AI基础

Lecture 13: Deep Learning

Bin Yang
School of Data Science and Engineering
byang@dase.ecnu.edu.cn

[Some slides adapted from Philipp Koehn, JHU]

Lecture 12 ILOs

- Linear regression and classification
    - Two different ways to optimize the parameters
    - Classification, logistic regression
- Regularization
    - L1, L2
    - Pruning and early stopping in decision trees
- Learning decision trees
    - How to select the best attribute to split, information gain
- Ensemble learning
    - Bagging, random forest, stacking, boosting
        - Where does the diversity come from?
        - Whether the base models can be executed in parallel.

	Bagging	Random Forest	Stacking	Boosting
How to introduce diversity	Different subsets of the training data Base models are of the same class.	Different trees. Base models are in the same category, but with different structures.	Different base models.	Sequential, different learning targets (wrongly classified examples, or the residual of the previous predictions)
Parallelism	Easy. Independent base models.	Easy. Independent base models.	Have multiple stacks. At the same stack, yes. But stack by stack is sequential.	Sequential. Next base model is based on the previous base models.

Lecture 13 ILOs

- Feedforward neural networks
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Outline

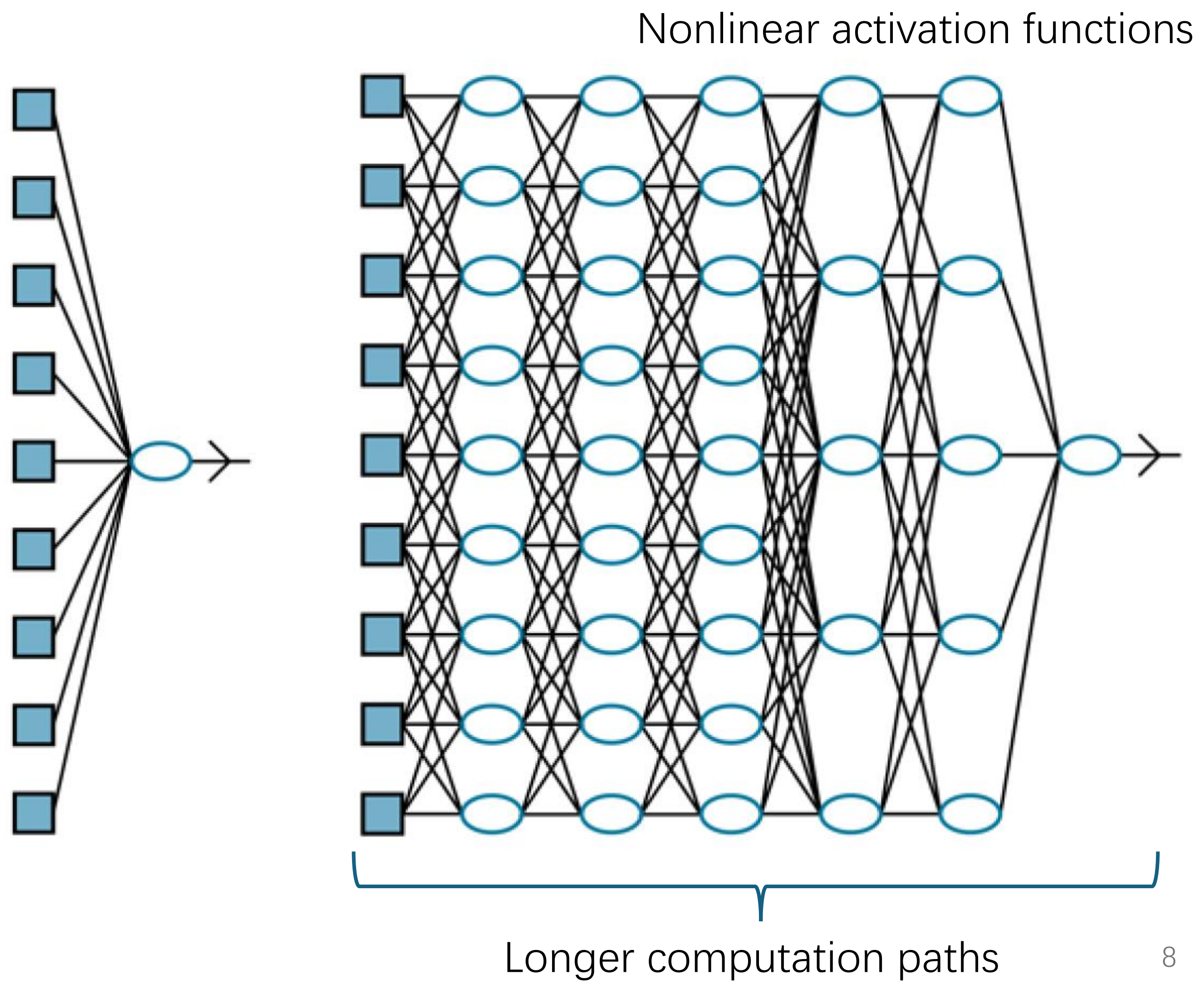
- Deep learning in a nutshell
- Simple Feedforward Network
- Computation graphs for deep learning
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Deep learning

- Deep learning is a broad family of techniques for machine learning in which hypotheses take the form of **complex algebraic circuits** with **tunable connection strengths**.
- The word “deep” refers to the fact that the circuits are typically organized into **many layers**, which means that computation paths from inputs to outputs have many steps.
- Deep learning is currently the most widely used approach for applications such as visual object recognition, machine translation, speech recognition, speech synthesis, and image synthesis; it also plays a significant role in reinforcement learning applications.
- Deep learning has its origins in early work that tried to model **networks of neurons in the brain** (McCulloch and Pitts, 1943) with computational circuits. For this reason, the networks trained by deep learning methods are often called **neural networks**, even though the resemblance to real neural cells and structures is superficial.

Shallow models vs. deep models

- Linear and logistic regression
  - Can handle a large number of input variables
  - The computation path from each input to the output is very short: multiplication by a single weight, then adding into the aggregate output.
- The different input variables contribute independently to the output, without interacting with each other.
  - This significantly limits the expressive power of such models.
  - They can represent only linear functions and boundaries in the input space, whereas most real-world concepts are far more complex.



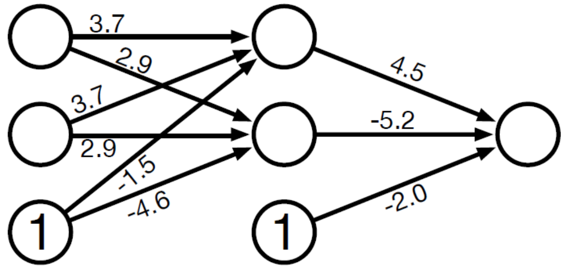
Outline

- Deep learning in a nutshell
- Simple Feedforward Network
    - Network as complex functions
    - Gradients and learning, backpropagation
- Computation graphs for deep learning
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Simple Feedforward Network

- A feedforward network
  - Connections only in one direction—that is, it forms a **directed acyclic graph (DAG)** with designated input and output nodes.
  - Each node computes a function of its inputs and passes the result to its successors in the network.
  - Information flows through the network from the input nodes to the output nodes, and there are **no loops**.
- A recurrent network
  - Feeds its intermediate or final outputs back into its own inputs.
  - This means that the signal values within the network form a dynamical system that has internal state or memory.

Networks as complex functions



bias units (no inputs, always value 1)

- Each node within a network is called a **unit**.
- A unit calculates the **weighted sum** of the inputs from predecessor nodes and then applies a **nonlinear function** to produce its output.
- Let a_j denote the output of unit j
- Let $w_{i,j}$ be the weight attached to the link from unit i to unit j
- Let g_i be a nonlinear activation function

$$a_j = g_j\left(\sum_i w_{i,j} a_i\right) \equiv g_j(in_j),$$

$$a_j = g_j(\mathbf{w}^\top \mathbf{x})$$

w: vector of weights leading into unit j

x: the vector of inputs to unit j

The universal approximation theorem

- The activation function is **nonlinear** is **important** because if it were not, any composition of units would still represent a linear function.
  - The nonlinearity is what allows sufficiently large networks of units to represent arbitrary functions.
- The universal approximation theorem states that a network with just two layers of computational units, the first nonlinear and the second linear, can approximate any continuous function to an arbitrary degree of accuracy.
  - The proof works by showing that an exponentially large network can represent exponentially many “bumps” of different heights at different locations in the input space, thereby approximating the desired function.

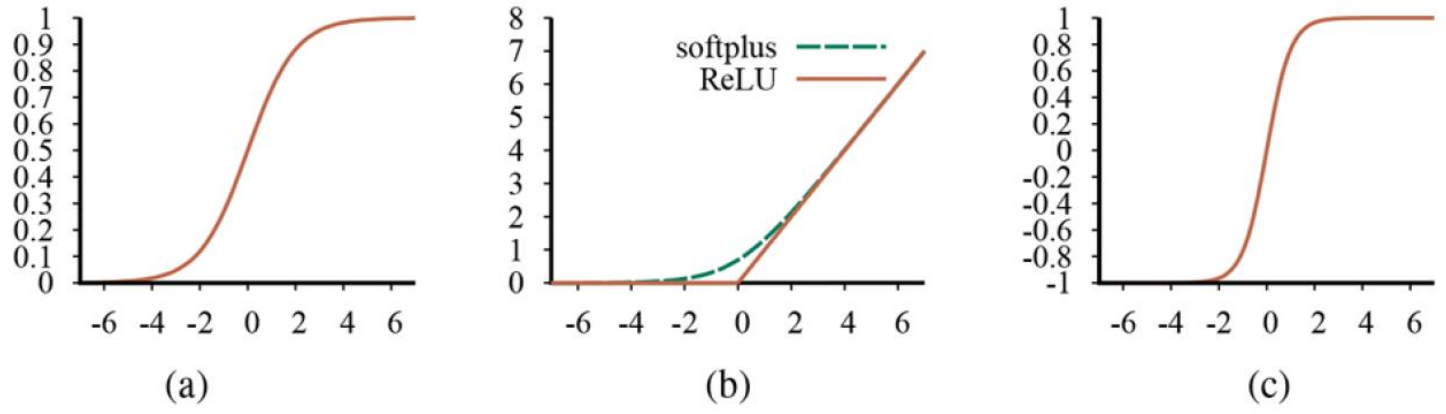
Nonlinear activation functions

- The logistic or sigmoid function
- The rectified linear unit (ReLU) function
- The softplus function, a smooth version of the ReLU function
- The tanh function

