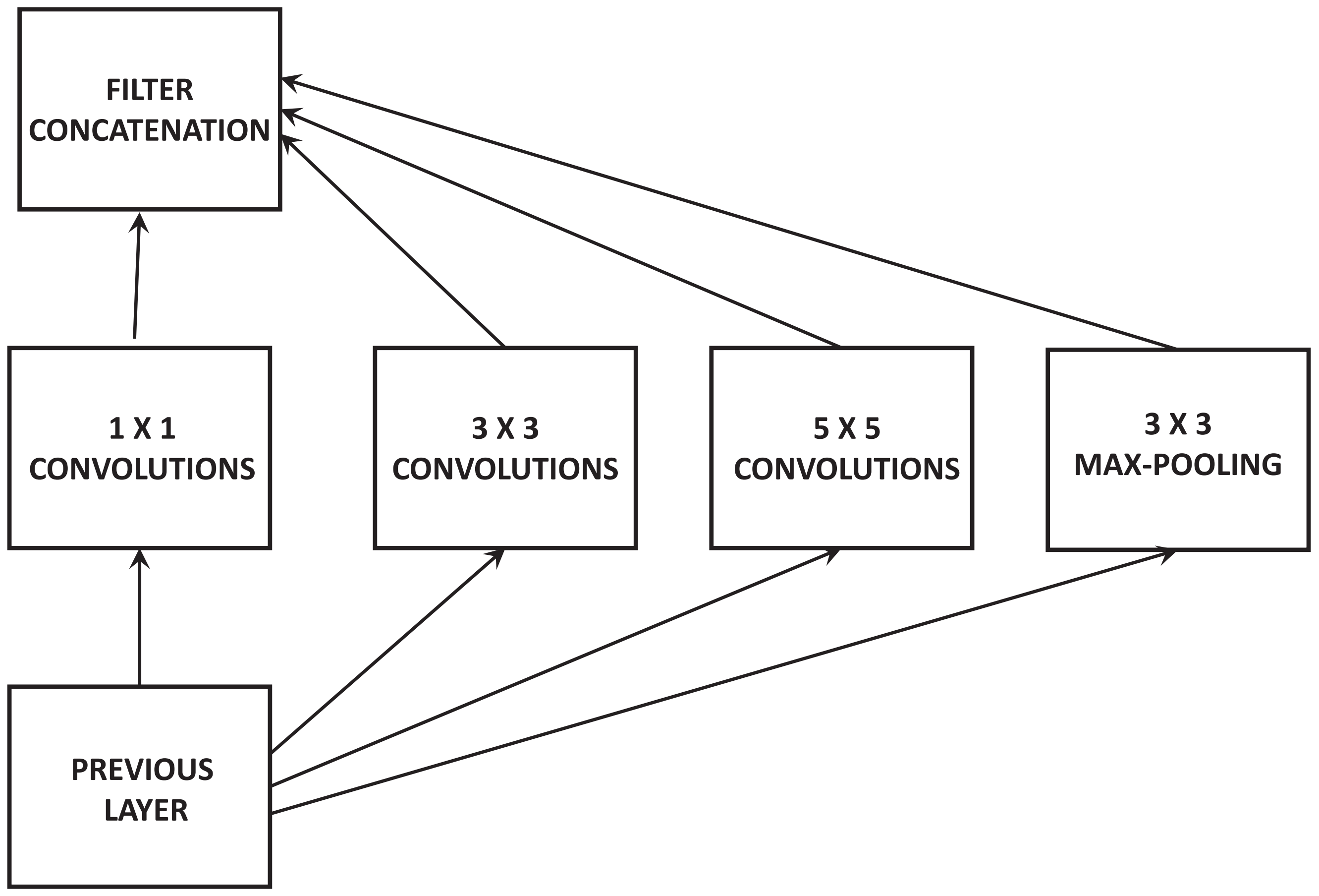
- Introduced the principle of *inception architecture*.
- The initial part of the architecture is much like a traditional convolutional network, and is referred to as the *stem*.
- The key part of the network is an intermediate layer, referred to as an *inception module*.
- Allows us to capture images at varying levels of detail in different portions of the network.

Inception Module: Motivation

- Key information in the images is available at different levels of detail.
  - Large filter can capture can information in a bigger area containing limited variation.
  - Small filter can capture detailed information in a smaller area.

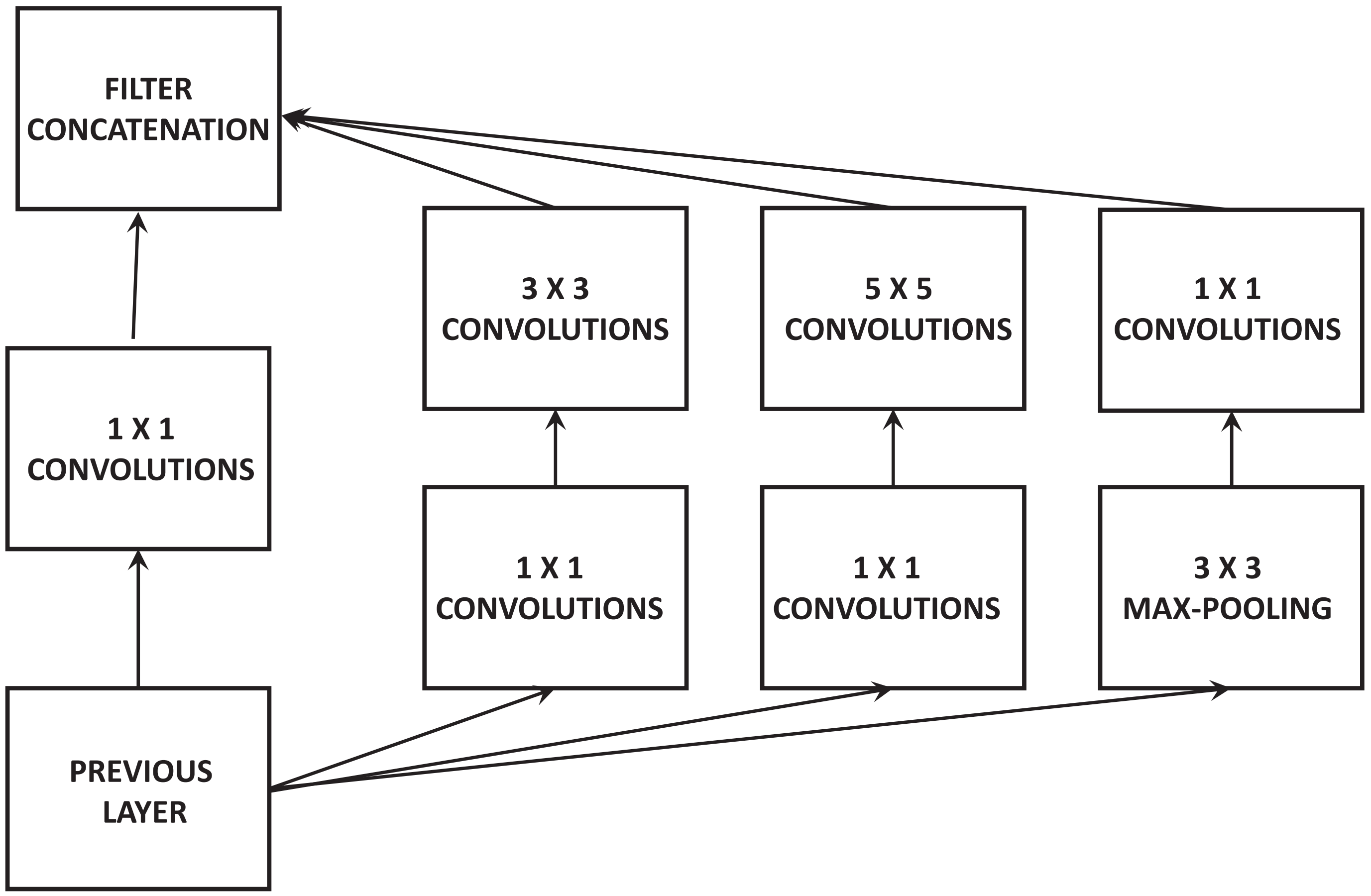
- Piping together many small filters is wasteful $\Rightarrow$ Why not let the neural network decide?

Basic Inception Module



- Main problem is computational efficiency $\Rightarrow$ First reduce depth

Computationally Efficient Inception Module



- Note the 1×1 filters

Design Principles of Output Layer

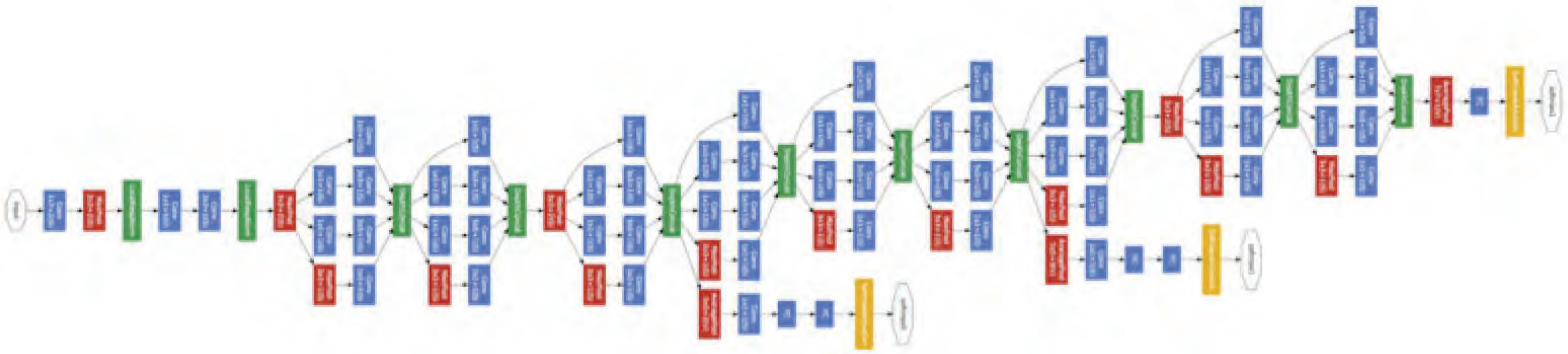
- It is common to use fully connected layers near the output.

- *GoogLeNet* uses average pooling across the whole spatial area of the final set of activation maps to create a single value.

- The number of features created in the final layer will be exactly equal to the number of filters.
  - Helps in reducing parameter footprint

- Detailed connectivity is not required for applications in which only a class label needs to be predicted.

GoogLeNet Architecture



Convolution
Pooling
Softmax
Other

Other Details on GoogLeNet

- Winner of ILSVRC 2014
- Reached top-5 error rate of 6.7%
- Contained 22 layers
- Several advanced variants with improved accuracy.
- Some have been combined with ResNet

ResNet Motivation

- Increasing depth has advantages but also makes the network harder to train.
- Even the error on the training data is high!
  - Poor convergence
- Selecting an architecture with unimpeded gradient flows is helpful!

ResNet

- ResNet brought the depth of neural networks into the hundreds.

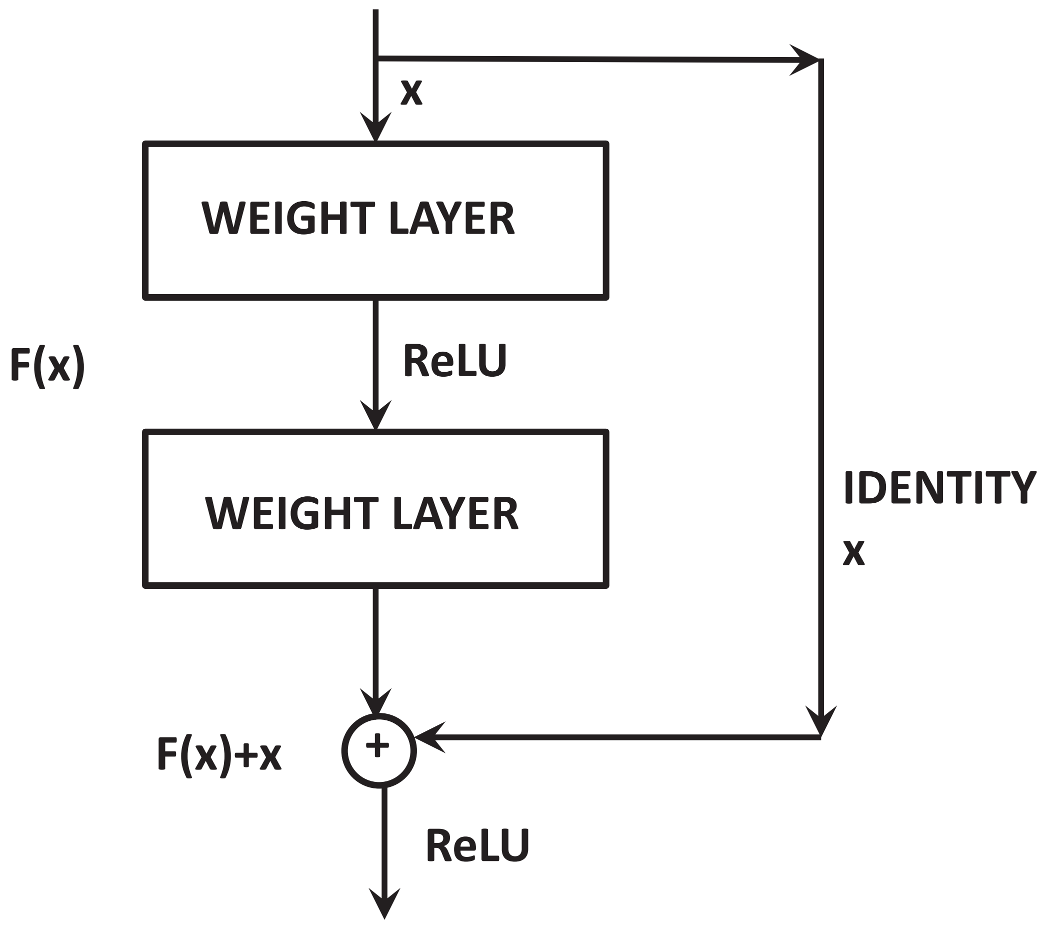
- Based on the principle of iterative feature engineering rather than hierarchical feature engineering.

- Principle of partially copying features across layers.

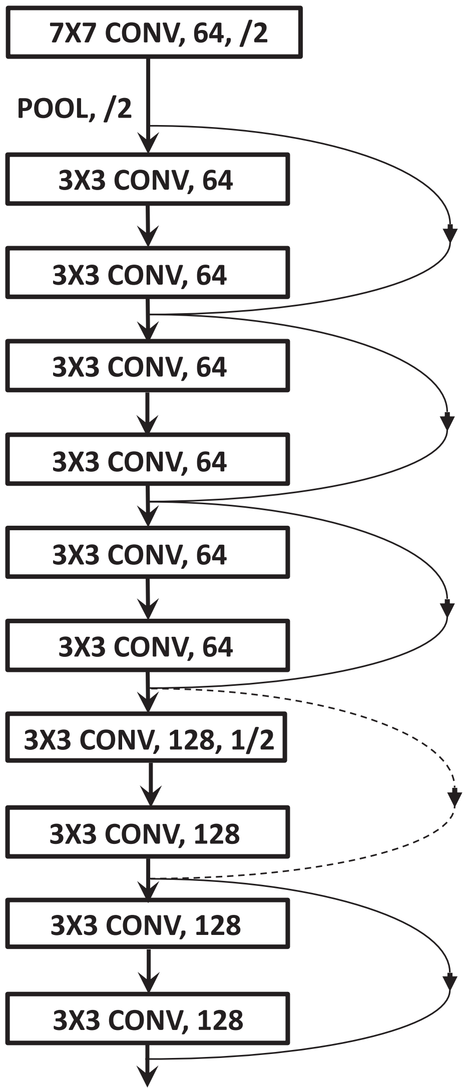
  – Where have we heard this before?

- Human-level performance with top-5 error rate of 3.6%.

Skip Connections in ResNet



(a) Residual module



(b) Partial architecture

Details of ResNet

- A 3×3 filter is used at a stride/padding of 1 $\Rightarrow$ Adopted from *VGG* and maintains dimensionality.

- Some layers use strided convolutions to reduce each spatial dimension by a factor of 2.
  - A linear projection matrix reduces the dimensionality.
  - The projection matrix defines a set of 1×1 convolution operations with stride of 2.

Importance of Depth

Name	**Year**	**Number of Layers**	**Top-5 Error**
-	Before 2012	≤ 5	$> 25\%$
AlexNet	2012	8	15.4%
ZfNet/Clarifai	2013	$8/>8$	14.8% / 11.1%
VGG	2014	19	7.3%
GoogLeNet	2014	22	6.7%
ResNet	2015	152	3.6%

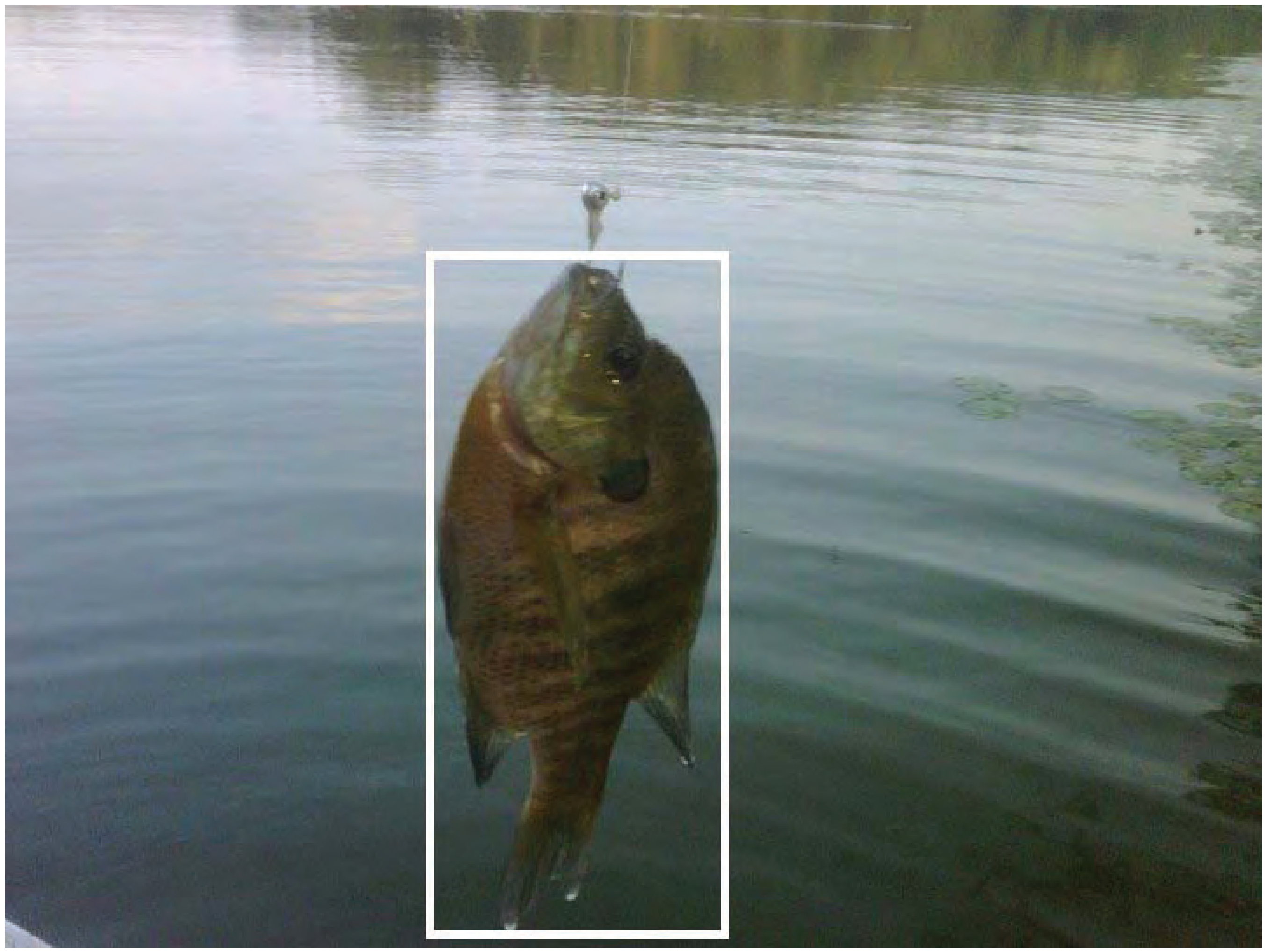
Pretrained Models

- Many of the models discussed in previous slides are available as pre-trained models with *ImageNet* data.

