Deep Learning

AI+X
INSTITUTE FOR ARTIFICIAL INTELLIGENCE

Introducing Deep Learning

- In recent years, so-called “deep learning” approaches to machine learning have had a major impact on speech recognition and computer vision
- Other disciplines, such as natural language processing, are also starting to see benefits
- A critical ingredient is the use of much larger quantities of data than has heretofore been possible
- Recent successes have arisen in settings involving *high capacity* models—ones with many parameters
- Here, deep learning methods create flexible models that exploit information buried in massive datasets far more effectively than do traditional machine learning techniques using hand-engineered features

Views on machine learning

One way to view machine learning is in terms of three general approaches:

1. Classical machine learning techniques,
   which make predictions directly from a set of features that have been pre-specified by the user;
2. Representation learning techniques,
   which transform features into some intermediate representation prior to mapping them to final predictions; and
3. Deep learning techniques,
   a form of representation learning that uses multiple transformation steps to create very complex features

The neural network renaissance and deep learning revolution

- The term “renaissance” captures a massive resurgence of interest in neural networks and deep learning techniques
- Many high-profile media (e.g. *The New York Times*) have documented the striking successes of deep learning techniques on key benchmark problems
- Starting around 2012, impressive results were achieved on long-standing problems in speech recognition and computer vision, and in competitive challenges such as the ImageNet *Large Scale Visual Recognition Challenge* and the *Labeled Faces in the Wild* evaluation

GPUs, graphs and tensors

- The easy availability of high-speed computation in the form of graphics processing units has been critical to the success of deep learning techniques
- When formulated in matrix-vector form, computation can be accelerated using optimized graphics libraries and hardware
- This is why we will study backpropagation in matrix-vector form
  - Readers unfamiliar with manipulating functions that have matrix arguments, and their derivatives are advised to consult Appendix A.1 for a summary of some useful background
- As network models become more complex, some quantities can only be represented using multidimensional arrays of numbers
  - Such arrays are sometimes referred to as tensors, a generalization of matrices that permit an arbitrary number of indices
- Software for deep learning supporting *computation graphs* and *tensors* is therefore invaluable for accelerating the creation of complex network structures and making it easier to learn them

Key developments

The following developments have played a crucial role in the resurgence of neural network methods:

- the proper evaluation of machine learning methods;
- vastly increased amounts of data;
- deeper and larger network architectures;
- accelerated training using GPU techniques

Mixed National Institute of Standards and Technology (MNIST)

- Is a database and evaluation setup for handwritten digit recognition
- Contains 60,000 training and 10,000 test instances of hand-written digits, encoded as 28×28 pixel grayscale images
- The data is a re-mix of an earlier NIST dataset in which adults generated the training data and high school students generated the test set
- Lets compare the performance of different methods

MNIST

Classifier	Test Error Rate (%)	References
Linear classifier (1-layer neural net)	12.0	LeCun et al. (1998)
K-nearest-neighbors, Euclidean (L2)	5.0	LeCun et al. (1998)
2-Layer neural net, 300 hidden units, mean square error	4.7	LeCun et al. (1998)
Support vector machine, Gaussian kernel	1.4	MNIST Website
Convolutional net, LeNet-5 (no distortions)	**0.95**	LeCun et al. (1998)
Methods using distortions		
Virtual support vector machine, deg-9 polynomial, (2-pixel jittered and deskewing)	0.56	DeCoste and Scholkopf (2002)
Convolutional neural net (elastic distortions)	0.4	Simard, Steinkraus, and Platt (2003)
6-Layer feedforward neural net (on GPU) (elastic distortions)	**0.35**	Ciresan, Meier, Gambardella, and Schmidhuber (2010)
Large/deep convolutional neural net (elastic distortions)	**0.35**	Ciresan, Meier, Masci, Maria Gambardella, and Schmidhuber (2011)
Committee of 35 convolutional networks (elastic distortions)	0.23	Ciresan, Meier, and Schmidhuber (2012)

Losses and regularization

- Logistic regression can be viewed as a simple neural network with no hidden units
- The underlying optimization criterion for predicting i=1,…,N labels y_i from features $\mathbf{x}_i$ with parameters θ consisting of a matrix of weights **W** and a vector of biases **b** can be viewed as

$$\sum_{i=1}^{N} -\log p(y_i \mid \mathbf{x}_i; \mathbf{W}, \mathbf{b}) + \lambda \sum_{j=1}^{M} w_j^2 = \sum_{i=1}^{N} L(f_i(\mathbf{x}_i; \theta), y_i) + \lambda R(\theta)$$

- where the first term, $L(f_i(\mathbf{x}_i; \theta), y_i)$, is the negative conditional log-likelihood or *loss*, and
- the second term, $\lambda R(\theta)$, is a weighted *regularizer* used to prevent overfitting

Empirical risk minimization

- This formulation as a loss- and regularizer-based objective function gives us the freedom to choose either probabilistic losses or other loss functions
- Using the *average loss* over the training data, called the *empirical risk*, leads to the following formulation of the optimization problem: minimize the empirical risk plus a regularization term, i.e.

$$\arg\min_{\theta}\left[\frac{1}{N}\sum_{i=1}^{N}L(f_i(\mathbf{x}_i;\theta),\mathbf{y}_i)+\lambda R(\theta)\right].$$

- Note that the factor *N* must be accounted for if one relates the regularization weight here to the corresponding parameter derived from a formal probabilistic model for a distribution on parameters

In practice

- In deep learning we are often interested in examining learning curves that show the loss or some other performance metric on a graph as a function of the number of passes that an algorithm has taken over the data.
- It is much easier to compare the *average* loss over a training set with the *average* loss over a validation set on the same graph, because dividing by *N* gives them the same scale.

Common losses for neural networks

- The final output function of a neural network typically has the form $f_k(\mathbf{x})=f_k(a_k(\mathbf{x}))$, where $a_k(\mathbf{x})$ is just one of the elements of vector fuction $\mathbf{a}(\mathbf{x})=\mathbf{W}\mathbf{h}(\mathbf{x})+\mathbf{b}$
- Commonly used output loss functions, output activation functions, and the underlying distributions from which they derive are shown below

Loss Name, $L(f_i(\mathbf{x}_i;\theta),\mathbf{y}_i)=$	**Distribution Name**, $P(f_i(\mathbf{x}_i;\theta),\mathbf{y}_i)=$	**Output Activation Function**, $f_k(a_k(\mathbf{x}))=$
Squared error, $\sum_{k=1}^{K}(f_k(\mathbf{x})-y_k)^2$	Gaussian, $N(\mathbf{y};\mathbf{f}(\mathbf{x};\theta),\mathbf{I})$	$\frac{1}{(1+\exp(-a_k(\mathbf{x})))}$
Cross entropy, $-\sum_{k=1}^{K}[y_k\log f_k(\mathbf{x})+(1-y_k)\log(1-f_k(\mathbf{x}))]$	Bernoulli, $\text{Bern}(\mathbf{y};\mathbf{f}(\mathbf{x};\theta))$	$\frac{1}{(1+\exp(-a_k(\mathbf{x})))}$
Softmax, $-\sum_{k=1}^{K}y_k\log f_k(\mathbf{x})$	Discrete or Categorical, $\text{Cat}(\mathbf{y};\mathbf{f}(\mathbf{x};\theta))$	$\frac{\exp(a_k(\mathbf{x}))}{\sum_{j=1}^{K}\exp(a_j(\mathbf{x}))}$

Deep neural network architectures

- Compose computations performed by many layers
- Denoting the output of hidden layers by $\mathbf{h}^{(l)}(\mathbf{x})$, the computation for a network with *L* hidden layers is:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}\left[\mathbf{a}^{(L+1)}\left(\mathbf{h}^{(L)}\left(\mathbf{a}^{(L)}\left(\ldots\left(\mathbf{h}^{(2)}\left(\mathbf{a}^{(2)}\left(\mathbf{h}^{(1)}\left(\mathbf{a}^{(1)}(\mathbf{x})\right)\right)\right)\right)\right)\right)\right]$$

- Where *pre-activation functions* $\mathbf{a}^{(l)}(\mathbf{x})$ are typically linear, of the form with matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$

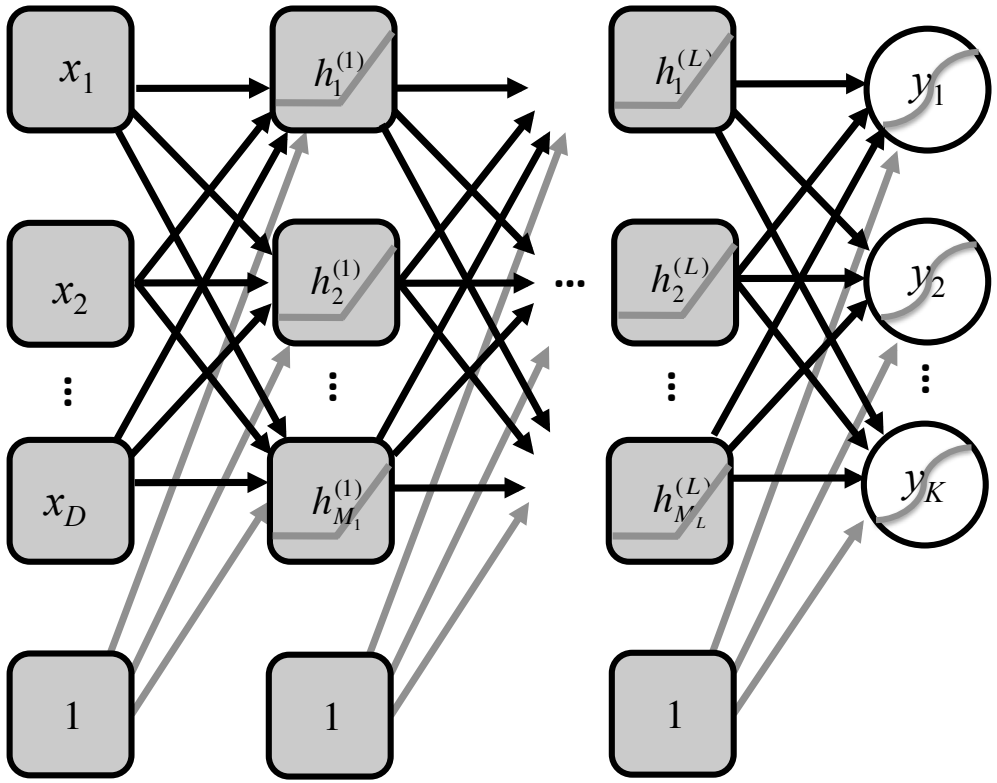
$$\mathbf{a}^{(l)}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}$$

- This formulation can be expressed using a single parameter matrix θ with the trick of defining $\hat{\mathbf{x}}$ as x with a 1 appended to the end of the vector; we then have

$$\mathbf{a}^{(l)}(\hat{\mathbf{x}}) = \theta^{(l)}\hat{\mathbf{x}} \qquad , l = 1$$

$$\mathbf{a}^{(l)}(\hat{\mathbf{h}}^{(l-1)}) = \theta^{(l)}\hat{\mathbf{h}}^{(l-1)} \qquad , l > 1$$

Deep feedforward networks

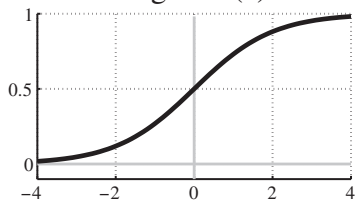
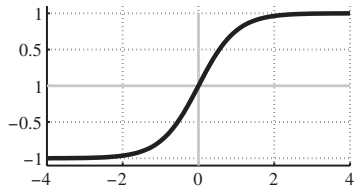
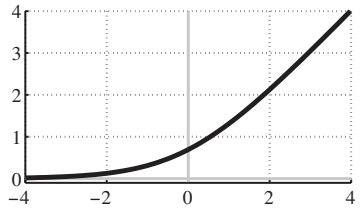
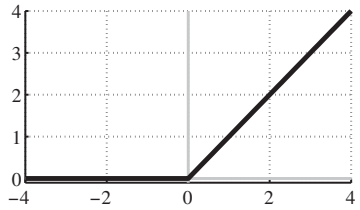


- Unlike Bayesian networks the hidden units here are *intermediate deterministic computations* not random variables, which is why they are not represented as circles
- However, the output variables y_k are drawn as circles because they can be formulated probabilistically

Activation functions

- Activation functions, $\mathbf{h}^{(l)}(\mathbf{x})$ generally operate on the pre-activation vectors in an *element-wise* fashion
- While sigmoid functions have been popular, the hyperbolic tangent function is sometimes preferred, partly because it has a steady state at 0
- More recently the *rectify*() function or rectified linear units (ReLUs) have been found to yield superior results in many different settings
  - Since ReLUs are 0 for negative argument values, some units in the model will yield activations that are 0, giving a sparseness property that is useful in many contexts
  - The gradient is particularly simple—either 0 or 1
  - This helps address the *exploding gradient problem*
- A number of software packages make it easy to use a variety of activation functions, determining gradients automatically using symbolic computations

Activation functions

Name and Graph	Function	Derivative
sigmoid(x) 	$h(x) = \frac{1}{1 + \exp(-x)}$	$h'(x) = h(x)[1 - h(x)]$
tanh(x) 	$h(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$h'(x) = 1 - h(x)^2$
softplus(x) 	$h(x) = \log(1 + \exp(x))$	$h'(x) = \frac{1}{1 + \exp(-x)}$
rectify(x) 	$h(x) = \max(0, x)$	$h'(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$

Bibliographic Notes & Further Reading

- The backpropagation algorithm has been known in close to its current form since Werbos (1974)'s PhD thesis
- In his extensive literature review of deep learning, Schmidhuber (2015) traces key elements of the algorithm back even further.
  - He also traces the idea of "deep networks" back to the work of Ivakhnenko and Lapa (1965).
- The popularity of neural network techniques has gone through several cycles and while some factors are social, there are important technical reasons behind the trends.
- A single-layer neural network cannot solve the XOR problem, a failing that was derided by Minsky and Papert (1969) and which stymied neural network development in the following decades.

Bibliographic Notes & Further Reading

- It is well known that networks with one additional layer can approximate any function (Cybenko, 1989; Hornik, 1991), and Rumelhart et al. (1986)'s influential work re-popularized neural network methods for a while.
- By the early 2000s neural network methods had fallen out of favor again – kernel methods like SVMs yielded state of the art results on many problems and were convex
- Indeed, the organizers of NIPS, the *Neural Information Processing Systems* conference, which was (and still is) widely considered to be the premier forum for neural network research, found that the presence of the term "neural networks" in the title was highly correlated with the paper's rejection!
  - A fact that is underscored by citation analysis of key neural network papers during this period.
- In this context, the recent resurgence of interest in deep learning really does feel like a "revolution."

Bibliographic Notes & Further Reading

- It is known that most complex Boolean functions require an exponential number of two-step logic gates for their representation (Wegener, 1987).
- The solution appears to be greater depth: according to Bengio (2014), the evidence strongly suggests that “functions that can be compactly represented with a depth-*k* architecture could require a very large number of elements in order to be represented by a shallower architecture”.