$\mathrm{ReLU}(x) = \max(0,x).$

$\mathrm{softplus}(x) = \log(1 + e^{x}).$

$\sigma(x) = 1/(1 + e^{-x}).$



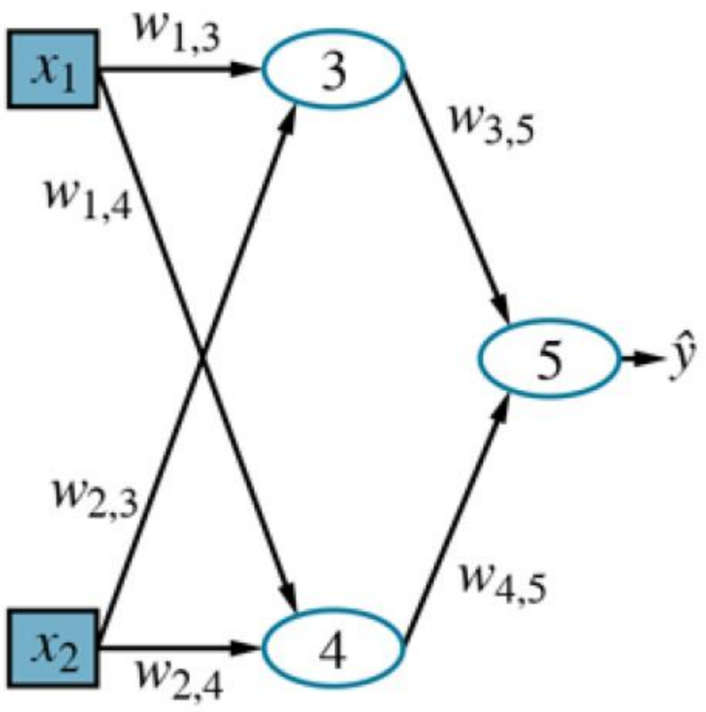
$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}.$

Tanh is a scaled and shifted version of the Sigmoid
$\tanh(x) = 2\sigma(2x) - 1$

Activation functions commonly used in deep learning systems: (a) the logistic or sigmoid function; (b) the ReLU function and the softplus function; (c) the tanh function.

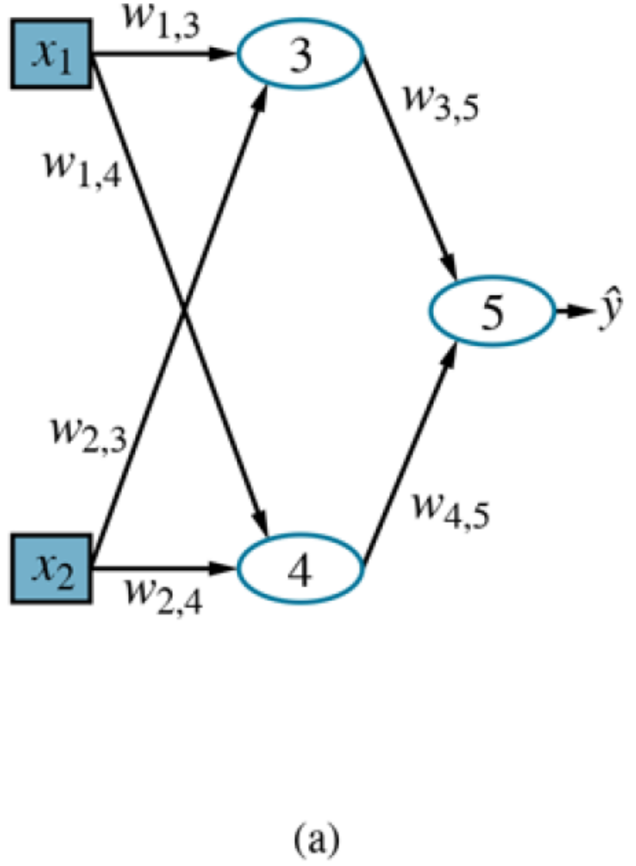
Example

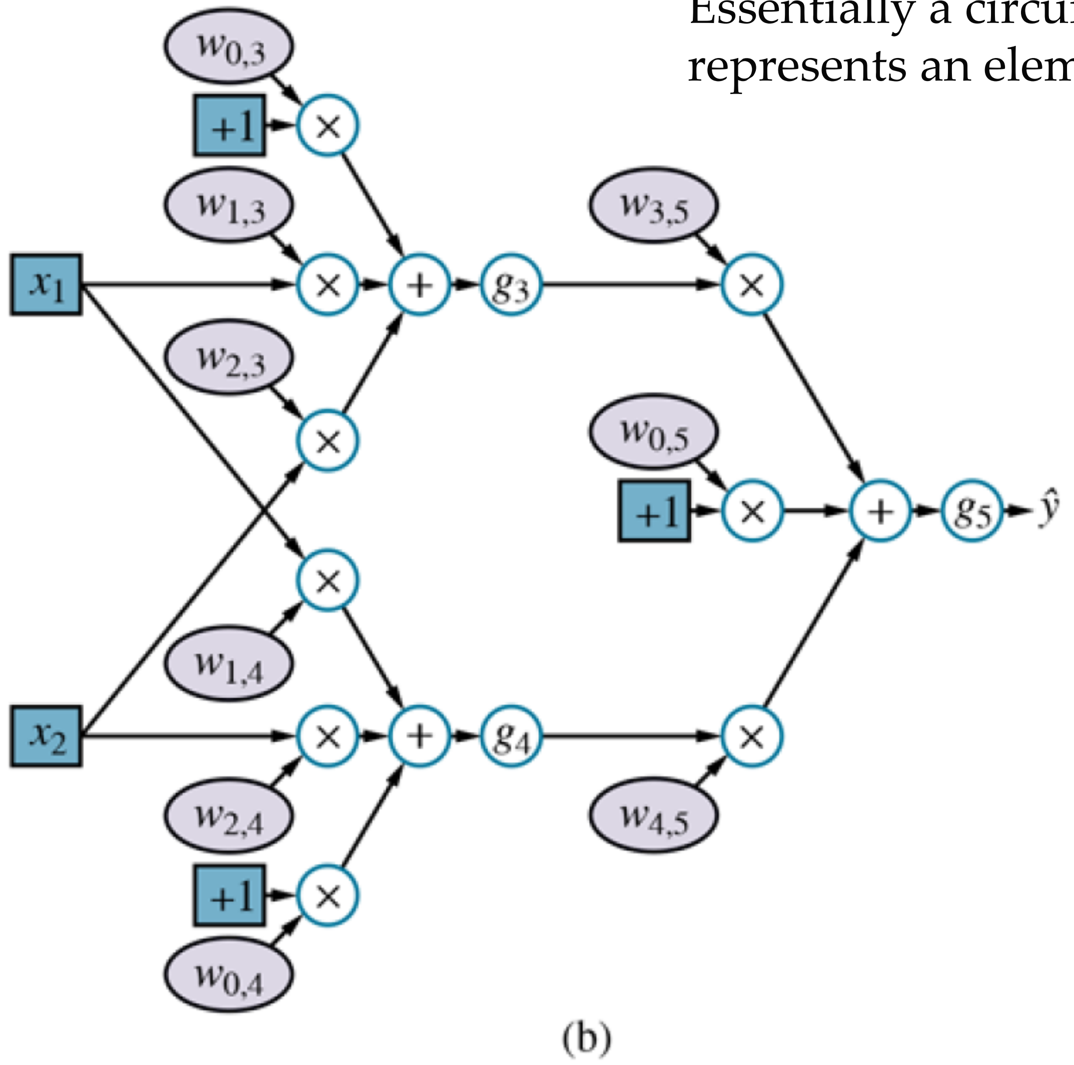
- Let a_j denote the output of unit j
- Let $w_{i,j}$ be the weight attached to the link from unit i to unit j
- Let g_i be a nonlinear activation function
- in_i denotes the input of unit i, weighted sum of inputs to unit i



$$\begin{aligned}\hat{y} &= g_5(in_5) = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= g_5(w_{0,5} + w_{3,5}g_3(in_3) + w_{4,5}g_4(in_4)) \\ &= g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\ &\qquad + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2)).\end{aligned}$$

Computation/dataflow graph





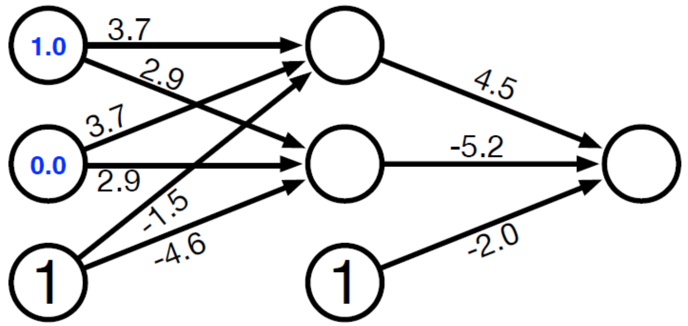
Essentially a circuit in which each node represents an elementary computation.

Inputs are blue.
Weights are in light mauve.

The weights can be adjusted to make the output $\hat{y}$ agree more closely with the true value y in the training data.

Each weight is like a volume control knob that determines how much the next node in the graph hears from that particular predecessor in the graph.