- One can use the features from fully connected layers for nearest neighbor search.
  - A nearest neighbor classifier on raw pixels vs features from fully connected layers will show huge differences.

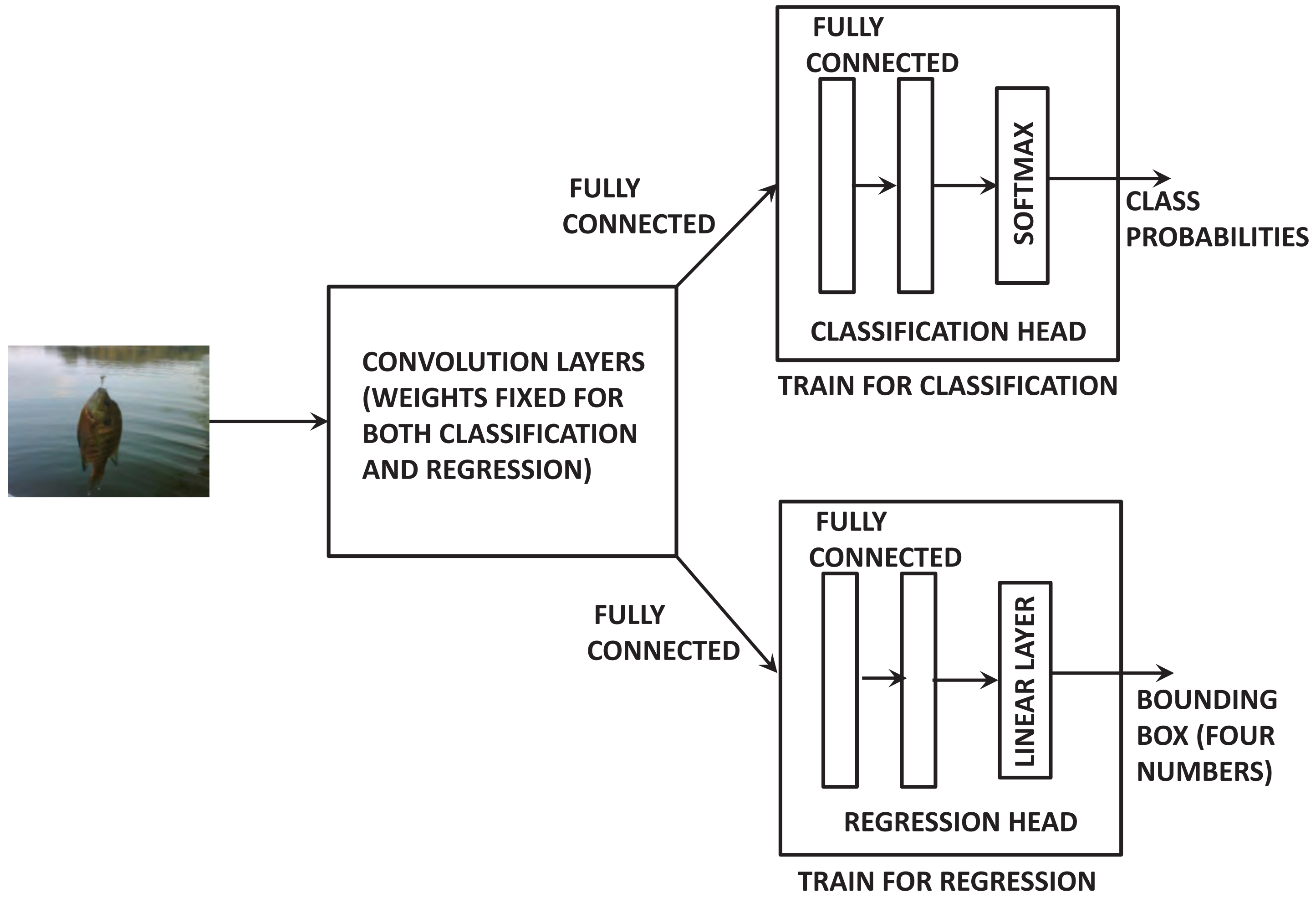
- One can train output layer for other applications like regression while retaining weights in other layers.

Object Localization



- Need to classify image together with bounding box (four numbers)

Object Localization



- Fully connected layers for classification and regression heads trained separately.

Other Applications

- Object detection: Multiple objects

- Text and sequence processing applications
  - 1-dimensional convolutions

- Video and spatiotemporal data
  - 3-dimensional convolutions

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Basic Principles of Reinforcement Learning [Motivating Deep Learning]

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.1–8.3

The Complexity of Human Intelligence is Quite Simple!

- Herbert Simon's ant hypothesis:

  "*Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves.*"

  - Humans are simple, because they are reward-driven entities.

  - All of biological intelligence is owed to this simple fact.

- Reinforcement learning attempts to simplify the learning of complex behaviors by using *reward-driven trial and error*.

When to Use Reinforcement Learning?

- Systems that are simple to judge but hard to specify.

- Easy to use trial-and-error to generate data.
  - Video games (e.g., Atari)
  - Board and card games (e.g., chess, Go, Poker)
  - Robot locomotion and visuomotor skills
  - Self-driving cars

- *Reinforcement learning is the gateway to general forms of artificial intelligence!*

Why Don't We have General Forms of Artificial Intelligence Yet?

- Reinforcement learning requires large amounts of data (generated by trial and error).
  - Possible to generate lots of data in some game-centric settings, but not other real-life settings.
- Biological reinforcement learning settings include some unsupervised learning.
  - The number of synapses in our brain is larger than the number of seconds we live!
  - There must be some unsupervised learning going on continuously $\Rightarrow$ We haven't mastered that art yet.
- Recent results do show promise for the future.

Simplest Reinforcement Learning Setting: Multi-armed Bandits

- Imagine a gambler in a casino faced with 2 slot machines.
- Each trial costs the gambler \$1, but pays \$100 with some unknown (low) probability.
- The gambler suspects that one slot machine is better than the other.
- What would be the optimal strategy to play the slot machines, assuming that the gambler's suspicion is correct?
- **Stateless Model:** Environment at every time-stamp is identical (although knowledge of *agent* improves).

Observations

- Playing both slot machines alternately helps the gambler learn about their payoff (over time).
  - However, it is wasteful *exploration*!
  - Gambler wants to *exploit* winner as soon as possible.
- Trade-off between exploration and exploitation $\Rightarrow$ Hallmark of reinforcement learning

Naïve Algorithm

- **Exploration:** Play each slot machine for a fixed number of trials.
- **Exploitation:** Play the winner forever.
  - Might require a large number of trials to robustly estimate the winner.
  - If we use too few trials, we might actually play the poorer slot machine forever.

ϵ-Greedy Strategy

- Probabilistically merge exploration and exploitation.
- Play a random machine with probability ϵ, and play the machine with highest current payoff with probability $1 - \epsilon$.
- Main challenge in picking the proper value of $\epsilon \Rightarrow$ Decides trade-off between exploration and exploitation.
- **Annealing:** Start with large values of ϵ and reduce slowly.

Upper Bounding: The Optimistic Gambler!

- Upper-bounding represents optimism towards unseen machines $\Rightarrow$ Encourages exploration.
- Empirically estimate mean μ_i and standard deviation σ_i of payoff of the ith machine using its n_i trials.
- Pick the slot machine with largest value of mean plus confidence interval $= \mu_i + K \cdot \sigma_i / \sqrt{n_i}$
  - Note the $\sqrt{n_i}$ in the denominator, because it is *sample* standard deviation.
  - Rarely played slot machines more likely to be picked because of optimism.
  - Value of K decides trade-off between exploration and exploitation.

Multi-Armed Bandits versus Classical Reinforcement Learning

- Multi-armed bandits is the simplest form of reinforcement learning.

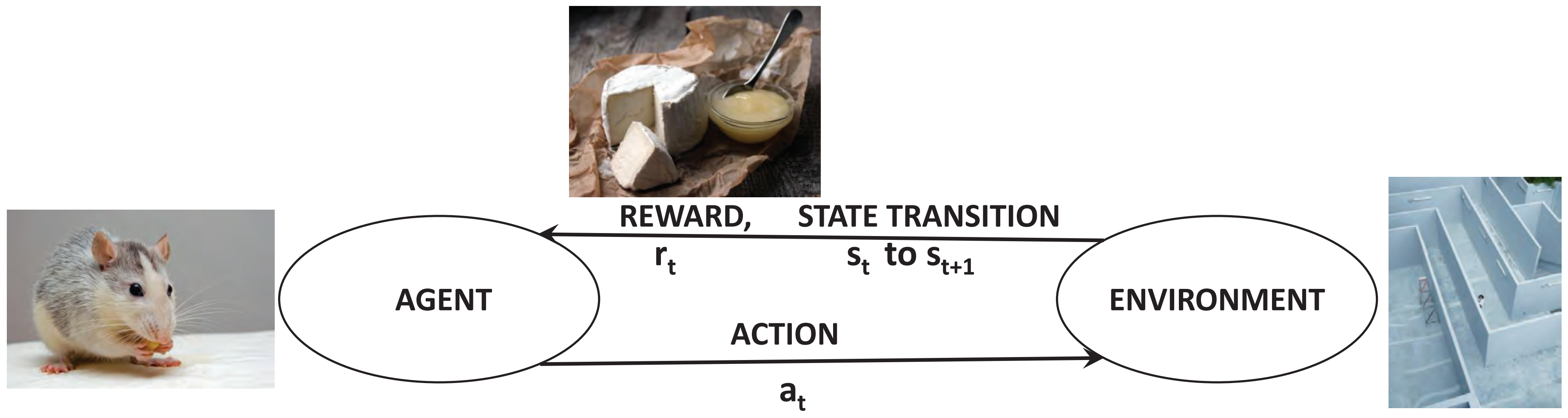
- The model is stateless, because the environment is identical at each time-stamp.

- Same action is optimal for each time-stamp.
  - Not true for classical reinforcement learning like Go, chess, robot locomotion, or video games.
  - *State* of the environment matters!

Markov Decision Process (MDP): Examples from Four Settings

- *Agent:* Mouse, chess player, gambler, robot
- *Environment:* maze, chess rules, slot machines, virtual test bed for robot
- *State:* Position in maze, chess board position, unchanged, robot joints
- *Actions:* Turn in maze, move in chess, pulling a slot machine, robot making step
- *Rewards:* cheese for mouse, winning chess game, payoff of slot machine, virtual robot reward

The Basic Framework of Reinforcement Learning



1. AGENT (MOUSE) TAKES AN ACTION a_t (LEFT TURN IN MAZE) FROM STATE (POSITION) s_t
2. ENVIRONMENT GIVES MOUSE REWARD r_t (CHEESE/NO CHEESE)
3. THE STATE OF AGENT IS CHANGED TO s_{t+1}
4. MOUSE'S NEURONS UPDATE SYNAPTIC WEIGHTS BASED ON WHETHER ACTION EARNED CHEESE

OVERALL: AGENT LEARNS OVER TIME TO TAKE STATE-SENSITIVE ACTIONS THAT EARN REWARDS

- The biological and AI frameworks are similar.

- MDP represented as $s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_n a_n r_n$

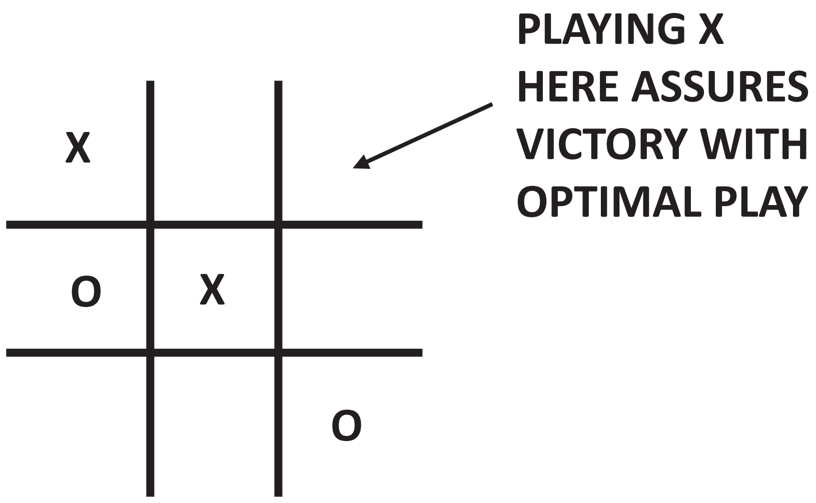
Examples of Markov Decision Process

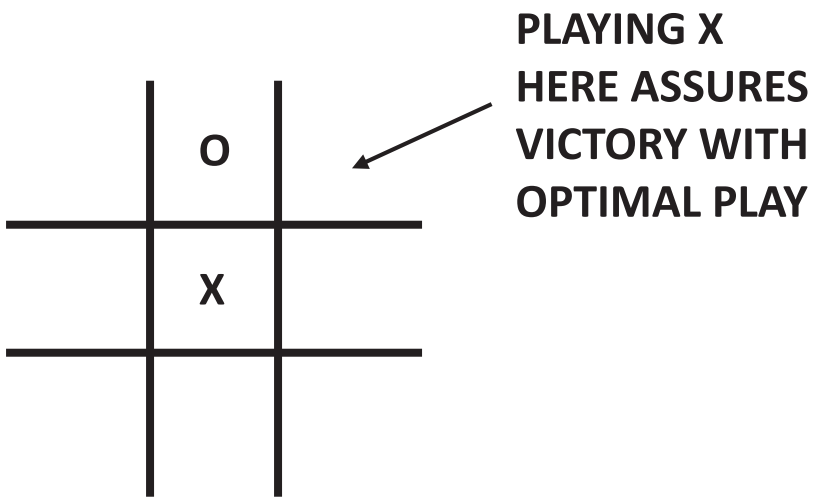
- *Game of tic-tac-toe, chess, or Go:* The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward is $+1$, 0, or -1 (depending on win, draw, or loss), *which is received at the end of the game*.

- *Robot locomotion:* The state corresponds to the current configuration of robot joints and its position. The actions correspond to the torques applied to robot joints. The reward at each time stamp is a function of whether the robot stays upright and the amount of forward movement.

- *Self-driving car:* The states correspond to the sensor inputs from the car, and the actions correspond to the steering, acceleration, and braking choices. The reward is a function of car progress and safety.

Role of Traditional Reinforcement Learning





- **Traditional reinforcement learning:** Learn through trial-and-error the *long-term* value of each state.

- Long-term values are not the same as rewards.
  - Rewards not realized immediately because of stochasticity (e.g., slot machine) or delay (board game).

Reinforcement Learning for Tic-Tac-Toe

- Main difference from multi-armed bandits is that we need to learn the long-term rewards for each action in each *state*.

- An eventual victory earns a reward from $\{+1, 0, -1\}$ with delay.

- A move occurring r moves earlier than the game termination earns *discounted* rewards of $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$.

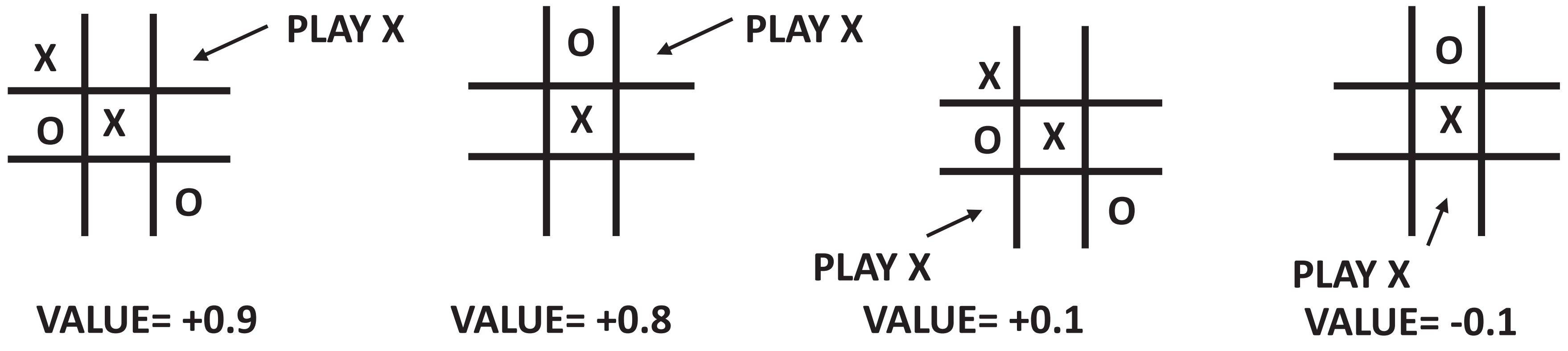
  – Future rewards would be less certain in a replay.