Backpropagation revisited in vector matrix form

Backpropagation in matrix vector form

- Backpropagation is based on the chain rule of calculus
- Consider the loss for a single-layer network with a softmax output (which corresponds exactly to the model for multinomial logistic regression)
- We use multinomial vectors **y**, with a single dimension $y_k = 1$ for the corresponding class label and whose other dimensions are 0
- Define $\mathbf{f} = [\ f_1(\mathbf{a}),\ \ldots,\ f_K(\mathbf{a})\ ]^T$, and $a_k(\mathbf{x};\boldsymbol{\theta}_k) = \boldsymbol{\theta}_k^{\mathrm{T}}\mathbf{x}$,

$\mathbf{a}(\mathbf{x};\boldsymbol{\theta}) = [\ a_1(\mathbf{x};\boldsymbol{\theta}_1),\ \ldots,\ a_K(\mathbf{x};\boldsymbol{\theta}_K)\ ]^T$ where θ_k is a column vector containing the k^{th} row of the parameter matrix

- Consider the softmax loss for **f**(**a**(**x**))

Logistic regression and the chain rule

- Given loss $L = -\sum_{k=1}^{K} y_k \log f_k(\mathbf{x}), \quad f_k(\mathbf{x}) = \frac{\exp(a_k(\mathbf{x}))}{\sum_{c=1}^{K} \exp(a_c(\mathbf{x}))}.$
- Use the chain rule to obtain

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} \frac{\partial \mathbf{f}}{\partial \mathbf{a}} \frac{\partial L}{\partial \mathbf{f}} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} \frac{\partial L}{\partial \mathbf{a}}.$$

- Note the order of terms - in vector matrix form terms build from right to left

$$\frac{\partial L}{\partial a_j} = \frac{\partial}{\partial a_j}\left[-\sum_{k=1}^{K} y_k \left[a_k - \log\left[\sum_{c=1}^{K} \exp(a_c)\right]\right]\right]$$

$$= -\left[y_{k=j} - \frac{\exp(a_{k=j})}{\sum_{c=1}^{K} \exp(a_c)}\right] = -\left[y_j - p(y_j \mid \mathbf{x})\right] = -\left[y_j - f_j(\mathbf{x})\right],$$

Matrix vector form of gradient

- We can write

$$\frac{\partial L}{\partial \mathbf{a}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] \equiv -\Delta$$

and since

$$\frac{\partial a_j}{\partial \boldsymbol{\theta}_k} = \begin{cases} \dfrac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{\theta}_k^{\mathrm{T}} \mathbf{x} = \mathbf{x} & , j = k \\ 0 & , j \neq k \end{cases}$$

we have

$$\frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} = \mathbf{H}_k = \begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}$$

- Notice that we avoid working with the partial derivative of the vector **a** with respect to the matrix θ, because it cannot be represented as a matrix — it is a multidimensional array of numbers (a tensor).

A compact expression for the gradient

- The gradient (as a column vector) for the vector in the *k*th row of the parameter matrix

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k}\frac{\partial L}{\partial \mathbf{a}} = -\begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right]$$

$$= -\mathbf{x}(y_k - f_k(x)).$$

- With a little rearrangement the gradient for the entire matrix of parameters can be written compactly:

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right]\mathbf{x}^{\mathrm{T}} = -\Delta\mathbf{x}^{\mathrm{T}}.$$

Consider now a multilayer network

- Using the same activation function for all L hidden layers, and a softmax output layer
- The gradient of the k^{th} parameter vector of the $L{+}1^{\text{th}}$ matrix of parameters is

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(L+1)}} = \frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}}, \qquad \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} = -\Delta^{(L+1)}$$

$$= -\frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \Delta^{(L+1)}$$

$$= -\mathbf{H}_k^L \Delta^{(L+1)} \quad \Rightarrow \quad \frac{\partial L}{\partial \boldsymbol{\theta}^{(L+1)}} = -\Delta^{(L+1)} \tilde{\mathbf{h}}_{(L)}^{\mathrm{T}}.$$

where $\mathbf{H}_k{}^L$ is a matrix containing the activations of the corresponding hidden layer, in column k

Backpropagating errors

- Consider the computation for the gradient of the k^{th} row of the L^{th} matrix of parameters
- Since the bias terms are constant, it is unnecessary to backprop through them, so

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(L)}} = \frac{\partial \mathbf{a}^{(L)}}{\partial \boldsymbol{\theta}_k^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}}$$

$$= -\frac{\partial \mathbf{a}^{(L)}}{\partial \boldsymbol{\theta}_k^{(L)}} \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \Delta^{(L+1)}, \qquad \Delta^{(L)} \equiv \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L+1)}}{\partial \mathbf{h}^{(L)}} \Delta^{(L+1)}$$

$$= -\frac{\partial \mathbf{a}^{(L)}}{\partial \boldsymbol{\theta}_k^{(L)}} \Delta^{(L)}$$

- Similarly, we can define $\Delta^{(l)}$ recursively in terms of $\Delta^{(l+1)}$

Backpropagating errors

- The backpropagated error can be written as simply

$$\Delta^{(l)} = \frac{\partial\, \mathbf{h}^{(l)}}{\partial\, \mathbf{a}^{(l)}} \frac{\partial\, \mathbf{a}^{(l+1)}}{\partial\, \mathbf{h}^{(l)}} \Delta^{(l+1)}, \qquad \frac{\partial\, \mathbf{h}^{(l)}}{\partial\, \mathbf{a}^{(l)}} = \mathbf{D}^{(l)}, \qquad \frac{\partial\, \mathbf{a}^{(l+1)}}{\partial\, \mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)},$$

$$\Delta^{(l)} = \mathbf{D}^{(l)} \mathbf{W}^{\mathrm{T}(l+1)} \Delta^{(l+1)}$$

  where $\mathbf{D}^{(l)}$ contains the partial derivatives of the hidden-layer activation function with respect to the pre-activation input.

- $\mathbf{D}^{(l)}$ is generally diagonal, because activation functions usually operate on an elementwise basis
- $\mathbf{W}^{\mathrm{T}(l+1)}$ arises from the fact that $\mathbf{a}^{(l+1)}(\mathbf{h}^{(l)}) = \mathbf{W}^{(l+1)}\mathbf{h}^{(l)} + \mathbf{b}^{(l+1)}$

A general form for gradients

- The gradients for the k^{th} vector of parameters of the l^{th} network layer can therefore be computed using products of matrices of the following form

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(l)}} = -\mathbf{H}_k^{(l-1)}\mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\cdots\mathbf{D}^{(L)}\mathbf{W}^{\mathrm{T}(L+1)}\Delta^{(L+1)}, \qquad \frac{\partial L}{\partial \boldsymbol{\theta}^{(l)}} = -\Delta^{(l)}\hat{\mathbf{h}}_{(l-1)}^{\mathrm{T}}$$

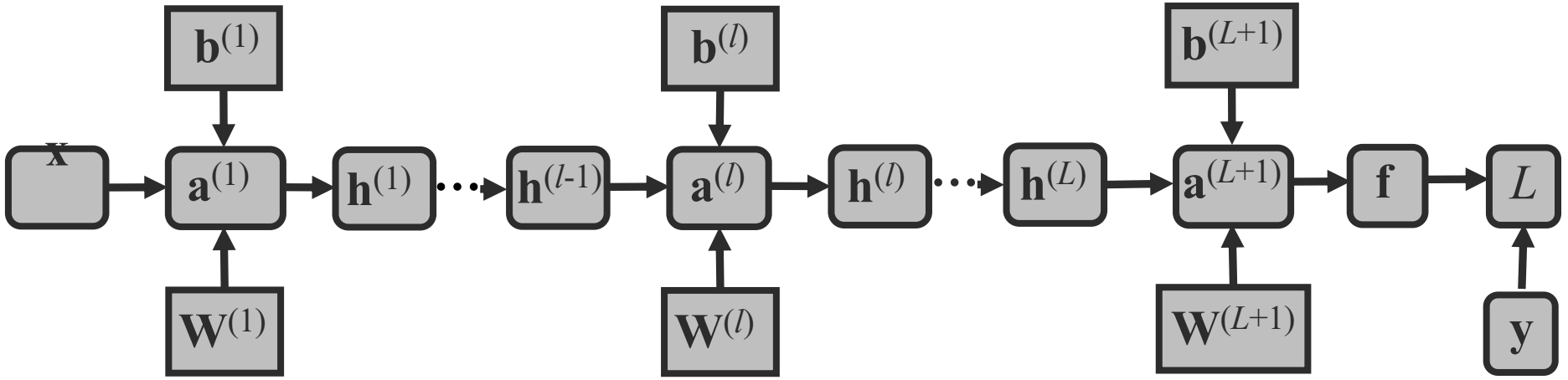
- When l=1, $\hat{\mathbf{h}}_{(0)} = \hat{\mathbf{x}}$, the input data with a 1 appended
- Note: since **D** is usually diagonal the corresponding matrix-vector multiply can be transformed into an element-wise product $\circ$ by extracting the diagonal for **d**

$$\Delta^{(l)} = \mathbf{D}^{(l)}(\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)}) = \mathbf{d}^{(l)} \circ (\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)})$$

Visualizing backpropagation

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \qquad \mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \qquad \mathbf{a}^{(L+1)} = \mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)}$$

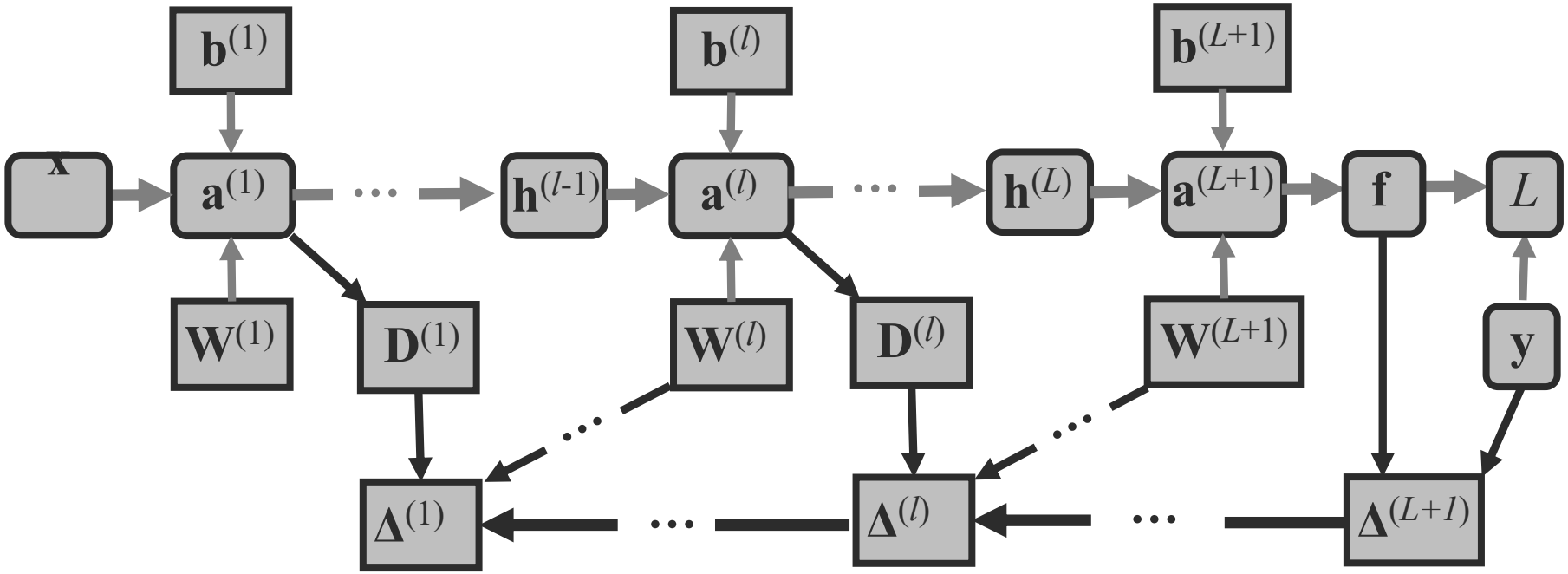
$$\mathbf{h}^{(1)} = \text{act}(\mathbf{a}^{(1)}) \qquad \mathbf{h}^{(l)} = \text{act}(\mathbf{a}^{(l)}) \qquad \mathbf{f} = \text{out}(\mathbf{a}^{(L+1)})$$



- In the forward propagation phase we compute terms of the form above
- The figure above is a type of computation graph, (which is different from the probability graphs we saw earlier)

Visualizing backpropagation

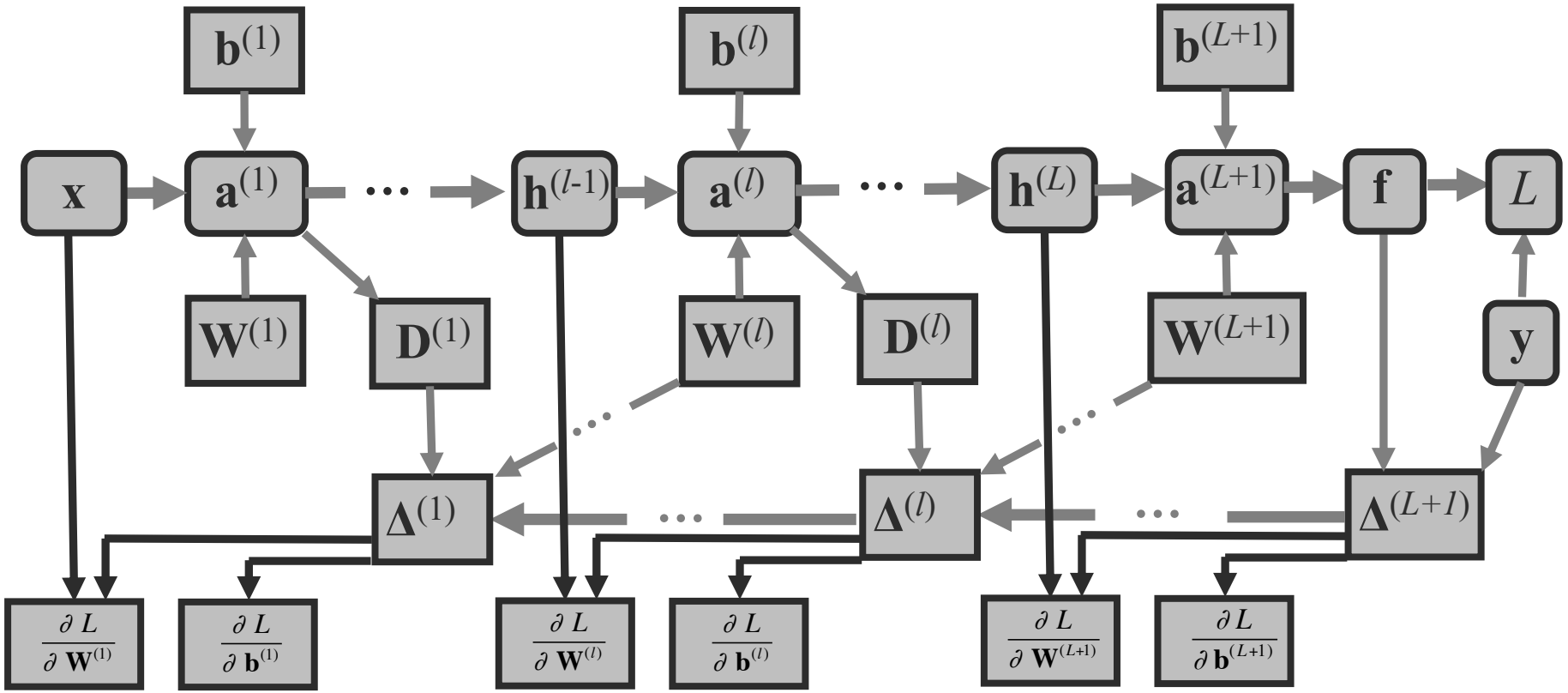
- In the backward propagation phase we compute terms of the form below



$$\Delta^{(l)} = \mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)} \qquad \frac{\partial\,\mathbf{h}^{(l)}}{\partial\,\mathbf{a}^{(l)}} = \mathbf{D}^{(l)} \qquad \frac{\partial\,\mathbf{a}^{(l+1)}}{\partial\,\mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)} \qquad \frac{\partial\,L}{\partial\,\mathbf{a}^{(L+1)}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] = -\Delta^{(L+1)}$$