Example



- Try out two input values
- Hidden unit computation

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

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

- Output unit computation

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

Vector form

- **$\mathbf{W}^{(i)}$** denotes the weights in the i-th layer
- $g^{(i)}$ denotes the activation function in the i-th layer

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{g}^{(2)}(\mathbf{W}^{(2)}\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{x})).$$

```
$\mathbf{w} \leftarrow$ any point in the parameter space
while not converged do
    for each $w_i$ in $\mathbf{w}$ do
        $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$
```

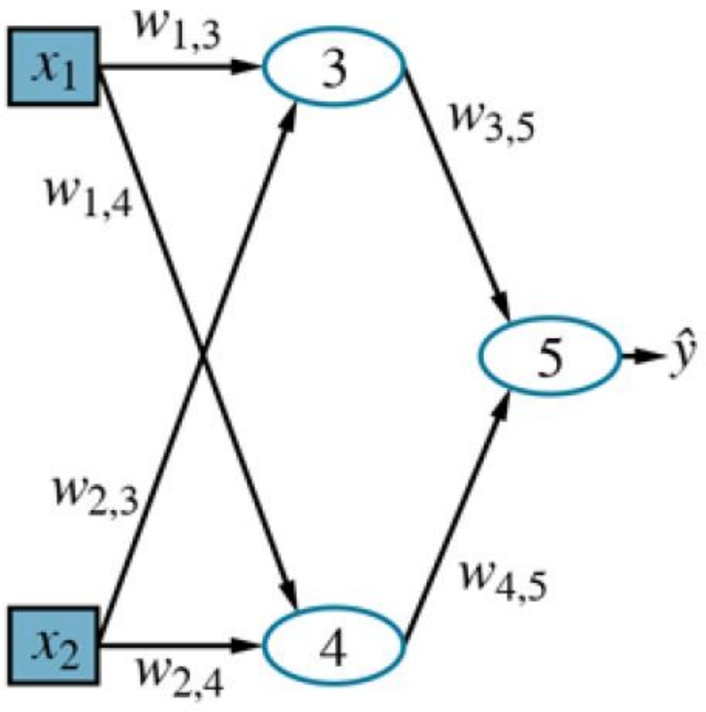
Gradients and learning

- Recall that we introduced an approach to supervised learning based on gradient descent: calculate the gradient of the loss function with respect to the weights, and adjust the weights along the gradient direction to reduce the loss.
- We can apply exactly the same approach to learning the weights in computation graphs.
- We will use the squared loss function

$$Loss(h_{\mathbf{w}}) = L_2(y, h_{\mathbf{w}}(\mathbf{x})) = \| y - h_{\mathbf{w}}(\mathbf{x}) \|^2 = (y - \hat{y})^2 .$$

$W_{3,5}$

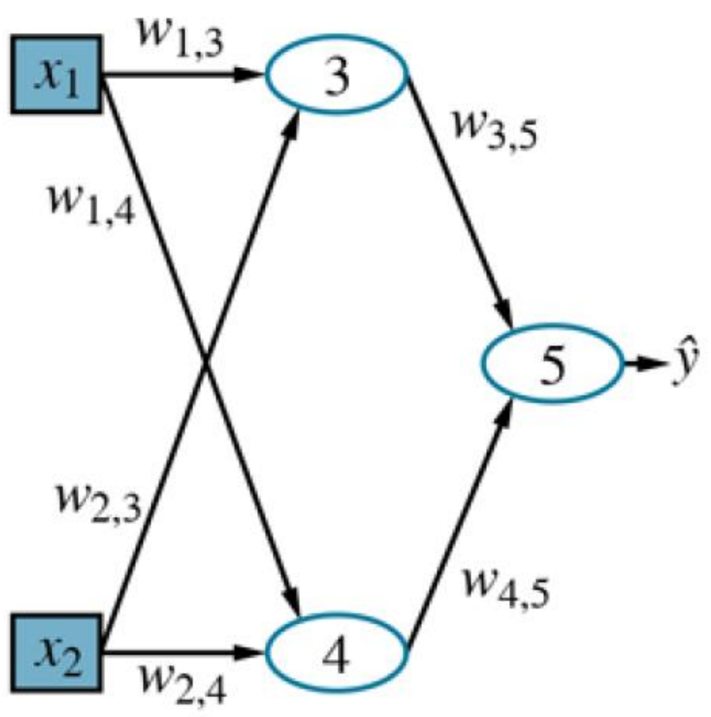
$$\begin{aligned}\hat{y} &= g_5(in_5) = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= g_5(w_{0,5} + w_{3,5}g_3(in_3) + w_{4,5}g_4(in_4)) \\ &= g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\ &\qquad + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2)).\end{aligned}$$



$$\begin{aligned}\frac{\partial}{\partial w_{3,5}} Loss(h_{\mathbf{w}}) &= \frac{\partial}{\partial w_{3,5}}(y-\hat{y})^2 = -2(y-\hat{y})\frac{\partial \hat{y}}{\partial w_{3,5}} \\ &= -2(y-\hat{y})\frac{\partial}{\partial w_{3,5}} g_5(in_5) = -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}} in_5 \\ &= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}}(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= -2(y-\hat{y})g_5'(in_5)a_3.\end{aligned}$$

$w_{1,3}$

$$
\begin{aligned}
\hat{y} &= g_5(in_5) = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\
&= g_5(w_{0,5} + w_{3,5}g_3(in_3) + w_{4,5}g_4(in_4)) \\
&= g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\
&\qquad + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2)).
\end{aligned}
$$



$$
\begin{aligned}
\frac{\partial}{\partial w_{1,3}} Loss(h_{\mathbf{w}}) &= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{1,3}}(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}\frac{\partial}{\partial w_{1,3}}a_3 \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}\frac{\partial}{\partial w_{1,3}}g_3(in_3) \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)\frac{\partial}{\partial w_{1,3}}in_3 \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)\frac{\partial}{\partial w_{1,3}}(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)x_1.
\end{aligned}
$$

```
$\mathbf{w} \leftarrow$ any point in the parameter space
while not converged do
    for each $w_i$ in $\mathbf{w}$ do
        $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$
```

$$\begin{aligned}
\frac{\partial}{\partial w_{3,5}} Loss(h_{\mathbf{w}}) &= \frac{\partial}{\partial w_{3,5}}(y-\hat{y})^2 = -2(y-\hat{y})\frac{\partial \hat{y}}{\partial w_{3,5}} \\
&= -2(y-\hat{y})\frac{\partial}{\partial w_{3,5}} g_5(in_5) = -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}} in_5 \\
&= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}}\left(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4\right) \\
&= -2(y-\hat{y})g_5'(in_5)a_3 .
\end{aligned}$$

- $\Delta_5 = 2(\hat{y} - y)g_5'(in_5)$ is perceived error at unit 5
- The gradient with respect to $w_{3,5}$ is $\triangle_5 a_3$.
- if $\triangle_5$ is positive, that means $\hat{y}$ is too big (recall that g’ is always nonnegative);
    - if a_3 is positive, then we should decrese $w_{3,5}$ to reduce the error.
    - if a_3 is negative, then we should increase $w_{3,5}$ to reduce the error.
- The magnitude of a_3 also matters: if a_3 is small for this training example, then $w_{3,5}$ didn’t play a major role in producing the error and doesn’t need to be changed much.
- **Back-propagation** for the way that the error at the output is passed back through the network.

Vanishing Gradient

- Another important characteristic of these gradient expressions is that they have as factors the local derivatives $g_j'(in_j)$.
- These derivatives are always nonnegative, but they can be
  - very close to zero (in the case of the sigmoid, softplus, and tanh functions) or
  - exactly zero (in the case of ReLUs),
  - if the inputs from the training example in question happen to put unit j in the flat operating region.
- If the derivative g_j' is small or zero, that means that changing the weights leading into unit j will have a negligible effect on its **output**.
- As a result, deep networks with many layers may suffer from a vanishing gradient—the error signals are extinguished altogether as they are propagated back through the network.

Outline

- Deep learning in a nutshell
- Simple Feedforward Network
- Computation graphs for deep learning
    - Input and output layers
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Computation graphs for deep learning

- The basic ideas of deep learning:
  - represent hypotheses as computation graphs with tunable weights
  - compute the gradient of the loss function with respect to those weights in order to fit the training data.
- Input layer: the training or test example is encoded as values of the input nodes.
- Output layer: the outputs are compared with the true values to derive a learning signal for tuning the weights.
- Hidden layers: in-between input and output layers.

Input encoding

- Boolean attributes
  - 0: false, 1: true
  - -1: false, 1: true
- Numeric attributes
  - Used as is
  - If the magnitudes for different examples vary enormously, the values can be mapped onto a log scale.
- Categorical attributes
  - One-hot encoding
  - An attribute with d possible values is represented by a vector of d bits.
  - For any given value, the corresponding input bit is set to 1 and all the others are set to 0.
  - Examples: French, Italian, Thai, or burger
  - <0001>: burger. <0100>: Italian

Output layers and loss functions

- Output encoding is similar as the input encoding.
- Loss functions
  - Squared-error, regression
  - Cross-entropy, classification
- Cross-entropy
  - Binary classification, logistics function on the output
  - Multiclass classification, softmax on the output

P being the true distribution over training examples,
Q being the predictive hypothesis

$$H(P,Q) = \mathbf{E}_{\mathbf{z}\sim P(\mathbf{z})}[\log Q(\mathbf{z})] = \int P(\mathbf{z}) \log Q(\mathbf{z}) d\mathbf{z}.$$

$$\text{softmax}(\mathbf{in})_k = \frac{e^{in_k}}{\sum_{k'=1}^{d} e^{in_{k'}}}.$$

$$\mathbf{in} = \langle 5,2,0,-2\rangle$$

$$\langle 0.946,0.047,0.006,0.001\rangle$$

Outline

- Deep learning in a nutshell
- Simple Feedforward Network
- Computation graphs for deep learning
- Convolutional neural networks
- Learning algorithms
- Generalization
- Recurrent neural networks

Convolutional networks

- How can we represent an image to a neural network?
- Problems of representing them as long vectors and use them in a fully connected network
- Huge weight size: n pixels and n units in the first hidden layer
  - Fully connected to the second layer
  - n^2 weights
  - Megapixel RGB image: $n=3*10^6$, $n^2 = 9*10^{12}$
- Insensitive to perturbation
  - We would get the same result whether we trained with unperturbed images or with images all of whose pixels had been randomly permuted.
- Each hidden unit receives input from only a small, local region of the image. This kills two birds with one stone.
  - It respects adjacency.
  - Cuts down the number of weights.

Convolutional neural networks (CNNs)

- A convolutional neural network (CNN) is one that contains spatially local connections, and has patterns of weights that are replicated across the units in each layer.
- A pattern of weights that is replicated across multiple local regions is called a **kernel** and the process of applying the kernel to the pixels of the image (or to spatially organized units in a subsequent layer) is called **convolution**.
- We expect image data to exhibit approximate **spatial invariance**, at least at small to moderate scales.
    - Anything that is detectable in one small, local region of the image—perhaps an eye or a blade of grass—would look the same if it appeared in another small, local region of the image.