- Assume that a fixed pool of humans is available as opponents to train the system (self-play possible).

Generalizing ϵ-Greedy to Tic-Tac-Toe

- Maintain table of values of state-action pairs (initialize to small random values).
  - In multi-armed bandits, we only had values on actions.
  - Table contains unnormalized total reward for each state-action pair $\Rightarrow$ Normalize to average reward.
- Use ϵ-greedy algorithm with *normalized* table values to simulate moves and create a game.
- **After game:** Increment at most 9 entries in the unnormalized table with values from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ for r moves to termination and win/loss.
- Repeat the steps above.

At the End of Training



- Typical examples of normalized values of moves

- ϵ-greedy will learn the strategic values of moves.

- Rather than state-action-value triplets, we can equivalently learn state-value pairs.

Where Does Deep Learning Fit In?

- The tic-tac-toe approach is a glorified "learning by rote" algorithm.

- Works only for toy settings with few states.

  - Number of board positions in chess is huge.

  - Need to be able to *generalize* to unseen states.

  - **Function Approximator:** Rather than a table of state-value pairs, we can have a *neural network* that maps states to values.

  - The *parameters* in the neural network substitute for the table.

Strawman ϵ-Greedy Algorithm with Deep Learning for Chess [Primitive: Don't Try It!]

- Convolutional neural network takes board position as input and produces position value as output.

- Use ϵ-greedy algorithm on output values to simulate a full game.

- **After game of X moves:** Create X training points with board position as input feature map and targets from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ depending on move number and win/loss.

- Update neural network with these X training points.

- Repeat the steps above.

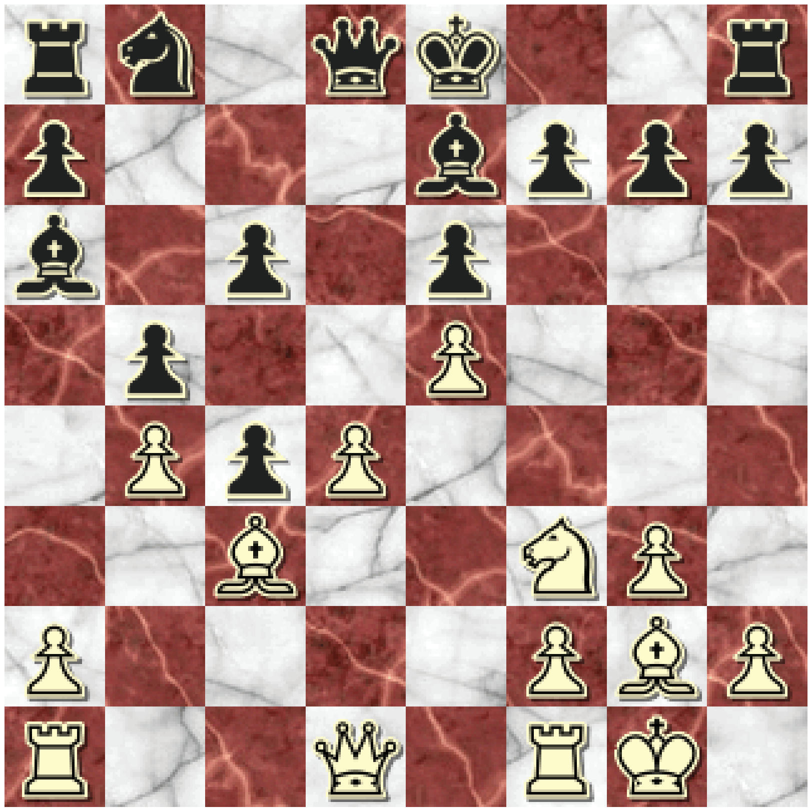
Reinforcement Learning in Chess and Go

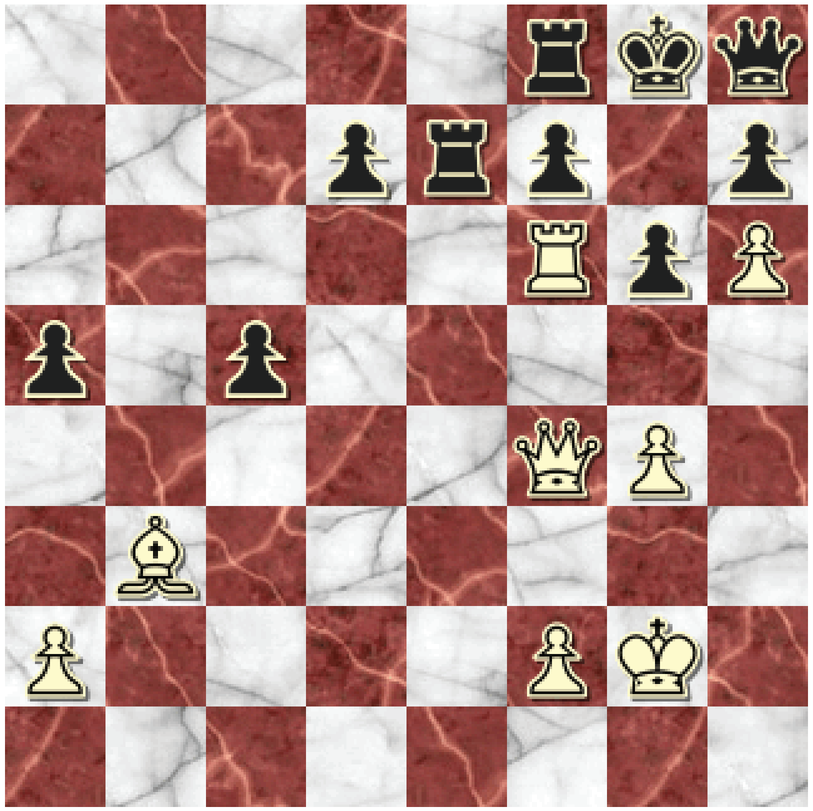
- The reinforcement learning systems, *AlphaGo* and *Alpha Zero*, have been designed for chess, Go, and shogi.

- Combines various advanced deep learning methods and Monte Carlo tree search.

- Plays positionally and sometimes makes sacrifices (much like a human).
  - Neural network encodes evaluation function learned from trial and error.
  - More complex and subtle than hand-crafted evaluation functions by conventional chess software.

Examples of Two Positions from Alpha Zero Games vs Stockfish





- Generalize to unseen states in training.
- Deep learner can recognize subtle positional factors because of trial-and-error experience with feature engineering.

Other Challenges

- Chess and tic-tac-toe are *episodic*, with a maximum length to the game (9 for tic-tac-toe and ≈6000 for chess).

- The ϵ-greedy algorithm updates episode by episode.

- What about infinite Markov decision processes like robots or long episodes?
  - Rewards received continuously.
  - Not optimal to update episode-by-episode.

- Value function and Q-function learning can update after each *step* with *Bellman's equations*.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Value Function Learning and Q-Learning

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.4

Challenges with Long and Infinite Markov Decision Processes

- Previous lecture discusses how value functions can be learned for shorter episodes.
  - Update state-action-value table for each episode with Monte Carlo simulation.

- Effective for games like tic-tac-toe with small episodes.

- What to do with continuous Markov decision processes?

An Infinite Markov Decision Process

- Sequence below is of infinite length (continuous process)

$$s_0 a_0 r_0 s_1 a_1 r_1 \ldots s_t a_t r_t \ldots$$

- The cumulative reward R_t at time t is given by the discounted sum of the immediate rewards for $\gamma \in (0, 1)$:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (1)$$

- Future rewards worth less than immediate rewards ($\gamma < 1$).

- Choosing $\gamma < 1$ is not essential for episodic processes but critical for long MDPs.

Recap of Episodic ϵ-Greedy for Tic-Tac-Toe

- Maintain table of average values of state-action pairs (initialize to small random values).

- Use ϵ-greedy algorithm with table values to simulate moves and create a game.

- **After game:** Update at most 9 entries in the table with new averages, based on the outcomes from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ depending on move number and win/loss.

- Repeat the steps above.

The Bootstrapping Intuition

- Consider a Markov decision process in which we are predicting values (e.g., long-term rewards) at each time-stamp.
  - A partial simulation of the future can improve the prediction at the current time-stamp.
  - This improved prediction can be used as the ground-truth at the current time stamp.

- **Tic-tac-toe:** Parameterized evaluation function for board.
  - After our opponent plays the next move, and board evaluation changes unexpectedly, we go back and correct parameters.

- **Temporal difference learning:** Use difference in prediction caused by partial lookaheads (treated as error for updates).

Example of Chess

- Why is the minimax evaluation of a chess program at 10-ply stronger than that using the 1-ply board evaluation?
  - Because evaluation functions are imperfect (can be strengthened by “cheating” with data from future)!
  - If chess were solved (like checkers today), the evaluation function at any ply would be the same.
  - The minimax evaluation at 10 ply can be used as a “ground truth” for updating a parameterized evaluation function at current position!
- Samuel’s checkers program was the pioneer (called *TD-Leaf* today)
- Variant of idea used by TD-Gammon, *Alpha Zero*.

Q-Learning

- Instead of minimax over a tree, we can use one-step lookahead

- Let $Q(s_t, a_t)$ be a table containing optimal values of state-action pairs (best value of action a_t in state s_t).

- Assume we play tic-tac-toe with ϵ-greedy and $Q(s_t, a_t)$ initialized to random values.

- Instead of Monte Carlo, make following update:

$$Q(s_t, a_t) = r_t + \gamma \mathrm{max}_a Q(s_{t+1}, a) \tag{2}$$

- Update: $Q(s_t, a_t) = Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma \mathrm{max}_a Q(s_{t+1}, a))$

Why Does this Work?

- Most of the updates we initially make are not meaningful in tic-tac-toe.
  - We started off with random values.
- However, the update of the value of a next-to-terminal state is informative.
- The next time the next-to-terminal state occurs on RHS of Bellman, the update of the next-to-penultimate state will be informative.
- Over time, we will converge to the proper values of all state-action pairs.

SARSA: ϵ-greedy Evaluation

- Let $Q(s_t, a_t)$ be the value of action a_t in state s_t when following the ϵ-greedy policy.

- **An improved estimate** of $Q(s_t, a_t)$ via bootstrapping is $r_t + \gamma Q(s_{t+1}, a_{t+1})$

- Follows from $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i} = r_t + \gamma R_{t+1}$

- SARSA: Instead of episodic update, we can update the table containing $Q(s_t, a_t)$ after performing a_t by ϵ-greedy, observing s_{t+1} and then computing a_{t+1} again using ϵ-greedy:

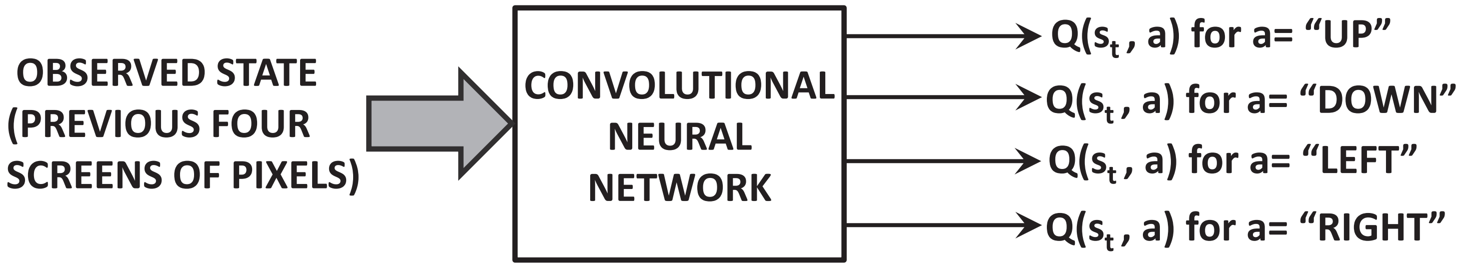
$$Q(s_t, a_t) \Leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) \tag{3}$$

- Gentler and stable variation: $Q(s_t, a_t) \Leftarrow Q(s_t, a_t)(1 - \alpha) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}))$

On-Policy vs Off-Policy Learning

- SARSA: On-policy learning is useful when learning and inference cannot be separated.
  - A robot who continuously learns from the environment.
  - The robot must be cognizant that exploratory actions have a cost (e.g., walking at edge of cliff).
- Q-learning: Off-policy learning is useful when we don't need to perform exploratory component during inference time (have non-zero ϵ during training but set to 0 during inference).
  - Tic-tac-toe can be learned once using Q-learning, and then the model is fixed.

Using Deep Learning

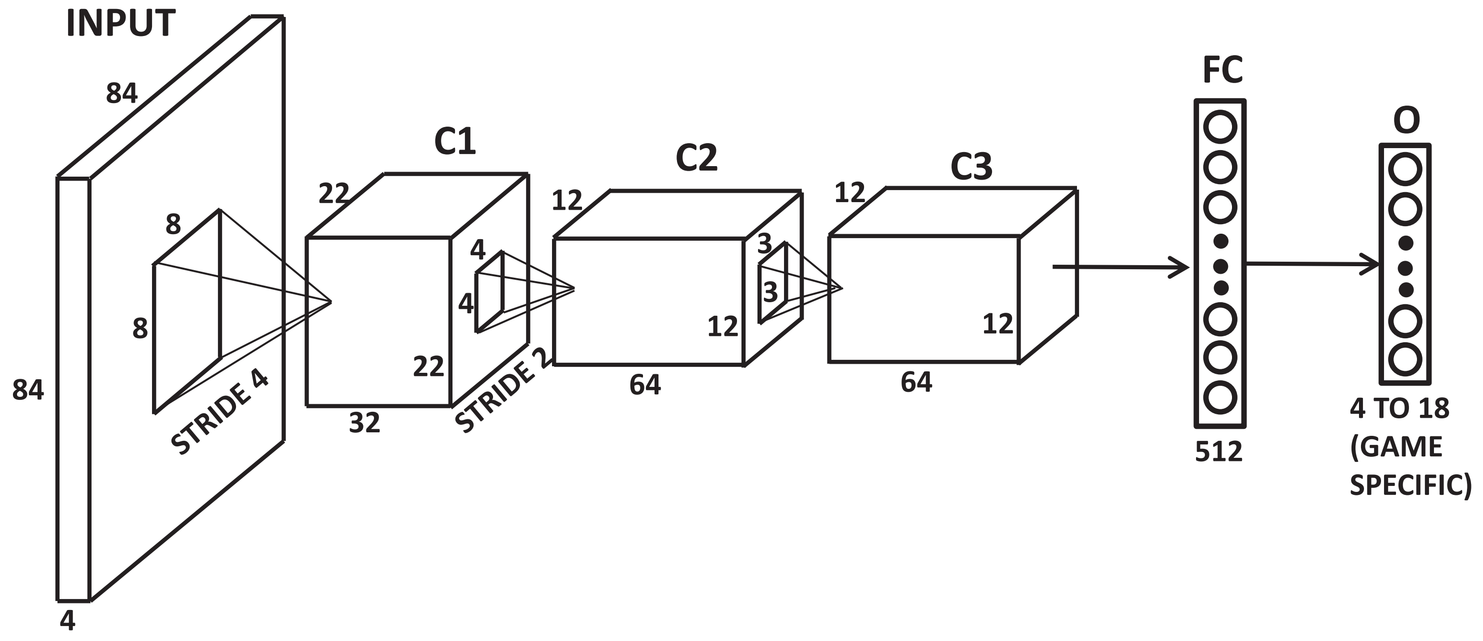


- When the number of states is large, the values $Q(s_t, a_t)$ are predicted from state s_t representation $\overline{X}_t$ rather than tabulated.

$$F(\overline{X}_t, \overline{W}, a) = \hat{Q}(s_t, a) \tag{4}$$

- $\overline{X}_t$: Previous four screens of pixels in Atari

Specific Details of Convolutional Network



- Same architecture with minor variations was used for all Atari games.

Neural Network Updates for Q-Learning

- The neural network outputs $F(\overline{X}_t, \overline{W}, a_t)$.

- We must wait to observe state $\overline{X}_{t+1}$ and then set up a "ground-truth" value for the output using Bellman's equations:

$$\text{Bootstrapped Ground-Truth} = r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a) \tag{5}$$

- Loss: $L_t = \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2$

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma \text{max}_a F(\overline{X}_{t+1}, \overline{W}, a)]}_{\text{Constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \tag{6}$$

Neural Network Updates for SARSA

- The neural network outputs $F(\overline{X}_t, \overline{W}, a_t)$.

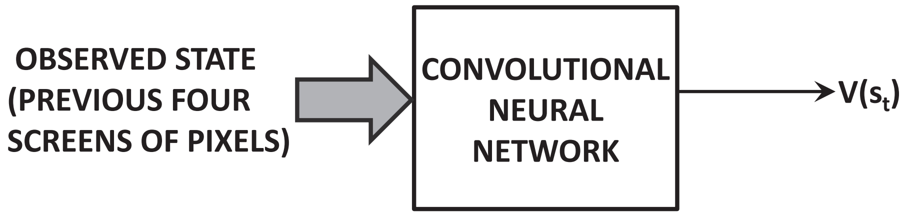
- We must wait to observe state $\overline{X}_{t+1}$, simulate a_{t+1} with ϵ-greedy and then set up a "ground-truth" value:

$$\text{Bootstrapped Ground-Truth} = r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1}) \quad (7)$$

- Loss: $L_t = \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Treat as constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\}^2$

$$\overline{W} \Leftarrow \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma F(\overline{X}_{t+1}, \overline{W}, a_{t+1})]}_{\text{Constant ground-truth}} - F(\overline{X}_t, \overline{W}, a_t) \right\} \frac{\partial F(\overline{X}_t, \overline{W}, a_t)}{\partial \overline{W}} \quad (8)$$

Value Function Learning


OBSERVED STATE (PREVIOUS FOUR SCREENS OF PIXELS)
CONVOLUTIONAL NEURAL NETWORK
V(s_t)

- Instead of outputting values of state-action pairs we can output just values.

- Q-Learning and SARSA can be implemented with this architecture as well.
  - General class of temporal difference learning $\Rightarrow$ Multi-step bootstrapping
  - Can explore a forward-looking tree for arbitrary bootstrapping.