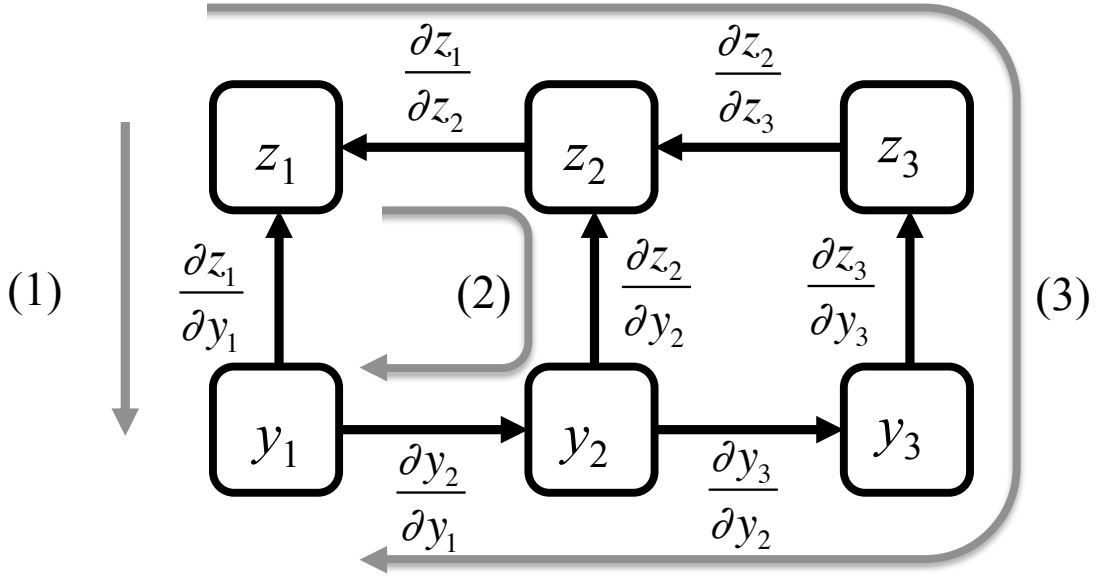
Visualizing backpropagation

- We update the parameters in our model using the simple computations below



$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = -\Delta^{(1)}\mathbf{x}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(1)}} = -\Delta^{(1)} \qquad \frac{\partial L}{\partial \mathbf{W}^{(l)}} = -\Delta^{(l)}\mathbf{h}_{(l-1)}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{b}^{(l)}} = -\Delta^{(l)}$$

Computation graphs



- For more complicated computations, computation graphs can help us keep track of how computations decompose, ex. $z_1 = z_1(y_1, z_2(y_2(y_1), z_3(y_3(y_2(y_1)))))$

$$\frac{\partial z_1}{\partial y_1} = \underbrace{\frac{\partial z_1}{\partial y_1}}_{(1)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(2)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_3}\frac{\partial z_3}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(3)}$$

Checking an implementation of backpropagation and software tools

- An implementation of the backpropagation algorithm can be checked for correctness by comparing the analytic values of gradients with those computed numerically
- For example, one can add and subtract a small perturbation to each parameter and then compute the symmetric finite difference approximation to the derivative of the loss:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta+\varepsilon)-L(\theta-\varepsilon)}{2\varepsilon}$$

- Many software packages use computation graphs to allow complex networks to be more easily defined and optimized
- Examples include: Theano, TensorFlow, Keras and Torch

Bibliographic Notes & Further Reading

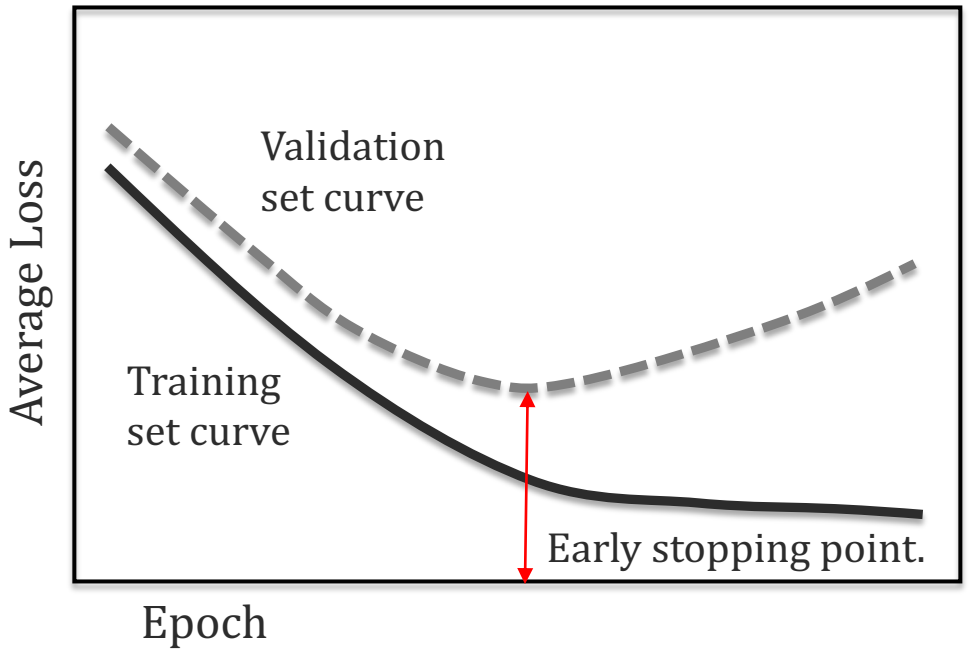
- Many neural network books (Haykin, 1994; Bishop, 1995; Ripley, 1996) do not formulate backpropagation in vector-matrix terms.
- However, recent online courses (e.g. by Hugo Larochelle), and Rojas (1996)'s text, do adopt this formulation, as we have done here

Training and evaluating deep networks

Early stopping

- Deep learning involves high capacity architectures, which are susceptible to overfitting even when data is plentiful,
- Early stopping is standard practice even when other methods to reduce overfitting are employed, ex. regularization and dropout
- The idea is to monitor learning curves that plot the average loss for the training and validation sets as a function of epoch
- The key is to find the point at which the validation set average loss begins to deteriorate

Early stopping



- In practice the curves above can be more noisy due to the use of stochastic gradient descent
- As such, it is common to keep the history of the validation set curve when looking for the minimum
  – even if it goes back up it might come back down

Validation sets and hyperparameters

- In deep learning hyperparameters are tuned by identifying what settings lead to best performance on the validation set, using early stopping
- Common hyperparameters include the strength of parameter regularization, but also model complexity in terms of the number of hidden units and layers and their connectivity, the form of activation functions, and parameters of the learning algorithm itself.
- Because of the many choices involved, performance monitoring on validation sets assumes an even more central role than it does with traditional machine learning methods.

Test sets

- *Should be set aside for a truly final evaluation*
- Repeated rounds of experiments using test set data give misleading (ex. optimistic) estimates of performance on fresh data
- For this reason, the research community has come to favor public challenges with hidden test-set labels, a development that has undoubtedly helped gauge progress in the field
- Controversy arises when participants submit multiple entries, and some favor a model where participants submit code to a competition server, so that the test data itself is hidden

Validation sets vs. cross-validation

- The use of a validation set is different from using *k*-fold cross-validation to evaluate a learning technique or to select hyperparameters.
- Cross-validation involves creating multiple training and testing partitions.
- Datasets for deep learning tend to be so massive that a single large test set adequately represents a model's performance, reducing the need for cross-validation
  - Since training often takes days or weeks, even using GPUs, cross-validation is often impractical anyway.
- If you do use cross validation you need to have an internal validation set *for each fold* to adjust hyperparameters or perform cross validation only using the training set

Validation set data and the ‘end game’

- To obtain the best possible results, one needs to tune hyperparameters, usually with a single validation set extracted from the training set.
- However, there is a dilemma: omitting the validation set from final training can reduce performance when testing
- It is advantageous to train on the combined training and validation data, but this risks overfitting.

Validation set data and the 'end game'

- One solution is to stop training after the same number of epochs that led to the best validation set performance; another is to monitor the average loss over the combined training set and stop when it reaches the level it was at when early stopping was performed using the validation set.
- One can use cross validation within the training set, treating each fold as a different validation set, then train the final model on the entire training data with the identified hyperparameters to perform the final test
- Snapshot learning may also be applied. Create an ensemble from models obtained at several different epochs

Hyperparameter tuning

- A weighted combination of L_2 and L_1 regularization is often used to regularize weights
- Hyperparameters in deep learning are often tuned heuristically by hand, or using grid search
- An alternative is random search, where instead of placing a regular grid over hyperparameter space, probability distributions are specified from which samples are taken
- Another approach is to use machine learning and Bayesian techniques to infer the next hyperparameter configuration to try in a sequence of experimental runs
- Keep in mind even things like the learning rate schedule (to be discussed) are forms of hyperparameters and you need to be careful **not to tune them on the test set**

Mini-batch based stochastic gradient descent (SGD)

- Stochastic gradient descent updates model parameters according to the gradient computed from one example
- The mini-batch variant uses a small subset of the data and bases updates to parameters on the average gradient over the examples in the batch
- This operates just like the regular procedure: initialize the parameters, enter a parameter update loop, and terminate by monitoring a validation set
- Normally these batches are randomly selected disjoint subsets of the training set, perhaps shuffled after each epoch, depending on the time required to do so

Mini-batch based SGD

- Each pass through a set of mini-batches that represent the complete training set is an *epoch*
- Using the empirical risk plus a regularization term as the objective function, updates are

$$\boldsymbol{\theta}^{\text{new}} \leftarrow \boldsymbol{\theta} - \eta_t \left[\frac{1}{B_k} \sum_{i \in \mathrm{I}} \left[\frac{\partial}{\partial \boldsymbol{\theta}} L(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i) \right] + \frac{B_k}{N} \lambda \frac{\partial}{\partial \boldsymbol{\theta}} R(\boldsymbol{\theta}) \right]$$

- η_t is the learning rate and may depend on the epoch t
- The batch is represented by a set of indices I=I(t,k) into the original data; the kth batch has B_k examples
- N is the size of the training set
- $L(f(\mathbf{x}_i;\theta),\mathbf{y}_i)$ is the loss for example $\mathbf{x}_i$, label $\mathbf{y}_i$, params θ
- R(θ) is the regularizer, with weight λ

Mini-batches

- Typically contain two to several hundred examples
  - For large models the choice may be constrained by resources
- Batch size often influences the stability and speed of learning; some sizes work particularly well for a given model and data set.
- Sometimes a search is performed over a set of potential batch sizes to find one that works well, before doing a lengthy optimization.
- The mix of class labels in the batches can influence the result
  - For unbalanced data there may be an advantage in pre-training the model using mini-batches in which the labels are balanced, then fine-tuning the upper layer or layers using the unbalanced label statistics.

Momentum

- As with regular gradient descent, 'momentum' can help the optimization escape plateaus in the loss
- Momentum is implemented by computing a moving average:

$$\Delta\theta = -\eta\nabla_{\theta}L(\theta) + \alpha\Delta\theta^{\text{old}}$$

  - where the first term is the current gradient of the loss times a learning rate
  - the second term is the previous update weighted by $\alpha \in [0,1]$
- Since the mini- batch approach operates on a small subset of the data, this averaging can allow information from other recently seen mini-batches to contribute to the current parameter update
- A momentum value of 0.9 is often used as a starting point, but it is common to hand-tune it, the learning rate, and the schedule used to modify the learning rate during the training process

Learning rate schedules

- The learning rate is a critical choice when using mini-batch based stochastic gradient descent.
- Small values such as 0.001 often work well, but it is common to perform a logarithmically spaced search, say in the interval [10^{-8}, 0.9], followed by a finer grid or binary search.
- The learning rate may be adapted over epochs *t* to give a learning rate schedule, ex.

$$\eta_t = \eta_0 (1 + \varepsilon t)^{-1}$$

- A fixed learning rate is often used in the first few epochs, followed by a decreasing schedule
- **Many** other options, ex. divide the rate by 10 when the validation error rate ceases to improve

Mini-batch SGD pseudocode

$\theta = \theta_o$ // initialize parameters

$\Delta\theta = 0$

$t = 0$

while converged == FALSE

$\{I_1, \ldots, I_K\} = \text{shuffle}(X)$ // create K mini-batches

for $k = 1 \ldots K$

$$\mathbf{g} = \frac{1}{B_k} \sum_{i \in I_k} \left[\frac{\partial}{\partial \theta} L(f(\mathbf{x}_i; \theta), \mathbf{y}_i) \right] + \frac{B_k}{N} \lambda \frac{\partial}{\partial \theta} R(\theta)$$

$\Delta\theta \leftarrow -\eta_t \mathbf{g} + \alpha \Delta\theta$

$\theta \leftarrow \theta + \Delta\theta$

end

$t = t + 1$

end

Dropout

- A form of regularization that randomly deletes units and their connections during training
- Intention: reducing hidden unit co-adaptation & combat over-fitting
- Has been argued it corresponds to sampling from an exponential number of networks with shared parameters & missing connections
- One averages over models at test time by using original network without dropped-out connections, but with scaled-down weights

Dropout

- If a unit is retained with probability *p* during training, its outgoing weights are rescaled or multiplied by a factor of *p* at test time
- By performing dropout a neural network with *n* units can be made to behave like an **ensemble** of 2^n smaller networks
- One way to implement it is with a binary mask vector $\mathbf{m}^{(l)}$ for each hidden layer *l* in the network: the dropped out version of $\mathbf{h}^{(l)}$ masks out units from the original version using element-wise multiplication, $\mathbf{h}_d^{(l)} = \mathbf{h}^{(l)} \odot \mathbf{m}^{(l)}$
- If the activation functions lead to diagonal gradient matrices, the backpropagation update is $\boldsymbol{\Delta}^{(l)} = \mathbf{d}^{(l)} \odot \mathbf{m}^{(l)} \odot (\mathbf{W}^{(l+1)} \boldsymbol{\Delta}^{(l+1)})$.

Batch normalization

- A way of accelerating training which many studies have found to be important to obtain state-of-the-art results
- Each element of a layer is normalized to zero mean and unit variance based on its statistics within a mini-batch
  - This can change the network's representational power
- Each activation has a learned scaling and shifting parameter
- Mini-batch based SGD is modified by calculating the mean μ_j and variance σ_j^2 over the batch for each hidden unit h_j in each layer, then the units are normalized
- Scale them using the learned scaling parameter γ_j and shift them by the learned shifting parameter β_j such that

$$\hat{h}_j \leftarrow \gamma_j \frac{h_j - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} + \beta_j.$$

- To update the γ_j and β_j one needs to backpropagate the gradient of the loss through these additional parameters

Parameter initialization

- Can be deceptively important!
- Bias terms are often initialized to 0 with no issues
- Weight matrices more problematic, ex.
  - If initialized to all 0s, can be shown that the tanh activation function will yield zero gradients
  - If the weights are all the same, hidden units will produce same gradients and behave the same as each other (wasting params)
- One solution: initialize all elements of weight matrix from uniform distribution over interval [–b, b]
- Different methods have been proposed for selecting the value of b, often motivated by the idea that units with more inputs should have smaller weights
- Weight matrices of rectified linear units have been successfully initialized using a zero-mean isotropic Gaussian distribution with standard deviation of 0.01

Unsupervised pre-training

- Idea: model the distribution of unlabeled data using a method that allows the parameters of the learned model to inform or be somehow transferred to the network
- Can be an effective way to both initialize and regularize a feedforward network
- Particularly useful when the volume of labeled data is small relative to the model's capacity.
- The use of activation functions such as rectified linear units (which improve gradient flow in deep networks), along with good parameter initialization techniques, can mitigate the need for sophisticated pre-training methods

Data augmentation

- Can be critical for best results
- As seen in MNIST table, augmenting even a large dataset with transformed data can increase performance
- A simple transformation is simply to jiggle the image
- If the object to be classified can be cropped out of a larger image, random bounding boxes can be placed around it, adding small translations in the vertical and horizontal directions
- Can also use rotations, scale changes, and shearing
- There is a hierarchy of rigid transformations that increase in complexity as parameters are added which can be used

Bibliographic Notes & Further Reading

- Bergstra and Bengio (2012) give empirical and theoretical justification for the use of random search for hyperparameter settings.
- Snoek et al. (2012) propose the use of Bayesian learning methods to infer the next hyperparameter setting to explore, and their Spearmint software package performs Bayesian optimizations of both deep network hyperparameters and general machine learning algorithm hyperparameters.
- Stochastic gradient descent methods go back at least as far as Robbins and Monro (1951).

Bibliographic Notes & Further Reading

- Bottou (2012) is an excellent source of tips and tricks for learning with stochastic gradient descent, while Bengio (2012) gives further practical recommendations for training deep networks.
- Glorot and Bengio (2010) cover various weight matrix initialization heuristics, and how the concepts of fan-in and fan-out can be used to justify them for networks with different kinds of activation functions.
- The origins of dropout and more details about it can be found in Srivastava et al. (2014).
- Ioffe and Szegedy (2015) proposed batch normalization and give more details on its implementation.