Convolution

We write the convolution operation using the $*$ symbol, for example: $\mathbf{z} = \mathbf{x} * \mathbf{k}$. The operation is defined as follows:

(21.8)

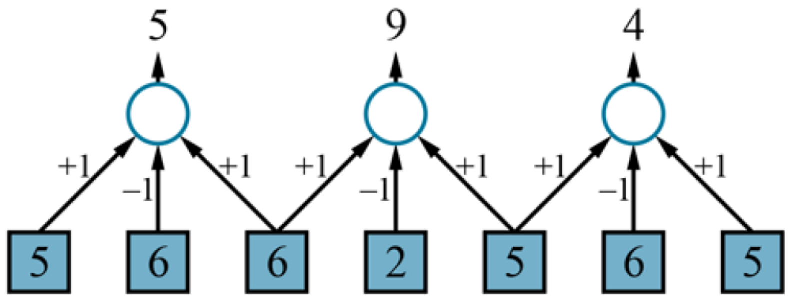
$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2}.$$

In other words, for each output position i, we take the dot product between the kernel $\mathbf{k}$ and a snippet of $\mathbf{x}$ centered on x_i with width l.

$$\begin{pmatrix} +1 & -1 & +1 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & -1 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & +1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 6 \\ 2 \\ 5 \\ 6 \\ 5 \end{pmatrix} = \begin{pmatrix} 5 \\ 9 \\ 4 \end{pmatrix}$$

A kernel vector [+1, -1, +1].
With a stride s=2

Figure 21.4

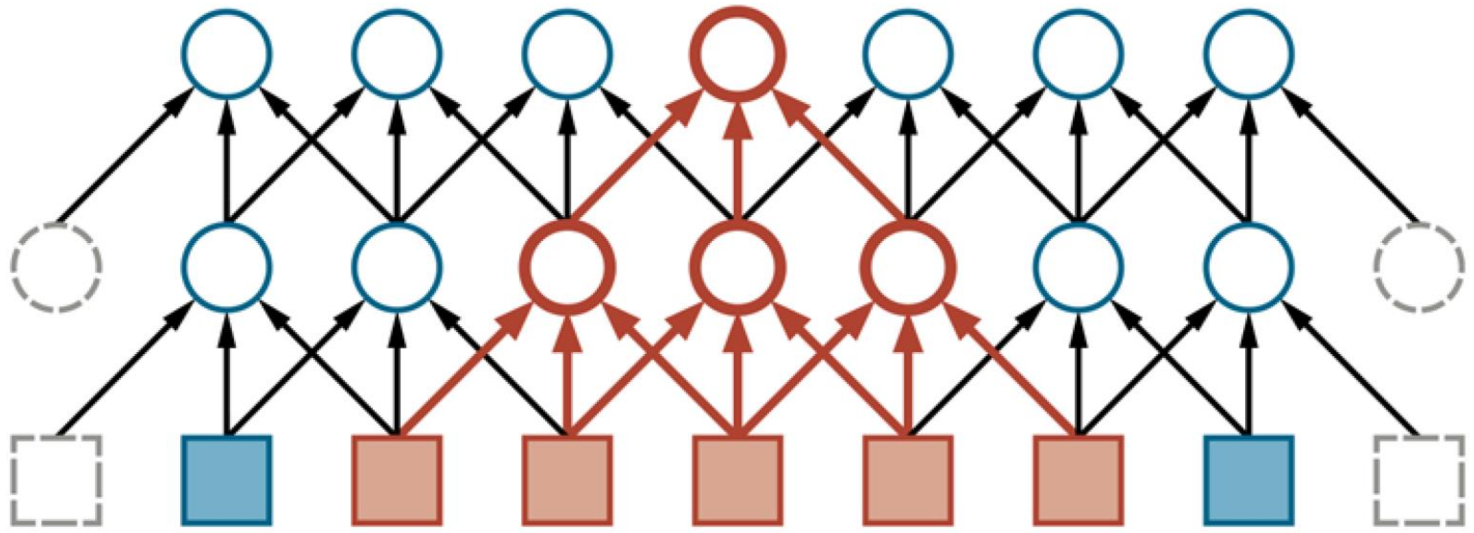


An example of a one-dimensional convolution operation with a kernel of size $l = 3$ and a stride $s = 2$. The peak response is centered on the darker (lower intensity) input pixel. The results would usually be fed through a nonlinear activation function (not shown) before going to the next hidden layer.

n/s

n

Receptive field



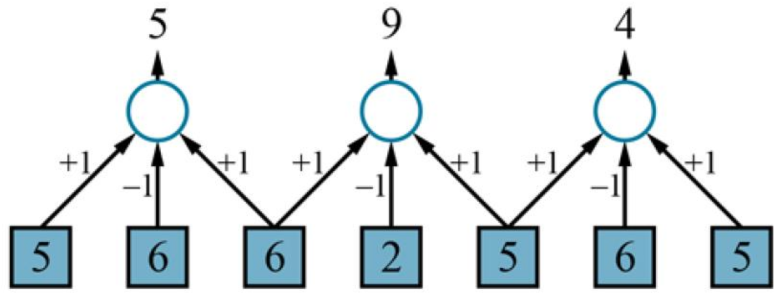
The first two layers of a CNN for a 1D image with a kernel size $l = 3$ and a stride $s = 1$. Padding is added at the left and right ends in order to keep the hidden layers the same size as the input. Shown in red is the receptive field of a unit in the second hidden layer. Generally speaking, the deeper the unit, the larger the receptive field.

CNNs were inspired originally by models of the visual cortex proposed in neuroscience. In those models, the receptive field of a neuron is the portion of the sensory input that can affect that neuron's activation.

In a CNN, the receptive field of a unit in the first hidden layer is small—just the size of the kernel, i.e., pixels. In the deeper layers of the network, it can be much larger.

Pooling and downsampling

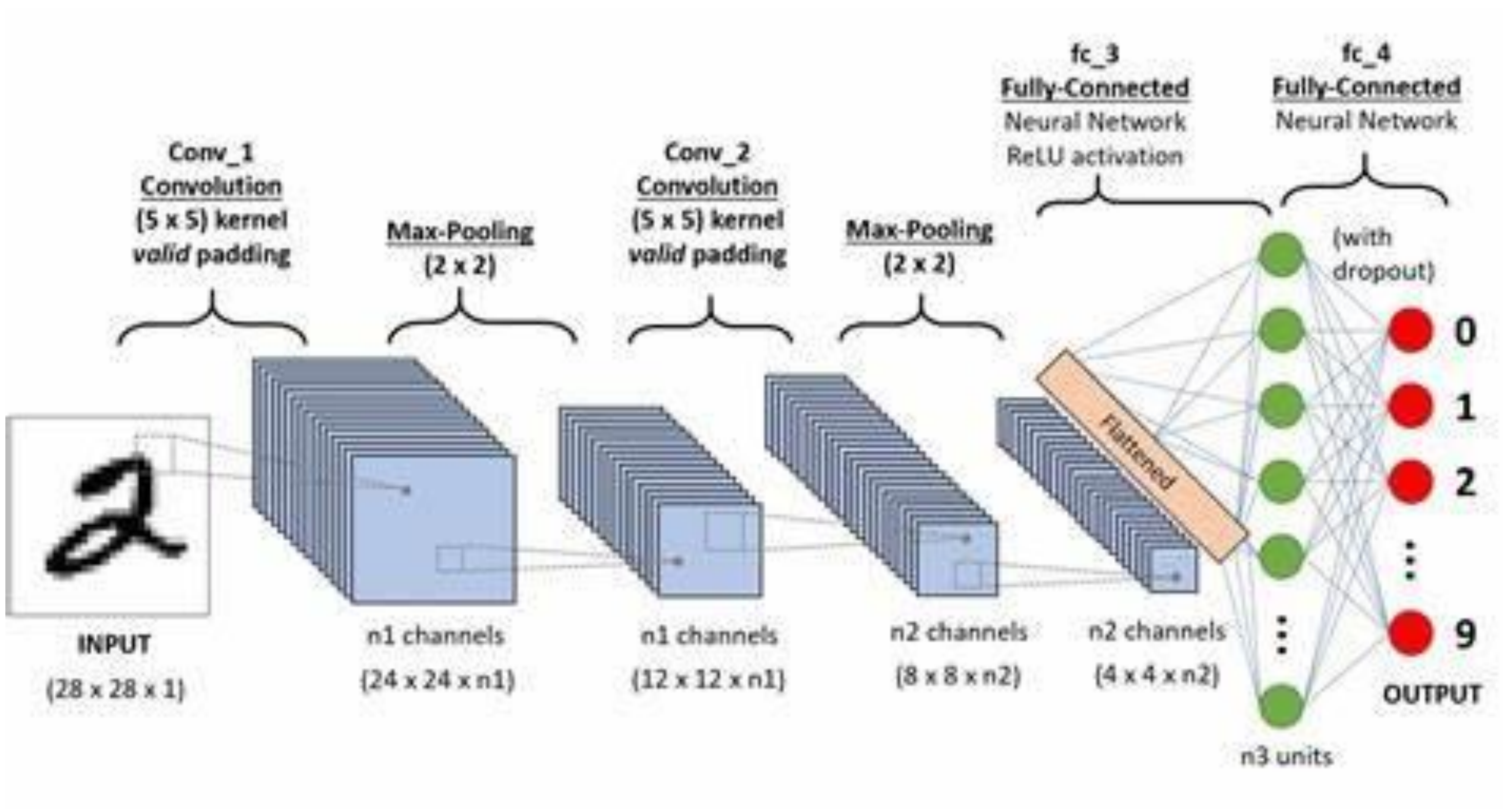
- A pooling layer in a neural network summarizes a set of adjacent units from the preceding layer with a single value.
- Average-pooling, max-pooling
  - To coarsen the resolution of the image—to downsample it.
  - Pooling facilitates multiscale recognition.
  - It also reduces the number of weights required in subsequent layers, leading to lower computational cost and possibly faster learning.



Average-pooling: avg(5, 9, 4)
max-pooling: max(5, 9, 4)

Size reduction

- If the goal is to classify the image into one of c categories, then the final layer of the network will be a softmax with c output units.
- The early layers of the CNN are image-sized, so somewhere in between there must be significant reductions in layer size.
  - Convolution layers and pooling layers with stride larger than 1 all serve to reduce the layer size.
  - A fully connected layer with fewer units than the preceding layer. CNNs often have one or two such layers preceding the final softmax layer.