Temporal Difference Learning $TD(0)$

- Value network produces $G(\overline{X}_t, \overline{W})$ and bootstrapped ground truth $= r_t + \gamma G(\overline{X}_{t+1}, \overline{W})$

- Same as SARSA: Observe next state by executing a_t according to current policy

- Loss: $L_t = \left\{ \underbrace{r_t + \gamma G(\overline{X}_{t+1}, \overline{W})}_{\text{“Observed” value}} - G(\overline{X}_t, \overline{W}) \right\}^2$

$$\overline{W} = \overline{W} + \alpha \left\{ \underbrace{[r_t + \gamma G(\overline{X}_{t+1}, \overline{W})]}_{\text{“Observed” value}} - G(\overline{X}_t, \overline{W}) \right\} \frac{\partial G(\overline{X}_t, \overline{W})}{\partial \overline{W}} \tag{9}$$

- Short notation: $\overline{W} \Leftarrow \overline{W} + \alpha \delta_t (\nabla G(\overline{X}_t, \overline{W}))$

Bootstrapping over Multiple Steps

- Temporal difference bootstraps only over one time-step.
  - A strategically wrong move will not show up immediately.
  - Can look at n-steps instead of one.
- On-policy looks at single sequence greedily (too weak)
- Off-policy (like Bellman) picks optimal over entire minimax tree (Samuel's checkers program).
- Any optimization heuristic for lookahead-based inference can be exploited.
  - Monte Carlo tree search explores *multiple branches* with upper-bounding strategy $\Rightarrow$ Statistically robust target.

Fixed Window vs Smooth Decay: Temporal Difference Learning $TD(\lambda)$

- Refer to one-step temporal difference learning as $TD(0)$
- Fixed Window n: $\overline{W} \Leftarrow \overline{W} + \alpha\delta_t \sum_{k=t-n+1}^{t} \gamma^{t-k}(\nabla G(\overline{X}_k, \overline{W}))$
- $TD(\lambda)$ corrects past mistakes with discount factor λ when new information is received.

$$\overline{W} \Leftarrow \overline{W} + \alpha\delta_t \sum_{k=0}^{t} (\lambda\gamma)^{t-k}(\nabla G(\overline{X}_k, \overline{W})) \qquad (10)$$

- Setting $\lambda = 1$ or $n = \infty$ is equivalent to Monte Carlo methods.
  - Details in book.

Monte Carlo vs Temporal Differences

- Not true that greater lookahead always helps!
  - The value of λ in $TD(\lambda)$ regulates the trade-off between bias and variance.
  - Using small values of λ is particularly advisable if data is limited.
- A temporal difference method places a different value on each position in a single chess game (that depends on the merits of the position).
  - Monte Carlo places a value that depends only on time discounting and final outcome.

Monte Carlo vs Temporal Differences: Chess Example

- Imagine a Monte Carlo rollout of chess game between two agents Alice and Bob.
  - Alice and Bob each made two mistakes but Alice won.
  - Monte Carlo training data does not differentiate between mistakes and good moves.
  - Using temporal differences might see an error after each mistake because an additional ply has *differential* insight about the effect of the move (bootstrapping).
- More data is needed in Monte Carlo rollouts to remove the effect of noise.
- On the other hand, TD(0) might favor learning of end games over openings.

Implications for Other Methods

- Policy gradients often use Monte Carlo rollouts.
  - Deciding the *advantage* of an action is often difficult in games without continuous rewards.
- Value networks are often used in combination with *policy-gradients* in order to design *actor-critic* methods.
  - Temporal differences are used to evaluate the advantage of an action.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Policy Gradients

Neural Networks and Deep Learning, Springer, 2018
Chapter 8.5

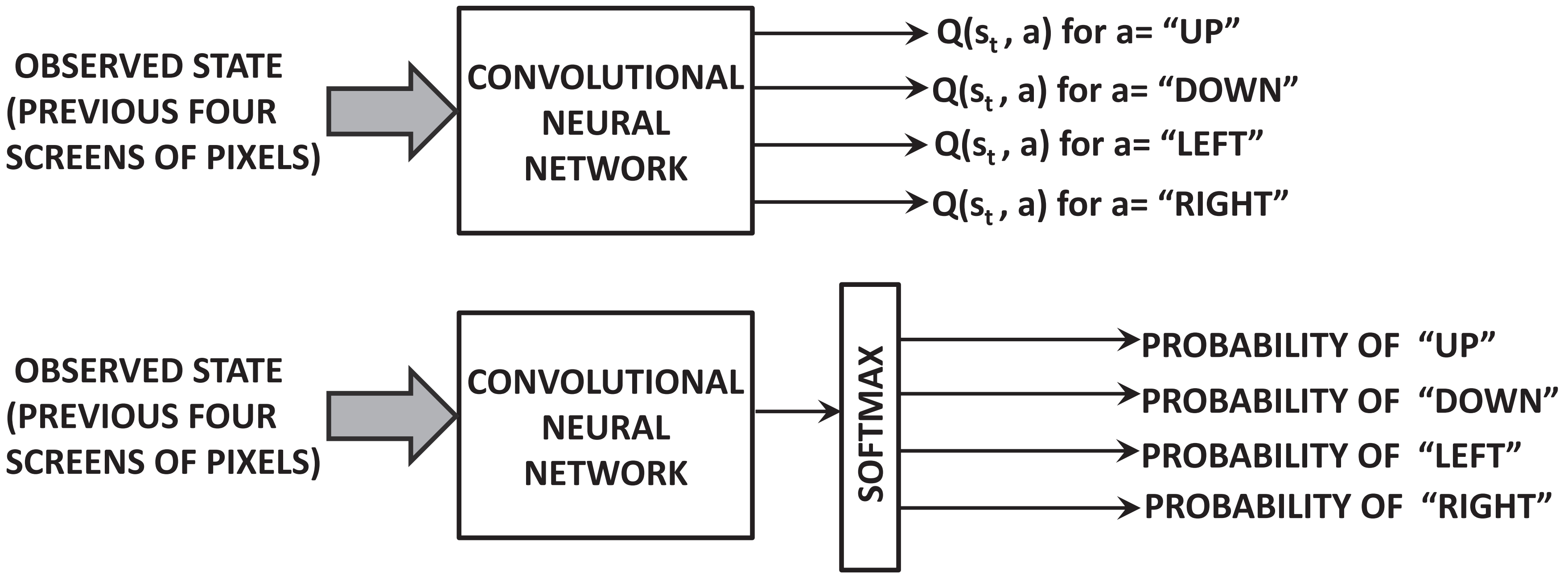
Difference from Value-Based Methods

- Value-based methods like Q-learning attempt to predict the value of an action with a *parameterized value function*.
    - Often coupled with a generic policy (like ϵ-greedy).

- Policy gradient methods estimate the *probability* of each action at each step with the goal of maximizing the overall reward.

- Policy is itself parameterized.

Policy Network vs Q-Network for Atari Game


OBSERVED STATE (PREVIOUS FOUR SCREENS OF PIXELS)
CONVOLUTIONAL NEURAL NETWORK
$Q(s_t, a)$ for a= "UP"
$Q(s_t, a)$ for a= "DOWN"
$Q(s_t, a)$ for a= "LEFT"
$Q(s_t, a)$ for a= "RIGHT"
OBSERVED STATE (PREVIOUS FOUR SCREENS OF PIXELS)
CONVOLUTIONAL NEURAL NETWORK
SOFTMAX
PROBABILITY OF "UP"
PROBABILITY OF "DOWN"
PROBABILITY OF "LEFT"
PROBABILITY OF "RIGHT"

- Output is *probability* of each action in policy network rather than *value* of each action.

Overview of Approach

- We want to update network to maximize *expected* future rewards $\Rightarrow$ We need to collect samples of long-term rewards for each simulated action.
  - **Method 1:** Use Monte Carlo policy rollouts to estimate the *simulated* long-term reward after each action.
  - **Method 2:** Use another value network to *model* long-term reward after each action (actor-critic methods).
- Main problem is in setting up a loss function that uses the simulated or modeled rewards to update the parameterized probabilities.

Example: Generating Training Data

- Training chess agent Alice (using pool of human opponents) with reward in $\{+1, 0, -1\}$.

- Consider a Monte Carlo simulation with win for agent Alice.

- Create training points for each board position faced by Alice and each action output a with long-term reward of 1.
  - If discount factor of γ then long-term reward is γ^{r-1}.

- **Backpropagated stochastic gradient ascent:** Somehow need to update neural network from samples of rewards to maximize expected rewards (non-obvious).

Nature of Training Data

- We have board positions together with output action *samples* and long-term rewards of each *sampled* action.
  - We do not have ground-truth *probabilities*.
- So we want to maximize *expected* long-term rewards from *samples* of the probabilistic output.
- How does one compute the gradient of an expectation from samples?

Log Probability Trick

- Let $Q^p(s_t, a)$ be the long-term reward of action a and policy p.

- The log probability trick of REINFORCE relates gradient of expectation to expectation of gradient:

$$\nabla E[Q^p(s_t, a)] = E[Q^p(s_t, a)\nabla \text{log}(p(a))] \tag{11}$$

- $Q^p(s_t, a)$ is estimated by Monte-Carlo roll-out and $\nabla\text{log}(p(a))$ is the log-likelihood gradient from backpropagation in the policy network for sampled action a.

$$\overline{W} \Leftarrow \overline{W} + \alpha Q^p(s_t, a)\nabla \text{log}(p(a)) \tag{12}$$

Baseline Adjustments

- Baseline adjustments change the reward into an "advantage" with the use of state-specific adjustments.
  - Subtract some quantity b from each $Q(s_t, a)$
  - Reduces variance of result without affecting bias.
- **State-independent:** One can choose b to be some long-term average of $Q^p(s_t, a)$.
- **State-specific:** One can choose b to be the value $V^p(s_t)$ of state $s_t \Rightarrow$ Advantage is same as temporal difference!

Why Does a State-Specific Baseline Adjustment Make Sense?

- Imagine a self-play chess game in which both sides make mistakes but one side wins.
  - Without baseline adjustments all training points from the game will have long-term rewards that depend only on final results.
  - Temporal difference will capture the *differential* impact of the error made in each action.
  - Gives more refined idea of the specific effect of that move $\Rightarrow$ Its advantage!

Problems with Monte Carlo Policy Gradients

- Full Monte Carlo simulation is best for episodic processes.

- Actor-critic methods allow online updating by combining ideas in policy gradients and value networks:

  - **Value-based:** The policy (e.g., ϵ-greedy) of the actor is subservient to the critic.

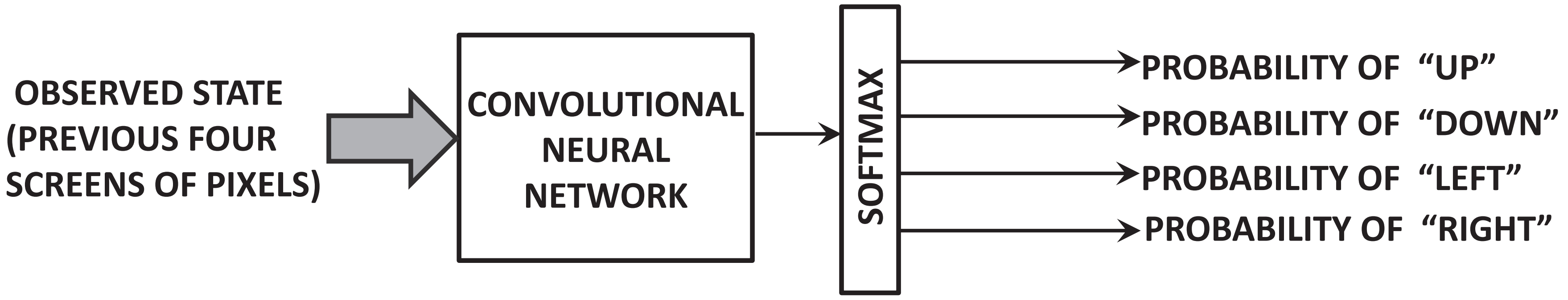
  - **Policy-based:** No notion of critic for value estimation (typical approach is Monte Carlo)

- **Solution:** Create separate neural network for estimating value/advantage.

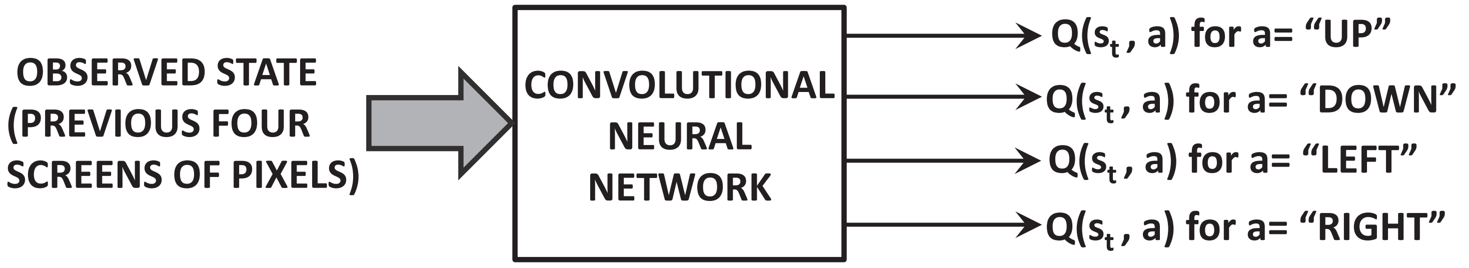
Actor-Critic Method

- Two separate neural networks:
  - **Actor:** Policy network with parameters $\overline{\Theta}$ that decides actions.
  - **Critic:** Value network or Q-network with parameter $\overline{W}$ that estimates long-term reward/advantage $\Rightarrow$ Advantage is temporal difference.
- The networks are trained simultaneously within an iterative loop.

Actor and Critic



(a) Actor (Decides actions as a probabilistic policy)


OBSERVED STATE (PREVIOUS FOUR SCREENS OF PIXELS)
CONVOLUTIONAL NEURAL NETWORK
$Q(s_t, a)$ for a= "UP"
$Q(s_t, a)$ for a= "DOWN"
$Q(s_t, a)$ for a= "LEFT"
$Q(s_t, a)$ for a= "RIGHT"

(b) Critic (Evaluates advantage in terms of temporal differences)

Steps in Actor-Critic Methods

- Sample the action a_{t+1} at state s_{t+1} using the policy network.

- Use Q-network to compute temporal difference error δ_t at s_t using bootstrapped target derived from value of s_{t+1}.

- **[Update policy network parameters]:** Update policy network using the Q-value of action a_t as its advantage (use temporal difference error for variance reduction).

- **[Update Q-Network parameters]:** Update the Q-network parameters using the squared temporal difference δ_t^2 as the error.

- Repeat the above updates.

Advantages and Disadvantages of Policy Gradients

- Advantages:
  - Work in continuous action spaces.
  - Can be used with stochastic policies.
  - Stable convergence behavior

- Main disadvantage is that they can reach local optima.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Attention Mechanisms

Neural Networks and Deep Learning, Springer, 2018
Chapter 10.2

The Biological Motivation

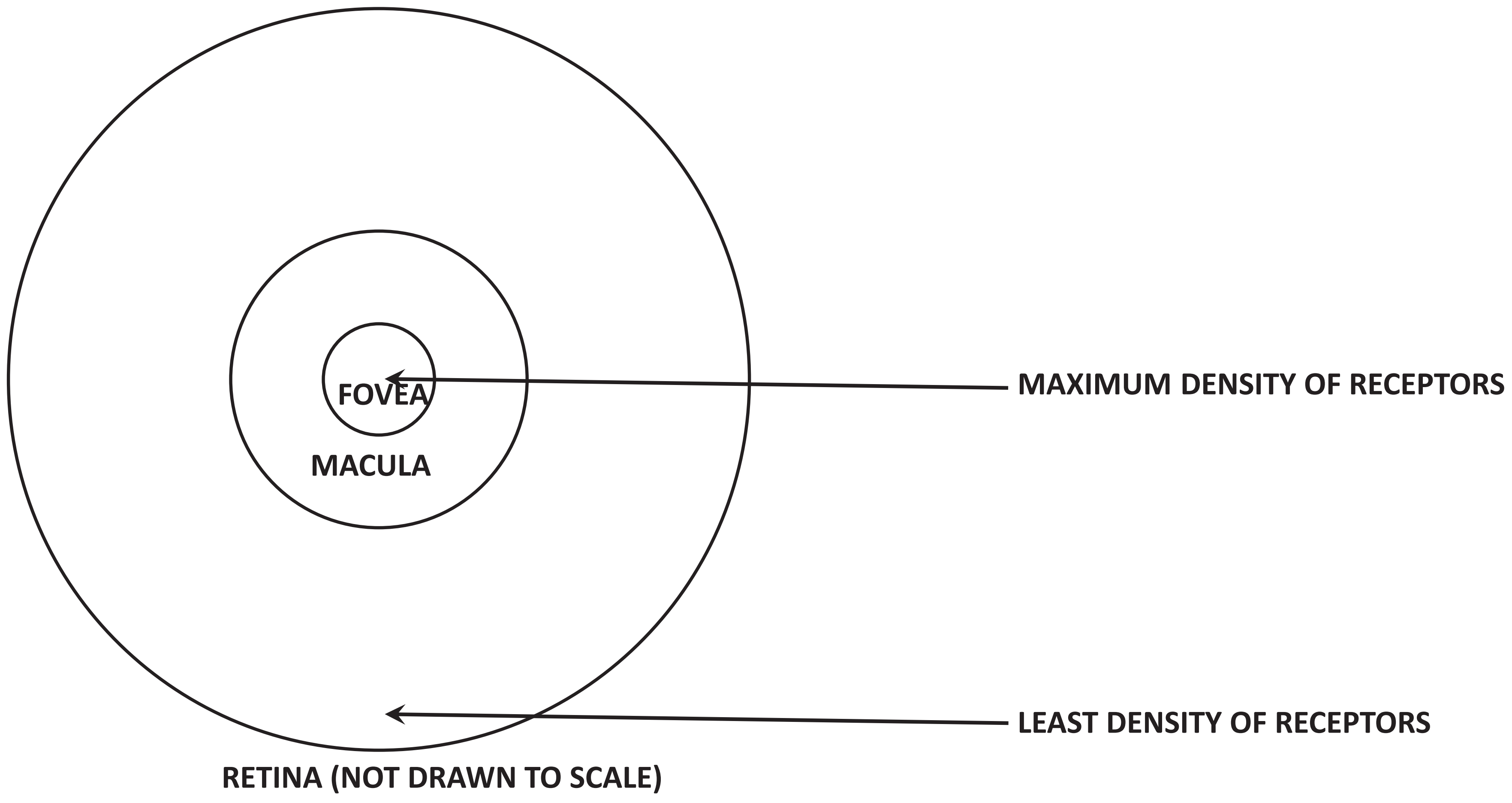
- Human beings rarely use all the available sensory inputs in order to accomplish specific tasks.
  - Problem of finding an address defined by a specific house number on a street.