Convolutional neural networks

Convolutional neural networks (CNNs)

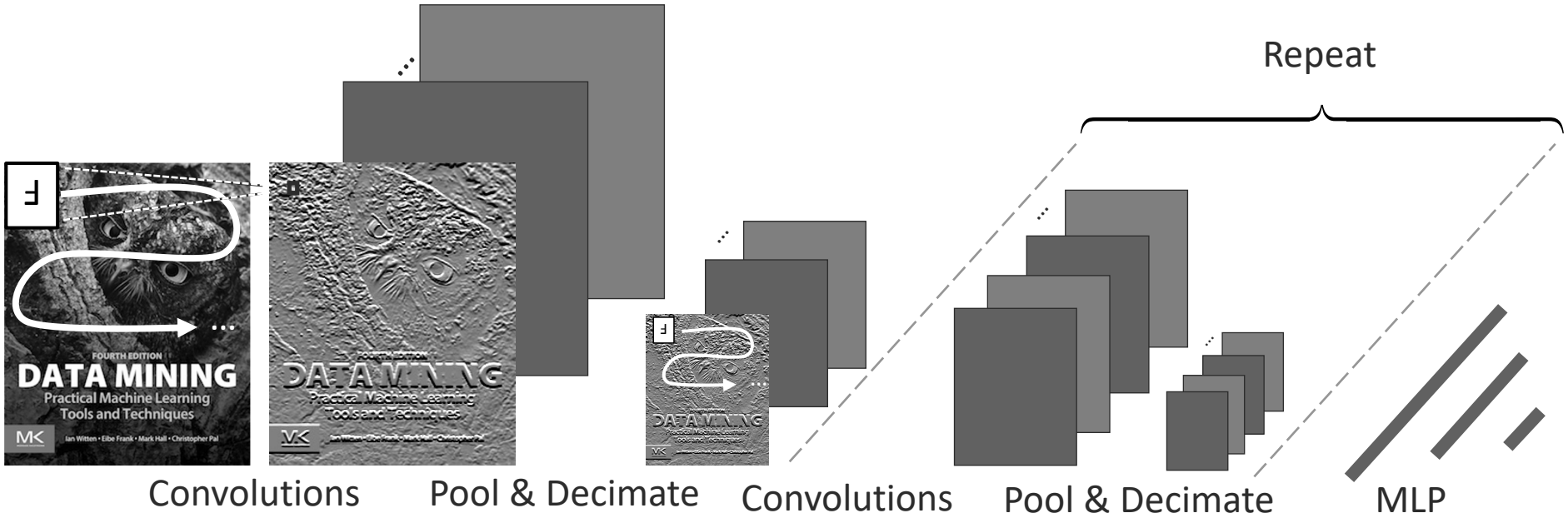
- Are a special kind of feedforward network that has proven *extremely* successful for image analysis
- Imagine filtering an image to detect edges, one could think of edges as a useful set of spatially organized ‘features’
- Imagine now if one could learn many such filters jointly along with other parameters of a neural network on top
- Each filter can be implemented by multiplying a relatively small spatial zone of the image by a set of weights and feeding the result to an activation function – just like those discussed above for vanilla feedforward networks
- Because this filtering operation is simply repeated around the image using the same weights, it can be implemented using convolution operations
- The result is a CNN for which it is possible to learn both the filters and the classifier using SGD and the backpropagation algorithm

Deep CNNs

- In a convolutional neural network, once an image has been filtered by several learnable filters, each filter bank's output is often aggregated across a small spatial region, using the average or maximum value.
- Aggregation can be performed within non-overlapping regions, or using subsampling, yielding a lower-resolution layer of spatially organized features—a process that is sometimes referred to as "decimation"
- This gives the model a degree of invariance to small differences as to exactly where a feature has been detected.
- If aggregation uses the max operation, a feature is activated if it is detected anywhere in the pooling zone
- The result can be filtered and aggregated again

A typical CNN architecture

- Many feature maps are obtained from convolving learnable filters across an image
- Results are aggregated or pooled & decimated
- Process repeats until last set of feature maps are given to an MLP for final prediction



CNNs in practice

- LeNet and AlexNet architectures are canonical models
- While CNNs are designed to have a certain degree of translational invariance, augmenting data through global synthetic transformations like the cropping trick can increase performance significantly
- CNNs are usually optimized using mini-batch-based stochastic gradient descent, so practical discussions above about learning deep networks apply
- The use of GPU computing is typically *essential* to accelerate convolution operations significantly
- Resource issues related to the amount of CPU vs. GPU memory available are often important to consider

The ImageNet challenge

- Crucial in demonstrating the effectiveness of deep CNNs
- Problem: recognize object categories in Internet imagery
- The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) classification task - classify image from Flickr and other search engines into 1 of 1000 possible object categories
- Serves as a standard benchmark for deep learning
- The imagery was hand-labeled based on the presence or absence of an object belonging to these categories
- There are 1.2 million images in the training set with 732-1300 training images available per class
- A random subset of 50,000 images was used as the validation set, and 100,000 images were used for the test set where there are 50 and 100 images per class respectively

A plateau, then rapid advances

- "Top-5 error" is the % of times that the target label does not appear among the 5 highest-probability predictions
- Visual recognition methods not based on deep CNNs hit a plateau in performance at 25%

Name	Layers	Top-5 Error (%)	References
AlexNet	8	15.3	Krizhevsky et al. (2012)
VGG Net	19	7.3	Simonyan and Zisserman (2014)
ResNet	152	3.6	He et al. (2016)

- Note: the performance for human agreement has been measured at 5.1% top-5 error
- Smaller filters have been found to lead to superior results in deep networks: the methods with 19 and 152 layers use filters of size 3×3

Starting simply: image filtering

- When an image is filtered, the output can be thought of as another image that contains the filter's response at each spatial location
- Consider filtering a 1D vector **x** by multiplication with a matrix **W** that has a special structure, such as

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \begin{bmatrix} w_1 & w_2 & w_3 & & & \\ & w_1 & w_2 & w_3 & & \\ & & & \ddots & & \\ & & & w_1 & w_2 & w_3 \end{bmatrix}$$

where the elements left blank in the matrix above are zero and we have used a simple filter having only three non-zero coefficients and a "stride" of one

Correlation and convolution

- Suppose our filter is centered, giving the first vector element an index of –1, or an index of –*K*, where *K* is the "radius" of the filter, then 1D filtering can be written

$$\mathbf{y}[n] = \sum_{k=-K}^{K} \mathbf{w}[k]\mathbf{x}[n+k]$$

- Directly generalizing this filtering to a 2D image **X** and filter **W** gives the *cross-correlation*, **Y**=**W**★**X**, for which the result for row *r* and column *c* is

$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[j,k]\mathbf{X}[r+j,c+k]$$

- The *convolution* of an image with a filter, **Y**=**W***__X__ , is obtained by simply flipping the sense of the filter

$$\mathbf{Y}[r,c] = \sum_{j=-J}^{J} \sum_{k=-K}^{K} \mathbf{W}[-j,-k]\mathbf{X}[r+j,c+k]$$

Simple filtering example

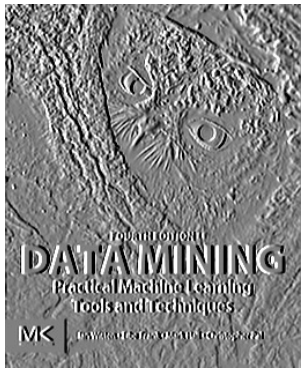
- Ex. consider the task of detecting edges in an image
- A well known technique is to filter an image with so-called “Sobel” filters, which involves convolving it with

$$\mathbf{W}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad \mathbf{W}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

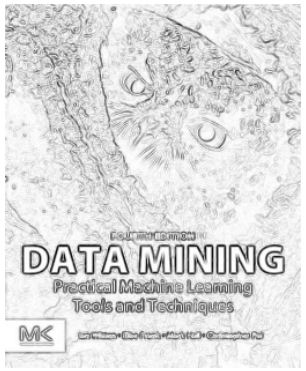
- Applied to the image X below, we have:

$\mathbf{X}$

$\mathbf{G}_x = \mathbf{W}_x * \mathbf{X}$



$\mathbf{G}_y = \mathbf{W}_y * \mathbf{X}$



$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$

Visualizing the filters learned by a CNN

- Learned edge-like filters and texture-like filters are frequently observed in the early layers of CNNs trained using natural images
- Since each layer in a CNN involves filtering the feature map below, so as one moves up the receptive fields become larger
- Higher- level layers learn to detect larger features, which often correspond to textures, then small pieces of objects

First Layer

Second Layer

Third Layer

  - (Imagery kindly provided by Matthew Zeiler)
- Above are the strongest activations of random neurons projecting the activation back into image space using the deconvolution approach of Zeiler and Fergus (2013).

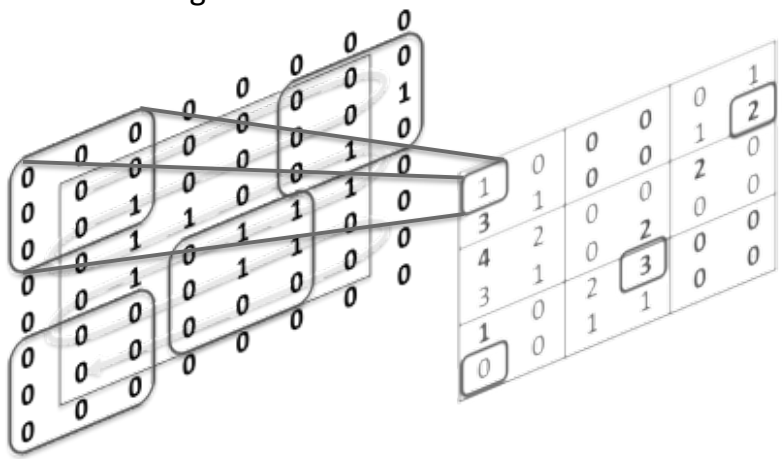
Simple example of: convolution, pooling, and decimation operations

Filter

-1	0	1
-2	0	2
-1	0	1

Image

Convolution



Max Pooling

3	3	0	0	2	2
3	3	0	0	2	2
4	4	2	2	2	2
4	4	2	2	2	2
1	1	3	3	0	0
1	1	3	3	0	0

Decimation

3	0	2
4	2	2
1	3	0

- An image is convolved with a filter; curved rectangular regions in the first large matrix depict a random set of image locations
- Maximum values within small 2×2 regions are indicated in bold in the central matrix
- The results are pooled, using max-pooling then decimated by a factor of two, to yield the final matrix

Convolutional layers and gradients

- Let's consider how to compute the gradients needed to optimize a convolutional network
- At a given layer we have i=1…$N^{(l)}$ feature filters and corresponding feature maps
- The convolutional kernel matrices $\mathbf{K}_i$ contain flipped weights with respect to kernel weight matrices $\mathbf{W}_i$
- With activation function act(), and for each feature type i, a scaling factor g_i and bias matrix $\mathbf{B}_i$, the feature maps are matrices $\mathbf{H}_i(\mathbf{A}_i(\mathbf{X}))$ and can be visualized as a set of images given by

$$\mathbf{H}_i = g_i \,\mathrm{act}\left[\mathbf{K}_i * \mathbf{X} + \mathbf{B}_i\right] = g_i \,\mathrm{act}\left[\mathbf{A}_i(\mathbf{X})\right]$$

Convolutional layers and gradients

- The loss is a function of the $N^{(l)}$ feature maps for a given layer, $L = L(\mathbf{H}_1^{(l)},\ldots,\mathbf{H}_{N^{(l)}}^{(l)})$
- Define **h**=vec(**H**), **x**=vec(**X**), **a**=vec(**A**), where the vec() function returns a vector with stacked columns of the given matrix argument,
- Choose an act() function that operates elementwise on an input matrix of pre-activations and has scale parameters of 1 and biases of 0.
- Partial derivatives of hidden layer output with respect to input **X** of the convolutional units are

$$\frac{\partial L}{\partial \mathbf{X}} = \sum_i \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{X}} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} = \sum_i \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}} \frac{\partial \mathbf{h}_i}{\partial \mathbf{a}_i} \frac{\partial L}{\partial \mathbf{h}_i} = \sum_i \left[\mathbf{W}_i * \mathbf{D}_i\right], \mathbf{D}_i = dL/\partial \mathbf{A}_i$$

Convolutional layers and gradients

- Matrix $\mathbf{D}_i$ in previous side is a matrix containing the partial derivative of the elementwise act() function's input with respect to its pre-activation value for the i^{th} feature type, organized according to spatial positions given by row j and column k.
- Intuitively, the result is a sum of the convolution of each of the (zero padded) filters $\mathbf{W}_i$ with an image-like matrix of derivatives $\mathbf{D}_i$.
- The partial derivatives of the hidden layer output are

$$\frac{\partial L}{\partial \mathbf{W}_i} = \sum_j \sum_k \frac{\partial a_{ijk}}{\partial \mathbf{W}_i} \frac{\partial \mathbf{H}_i}{\partial a_{ijk}} \frac{\partial L}{\partial \mathbf{H}_i} = \left[\mathbf{X}^{\dagger} * \mathbf{D}_i \right],$$

- Where $\mathbf{X}^{\dagger}$ is the row & column-flipped version of $\mathbf{X}$

Pooling and subsampling layers

- What are the consequences of backpropagating gradients through max or average pooling layers?
- In the former case, the units that are responsible for the maximum within each zone *j*, *k* —the "winning units"— get the backpropagated gradient
- For average pooling, the averaging is simply a special type of convolution with a fixed kernel that computes the (possibly weighted) average of pixels in a zone
  - the required gradients are therefore like std conv. layers
- The subsampling step either samples every n^{th} output, or avoids needless computation by only evaluating every n^{th} pooling computation

Implementing CNNs

- Convolutions are very well suited for acceleration using GPUs
- Since graphics hardware can accelerate convolutions by an *order of magnitude* or more over CPU implementations, they play an important often *critical* role in training CNNs
- An experimental turn-around time of days rather than weeks makes a huge difference to model development times!
- Can also be challenging to construct software for learning a convolutional neural network in such a way that alternative architectures can be explored
- Early GPU implementations were hard to extend, newer tools allow for both fast computation and flexible high-level programming primitives
- Many software tools allow gradient computations and the backpropagation algorithm for large networks to be almost completely automated.

Bibliographic Notes & Further Reading

Convolutional Networks

- Modern convolutional neural networks are widely acknowledged as having their roots with the “neocognitron” proposed by Fukushima (1980); however
- The work of LeCun et al. (1998) on the LeNet convolutional network architecture has been extremely influential.
- The MNIST dataset containing 28×28 pixel images of handwritten digits has been popular in deep learning research community since 1998
- However, it was the ImageNet challenge (Russakovsky et al., 2015), with a variety of much higher resolutions, that catapulted deep learning into the spotlight in 2012.
  - The winning entry from the University of Toronto (Krizhevsky et al. , 2012) processed the images at a resolution of 256×256 pixels.
  - Up till then, CNNs were simply incapable of processing such large volumes of imagery at such high resolutions in a reasonable amount of time.

Bibliographic Notes & Further Reading

Convolutional Networks

- Krizhevsky et al. (2012)'s dramatic ImageNet win used a GPU accelerated convolutional neural networks.
  - This spurred a great deal of development, reflected in rapid subsequent advances in visual recognition performance and on the ImageNet benchmark.
- In the 2014 challenge, the Oxford Visual Geometry Group and a team from Google pushed performance even further using much deeper architectures: 16-19 weight layers for the Oxford group, using tiny 3×3 convolutional filters (Simonyan and Zisserman, 2014); 22 layers, with filters up to 5×5 for the Google team (Szegedy et al., 2015).
- The 2015 ImageNet challenge was won by a team from Microsoft Research Asia (MSRA) using an architecture with 152 layers (He et al., 2015), using tiny 3×3 filters combined with "shortcut" connections that skip over layers, and pooling and decimating the result of multiple layers of small convolution operations

Bibliographic Notes & Further Reading

Convolutional Networks

- Good parameter initialization can be critical for the success of neural networks, as discussed in LeCun et al. (1998)'s classic work and the more recent work of Glorot and Bengio (2010).
- Krizhevsky et al. (2012)'s convolutional network of rectified linear units (ReLUs) initialized weights using 0-mean isotropic Gaussian distributions with a standard deviation of 0.01, and initialized the biases to 1 for most hidden convolutional layers as well as their model's hidden fully connected layers.
- They observed that this initialization accelerated the early phase of learning by providing ReLUs with positive inputs.