Conv_1
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
Conv_2
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
fc_3
Fully-Connected
Neural Network
ReLU activation
fc_4
Fully-Connected
Neural Network
(with dropout)
Flattened
INPUT
(28 x 28 x 1)
n1 channels
(24 x 24 x n1)
n1 channels
(12 x 12 x n1)
n2 channels
(8 x 8 x n2)
n2 channels
(4 x 4 x n2)
n3 units
0
1
2
9
OUTPUT

Residual networks

- Residual networks are a popular and successful approach to building very deep networks that avoid the problem of **vanishing gradients**.
- Typical deep models use layers that learn a new representation at layer i by completely replacing the representation at layer i-1.

$$\mathbf{z}^{(i)} = f(\mathbf{z}^{(i-1)}) = \mathbf{g}^{(i)}(\mathbf{W}^{(i)}\mathbf{z}^{(i-1)}).$$

$$\mathbf{z}^{(i)} = \mathbf{g}_r^{(i)}(\mathbf{z}^{(i-1)} + f(\mathbf{z}^{(i-1)})),$$

Outline

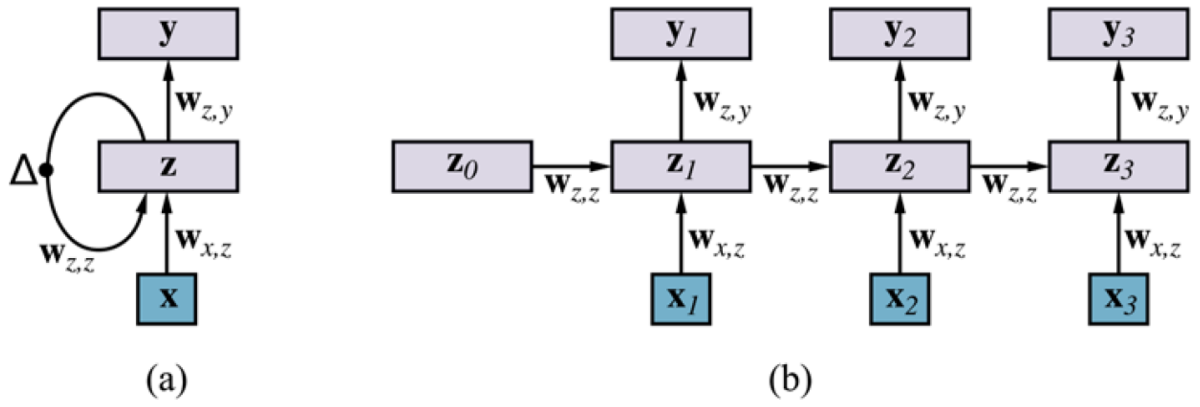
- Deep learning in a nutshell
- Simple Feedforward Network
- Computation graphs for deep learning
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Recurrent neural networks

- Recurrent neural networks (RNNs) are distinct from feedforward networks in that they allow cycles in the computation graph.
- In all the cases we will consider, each cycle has a delay, so that units may take as input a value computed from their own output at an earlier step in the computation.

$$\mathbf{z}_t = f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t) = \mathbf{g}_z(\mathbf{W}_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_t) \equiv \mathbf{g}_z(\mathbf{in}_{z,t})$$
$$\hat{\mathbf{y}}_t = \mathbf{g}_y(\mathbf{W}_{z,y}\mathbf{z}_t) \equiv \mathbf{g}_y(\mathbf{in}_{y,t}),$$

Figure 21.8



(a) Schematic diagram of a basic RNN where the hidden layer **z** has recurrent connections; the Δ symbol indicates a delay. (b) The same network unrolled over three time steps to create a feedforward network. Note that the weights are shared across all time steps.

We show the gradient calculation for an RNN with just one input unit, one hidden unit, and one output unit.

$$z_t = g_z(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z}) \text{ and } \hat{y}_t = g_y(w_{z,y}z_t + w_{0,y}).$$

$$\begin{aligned}
\frac{\partial L}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2 = \sum_{t=1}^{T} -2(y_t - \hat{y}_t) \frac{\partial \hat{y}_t}{\partial w_{z,z}} \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) \frac{\partial}{\partial w_{z,z}} g_y(in_{y,t}) = \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g_y'(in_{y,t}) \frac{\partial}{\partial w_{z,z}} in_{y,t} \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g_y'(in_{y,t}) \frac{\partial}{\partial w_{z,z}} (w_{z,y}z_t + w_{0,y}) \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g_y'(in_{y,t}) w_{z,y} \frac{\partial z_t}{\partial w_{z,z}}.
\end{aligned}$$

$$\begin{aligned}
\frac{\partial z_t}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}} g_z(in_{z,t}) = g_z'(in_{z,t}) \frac{\partial}{\partial w_{z,z}} in_{z,t} = g_z'(in_{z,t}) \frac{\partial}{\partial w_{z,z}} (w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z}) \\
&= g_z'(in_{z,t}) \left(z_{t-1} + w_{z,z} \frac{\partial z_{t-1}}{\partial w_{z,z}}\right),
\end{aligned}$$

Back-propagation through time:
the contribution to the gradient from time step t is calculated using the contribution from time step t-1. If we order the calculations in the right way, the total runtime for computing the gradient will be linear in the size of the network.

We see that gradients at will include terms proportional to $w_{z,z} \prod_{t=1}^{T} g_z'(in_{z,t})$

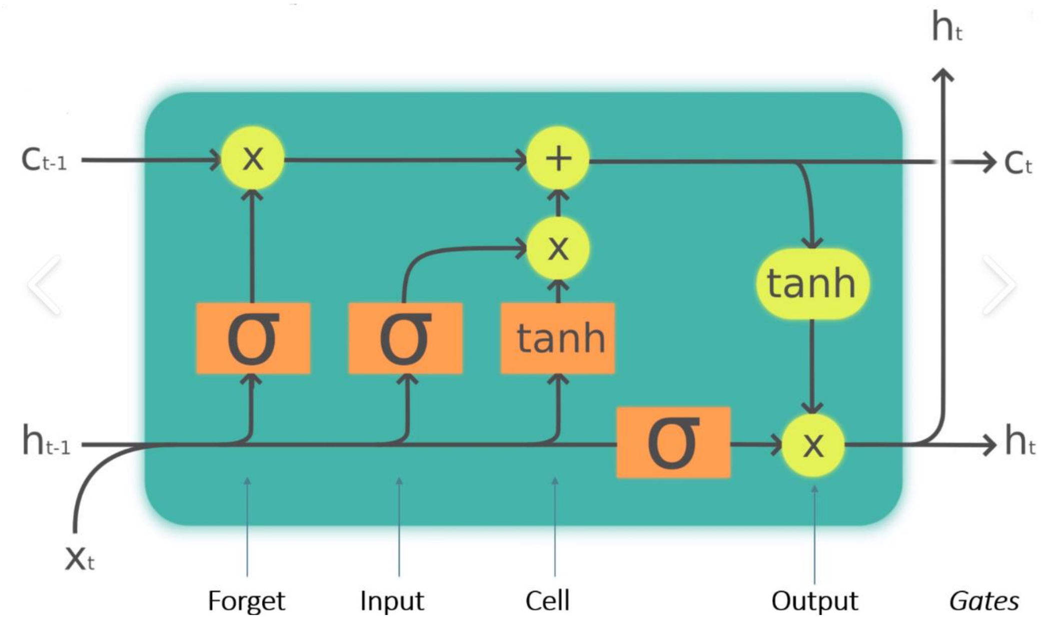
g'<=1, if w_{zz}<1, gradient vanishing.

Long-short term memory (LSTM)

- The long-term memory component of an LSTM, called the memory cell and denoted by c, is essentially copied from time step to time step.
  - The forget gate f determines if each element of the memory cell is remembered (copied to the next time step) or forgotten (reset to zero).
  - The input gate i determines if each element of the memory cell is updated additively by new information from the input vector at the current time step.
  - The output gate o determines if each element of the memory cell is transferred to the short-term memory , which plays a similar role to the hidden state in basic RNNs.

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_{x,f}\mathbf{x}_t + \mathbf{W}_{z,f}\mathbf{z}_{t-1}) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_{x,i}\mathbf{x}_t + \mathbf{W}_{z,i}\mathbf{z}_{t-1}) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{x,o}\mathbf{x}_t + \mathbf{W}_{z,o}\mathbf{z}_{t-1}) \\
\mathbf{c}_t &= \mathbf{c}_{t-1} \odot \mathbf{f}_t + \mathbf{i}_t \odot \tanh(\mathbf{W}_{x,c}\mathbf{x}_t + \mathbf{W}_{z,c}\mathbf{z}_{t-1}) \\
\mathbf{z}_t &= \tanh(\mathbf{c}_t) \odot \mathbf{o}_t,
\end{aligned}
$$

Sigmoid for gates [0, 1]



Outline

- Deep learning in a nutshell
- Simple Feedforward Network
- Computation graphs for deep learning
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Stochastic gradient descent (SGD)

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}),$$

where α is the learning rate. For standard gradient descent, the loss L is defined with respect to the entire training set. For SGD, it is defined with respect to a minibatch of m examples chosen randomly at each step.

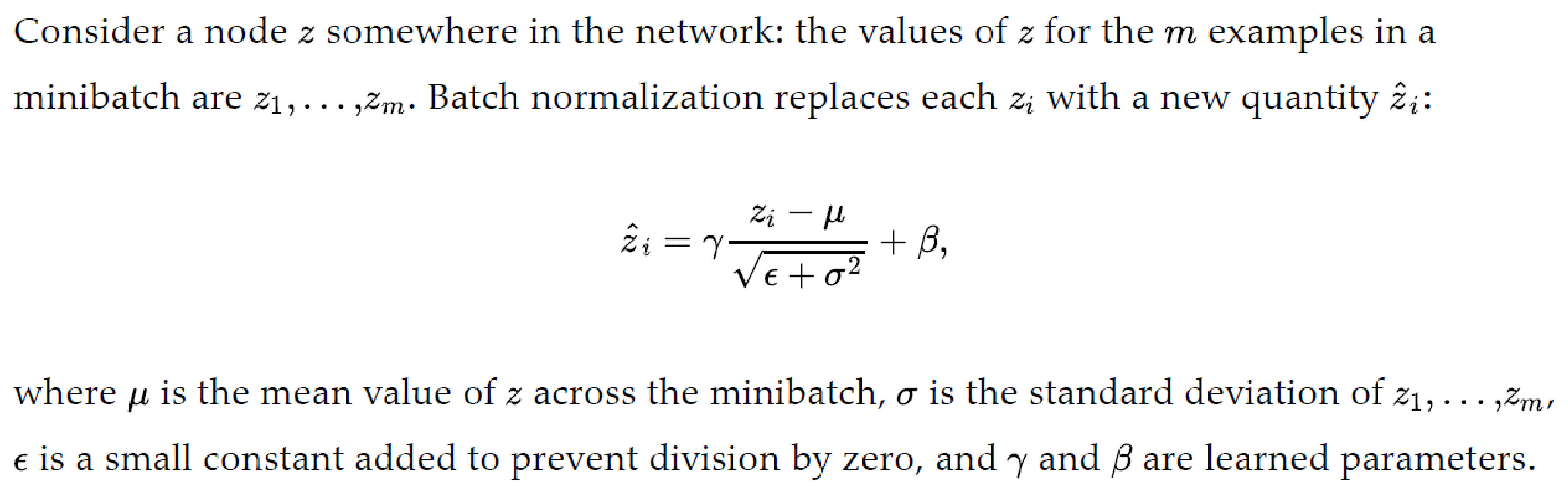
- For most networks that solve real-world problems, both the dimensionality of w and the size of the training set are very large. These considerations militate strongly in favor of using SGD with a relatively small minibatch size: **stochasticity** helps the **algorithm escape small local minima** in the high-dimensional weight space (as in simulated annealing); and **the small minibatch size** ensures that **the computational cost of each weight update step is a small constant**, independent of the training set size.
- Because the gradient contribution of each training example in the SGD minibatch can be computed independently, the minibatch size is often chosen so as to take maximum advantage of hardware **parallelism** in GPUs or TPUs.