- An important component of the task is to identify the number written either on the door or the mailbox of a house.

- The retina often has an image of a broader scene, although one rarely focuses on the full image.

- One pays greater *attention* to the *relevant* parts of the image.

The Notion of Attention in the Retina


FOVEA
MACULA
MAXIMUM DENSITY OF RECEPTORS
LEAST DENSITY OF RECEPTORS
RETINA (NOT DRAWN TO SCALE)

- Only a small portion of the image in the retina is carried in high resolution.

How Does the Process Work?

- Need to systematically focus on small parts of the image to find what one is looking for.

- Biological organisms draw quick visual cues from whatever they are focusing on in order to identify *where to next look* to get what they want.

  – If we first focus on the door knob by chance, then we know from experience (i.e., our trained neurons tell us) to look to its upper left or right to find the street number.

- Neurons were trained by past trial-and-error $\Rightarrow$ Use reinforcement learning methods.

- Some attention-based methods are paired with reinforcement learning.

Recurrent Models of Visual Attention

- Use a simple neural network in which only the resolution of specific portions of the image centered at a particular location is high.

- This location can change with time, as the model learns more about the relevant portions of the image.
  - Selecting a particular location in a given time-stamp is referred to as a *glimpse*.

- A recurrent neural network is used as the controller to identify the precise location in each time-stamp.
  - This choice is based on the feedback from the glimpse in the previous time-stamp.

Components of Neural Architecture

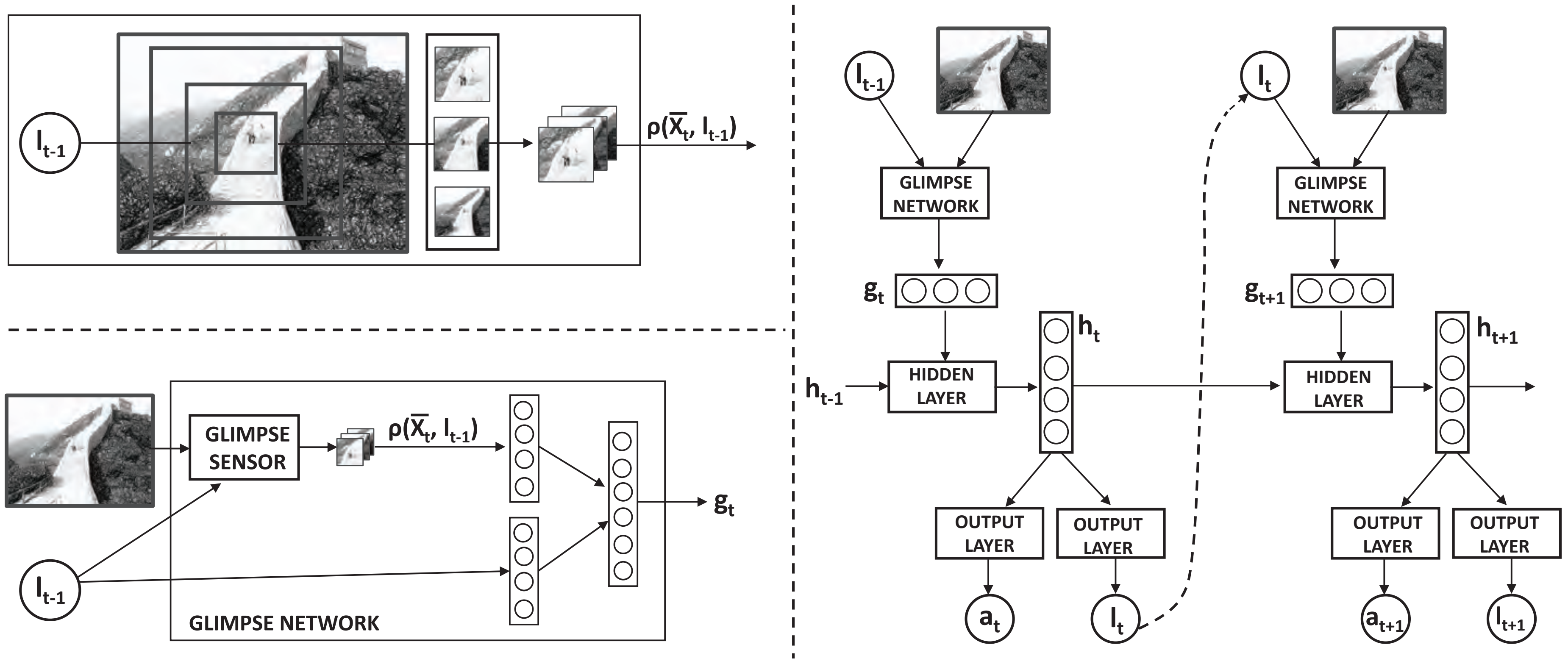
- *Glimpse Sensor:* Creates a retina-like representation $\rho(\overline{X}_t, l_{t-1})$ of the image $\overline{X}_t$ based on location l_{t-1}.

- *Glimpse Network:* The glimpse network contains the glimpse sensor and encodes both the glimpse location l_{t-1} and the glimpse representation $\rho(\overline{X}_t, l_{t-1})$ into hidden spaces.
  - Key image-processing component that is much simpler than a convolutional neural network.

- *Recurrent Neural Network:* The recurrent neural network outputs locations l_t for the next time stamp.

- **Important result:** The relatively simple glimpse network is able to outperform a convolutional neural network because of the attention mechanism.

Neural Architecture



Reinforcement Learning

- Action corresponds to choosing the class label at each time-stamp.

- The reward at time-stamp t is 1 if the classification is correct after t time-stamps.

  - Sum up discounted rewards over all time stamps.

  - Common to subtract baseline to reduce variance.

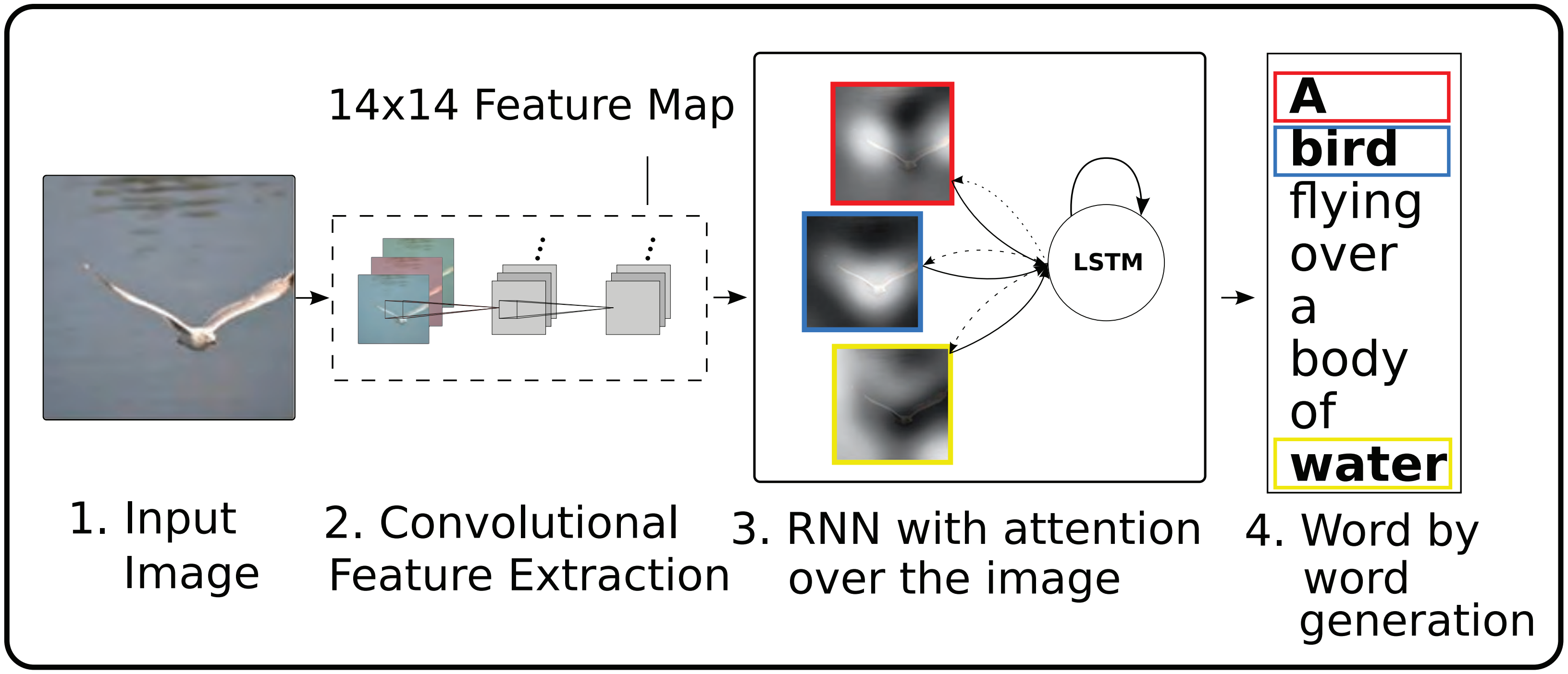
- Trained using the REINFORCE framework (policy gradients).

- Outperforms a convolutional neural network in spite of relatively simple architecture within the glimpse network and T between 6 and 8.

Image Captioning (Xu et al.)

- Modified version of classification framework.
- Instead of using glimpse sensor outputting location l_t, we use L preprocessed variants centered at different positions.
- Reinforcement learning selects one of these L actions (discrete output).
- The output at the next time stamp is the subsequent word in the image caption.
- Reward based on caption prediction accuracy.

Image Captioning Architecture


14x14 Feature Map
LSTM
A
bird
flying
over
a
body
of
water
1. Input Image
2. Convolutional Feature Extraction
3. RNN with attention over the image
4. Word by word generation

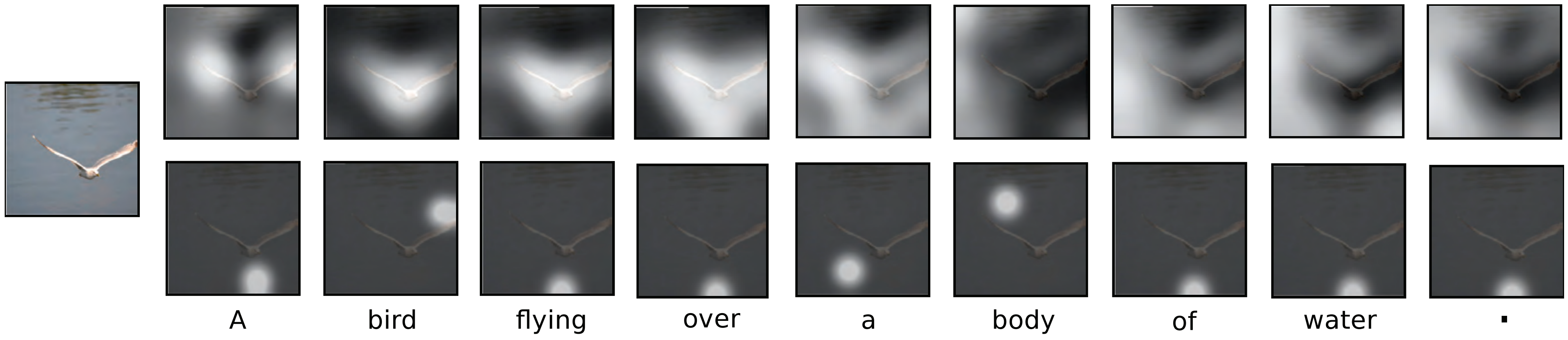
- K. Xu *et al.* Show, attend, and tell: Neural image caption generation with visual attention. *International Conference on Machine Learning*, 2015.

Hard Attention versus Soft Attention

- Hard attention selects specific locations.
  - Uses reinforcement learning because of hard selection of locations.

- Soft attention gives soft weights to various locations.
  - More conventional models (later slides).

Examples of Attention Locations

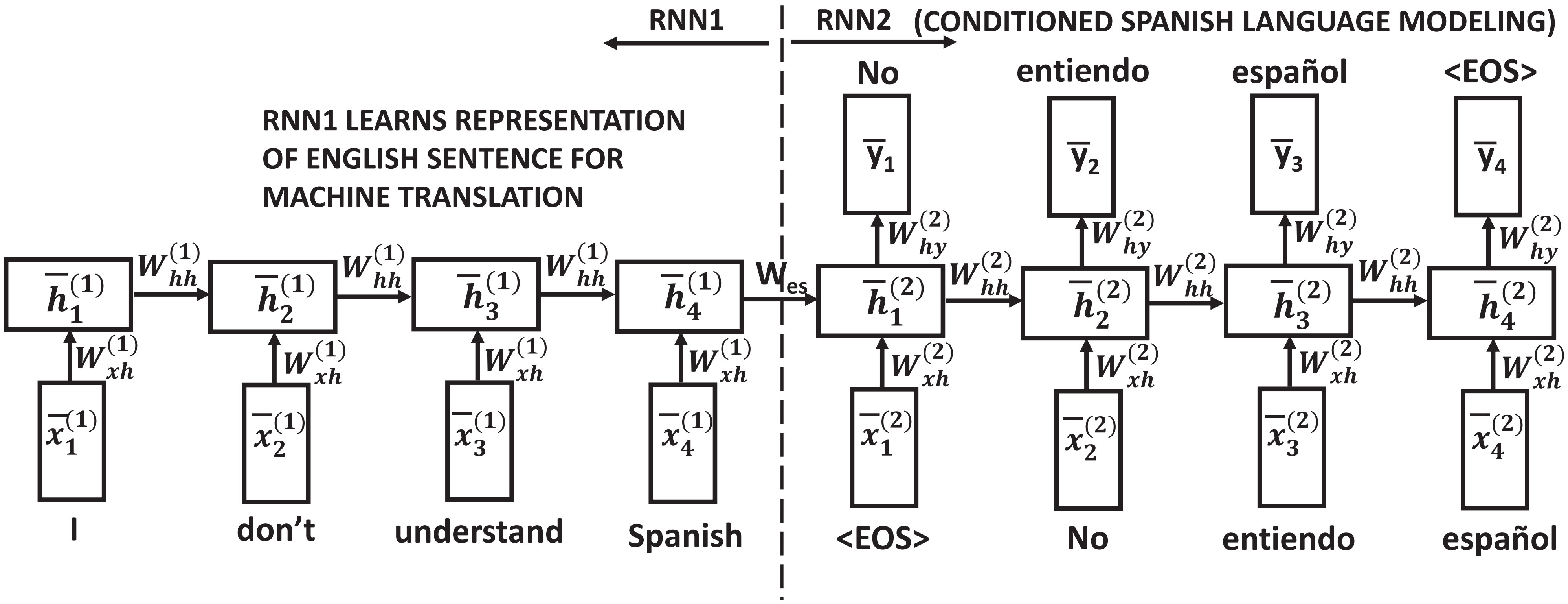


- K. Xu *et al.* Show, attend, and tell: Neural image caption generation with visual attention. *International Conference on Machine Learning*, 2015.

Application to Machine Translation

- A basic machine translation model hooks up two recurrent neural networks.
  - Typically, an advanced variant like LSTM is used.
  - Show basic version for simplicity.
- An attention model focuses on small portions of the sentence while translating a word.
- Use soft attention model in which the individual words are weighted.

The Basic Machine Translation Model



RNN1
RNN2 (CONDITIONED SPANISH LANGUAGE MODELING)
RNN1 LEARNS REPRESENTATION OF ENGLISH SENTENCE FOR MACHINE TRANSLATION
$\overline{h}_1^{(1)}$ $\overline{h}_2^{(1)}$ $\overline{h}_3^{(1)}$ $\overline{h}_4^{(1)}$
$W_{hh}^{(1)}$ $W_{xh}^{(1)}$
$\overline{x}_1^{(1)}$ $\overline{x}_2^{(1)}$ $\overline{x}_3^{(1)}$ $\overline{x}_4^{(1)}$
I
don't
understand
Spanish
W_{es}
$\overline{h}_1^{(2)}$ $\overline{h}_2^{(2)}$ $\overline{h}_3^{(2)}$ $\overline{h}_4^{(2)}$
$W_{hh}^{(2)}$ $W_{xh}^{(2)}$ $W_{hy}^{(2)}$
$\overline{y}_1$ $\overline{y}_2$ $\overline{y}_3$ $\overline{y}_4$
No
entiendo
español
<EOS>
$\overline{x}_1^{(2)}$ $\overline{x}_2^{(2)}$ $\overline{x}_3^{(2)}$ $\overline{x}_4^{(2)}$
<EOS>
No
entiendo
español

- Details discussed in lecture on applications of RNNs.

What Does Attention Do?

- The hidden states $h_t^{(2)}$ are transformed to enhanced states $H_t^{(2)}$ with some additional processing from an *attention layer*.
  - Attention layer incorporates context from the source hidden states into the target hidden states.

- Find a source representation that is close to the current target hidden state $h_t^{(2)}$ being processed.

- Use similarity-weighted average of the source vectors to create a context vector $\overline{c}_t$.