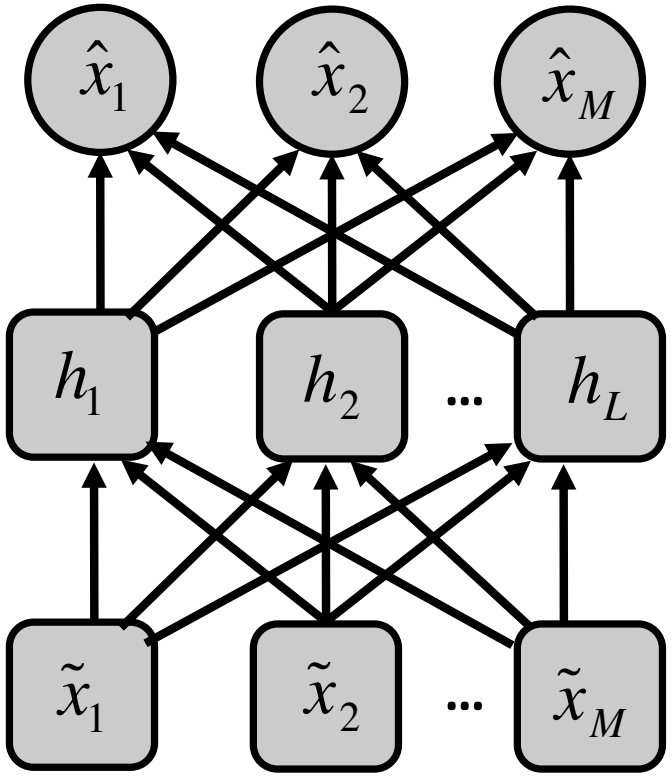
Autoencoders

Autoencoders

- Used for unsupervised learning
- It is a network that learns an efficient coding of its input.
- The objective is simply to reconstruct the input, but through the intermediary of a compressed or reduced-dimensional representation.
- If the output is formulated using probability, the objective function is to optimize $p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}})$, that is, the probability that the model gives a random variable **x** the value $\hat{\mathbf{x}}$ given the observation $\tilde{\mathbf{x}}$, where $\hat{\mathbf{x}} = \tilde{\mathbf{x}}$.
- In other words, the model is trained to predict its own input—but it must map it through a representation created by the hidden units of a network.

A simple autoencoder

- Predicts its own input, ex. $p(\hat{\mathbf{x}} \mid \tilde{\mathbf{x}}) = p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}}; \mathbf{f}(\tilde{\mathbf{x}}))$
- Going through an encoding, $\mathbf{e} = \mathbf{h} = \text{act}(\mathbf{a}^{(1)})$



where

$f(\hat{x})=f(d(e(\hat{x})))$,
$d=\text{out}(\mathbf{a}^{(2)})$
$\mathbf{a}^{(2)}=\mathbf{W}^T\mathbf{h}+\mathbf{b}^{(2)}$
$\mathbf{h}=\text{act}(\mathbf{a}^{(1)})$,
$\mathbf{a}^{(1)}=\mathbf{W}^T\tilde{x} + \mathbf{b}^{(1)}$

Autoencoders

- Since the idea of an autoencoder is to compress the data into a lower-dimensional representation, the number *L* of hidden units used for encoding is less than the number *M* in the input and output layers
- Optimizing the autoencoder using the negative log probability over a data set as the objective function leads to the usual forms
- Like other neural networks it is typical to optimize autoencoders using backpropagation with mini-batch based SGD

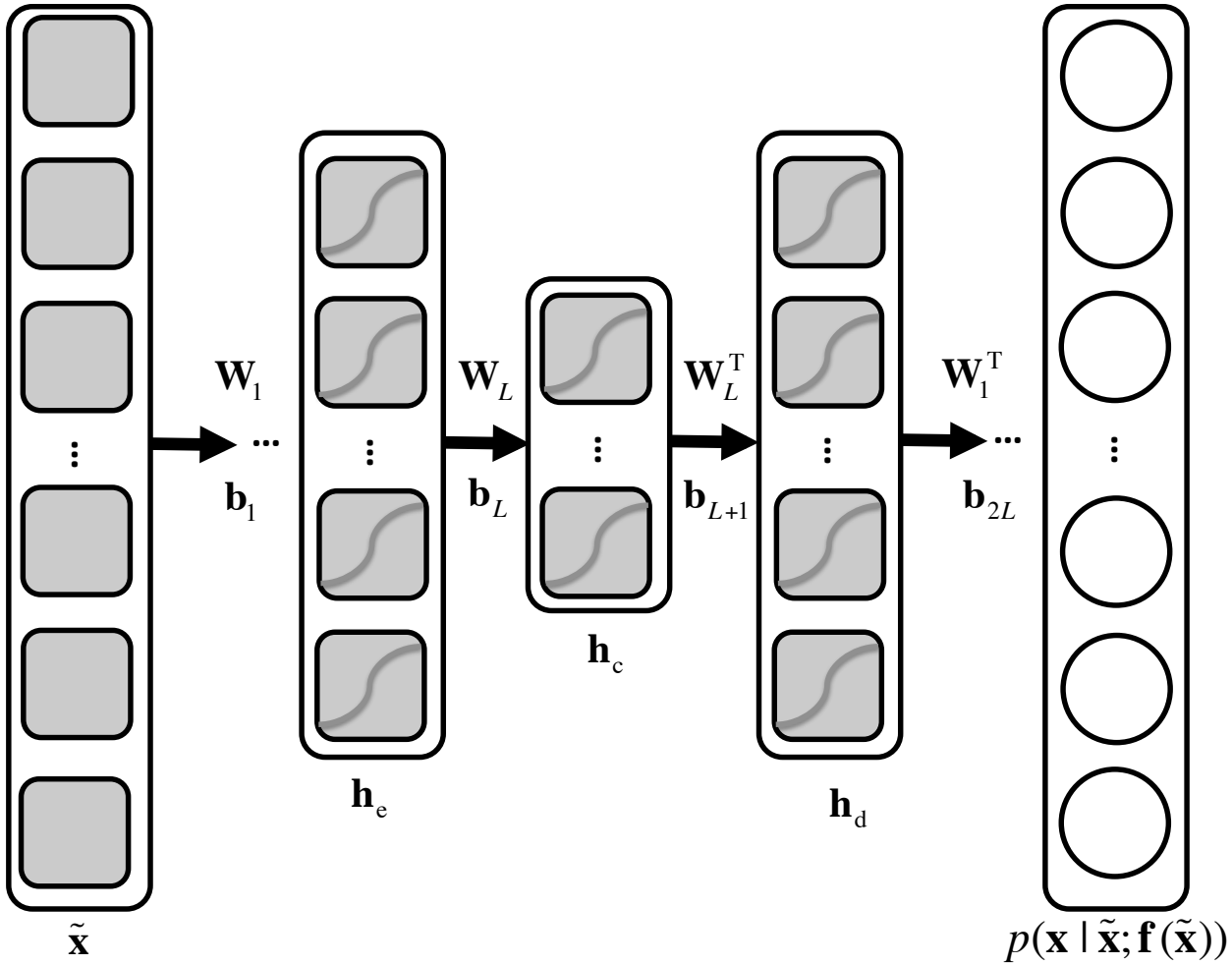
Linear autoencoders and PCA

- Both the encoder activation function act() and the output activation function out() in the simple autoencoder model could be defined as the sigmoid function
- However, it can be shown that with no activation function, $\mathbf{h}^{(i)}=\mathbf{a}^{(i)}$, the resulting "linear autoencoder" will find the same subspace as PCA, (assuming a squared-error loss function and normalizing the data using mean centering)
  - Can be shown to be optimal in the sense that any model with a non-linear activation function would require a weight matrix with more parameters to achieve the same reconstruction error
- Even with non-linear activation functions such as a sigmoid, optimization finds solutions where the network operates in the linear regime, replicating the behavior of PCA
- This might seem discouraging; however, using a neural network with even one hidden layer to create much more flexible transformations, and
  - There is growing evidence deeper models can learn more useful representations

Deep autoencoders

- When building autoencoders from more flexible models, it is common to use a *bottleneck* in the network to produce an under-complete representation, providing a mechanism to obtain an encoding of lower dimension than the input.
- Deep autoencoders are able to learn low-dimensional representations with smaller reconstruction error than PCA using the same number of dimensions.
- Can be constructed by using L layers to create a hidden layer representation $\mathbf{h}_c^{(L)}$ of the data, and following this with a further L layers $\mathbf{h}_d^{(L+1)}$… $\mathbf{h}_d^{(2L)}$ to decode the representation back into its original form
- The j=1,…,2L weight matrices for each of the i=1,…,L encoding and decoding layers are constrained by $\mathbf{W}_{L+i} = \mathbf{W}_{L+1-i}^{\mathrm{T}}$

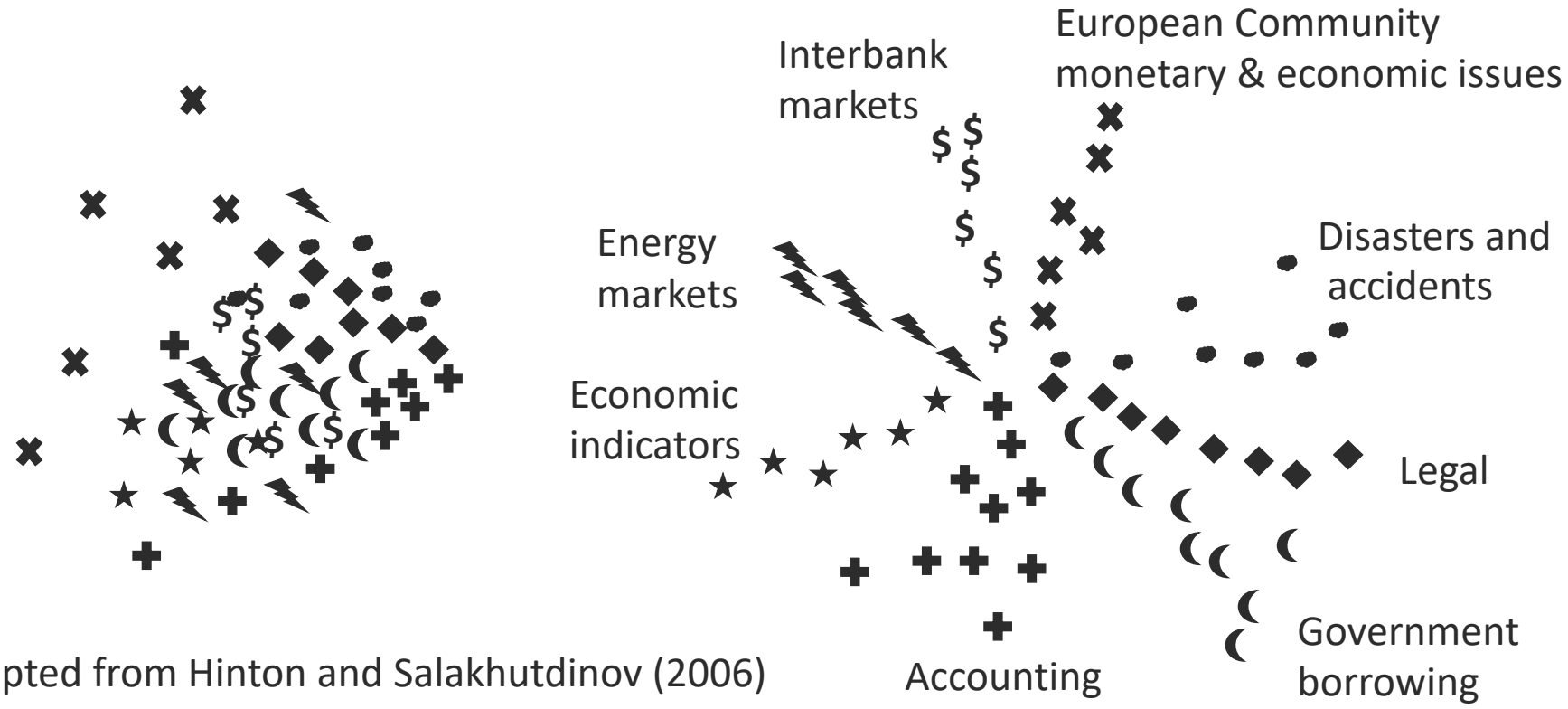
A deep autoencoder



$$\mathbf{f}(\mathbf{x}) = \mathbf{f}_{\mathbf{d}}(\mathbf{a}_{\mathbf{d}}^{(2L)}(\ldots\mathbf{h}_{\mathbf{d}}^{(L+1)}(\mathbf{a}_{\mathbf{d}}^{(L+1)}(\mathbf{h}_{\mathbf{c}}^{(L)}(\mathbf{a}_{\mathbf{e}}^{(L)}(\ldots\mathbf{h}_{\mathbf{e}}^{(1)}(\mathbf{a}_{\mathbf{e}}^{(1)}(\mathbf{x}))))))).$$

Deep autoencoders

- A comparison of data projected into a 2D space with PCA (left) vs a deep autoencoder (right) for a text dataset
- The non-linear autoencoder can arrange the learned space in such that it better separates natural groupings of the data



Adapted from Hinton and Salakhutdinov (2006)

Training autoencoders

- Deep autoencoders are an effective framework for non-linear dimensionality reduction
- It can be difficult to optimize autoencoders; being careful about activation function choice and initialization can help
- Once such a network has been built, the top-most layer of the encoder, the code layer $\mathbf{h}_c$, can be input to a supervised classification procedure
- One can pre-train a discriminative neural net with an autoencoder
- One can also use a composite loss from the start, with a reconstructive (unsupervised) and a discriminative (supervised) criterion

$$L(\theta) = (1-\lambda)L_{\text{sup}}(\theta) + \lambda L_{\text{unsup}}(\theta)$$

  where λ is a hyperparameter that balances the two objectives

- Another approach to training autoencoders is based on pre-training by stacking two-layered restricted Boltzmann machines RBMs

Denoising autoencoders

- Autoencoders can be trained layerwise, using autoencoders as the underlying building blocks
- One can use greedy layerwise training strategies involving plain autoencoders to train deep autoencoders, but attempts to do this for networks of even moderate depth have been problematic
- Procedures based on stacking denoising autoencoders have been found to work better
- Denoising autoencoders are trained to remove different types of noise that has been added synthetically to their input
- Autoencoder inputs can be corrupted with noise such as: Gaussian noise; masking noise, where some elements are set to 0; and salt-and-pepper noise, where some elements are set to minimum and maximum input values (such as 0 and 1)

Bibliographic Notes & Further Reading

Autoencoders

- Hinton and Salakhutdinov (2006) noted that it has been known since the 1980s that deep autoencoders, optimized through backpropagation, could be effective for non-linear dimensionality reduction.
- The key limiting factors were the small size of the datasets used to train them, coupled with low computation speeds; plus the old problem of local minima.
- By 2006, datasets such as the MNIST digits and the 20 Newsgroups collection were large enough, and computers were fast enough, for Hinton and Salakhutdinov to present compelling results illustrating the advantages of deep autoencoders over principal component analysis.
  - Their experimental work used generative pre-training to initialize weights to avoid problems with local minima.

Bibliographic Notes & Further Reading

Autoencoders

- Bourlard and Kamp (1988) provide a deep analysis of the relationships between autoencoders and principal component analysis.
- Vincent et al. (2010) proposed stacked denoising autoencoders and found that they outperform both stacked standard autoencoders and models based on stacking restricted Boltzmann machines.
- Cho and Chen (2014) produced state-of-the-art results on motion capture sequences by training deep autoencoders with rectified linear units using hybrid unsupervised and supervised learning.