SGD

- To improve convergence, it is usually a good idea to use a learning rate that decreases over time. Choosing the right schedule is usually a matter of trial and error.
- Near a local or global minimum of the loss function with respect to the entire training set, the gradients estimated from small minibatches will often have high variance and may point in entirely the wrong direction, making convergence difficult. One solution is to increase the minibatch size as training proceeds; another is to incorporate the idea of momentum, which keeps a running average of the gradients of past minibatches in order to compensate for small minibatch sizes.
- Care must be taken to mitigate numerical instabilities that may arise due to overflow, underflow, and rounding error. These are particularly problematic with the use of exponentials in softmax, sigmoid, and tanh activation functions, and with the iterated computations in very deep networks and recurrent networks that lead to vanishing and exploding activations and gradients.

Batch normalization

- Batch normalization is a commonly used technique that improves the rate of convergence of SGD by rescaling the values generated at the internal layers of the network from the examples within each minibatch.



Consider a node z somewhere in the network: the values of z for the m examples in a minibatch are $z_1, \ldots, z_m$. Batch normalization replaces each z_i with a new quantity $\hat{z}_i$:

$$\hat{z}_i = \gamma \frac{z_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta,$$

where μ is the mean value of z across the minibatch, σ is the standard deviation of $z_1, \ldots, z_m$, ϵ is a small constant added to prevent division by zero, and γ and β are learned parameters.

Outline

- Deep learning in a nutshell
- Simple Feedforward Network
- Computation graphs for deep learning
- Convolutional neural networks
- Recurrent neural networks
- Learning algorithms
- Generalization

Generalization

- So far, we have described how to fit a neural network to its training set, but in machine learning the goal is to generalize to new data that has not been seen previously, as measured by performance on a test set.
  - Avoid over-fitting
- Three approaches to improving generalization performance:
  - Choosing the right network architecture
  - Penalizing large weights
  - Randomly perturbing the values passing through the network during training

Choosing a network architecture

- A great deal of effort in deep learning research has gone into finding network architectures that generalize well.
- Convolutional architectures are expected to generalize well on images and recurrent networks to generalize well on text and audio signals.
- Adversarial examples
  - Deep learning models tend to produce input–output mappings that are discontinuous, so that a small change to an input can cause a large change in the output.
  - For example, it may be possible to alter just a few pixels in an image of a dog and cause the network to classify the dog as an ostrich or a school bus—even though the altered image still looks exactly like a dog by a human.
  - One to find learning algorithms and network architectures that would not be susceptible to adversarial attack,
  - Another to create ever-more-effective adversarial attacks against all kinds of learning systems.

Neural architecture search

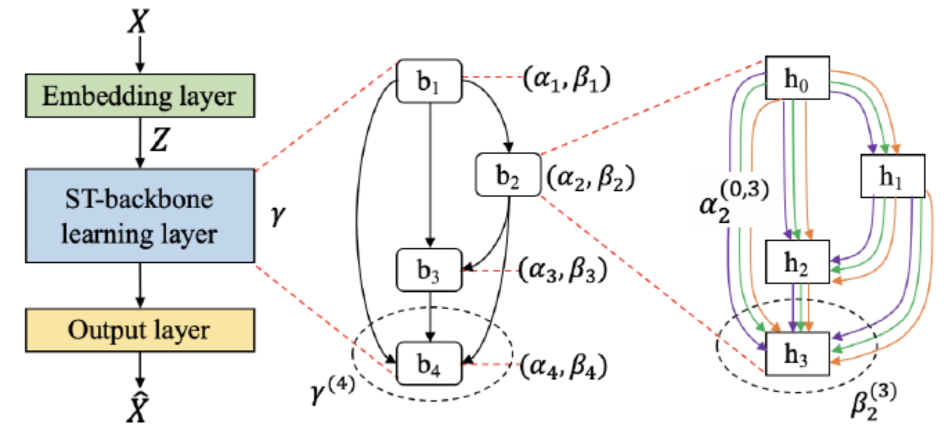
- We do not yet have a clear set of guidelines to help you choose the best network architecture for a particular problem. Success in deploying a deep learning solution requires experience and good judgment.
- Automate the process of architecture selection.
  - A case of hyperparameter tuning, where the hyperparameters determine the depth, width, connectivity, and other attributes of the network.
  - Evolutionary algorithm, continuous differentiable space
- Major challenge: estimating the value of a candidate architecture
  - Takes a long time
  - Start by choosing a few hundred network architectures and train and evaluate them. That gives us a data set of (network, score) pairs. Then learn a mapping from the features of a network to a predicted score.

Our research on NAS

AutoCTS: Automated Correlated Time Series Forecasting

Xinle Wu[1], Dalin Zhang[1], Chenjuan Guo[1], Chaoyang He[2], Bin Yang[1]*, Christian S. Jensen[1]
[1]Aalborg University, Denmark [2]University of Southern California, USA
[1]{xinlewu, dalinz, cguo, byang, csj}@cs.aau.dk [2]chaoyang.he@usc.edu

PVLDB 2022



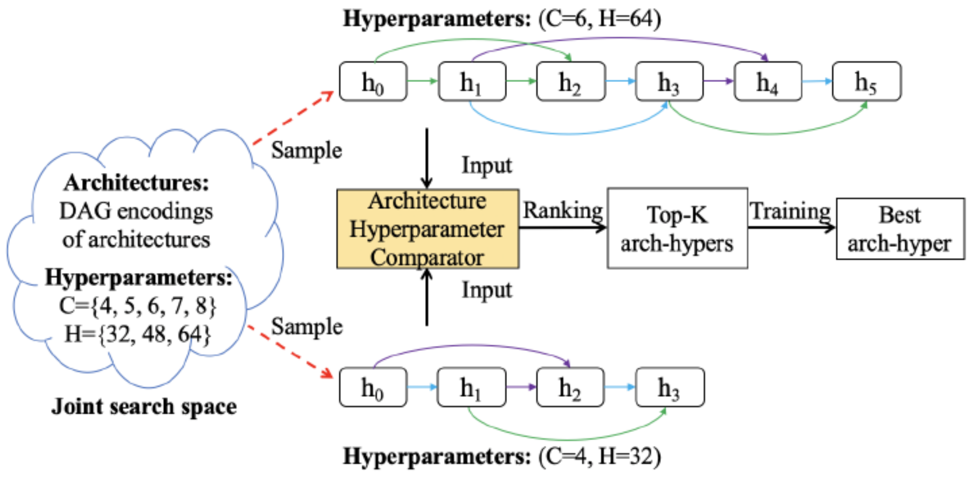
(a) Overall framework (b) ST-backbone (c) ST-block

continuous differentiable space

AutoCTS+: Joint Neural Architecture and Hyperparameter Search for Correlated Time Series Forecasting

XINLE WU, Aalborg University, Denmark
DALIN ZHANG*, Aalborg University, Denmark
MIAO ZHANG, Harbin Institute of Technology, China and Aalborg University, Denmark
CHENJUAN GUO, East China Normal University, China and Aalborg University, Denmark
BIN YANG*, East China Normal University, China and Aalborg University, Denmark
CHRISTIAN S. JENSEN, Aalborg University, Denmark

SIGMOD 2023


Hyperparameters: (C=6, H=64)
h_0 h_1 h_2 h_3 h_4 h_5
Sample
Input
Architectures: DAG encodings of architectures
Hyperparameters: C={4, 5, 6, 7, 8} H={32, 48, 64}
Joint search space
Architecture Hyperparameter Comparator
Ranking
Top-K arch-hypers
Training
Best arch-hyper
Input
Sample
h_0 h_1 h_2 h_3
Hyperparameters: (C=4, H=32)

Start by choosing a few hundred network architectures and train and evaluate them. That gives us a data set of (network, score) pairs. Then learn a mapping from the features of a network to a predicted score.