Context Vector and Attention Layer

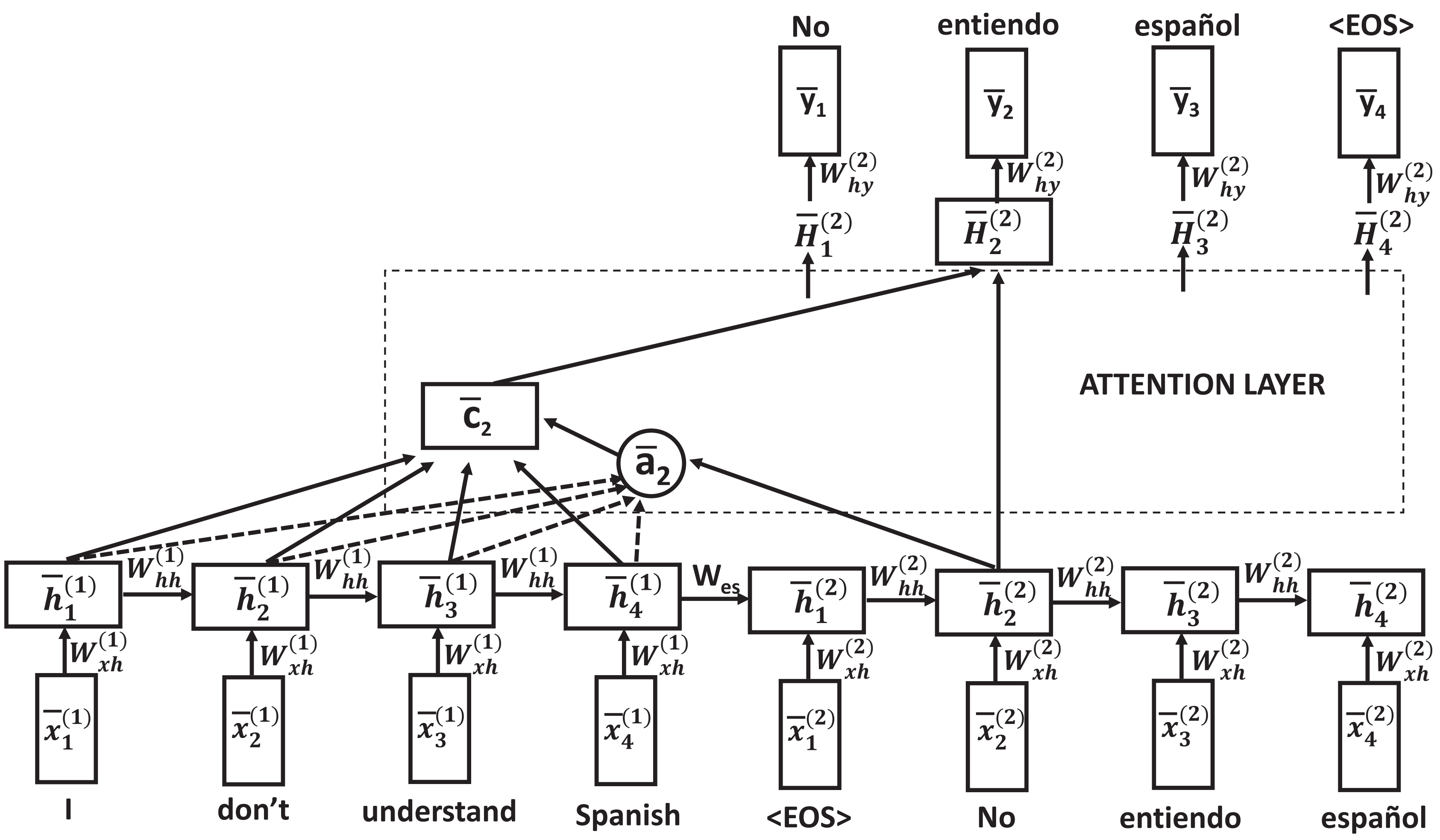
- Context vector is defined as follows:

$$\overline{c}_t = \frac{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})\overline{h}_j^{(1)}}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} = \sum_{j=1}^{T_s} a(t,j)\overline{h}_j^{(1)} \tag{1}$$

- **Attention Layer:** Create a new target hidden state $H_t^{(2)}$ that combines the information in the context and the original target hidden state as follows:

$$\overline{H}_t^{(2)} = \tanh\left(W_c \left[\begin{array}{c} \overline{c}_t \\ \overline{h}_t^2 \end{array} \right]\right) \tag{2}$$

The Attention-Centric Machine Translation Model



- Enhanced hidden states used for prediction in lieu of original hidden states.

What Have We Done?

- The hidden states will be more weighted towards specific words in the source sentence.

- The context vector helps focus the prediction towards the portion of the source sentence that is more relevant to the target word.

- The value of the attention score $a(t, j)$ while predicting a target word will be higher for relevant portions of the source sentence.

Refinements

- The previous model uses simple dot products for similarity.

- Can also use parameterized variants:

$$\text{Score}(t,s) = \begin{cases} \overline{h}_s^{(1)} \cdot \overline{h}_t^{(2)} & \text{Dot product} \\ (\overline{h}_t^{(2)})^T W_a \overline{h}_s^{(1)} & \text{Parameters } W_a \\ \overline{v}_a^T \tanh\left(W_a \begin{bmatrix} \overline{h}_s^{(1)} \\ \overline{h}_t^2 \end{bmatrix} \right) & \text{Concat: Parameters } W_a,\ \overline{v}_a \end{cases} \tag{3}$$

- The first of these options is identical to that in previous slides.

$$a(t,s) = \frac{\exp(\text{Score}(t,s))}{\sum_{j=1}^{T_s} \exp(\text{Score}(t,j))} \tag{4}$$

Observations

- The refined variants do not necessarily help too much for soft attention models.

- More details of hard attention models are available in:

  - M. Luong, H. Pham, and C. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

- Attention ideas have been generalized to neural Turing machines.

  - Discussed in book.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Generative Adversarial Networks

Neural Networks and Deep Learning, Springer, 2018
Chapter 10.4

Generative Adversarial Network

- Generative adversarial network creates an unsupervised generative model of the data.
  - Alternative to variational autoencoder

- Unlike the variational autoencoder, the generative portion does not directly see the training data.
  - Indirectly receives feedback from a discriminator as to whether or not its generated samples are realistic.

Adversarial Training

- We have a *generator* and a *discriminator*.

- The discriminator has access to training samples and is a classifier designed to distinguish between real and fake samples.

- The generator tries to create samples whose main goal is to fool the discriminator.

- Simultaneous training of generators and discriminators with opposite objectives.

- Helpful to think of generator as counterfeiter and discriminator as police.

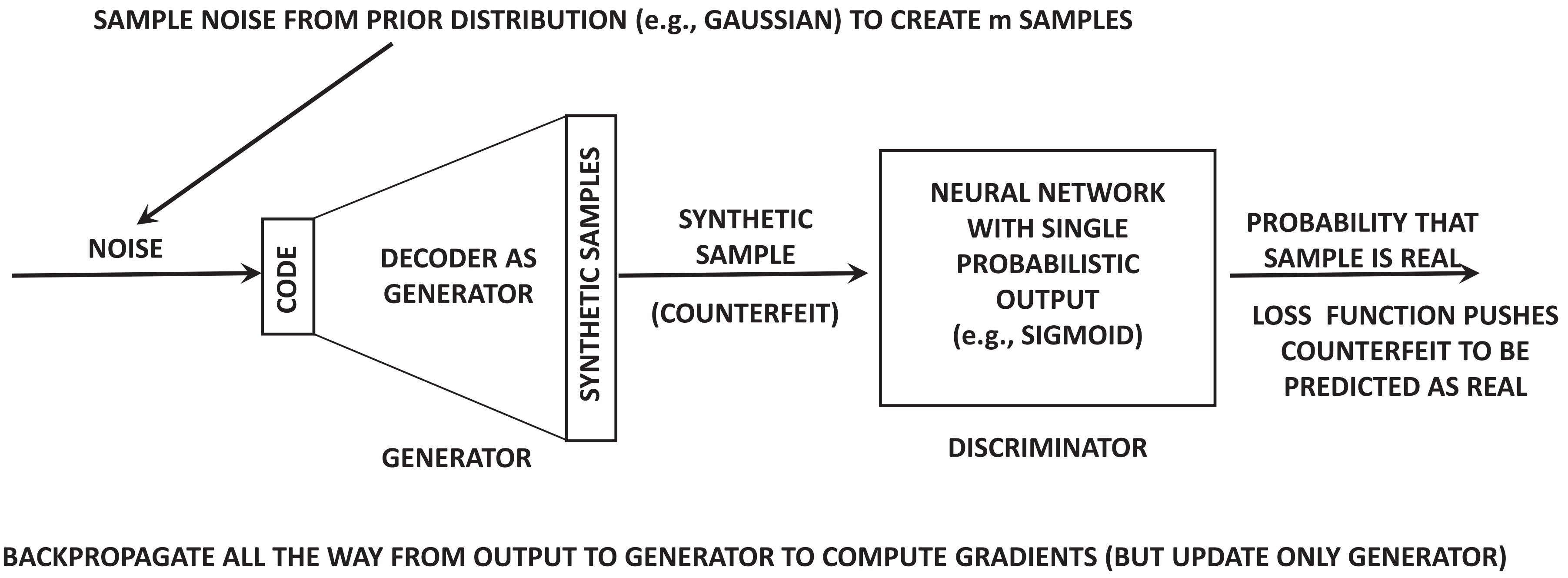
Generator and Discriminator

- R_m: Set of m randomly sampled examples from the real data set.

- S_m: Set of m synthetically generated samples.

- Synthetic samples are generated by first creating a set N_m of p-dimensional Gaussian noise samples $\{\overline{Z}_m \ldots \overline{Z}_m\}$.
  - Apply the generator to these noise samples as the input to create the data samples $S_m = \{G(\overline{Z}_1) \ldots G(\overline{Z}_m)\}$.

Neural Architecture for GAN


SAMPLE NOISE FROM PRIOR DISTRIBUTION (e.g., GAUSSIAN) TO CREATE m SAMPLES
NOISE
CODE
DECODER AS GENERATOR
SYNTHETIC SAMPLES
GENERATOR
SYNTHETIC SAMPLE
(COUNTERFEIT)
NEURAL NETWORK WITH SINGLE PROBABILISTIC OUTPUT (e.g., SIGMOID)
DISCRIMINATOR
PROBABILITY THAT SAMPLE IS REAL
LOSS FUNCTION PUSHES COUNTERFEIT TO BE PREDICTED AS REAL
BACKPROPAGATE ALL THE WAY FROM OUTPUT TO GENERATOR TO COMPUTE GRADIENTS (BUT UPDATE ONLY GENERATOR)

Discriminator Objective Function

- $D(\overline{X})$: Discriminator output probability that sample $\overline{X}$ is real.

- The *maximization* objective function J_D for the discriminator is as follows:

$$\text{Maximize}_D \, J_D = \underbrace{\sum_{\overline{X} \in \mathcal{R}_m} \log \left[D(\overline{X})\right]}_{m \text{ real examples}} + \underbrace{\sum_{\overline{X} \in \mathcal{S}_m} \log \left[1 - D(\overline{X})\right]}_{m \text{ synthetic examples}}$$

- The objective function will be maximized when real examples are correctly classified to 1 and synthetic examples are correctly classified to 0.

Generator Objective Function

- The generator creates m synthetic samples, S_m, and the goal is to fool discriminator.

- The objective function is to *minimize* the likelihood that these samples are flagged as synthetic.

- The objective function, J_G, for the generator can be written as follows:

$$\begin{aligned}\text{Minimize}_G\, J_G &= \underbrace{\sum_{\overline{X} \in S_m} \log\left[1 - D(\overline{X})\right]}_{m \text{ synthetic examples}}\\ &= \sum_{\overline{Z} \in N_m} \log\left[1 - D(G(\overline{Z}))\right]\end{aligned}$$

Minimax Formulation

- Note that $J(D) = J(G)+$Term indpendent of generator parameters
- So minimizing J_G with respect to generator parameters is the same as minimizing J_D with respect to generator parameters.
- So we want to maximize J_D with respect to discriminator parameters and minimize it with respect to generator parameters.
- Standard minimax formulation: $\text{min}_G \text{max}_D J_D$

How to Solve?

- Alternately perform updates with respect to generator and discriminator parameters.
  - Updates for generator parameters are gradient descent updates.
  - Updates for discriminator parameters are gradient ascent updates.
- Common to use k steps of the discriminator for each step of the generator (backprop).
- Increasingly common to train neural networks simultaneously in many applications.

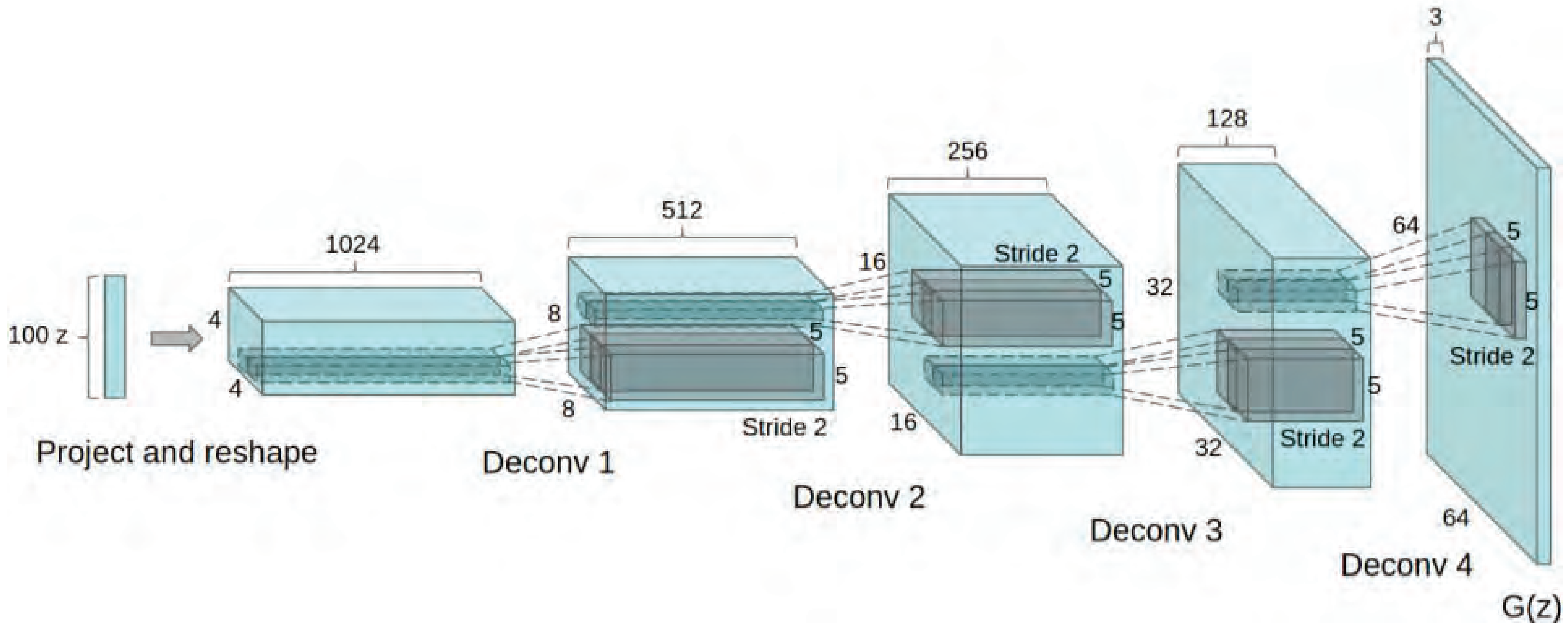
Adjustments During Early Optimization Iterations

- Maximize log $\left[D(\overline{X})\right]$ for each $\overline{X} \in S_m$ instead of minimizing log $\left[1 - D(\overline{X})\right]$.

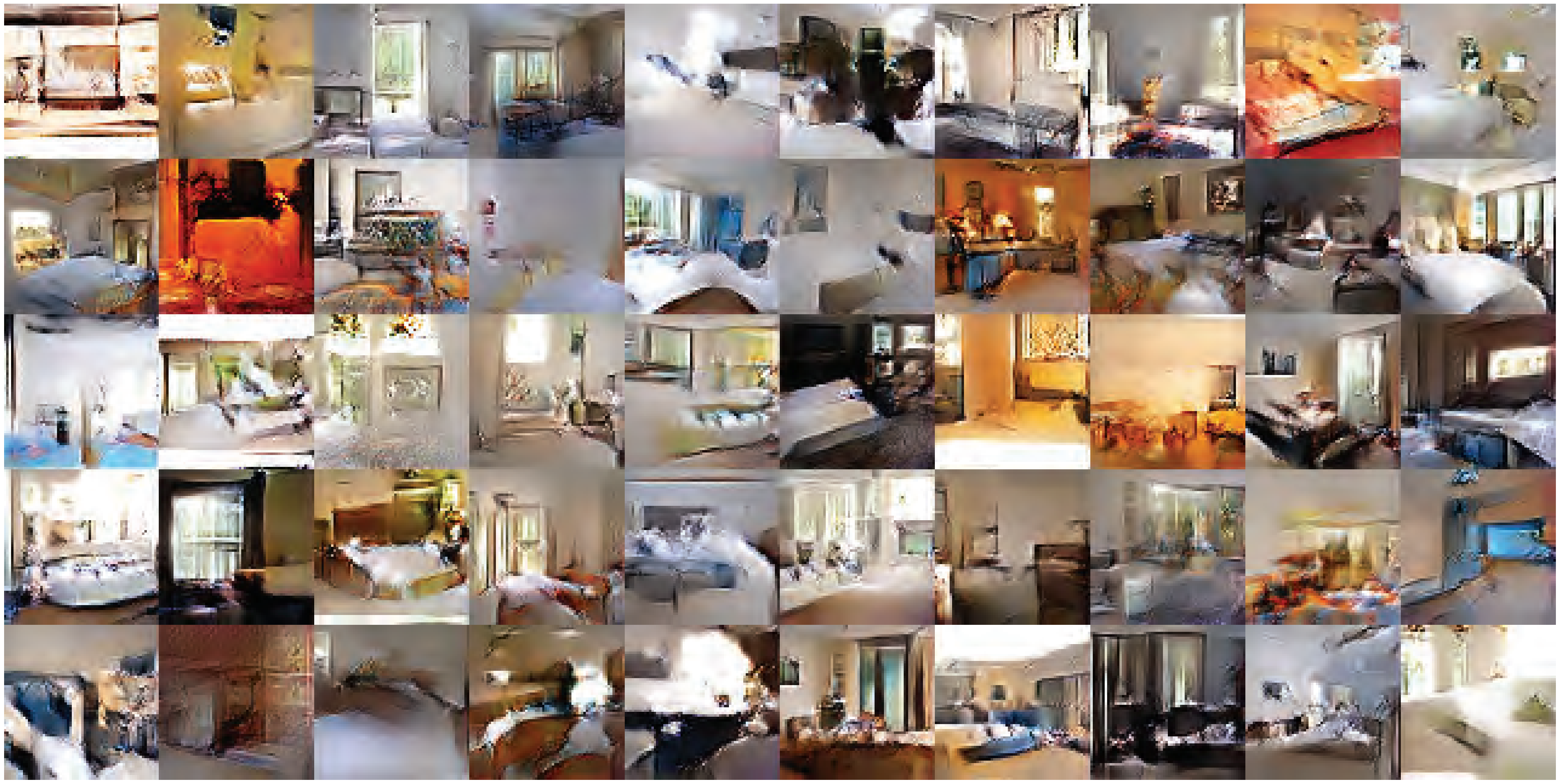
- This alternative objective function sometimes works better during the early iterations of optimization.