Stochastic methods

Boltzmann machines

- Are a type of Markov random field often used for unsupervised learning
- Unlike the units of a feedforward neural network, the units in Boltzmann machines correspond to random variables, such as are used in Bayesian networks
- Older variants of Boltzmann machines were defined using exclusively binary variables, but models with continuous and discrete variables are also possible
- They became popular prior to the impressive results of convolutional neural networks on the ImageNet challenge, but have since waned in popularity because they are more difficult to work with

Boltzmann machines

- To create a Boltzmann machine we partitioning variables into ones that are visible, using a *D*-dimensional binary vector **v**, and ones that are hidden, defined by a *K*-dimensional binary vector **h**
- A Boltzmann machine is a joint probability model of the form

$$p(\mathbf{v},\mathbf{h};\theta) = \frac{1}{Z(\theta)}\exp(-E(\mathbf{v},\mathbf{h};\theta)), \quad Z(\theta) = \sum_{\mathbf{v}}\sum_{\mathbf{h}}\exp(-E(\mathbf{v},\mathbf{h};\theta)),$$

$$E(\mathbf{v},\mathbf{h};\theta) = -\frac{1}{2}\mathbf{v}^T\mathbf{A}\mathbf{v} - \frac{1}{2}\mathbf{h}^T\mathbf{B}\mathbf{h} - \mathbf{v}^T\mathbf{W}\mathbf{h} - \mathbf{a}^T\mathbf{v} - \mathbf{b}^T\mathbf{h},$$

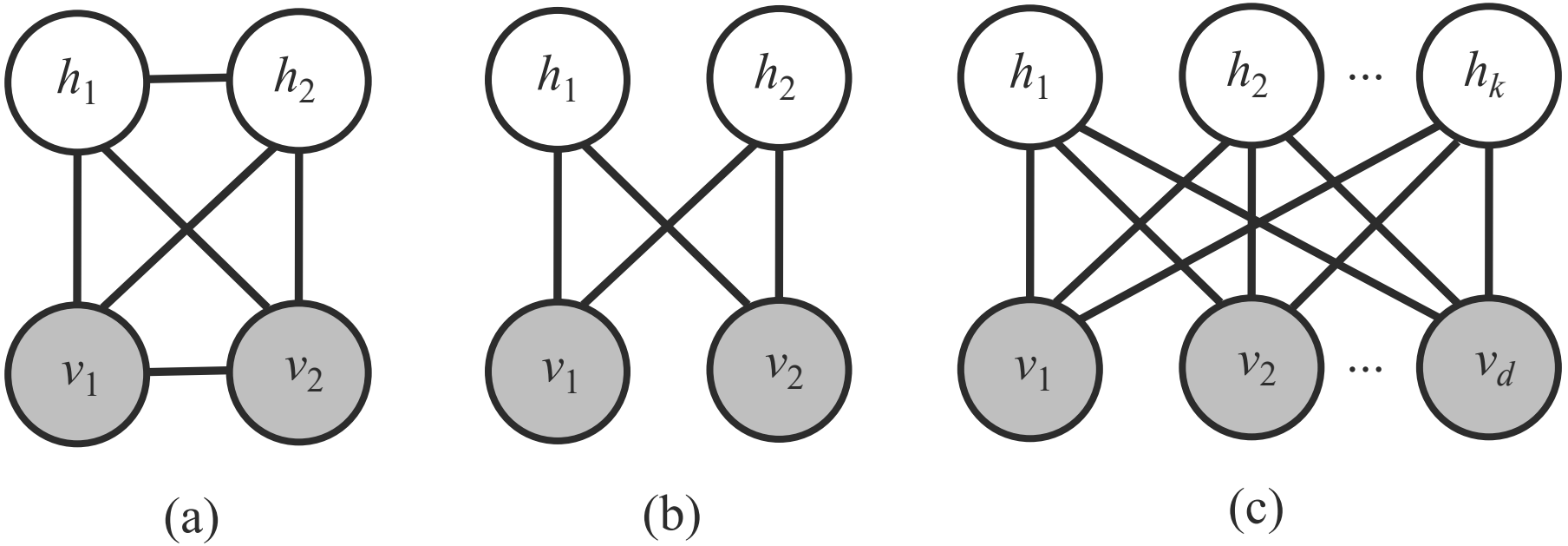
- where E(**v**,**h**;θ) is the energy function
- Z(θ) normalizes *E* so that it defines a valid joint probability
- matrices **A**, **B** and **W** encode the visible-to-visible, hidden-to-hidden and the visible-to-hidden variable interactions respectively
- vectors **a** and **b** encode the biases associated with each variable
- matrices **A** and **B** are symmetric, and their diagonal elements are 0

Boltzmann machines

(a) Boltzman Machines are binary Markov random field with pairwise connections between all variables

(b) Restricted Boltzmann machines (RBMs) do not have connections between the variables in a layer

(c) RBMs can be extended to many variables as shown



Key feature of Boltzmann machines

- A key feature of Boltzmann machines (and binary Markov random fields in general) is that the conditional distribution of one variable given the others is a sigmoid function whose argument is a weighted linear combination of the states of the other variables

$$p(h_j = 1 \mid \mathbf{v}, \mathbf{h}_{\neg j}; \theta) = \text{sigmoid}\left(\sum_{i=1}^{D} W_{ij} v_i + \sum_{k=1}^{K} B_{jk} h_k + b_j \right),$$

$$p(v_i = 1 \mid \mathbf{h}, \mathbf{v}_{\neg i}; \theta) = \text{sigmoid}\left(\sum_{j=1}^{K} W_{ij} h_j + \sum_{d=1}^{D} A_{id} v_d + c_i \right),$$

where the notation $\neg i$ indicates all elements with subscript other than *i*.

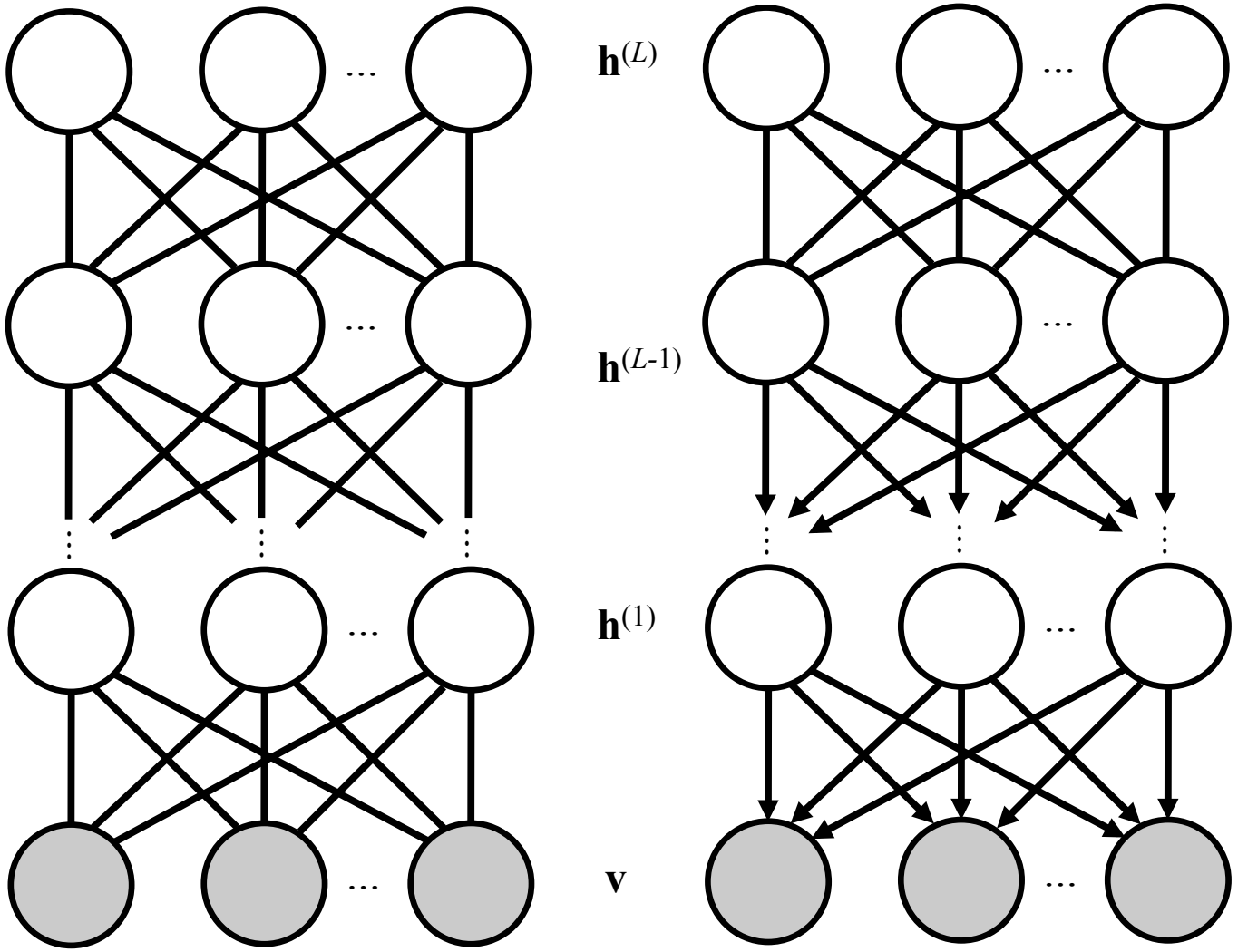
Contrastive divergence

- Running a Gibbs sampler for a Boltzmann machine often requires many iterations,
- A technique called “contrastive divergence” is a popular alternative that initializes the sampler to the observed data instead of randomly and performs a limited number of Gibbs updates.
- In an RBM a sample can be generated from the sigmoid distributions for all the hidden variables given the observed; then samples can be generated for the observed variables given the hidden variable sample
- This single step often works well in practice, although the process of alternating the sampling of hidden and visible units can be continued for multiple steps.

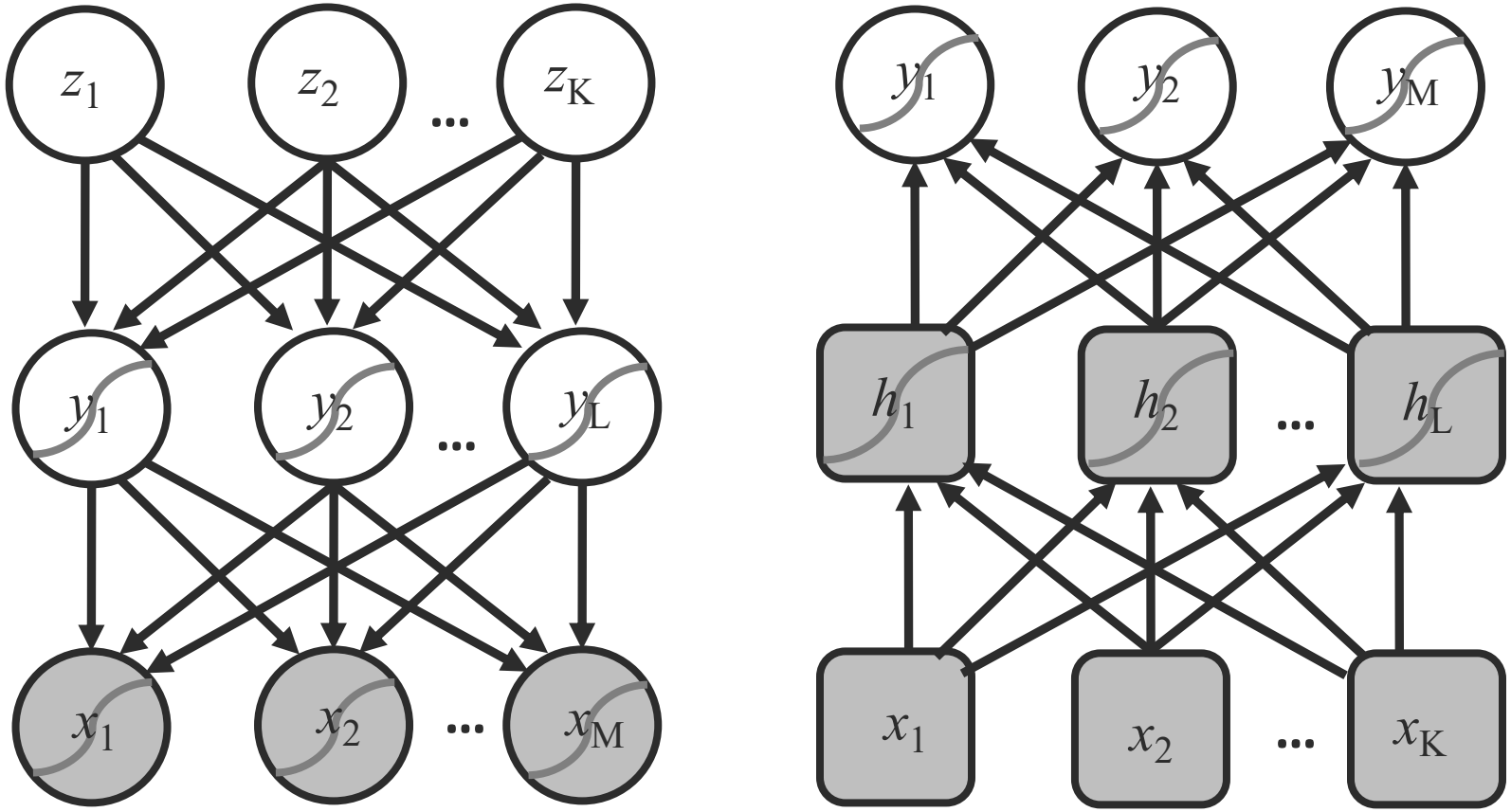
Deep RBMs and deep belief networks

- Deep Boltzmann machines involve coupling layers of random variables using restricted Boltzmann machine connectivity
- While any deep Bayesian network is technically a deep belief network, the term “deep belief network” has become strongly associated with a particular type of deep architecture that can be constructed by training restricted Boltzmann machines incrementally.
- The procedure is based on converting the lower part of a growing model into a Bayesian belief network, adding an RBM for the upper part of the model, then continuing the training, conversion and stacking process.

A deep RBM vs a deep belief network



A sigmoidal belief network vs a neural network



Bibliographic Notes & Further Reading

Stochastic methods – Boltzmann machines

- The history of Markov random fields has roots in statistical physics in the 1920s with so-called “Ising models” of ferromagnetism
- Our presentation of Boltzmann machines follows Hinton and Sejnowski (1983), but we use matrix-vector notation and our exposition more closely resembles formulations such as that of Salakhutdinov and Hinton (2009)
- Harmonium networks proposed in Smolensky (1986) are essentially equivalent to what are now commonly referred to as restricted Boltzmann machines
- Contrastive divergence was proposed by Hinton (2002)

Bibliographic Notes & Further Reading

Stochastic methods – Boltzmann machines

- The idea of using unsupervised pre-training to initialize deep networks using stacks of restricted Boltzmann machines was popularized by Hinton and Salakhutdinov (2006)
- Salakhutdinov and Hinton (2009) give further details on the use of deep Boltzmann machines and training procedures for deep belief networks, including other nuances for greedy training of deep restricted Boltzman machines
- Neal (1992) introduced sigmoidal belief networks
- Welling et al. (2004) showed how to extend Boltzmann machines to categorical and continuous variables using exponential-family models
- A greedy layer-wise training procedure for deep Boltzmann machines was proposed by Hinton and Salakhutdinov (2006) and refined by Murphy (2012)

Bibliographic Notes & Further Reading

Stochastic methods – Boltzmann machines

- Hybrid supervised and unsupervised learning procedures for restricted Boltzman machines were proposed by McCallum et al. (2005) and further explored by Larochelle and Bengio (2008).
- Vincent et al. (2010) proposed the autoencoder approach to unsupervised pre-training; they also explored various layer-wise stacking and training strategies and compared stacked restricted Boltzmann machines with stacked autoencoders.