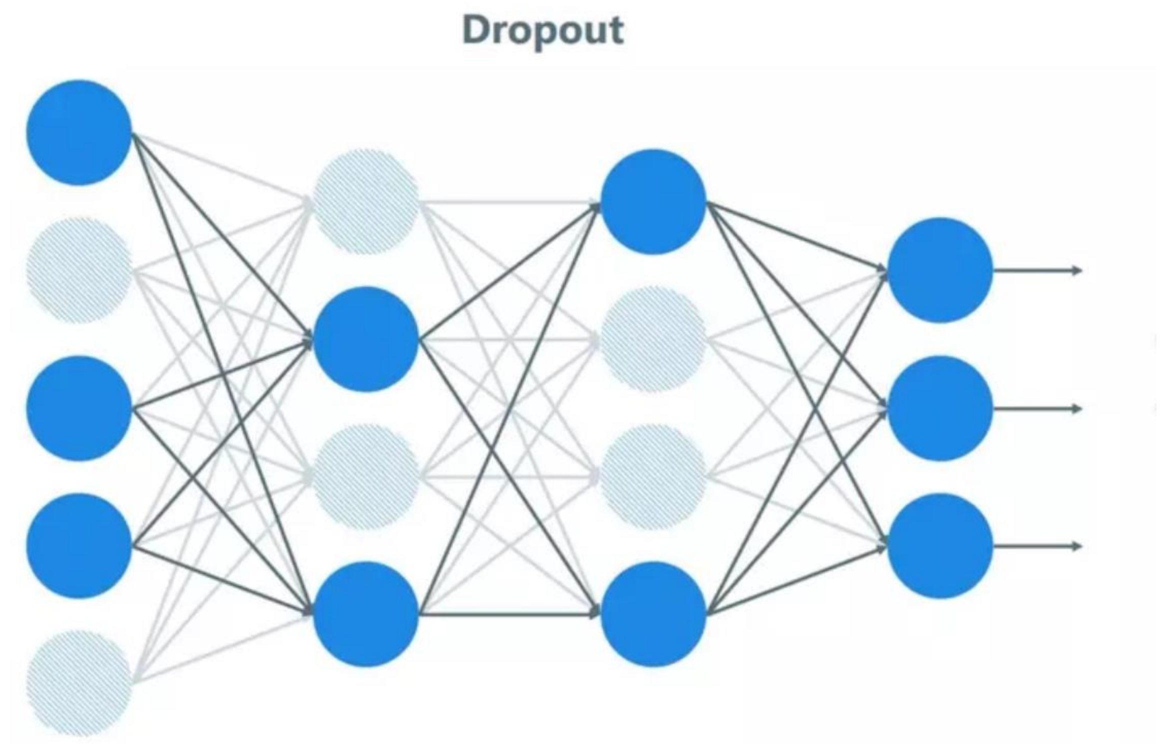
Weight Decay

- Regularization for deep learning

Weight decay consists of adding a penalty $\lambda \sum_{i,j} W_{i,j}^2$ to the loss function used to train the neural network, where λ is a hyperparameter controlling the strength of the penalty and the sum is usually taken over all of the weights in the network. Using $\lambda = 0$ is equivalent to not using weight decay, while using larger values of λ encourages the weights to become small. It is common to use weight decay with λ near 10^{-4}.

Drop out

- At each step of training, dropout applies one step of back-propagation learning to a new version of the network that is created by deactivating a randomly chosen subset of the units.



Beyond supervised learning

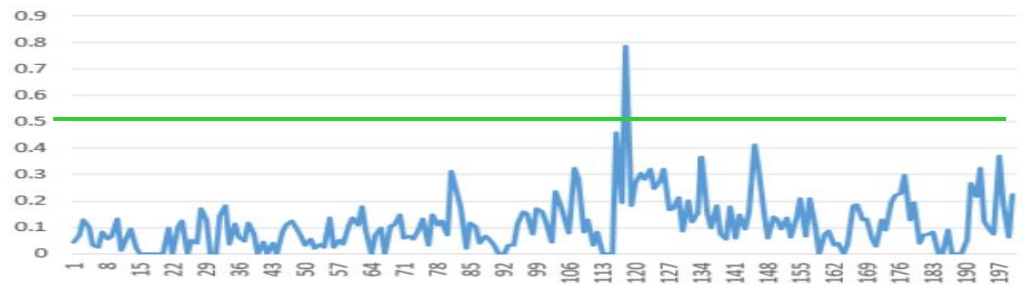
- The deep learning systems we have discussed so far are based on supervised learning, which requires each training example to be labeled with a value for the target function.
- Although such systems can reach a high level of test-set accuracy—as shown by the ImageNet competition results, for example—they often require far more labeled data than a human would for the same task.
    - For example, a child needs to see only one picture of a giraffe, rather than thousands, in order to be able to recognize giraffes reliably in a wide range of settings and views.
- Clearly, something is missing in our deep learning story; indeed, it may be the case that our current approach to supervised deep learning renders some tasks completely unattainable because the requirements for labeled data would exceed what the human race (or the universe) can supply.
- Moreover, even in cases where the task is feasible, labeling large data sets usually requires scarce and expensive human labor.
- Unsupervised learning, transfer learning, and semi-supervised learning.

Beyond supervised learning

- Unsupervised learning algorithms learn solely from unlabeled data, which are often more abundantly available than labeled examples. Unsupervised learning algorithms typically produce generative models, which can produce realistic text, images, audio, and video, rather than simply predicting labels for such data.
  - See autoencoder example later.
- Transfer learning algorithms require some labeled examples but are able to improve their performance further by studying labeled examples for different tasks, thus making it possible to draw on more existing sources of data.
- Semi-supervised learning algorithms require some labeled examples but are able to improve their performance further by also studying unlabeled examples.
  - Some research on graph semi-supervised learning, crystal property estimation

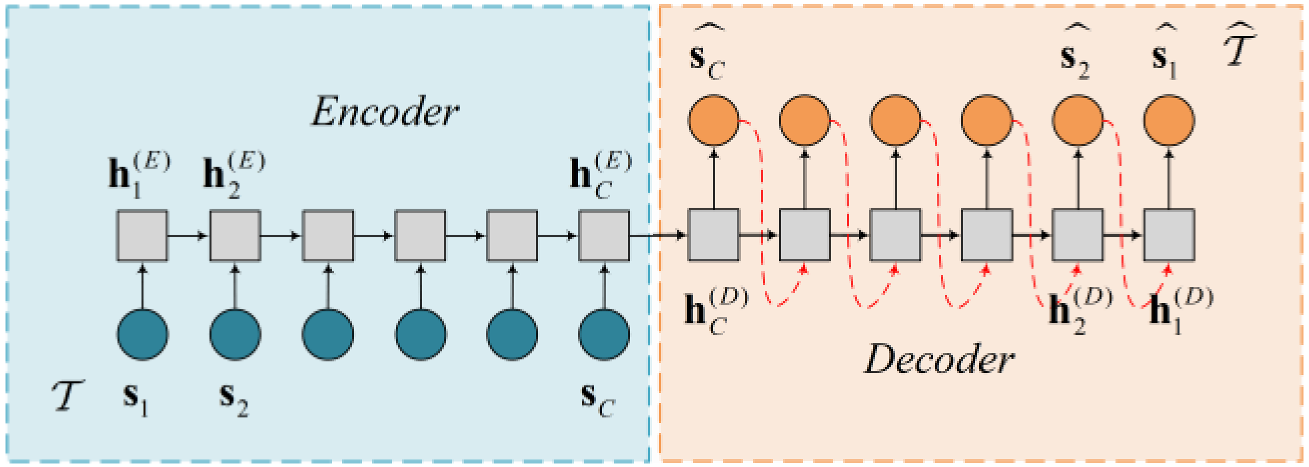
Our research: Recurrent autoencoder ensembles

- Recurrent neural networks + ensembles + autoencoders (unsupervised learning)
- Unsupervised outlier detection
  - For time series
- A data point is an outlier if it differs significantly from other data points.



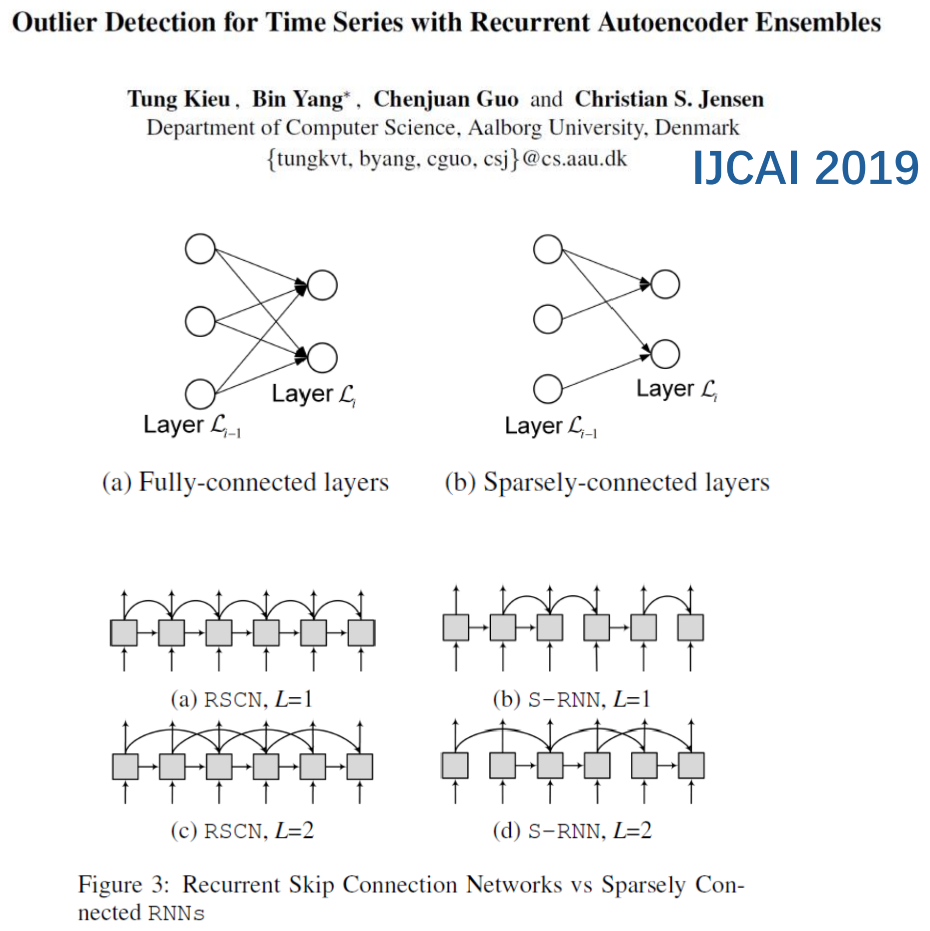
Autoencoder

- Encoder: Compress original TS into a compact representation.
- Decoder: Reconstruct the TS from the compact representation.
- Use the difference between the reconstructed TS vs. original TS to identify outliers.