  – Faster learning.

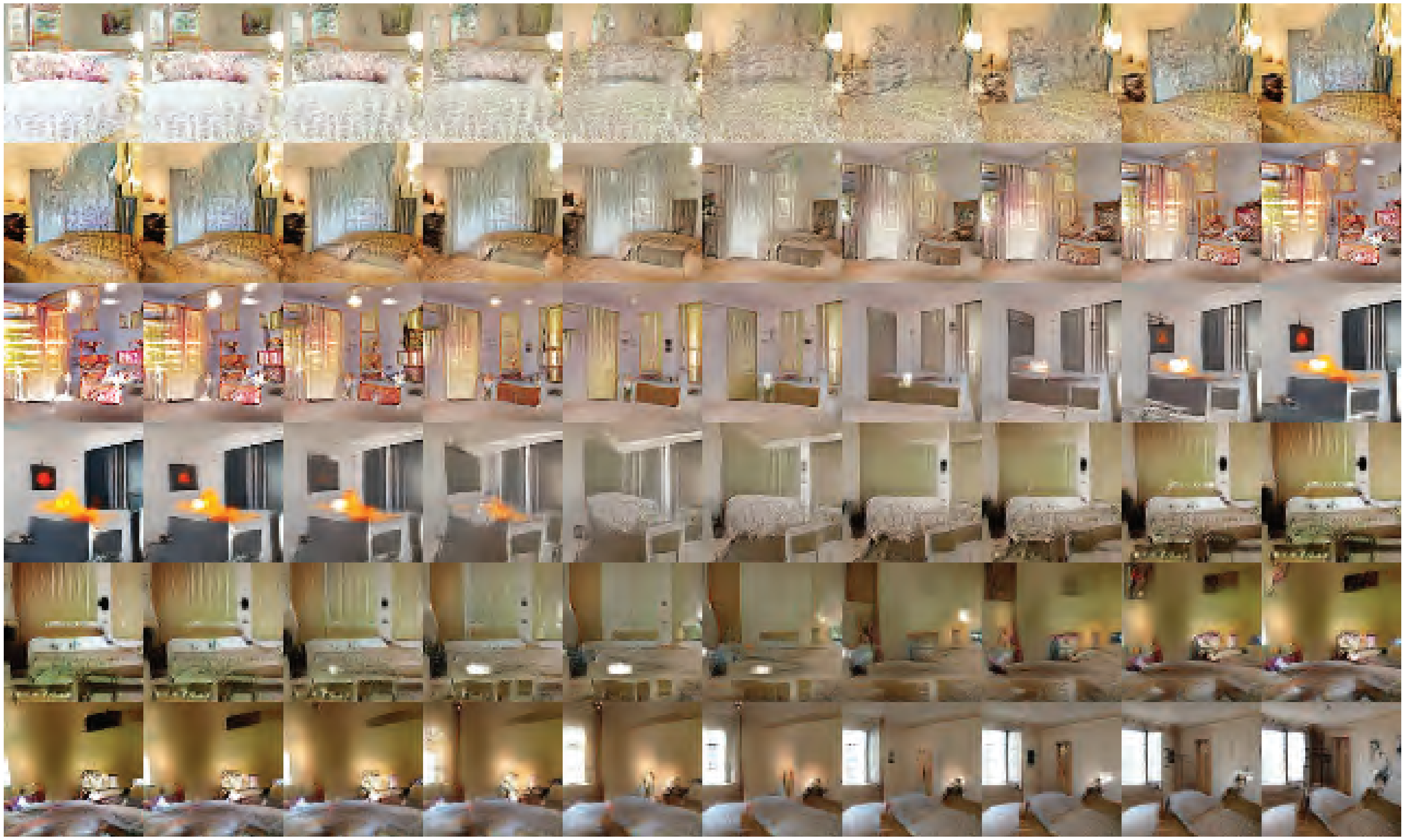
Example for Image Generation [Radford, Metz, Chintala]



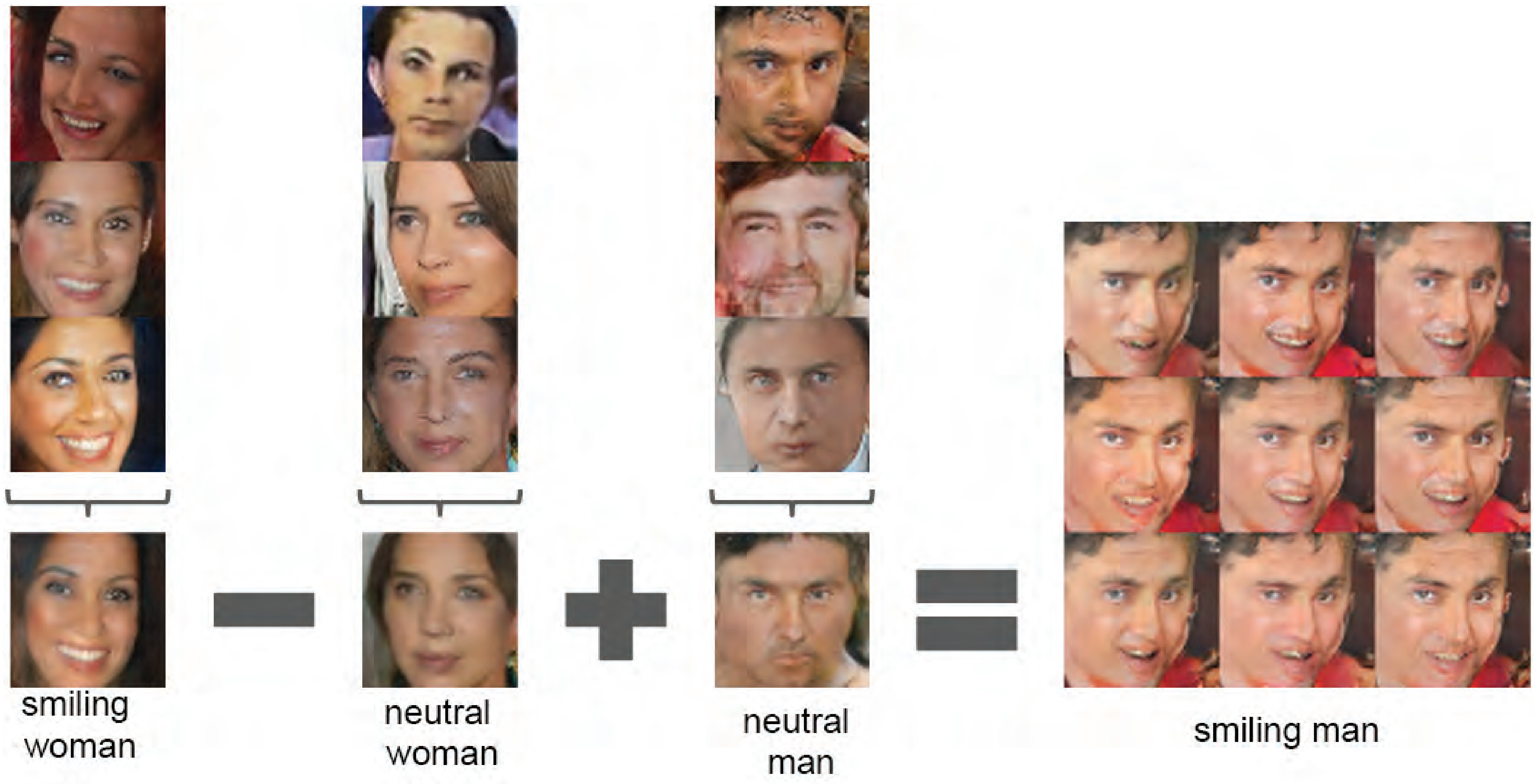
Generated Bedrooms [Radford, Metz, Chintala]



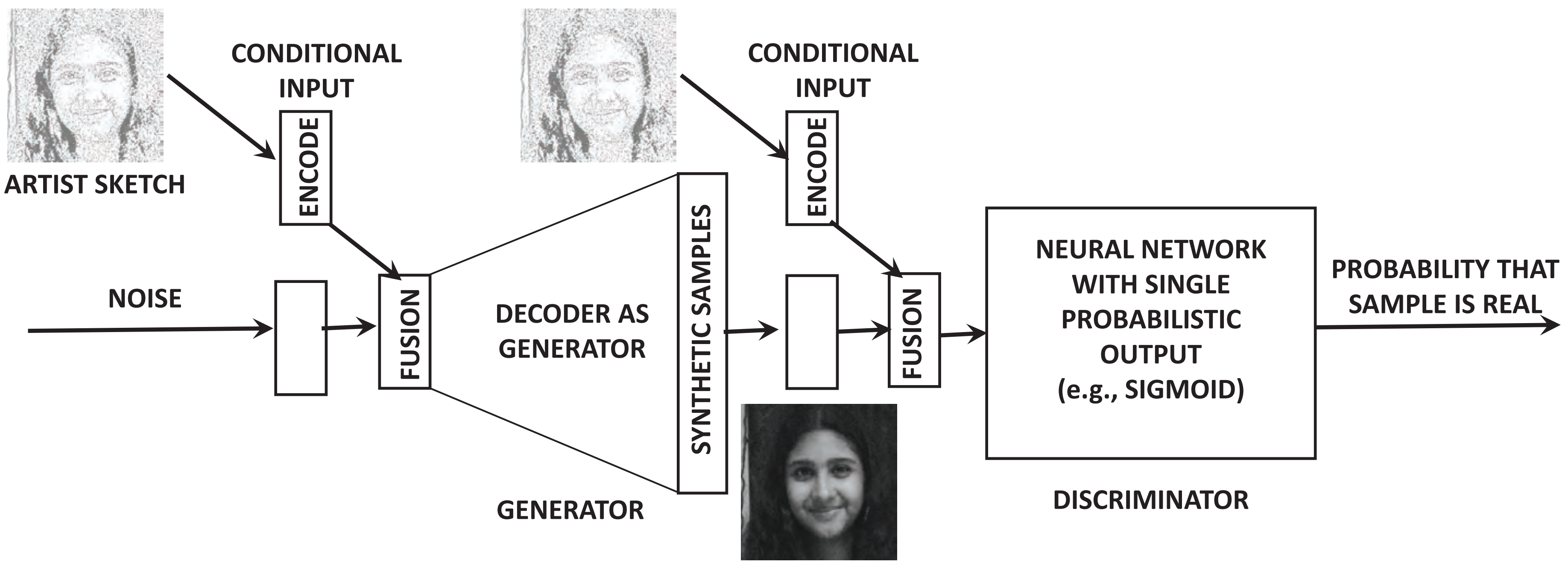
Changing the Synthetic Noise Sample [Radford, Metz, Chintala]



Vector Arithmetic on Synthetic Noise Samples [Radford, Metz, Chintala]

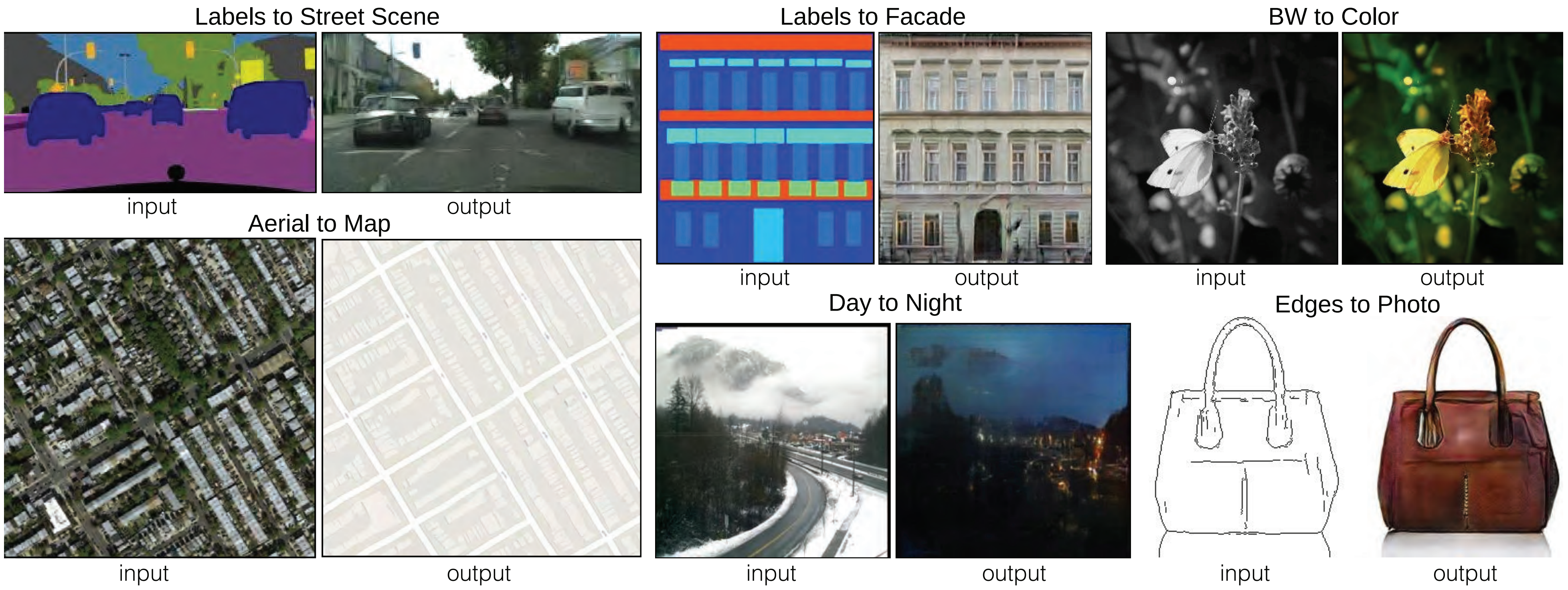


Conditional Generative Adversarial Networks (CGAN)

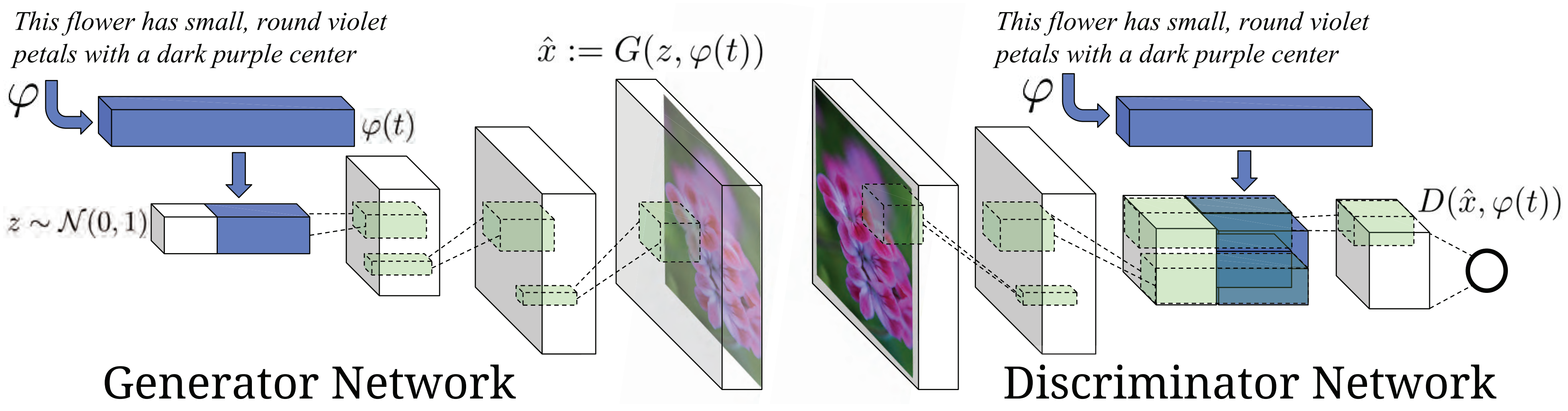


GENERATOR EXTRAPOLATION FROM SKETCH (SYNTHETIC SAMPLE)

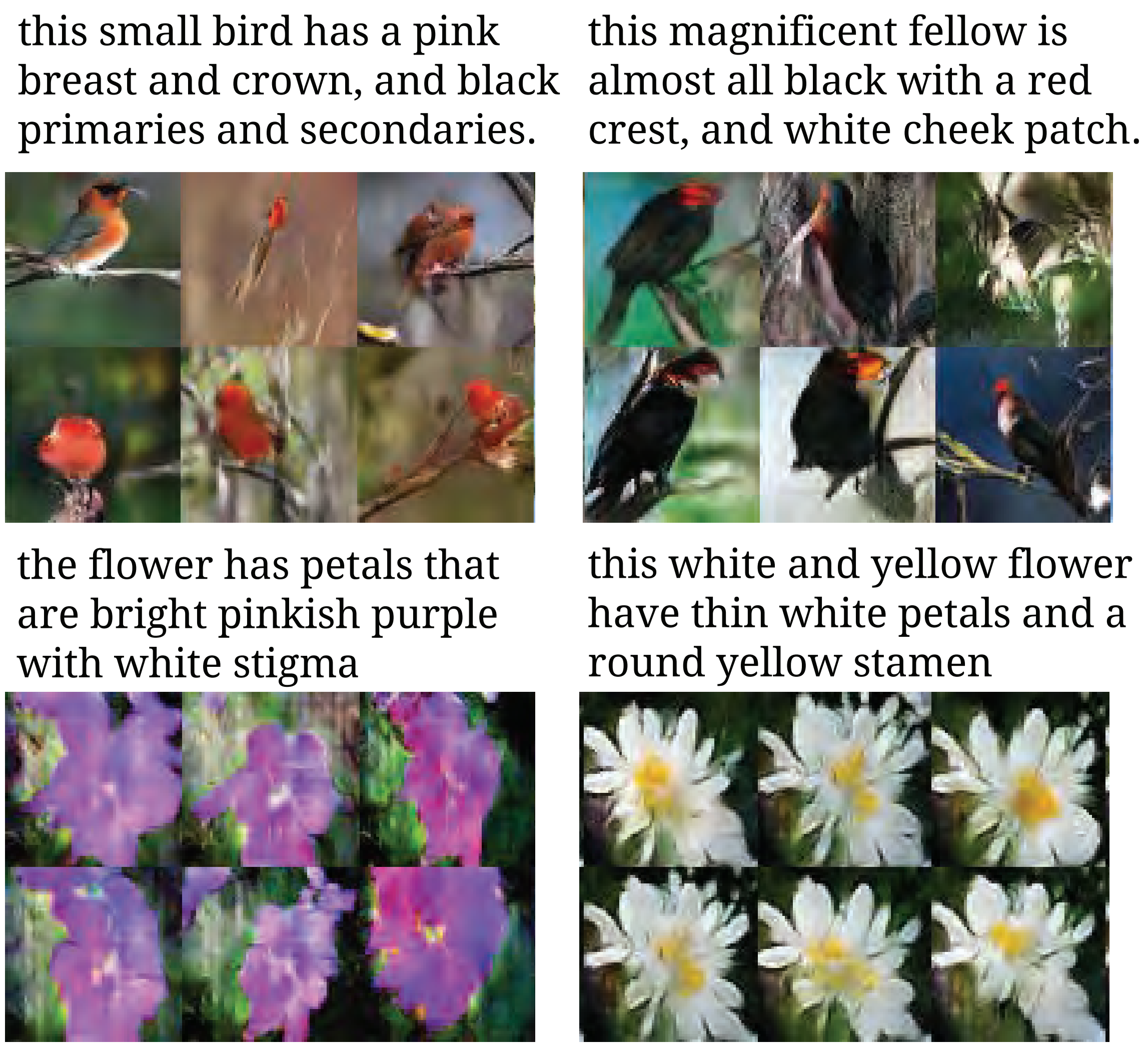
Image-to-Image Translation with CGAN [Isola, Zhu, Zhou, Efros]



Text-to-Image Translation with CGAN: Scott Reed et al.