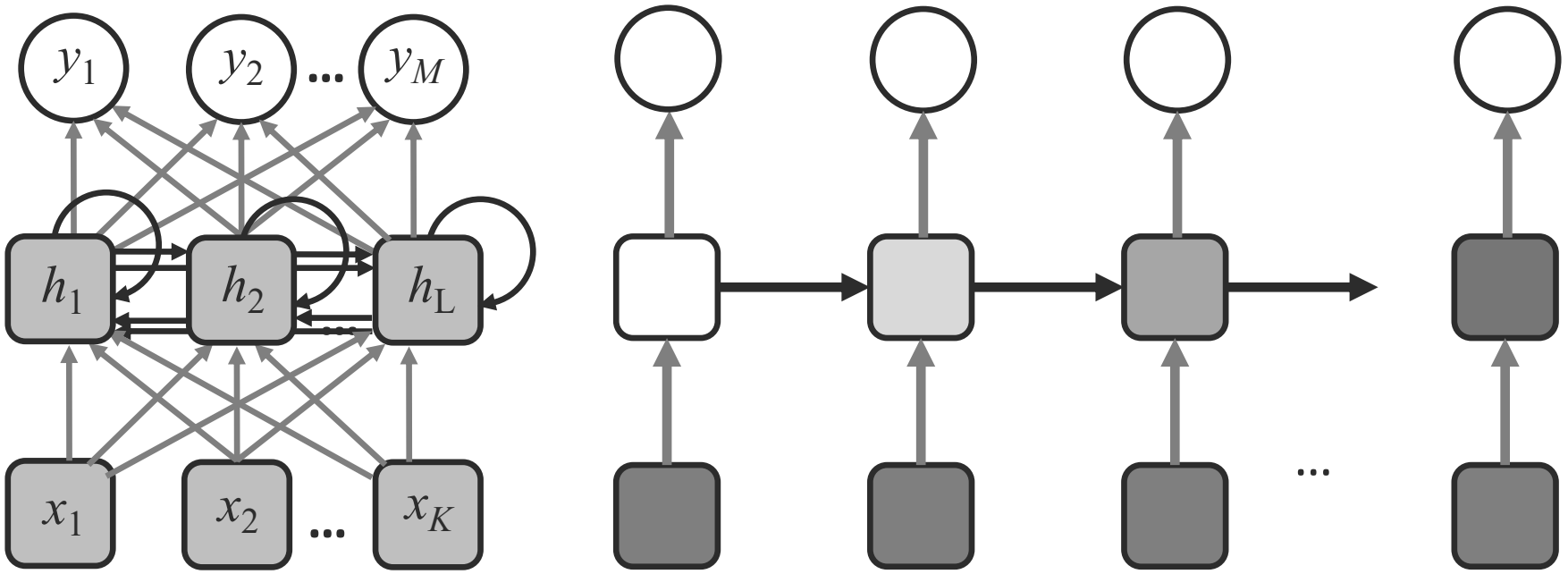
Recurrent neural networks

Recurrent neural networks

- Recurrent neural networks are networks with connections that form directed cycles.
- As a result, they have an internal state, which makes them prime candidates for tackling learning problems involving sequences of data—such as handwriting recognition, speech recognition, and machine translation.
- A feedforward network can be transformed into a recurrent network by adding connections from all hidden units h_i to h_j.
- Each hidden unit has connections to both itself and other hidden units.
- Imagine unfolding a recurrent network over time by following the sequence of steps that perform the underlying computation.
- Like a hidden Markov model, a recurrent network can be unwrapped and implemented using the same weights and biases at each step to link units over time.

Recurrent neural networks (RNNs)

- An RNN can be unwrapped and implemented using the same weights and biases at each step to link units over time as shown below
- The resulting unwrapped RNN is similar to a hidden Markov model, but keep in mind that the hidden units in RNNs are not stochastic



Recurrent neural networks (RNNs)

- Recurrent neural networks apply linear matrix operations to the current observation and the hidden units from the previous time step, and the resulting linear terms serve as arguments of activation functions act():

$$\mathbf{h}_t = \text{act}(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{o}_t = \text{act}(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$$

- The same matrix $\mathbf{U}_h$ is used at each time step
- The hidden units in the previous step $\mathbf{h}_{t-1}$ influence the computation of $\mathbf{h}_t$ where the current observation contributes to a $\mathbf{W}_h\mathbf{x}$ term that is combined with $\mathbf{U}_h\mathbf{h}_{t-1}$ and bias $\mathbf{b}_h$ terms
- Both $\mathbf{W}_h$ and $\mathbf{b}_h$ are typically replicated over time
- The output layer is modeled by a classical neural network activation function applied to a linear transformation of the hidden units, the operation is replicated at each step.

The loss, exploding and vanishing gradients

- The loss for a particular sequence in the training data can be computed either at each time step or just once, at the end of the sequence.
- In either case, predictions will be made after many processing steps and this brings us to an important problem.
- The gradient for feedforward networks decomposes the gradient of parameters at layer *l* into a term that involves the product of matrix multiplications of the form $\mathbf{D}^{(l)}\mathbf{W}^{T(l+1)}$ (see the analysis for feedforward networks above)
- A recurrent network uses the same matrix at each time step, and over many steps the gradient can very easily either diminish to zero or explode to infinity—just as the magnitude of any number other than one taken to a large power either approaches zero or increases indefinitely

Dealing with exploding gradients

- The use of L_1 or L_2 regularization can mitigate the problem of exploding gradients by encouraging weights to be small.
- Another strategy is to simply detect if the norm of the gradient exceeds some threshold, and if so, scale it down.
- This is sometimes called gradient (norm) clipping where for a gradient vector $\mathbf{g}$ and threshold T,

$$\text{if} \quad \|\mathbf{g}\| \geq T \quad \text{then} \quad \mathbf{g} \leftarrow \frac{T}{\|\mathbf{g}\|}\mathbf{g}$$

where T is a hyperparameter, which can be set to the average norm over several previous updates where clipping was not used.

LSTMs and vanishing gradients

- The so-called “long short term memory” (LSTM) RNN architecture was specifically created to address the vanishing gradient problem.
- Uses a combination of hidden units, elementwise products and sums between units to implement gates that control “memory cells”.
- Memory cells are designed to retain information without modification for long periods of time.
- They have their own input and output gates, which are controlled by learnable weights that are a function of the current observation and the hidden units at the previous time step.
- As a result, *backpropagated error terms from gradient computations can be stored and propagated backwards without degradation*.

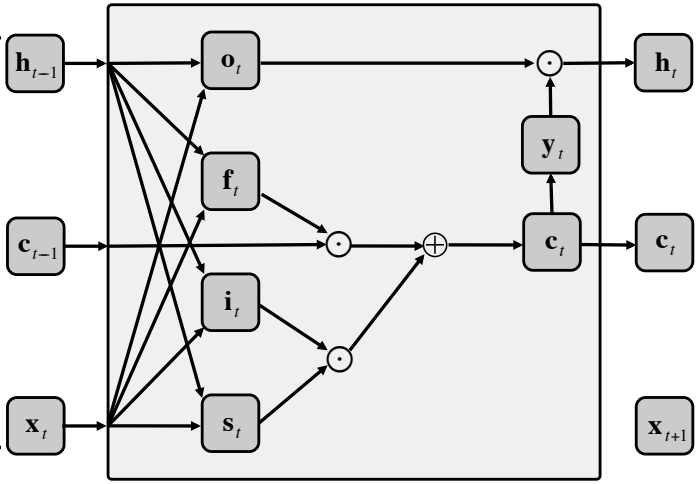
LSTMs and vanishing gradients

- The original LSTM formulation consisted of *input gates* and *output gates*, but *forget gates* and “peephole weights” were added later.
- Below we present the most popular variant of LSTM RNNs which does not include peephole weights, but which does use forget gates.
- The architecture is complex, but has produced state-of-the-art results on a wide variety of problems.

LSTM architecture

- At each time step there are three types of gates: input $\mathbf{i}_t$, forget $\mathbf{f}_t$, and output $\mathbf{o}_t$.
- Each are a function of both the underlying input $\mathbf{x}_t$ at time *t* as well as the hidden units at time *t*-1, $\mathbf{h}_{t-1}$
- Each gate multiplies $\mathbf{x}_t$ by its own gate specific **W** matrix, by its own **U** matrix, and adds its own bias vector **b**.
- This is usually followed by the application of a sigmoidal elementwise non-linearity.

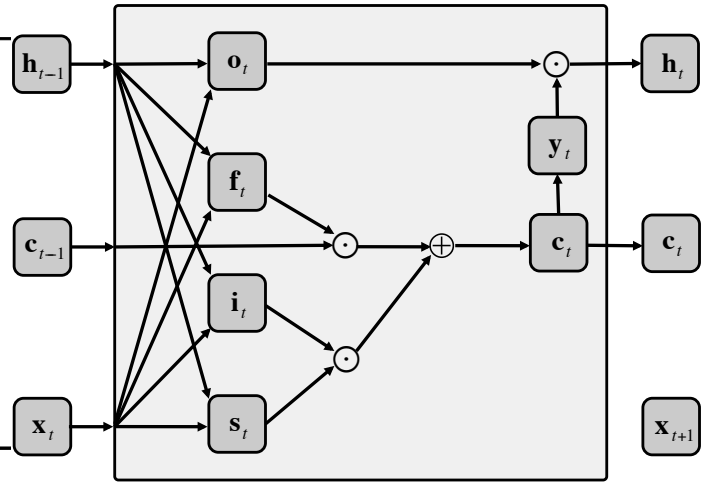
LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$



LSTM architecture

- At each time step *t*, input gates $\mathbf{i}_t$ are used to determine when a potential input given by $\mathbf{s}_t$ is important enough to be placed into the memory unit or cell, $\mathbf{c}_t$
- Forget gates $\mathbf{f}_t$ allow memory unit content to be erased
- Output gates $\mathbf{o}_t$ determine whether $\mathbf{y}_t$, the content of the memory units transformed by activation functions, should be placed in the hidden units $\mathbf{h}_t$
- Typical gate activation functions and their dependencies are shown below

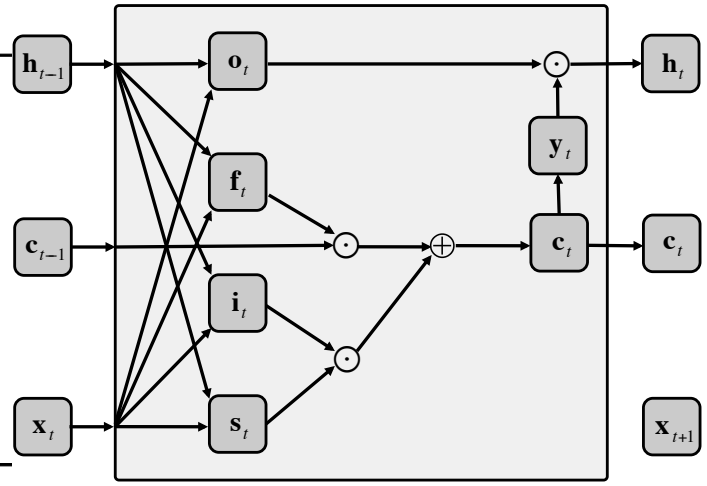
LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$



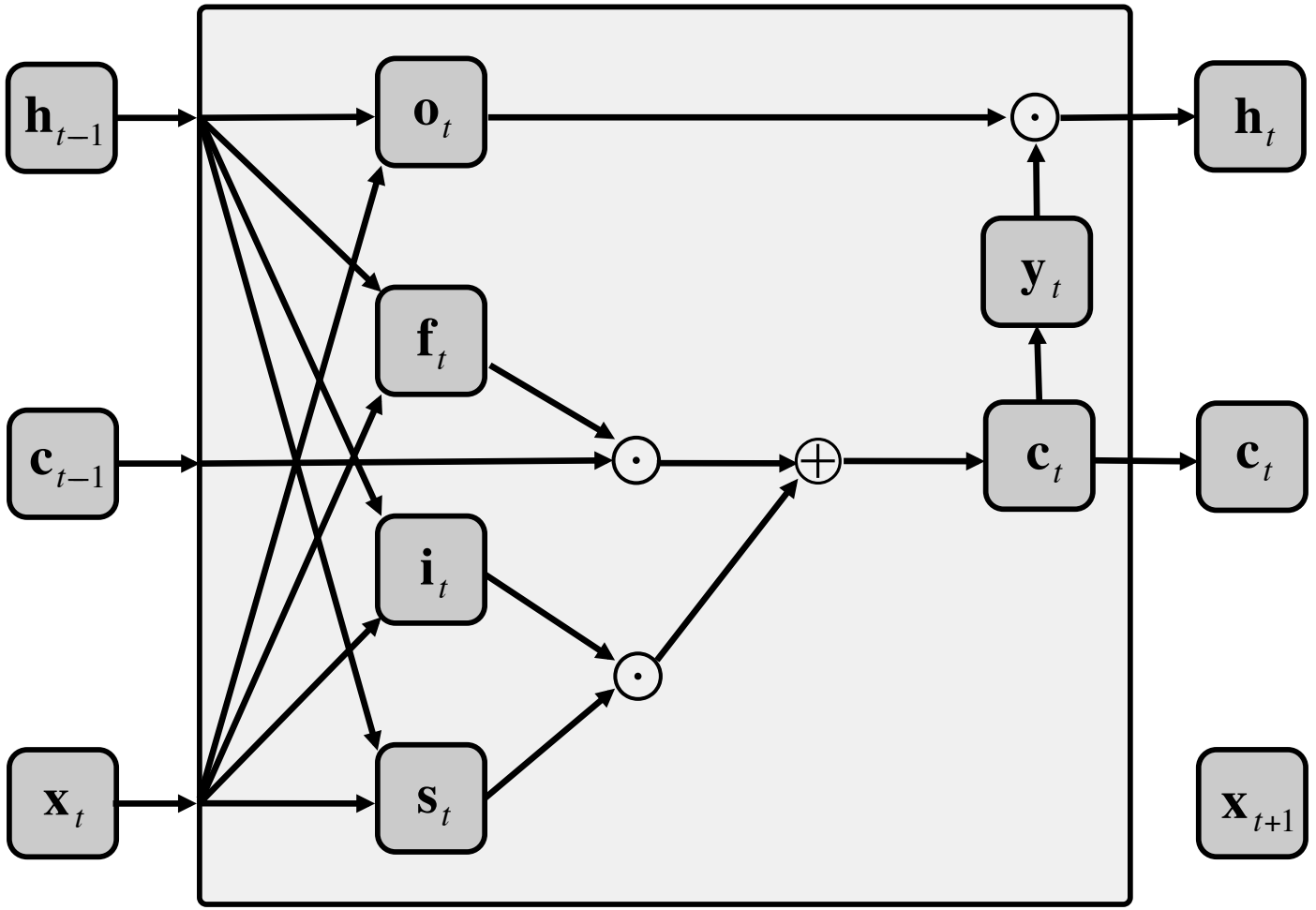
LSTM achitecture

- The final gating is implemented as an elementwise product between the output gate and the transformed memory contents, $\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
- Memory units are typically transformed by the tanh function prior to the gated output, such that $\mathbf{y}_t$=tanh($\mathbf{c}_t$)
- Memory units or cells are updated by $\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$ an elementwise product between the forget gates and the previous contents of the memory units, plus the elementwise product of the input gates and the new potential inputs .

LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$



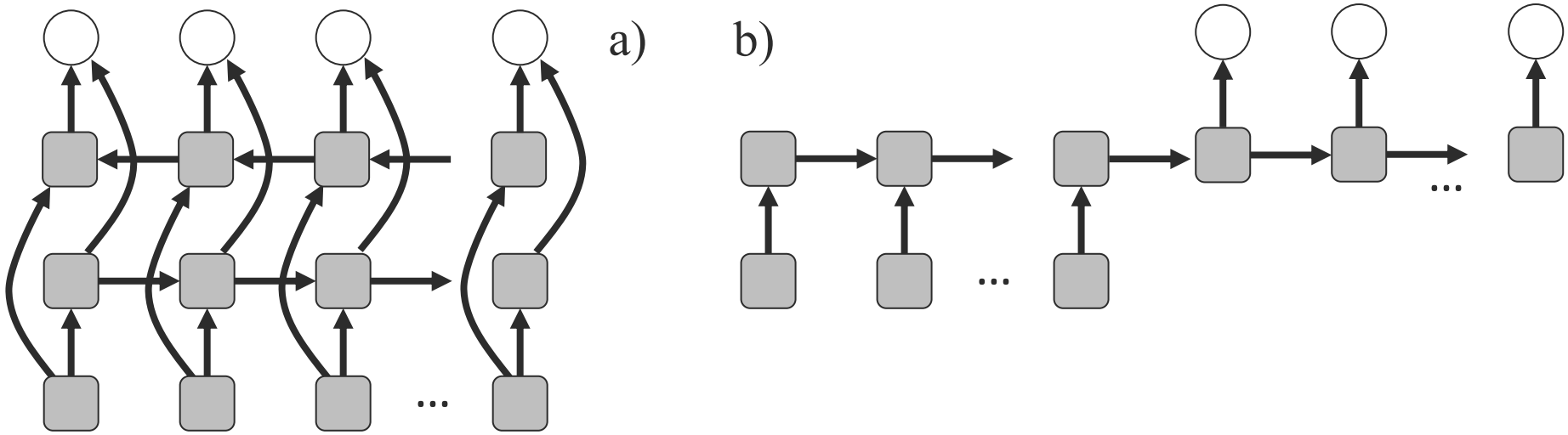
LSTM achitecture



Other RNN architectures

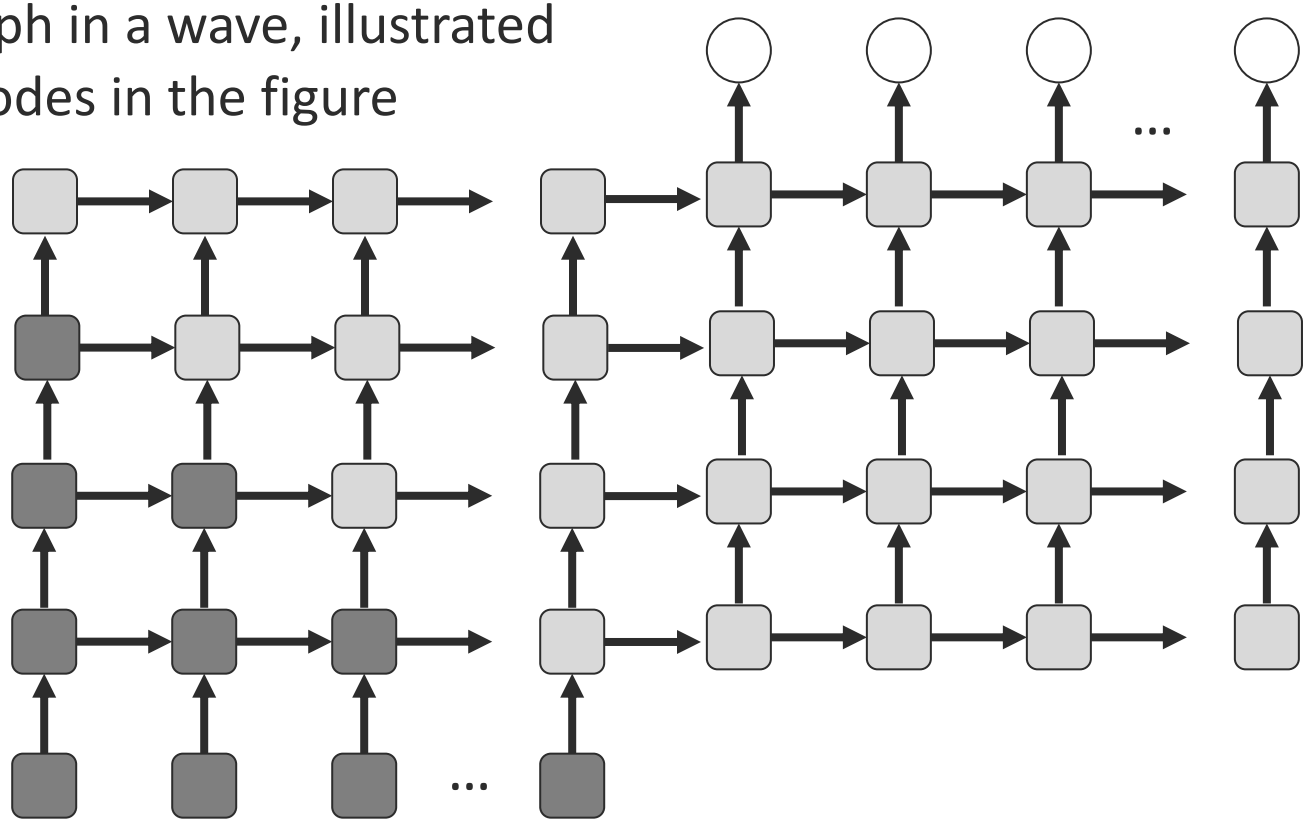
a) Recurrent networks can be made bidirectional, propagating information in both directions
   - They have been used for a wide variety of applications, including protein secondary structure prediction and handwriting recognition

b) An "encoder-decoder" network creates a fixed-length vector representation for variable-length inputs, the encoding can be used to generate a variable-length sequence as the output
   - Particularly useful for machine translation



Deep encoder-decoder architectures

- Given enough data, a deep encoder-decoder architecture such as that below can yield results that compete with translation systems that have been hand-engineered over decades of research.
- The connectivity structure means that partial computations in the model can flow through the graph in a wave, illustrated by the darker nodes in the figure



Bibliographic Notes & Further Reading

Recurrent neural networks

- Graves et al. (2009) demonstrate how recurrent neural networks are particularly effective at handwriting recognition,
- Graves et al. (2013) apply recurrent neural networks to speech.
- The form of gradient clipping presented above was proposed by Pascanu et al. (2013).
- Hochreiter and Schmidhuber (1997) is the seminal work on the "Long Short-term Memory" architecture for recurrent neural networks;
  - our explanation follows Graves and Schmidhuber (2005)'s formulation.
- Greff et al. (2015)'s paper "LSTM: A search space odyssey" explored a wide variety of variants and finds that:
  a) none of them significantly outperformed the standard LSTM architecture; and
  b) forget gates and the output activation function were the most critical components. Forget gates were added by Gers et al. (2000).

Bibliographic Notes & Further Reading

Recurrent neural networks

- IRNNs were proposed by Le et al. (2015)
- Chung et al. (2014) proposed gated recurrent units
- Schuster and Paliwal (1997) proposed bidirectional recurrent neural networks
- Chen and Chaudhari (2004) used bi-directional networks for protein structure prediction; Graves et al. (2009) used them for handwriting recognition
- Cho et al. (2014) used encoder-decoder networks for machine translation, while Sutskever et al. (2014) proposed deep encoder-decoder networks and used them with massive quantities of data
- For further accounts of advances in deep learning and a more extensive history of the field, consult the reviews of LeCun et al. (2015), Bengio (2009), and Schmidhuber (2015)

Deep learning software

Theano

- A library in Python which has been developed with the specific goal of facilitating research in deep learning (Bergstra et al., 2010; Theano Development Team, 2016)
- It is also a powerful general purpose tool for general mathematical programming
- Theano extends NumPy (the main Python package for scientific computing) by adding symbolic differentiation and GPU support, among various other functions
- It provides a high-level language for creating the mathematical expressions that underlie deep learning models, and a compiler that takes advantage of deep learning techniques, including calls to GPU libraries, to produce code that executes quickly
- Theano supports execution on multiple GPUs

Deep learning software

Theano -> Aesara

- Allows the user to declare symbolic variables for inputs and targets, and supply numerical values only when they are used
- Shared variables such as weights and biases are associated with numerical values stored in NumPy arrays
- Aesara creates symbolic graphs as a result of defining mathematical expressions involving the application of operations to variables
  - These graphs consist of *variable*, *constant*, *apply* and *operation* nodes
  - Constants, and constant nodes, are a subclass of variables, and variable nodes, which hold data that will remain constant and can therefore be subjected to various optimizations by the compiler
- Aesara is an open-source project using a BSD license

Deep learning software

Tensor Flow

- C++ and Python based software library for the types of numerical computation typically associated with deep learning (Abadi et al., 2016)
- It is heavily inspired by Theano, and, like it, uses dataflow graphs to represent the ways in which multidimensional data arrays communicate between one another
- These multidimensional arrays are referred to as “tensors.”
- Tensor Flow also supports symbolic differentiation and execution on multiple GPUs
- It was released in 2015 and is available under the Apache 2.0 license

Deep learning software

Torch -> PyTorch

- An open-source machine learning library built using C and a high-level scripting language known as Lua (Collobert et al., 2011)
- It uses multidimensional array data structures, and supports various basic numerical linear algebra manipulations
- It has a neural network package with modules that permit the typical forward and backward methods needed for training neural networks
- It also supports automatic differentiation

Deep learning software

Computational Network Toolkit (CNTK)

- C++ library for manipulating computational networks (Yu et al., 2014)
- It was produced by Microsoft Research, but has been released under a permissive license
- It has been popular for speech and language processing, but also supports convolutional networks of the type used for images
- It supports execution on multiple machines and using multiple GPUs

Deep learning software

Caffe

- C++ and Python based BSD-licensed convolutional neural network library (Jia et al., 2014).
- Has a clean and extensible design which makes it a popular alternative to the original open-source implementation of Krizhevsky et al. (2012)'s famous AlexNet that won the 2012 ImageNet challenge.

Deep learning software

Deeplearning4j

- Java-based open-source deep learning library available under the Apache 2.0 license
- Uses an multidimensional array class and provides linear algebra and matrix manipulation support similar to that provided by Numpy

Deep learning software

Keras and cuDNN

- Keras is a Python library that runs on top of TensorFlow (Chollet, 2015) that allows one to quickly define a network architecture in terms of layers and also includes functionality for image and text preprocessing
- cuDNN is a highly optimized GPU library for NVIDIA units that allows deep learning networks to be trained more quickly
  - It can dramatically accelerate the performance of a deep network and is often called by the other packages above.

Weka support for deep learning

Deep learning can be implemented in WEKA using three methods:

- With the wrapper classifiers for the third-party *DeepLearningForJ* package that are available in the *deepLearningForJ* package
- Using the MLRClassifier from the *RPlugin* package to exploit deep learning implementations in R
- By accessing Python-based deep learning libraries using the *PyScript* package

You have reached the end of the lecture.

Reference:

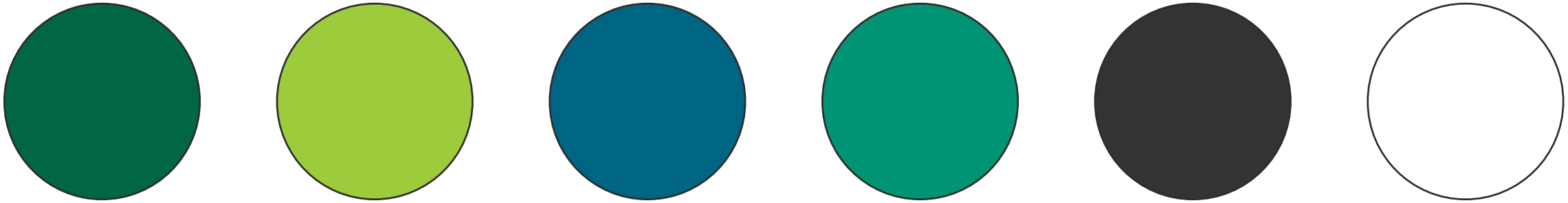
I. H. Witten, E. Frank,M. A. Hall and C. J. Pal(2016).*Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann

Design Overview

Title (Tahoma– 40)

Subtitle (Tahoma– 24)

Body (Calibri-16-20)



006646
(00,102,70)

9CCB3B
(156, 203, 59)

006484
0, 100, 131

009374
(0, 147, 116)

333333
(51, 51, 51)

FFFFFF
(255, 255, 255)

Assets 01 Structural Assets

Label

Label
To emphasize and draw attention to a main point or content. To label content

Main Point
Use to present main points in a list.
can be used in cyan, slate and lime.

01

Numbered Point
Use to present main steps/process/points that require numbering, or lettering.

Arrow
highlight different elements of an image

A Clickable Quiz

To highlight important quotes.

- Angela Forero

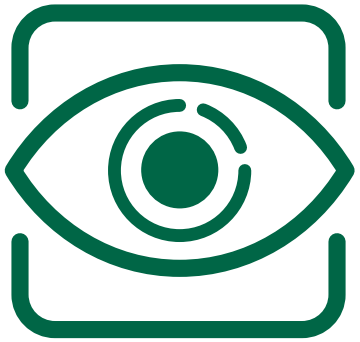
Propose a question using this asset

Important information using this asset

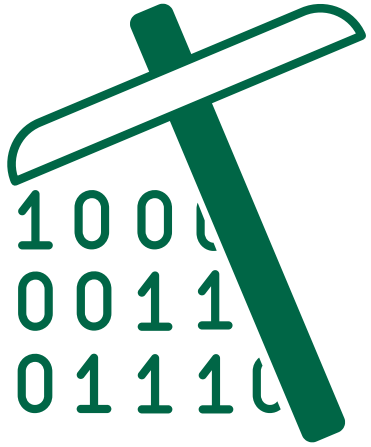
Term: definition

Title Slide Course Icons

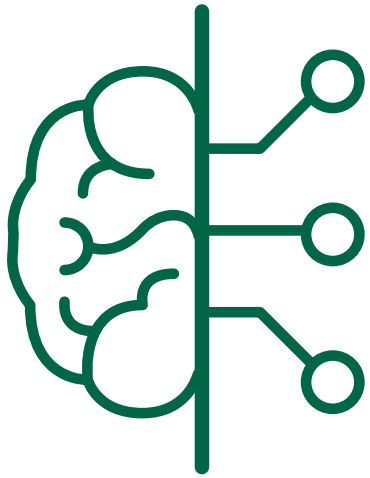
Intro Artificial Intelligence



Computer Vision



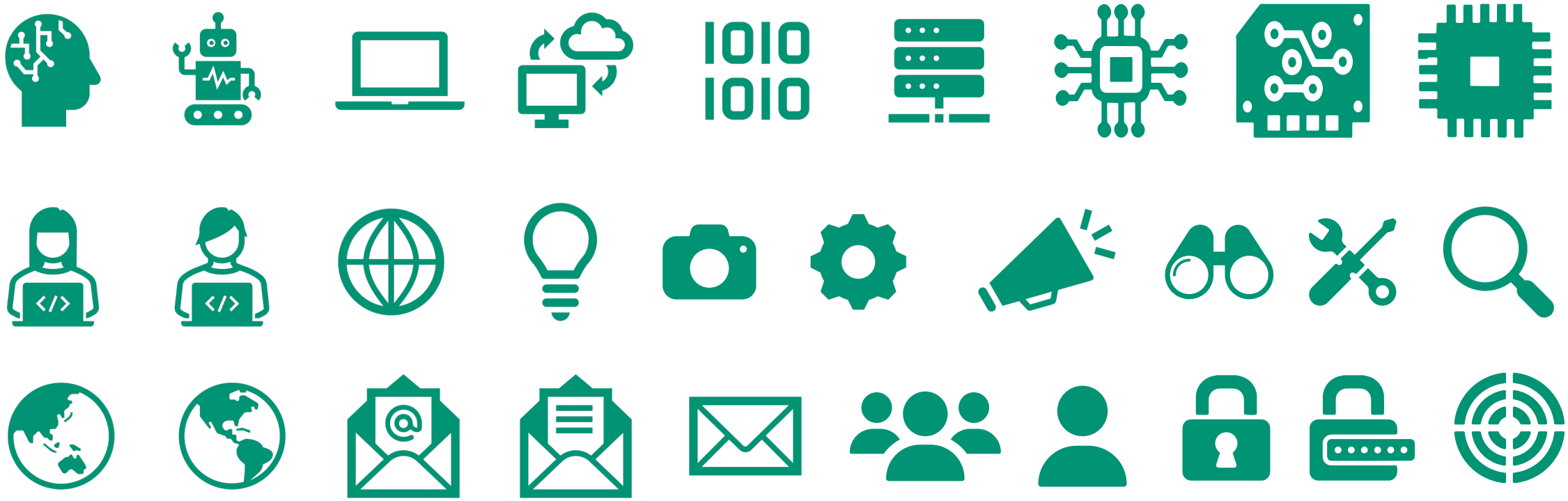
Data Mining



Deep Learning

Designer Note: Please use the corresponding icon for each course on the title page.

Assets 02 Icons



Assets 03 Media Icons

Start

Next

Submit

Previous