Recurrent Autoencoder Ensembles

- Multiple autoencoders with different structures are combined together (IJCAI 2019)



Outlier Detection for Time Series with Recurrent Autoencoder Ensembles

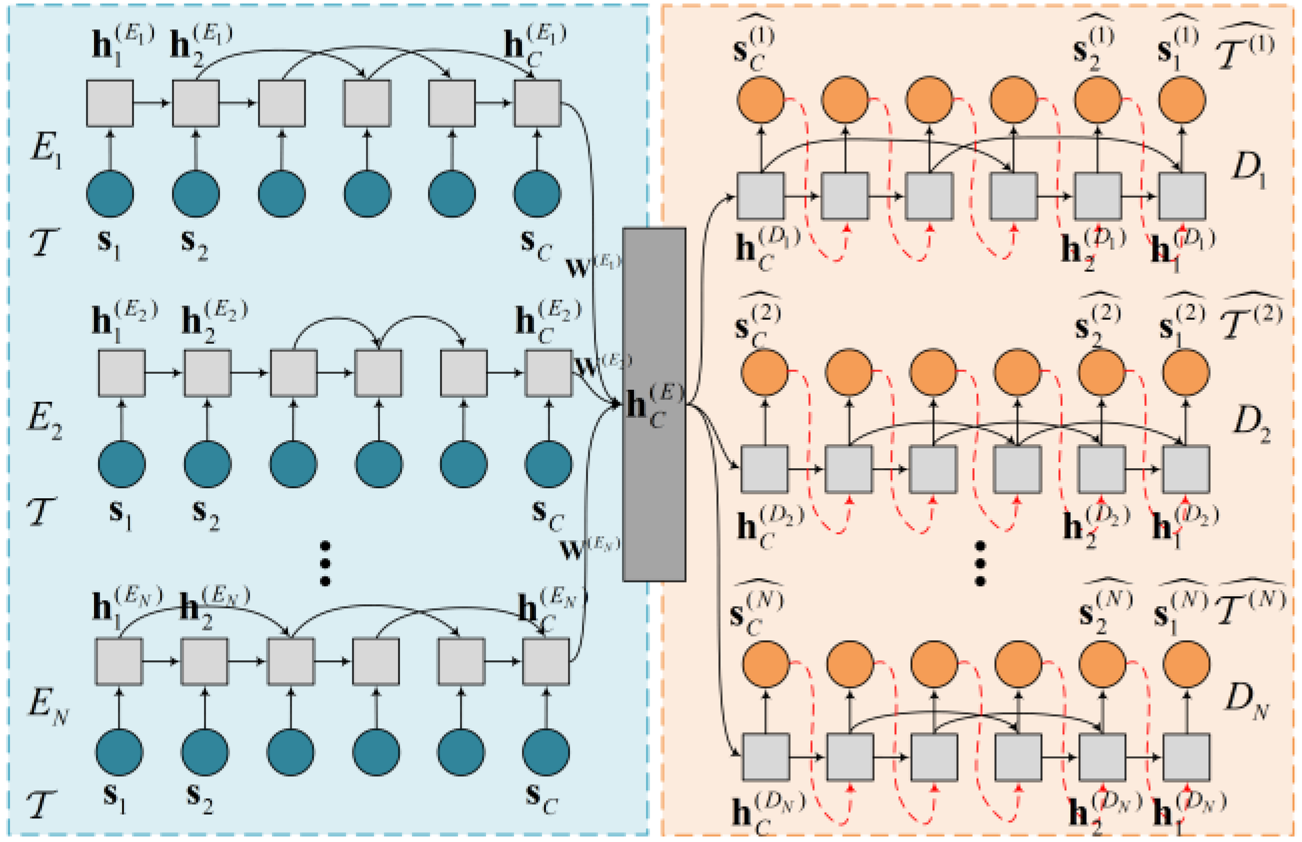
Tung Kieu, **Bin Yang***, **Chenjuan Guo** and **Christian S. Jensen**
Department of Computer Science, Aalborg University, Denmark
{tungkvt, byang, cguo, csj}@cs.aau.dk

IJCAI 2019

(a) Fully-connected layers (b) Sparsely-connected layers

(a) RSCN, L=1 (b) S−RNN, L=1
(c) RSCN, L=2 (d) S−RNN, L=2

Figure 3: Recurrent Skip Connection Networks vs Sparsely Connected RNNs



Unsupervised Time Series Outlier Detection with Diversity-Driven Convolutional Ensembles

David Campos[1*], Tung Kieu[1*], Chenjuan Guo[1+], Feiteng Huang[2], Kai Zheng[3], Bin Yang[1], and Christian S. Jensen[1]
[1]Aalborg University, Denmark [2]Huawei Cloud Database Innovation Lab, China
[3]University of Electronic Science and Technology of China, China
[1]{dgcc, tungkvt, cguo, byang, csj}@cs.aau.dk, [2]huangfeiteng@huawei.com, [3]zhengkai@uestc.edu.cn

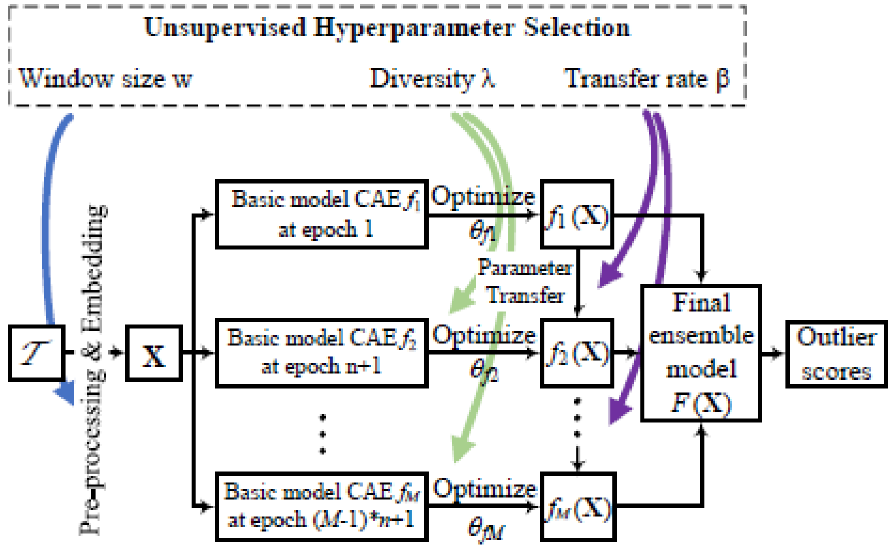

Unsupervised Hyperparameter Selection
Window size w
Diversity λ
Transfer rate β
Pre-processing & Embedding
T
X
Basic model CAE f1 at epoch 1
Basic model CAE f2 at epoch n+1
Basic model CAE fM at epoch (M-1)*n+1
Optimize θf1
Optimize θf2
Optimize θfM
Parameter Transfer
f1 (X)
f2 (X)
fM (X)
Final ensemble model F (X)
Outlier scores

Figure 8: CAE-Ensemble overview.

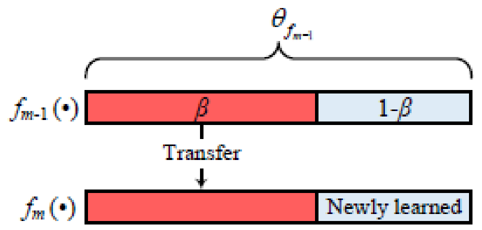
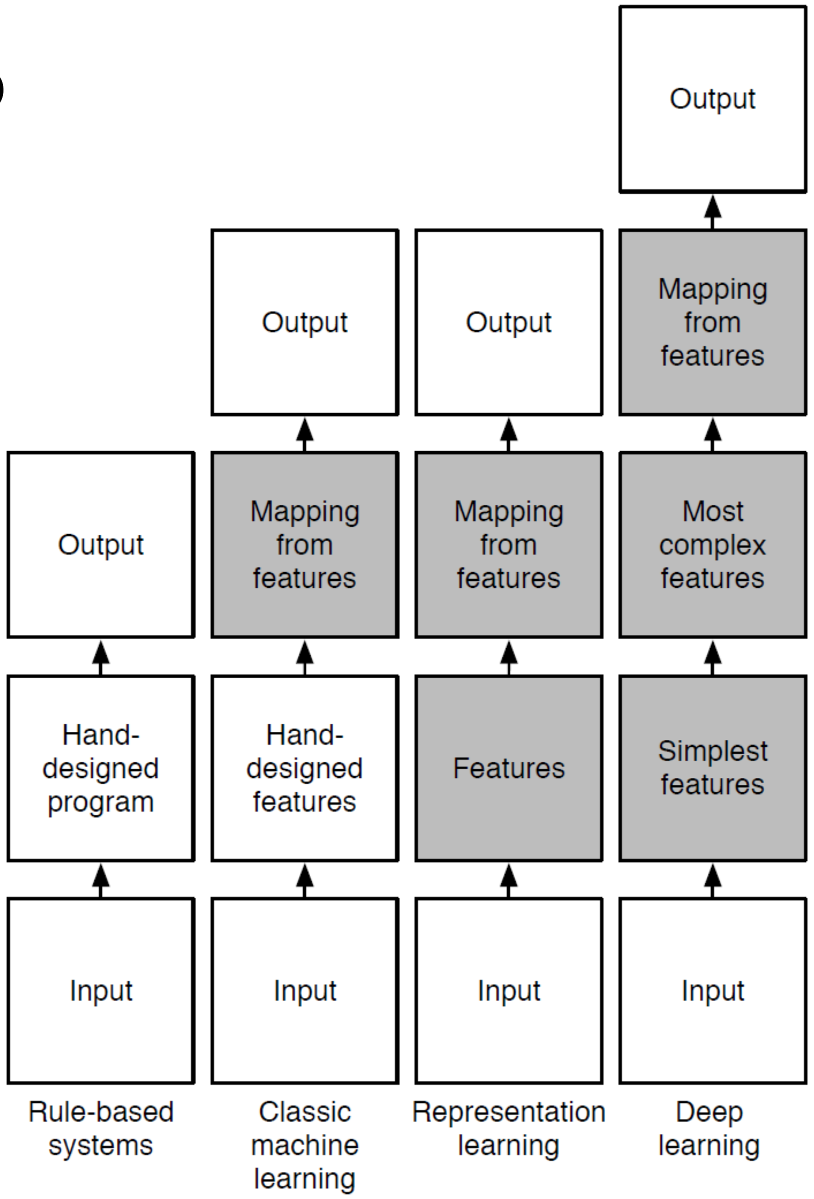


Figure 9: Basic model generation with parameter transfer.

$$DIV_F(\mathbf{X}) = \frac{2}{M(M-1)} \sum_{m=1}^{M} \sum_{n=m+1}^{M} DIV_{f_m,f_n}(\mathbf{X}),$$

Why deep learning is good?



Lecture 13 ILOs

- Feedforward neural networks
    - Universal approximation theorem
    - Nonlinear activation functions
    - Computation graphs
- Convolutional neural networks
    - Kernel, Receptive field
    - Pooling, down-sampling
- Recurrent neural networks
    - Back-propagation through time
    - Long-short term memory
- Learning algorithms
    - Stochastic gradient descent
    - Batch normalization
- Generalization
    - Network architecture, neural architecture search
    - Weight decay
    - Drop out
- Beyond supervised learning
    - Unsupervised learning, transfer learning, semi-supervised learning