This flower has small, round violet petals with a dark purple center
φ
$\varphi(t)$
$z \sim \mathcal{N}(0,1)$
$\hat{x} := G(z, \varphi(t))$
Generator Network
This flower has small, round violet petals with a dark purple center
φ
$D(\hat{x}, \varphi(t))$
Discriminator Network

Text-to-Image Translation with CGAN: Scott Reed et al.


this small bird has a pink breast and crown, and black primaries and secondaries.

this magnificent fellow is almost all black with a red crest, and white cheek patch.

the flower has petals that are bright pinkish purple with white stigma

this white and yellow flower have thin white petals and a round yellow stamen

Figure 1. Examples of generated images from text descriptions.

Comments on CGAN

- Capabilities are similar to conditional variational autoencoder
  - Special case is captioning (conditioning on image and target is caption)
  - Special case is classification (conditioning on object and target is class)

- Simpler special cases can be handled by supervised learning

- Makes a lot more sense to use when target is more complex than the conditioning $\Rightarrow$ Generative creativity required

Comparison with Variational Autoencoder

- Only a decoder (i.e., generator) is learned, and an encoder is not learned in the training process of the generative adversarial network.

- A generative adversarial network is not designed to reconstruct specific input samples like a variational autoencoder.

- The generative adversarial network produces samples of better quality than a variational autoencoder.

  - The adversarial approach is specifically designed to produce realistic images.

  - The regularization of the variational autoencoder actually hurts the quality of the generated objects.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Kohonen Self-Organizing Maps

Neural Networks and Deep Learning, Springer, 2018
Chapter 10.5

Introduction and Motivation

- The Kohonen self-organizing map belongs to the class of competitive learning algorithms.
  - Competitive learning algorithms are a broader class than the Kohonen self-organizing map.
  - Used for clustering, compression, and visualization.
  - Kohonen self-organizing map is a special case that is designed for visualization.
- First discuss competitive learning and then the Kohonen map.

Competitive Learning

- The neurons compete for the right to respond to a subset of the input data.

- The activation of an output neuron increases with greater similarity between the weight vector of the neuron and the input.

  - Weight vector and input have same dimensionality.

- A common approach is to use the Euclidian distance between the input and the weight vector in order to compute the activation.

- The output unit that has the highest activation (smallest distance) to a given input is declared the winner and moved closer to the input.

Notations

- Let $\overline{X}$ be an input vector in d dimensions.

- Let $\overline{W}_i$ be the weight vector associated with the ith neuron in the same number of dimensions.

- Number of neurons m is typically much less than the size of the data set n.
  - Intuitively, consider the neurons like k-means centroids.

- Moving a neuron closer to the input is similar to how prototypes are always moved closer to their relevant clusters in algorithms like k-means.

Iterative Steps for Each Input Point

- The Euclidean distance $||\overline{W}_i - \overline{X}||$ is computed for each i (activation value is higher for smaller distance).

- If the pth neuron has the smallest value of the Euclidean distance, then it is declared as the winner.

- The pth neuron is updated using the following rule and learning rate $\alpha \in (0, 1)$:

$$\overline{W}_p \Leftarrow \overline{W}_p + \alpha(\overline{X} - \overline{W}_p) \tag{5}$$

Comparison with Prototype-Based Clustering

- The basic idea in competitive learning is to view the weight vectors as prototypes (like the centroids in k-means clustering).

- The value of α regulates the fraction of the distance between the point and the weight vector, by which the movement of $\overline{W}_p$ occurs.

- The k-means clustering also achieves similar goals.
  - When a point is assigned to the winning centroid, it moves that centroid by a small distance towards the training instance at the end of the iteration.

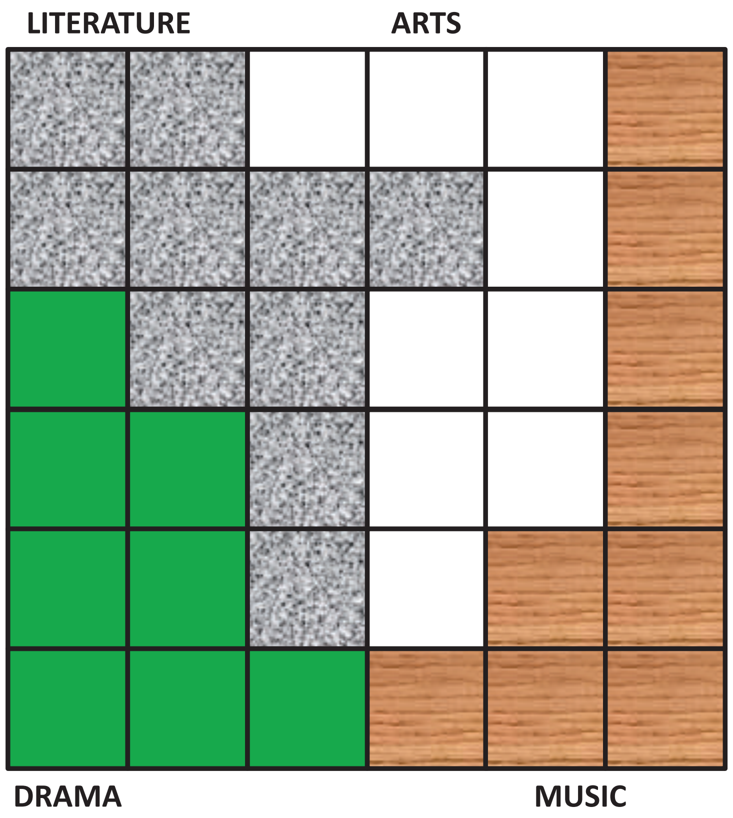
- Competitive learning is a natural variation of this framework.

Physically Arranging the Clusters

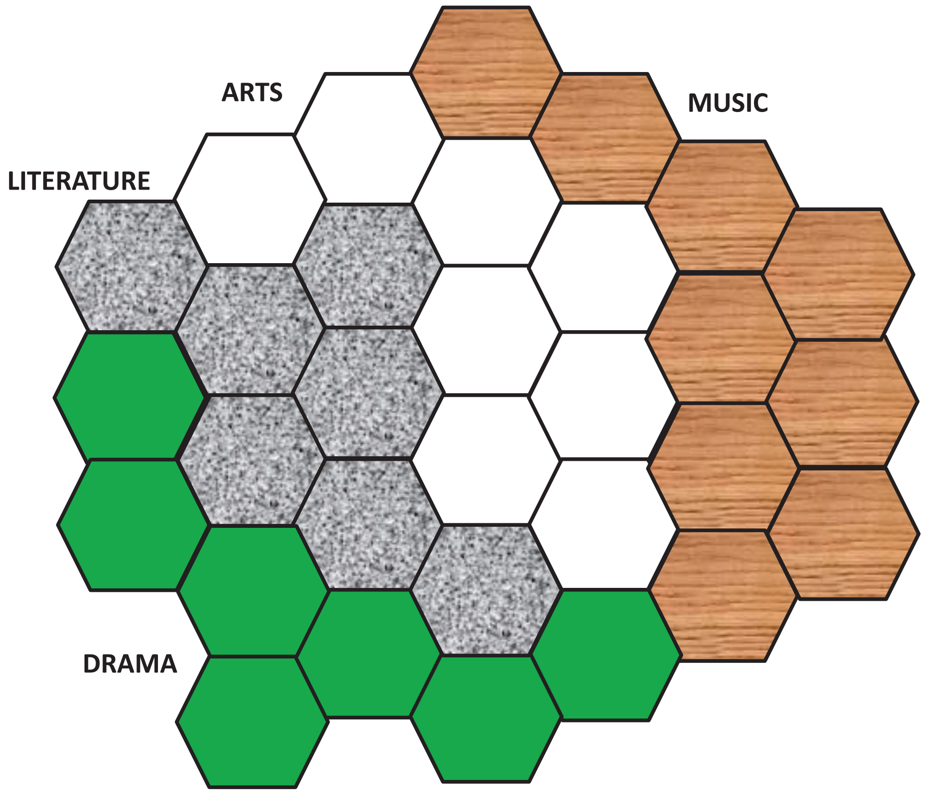
- Pure competitive learning does not impose any relationships among clusters.
  - Clusters often have related content.
  - Can we construct and place the clusters in 2-dimensions, so that physically adjacent clusters have related points?
  - Important to keep need for physical placement in mind during cluster construction.

- Kohonen's self organizing map gives the same shape to each cluster (e.g., rectangle, hexagon) and places then in a 2-dimensional hexagon or grid-like structure.

Illustrative Example of Visualization



(a) Rectangular lattice



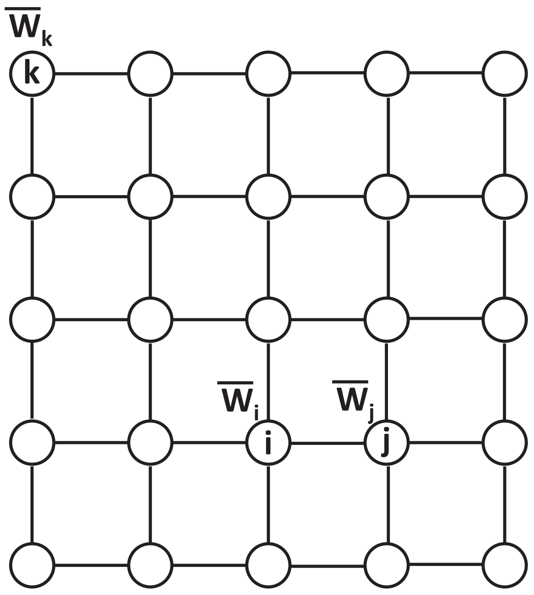
(b) Hexagonal lattice

- All clusters are either rectangles or hexagons

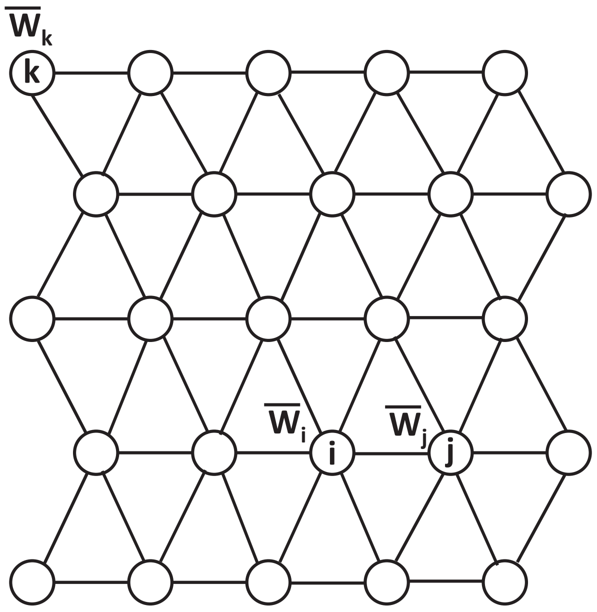
Kohonen Self-Organizing Map

- The Kohonen self-organizing map is a variation on the competitive learning paradigm in which a 2-dimensional lattice-like structure is imposed on the neurons (cluster prototypes).
  - Vanilla competitive learning does not force clusters to have relationships with one another.
  - Imposing 2-dimensional adjacency on (similar) clusters is useful for visualization.
  - All points assigned to a neuron can be assigned to a 2-d cell of a particular shape.
- The values of $\overline{W}_i$ in lattice-adjacent neurons are encouraged to be similar (type of regularization).

Different Types of Lattices



(a) Rectangular



(b) Hexagonal

- Rectangular lattice will lead to rectangular regions and hexagonal lattice will lead to hexagonal regions.

Modifications to Basic Competitive Learning

- The weights in the winner neuron are updated in a manner similar to the vanilla competitive learning algorithm in the Kohonen map.
  - The main difference is that *a damped version of this update is also applied to the lattice-neighbors of the winner neuron.*
- Has the effect of moving similar points to lattice-adjacent clusters $\Rightarrow$ Useful for visualization.
  - Provides a 2-dimensional organization of clusters.

The Kohonen SOM Algorithm

- $LDist(i, j)$ represents the *lattice distance* between neurons i and j.

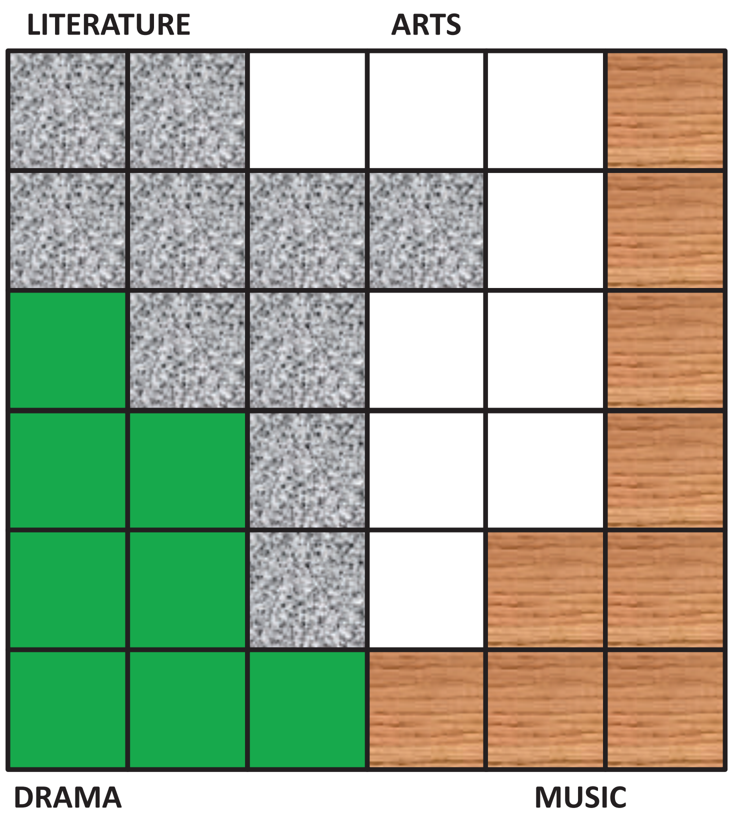
$$Damp(i, j) = \exp\left(-\frac{LDist(i, j)^2}{2\sigma^2}\right) \tag{6}$$

- Here, σ is the bandwidth of the Gaussian kernel.

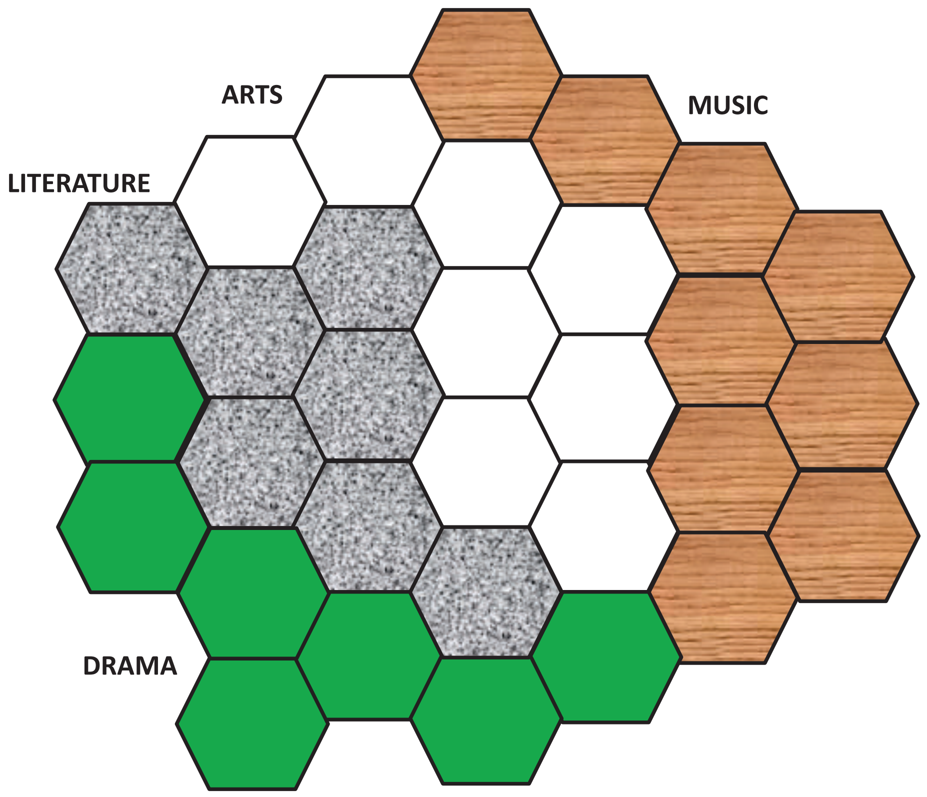
$$\overline{W}_i \Leftarrow \overline{W}_i + \alpha \cdot Damp(i, p) \cdot (\overline{X} - \overline{W}_i) \quad \forall i \tag{7}$$

- Using extremely small values of σ reverts to pure winner-take-all learning,

Illustrative Example of Visualization



(a) Rectangular lattice



(b) Hexagonal lattice

- Color each region with majority class $\Rightarrow$ Similar documents are mapped to same/adjacent regions.