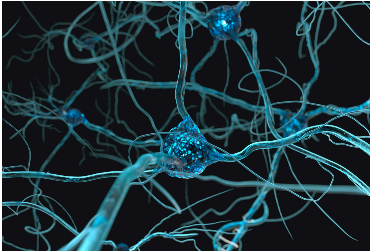
Chapter 8
Neural Networks

supplementary slides to
Machine Learning Fundamentals
©Hui Jiang 2020
published by Cambridge University Press

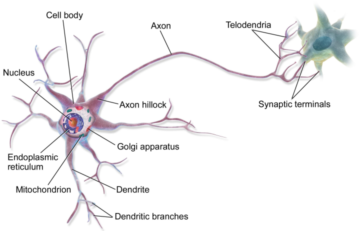
August 2020

Outline

Biological Neuronal Networks



(a) neuronal networks

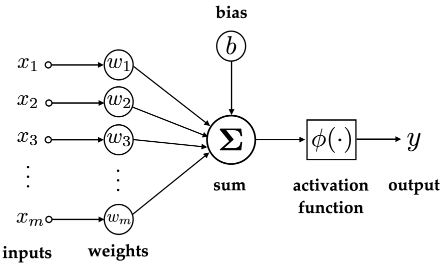


(b) biological neuron

- brain: a large number of inter-connected neurons
- neuron: axon, dendrites and synapse
- mechanisms of biological neuronal networks

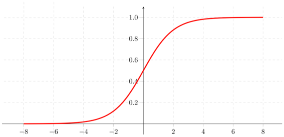
Artificial Neural Networks (ANNs)

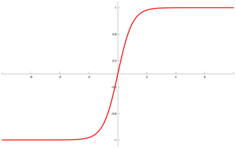
- motivated by biological neuronal networks
- artificial neuron: a simplified computational model to simulate a biological neuron $y = \phi(\sum_i w_i x_i + b)$
  - nonlinear activation function: sigmoid, tanh, ReLU, etc.

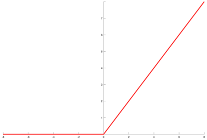


- ANNs consist of a large number of artificial neurons

Nonlinear Activation Functions







$$\text{sigmoid}: y = \frac{1}{1+e^{-x}}$$

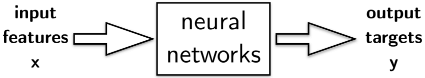
$$\tanh: y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{ReLU}: y = \max(0, x)$$

- sigmoid: $(0, 1)$, monotonically increasing, differentiable everywhere
- tanh: $(-1, 1)$, monotonically increasing, differentiable everywhere
- ReLU: $[0, \infty)$, monotonically non-decreasing, unbounded

Neural Networks: Mathematical Justification

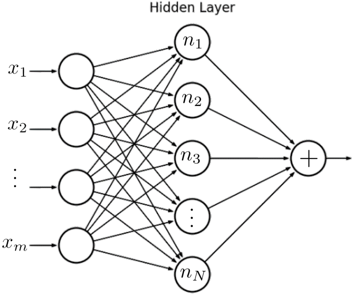
- neural networks are primarily used as a function approximator



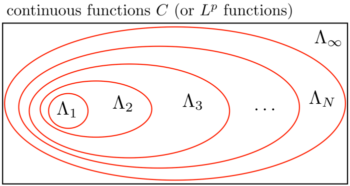
- what is the modeling power of neural networks?
- linear functions vs. nonlinear functions
- $f(\mathbf{x})$ is an L^p function $(\forall p > 0)$ iff $\int_{\mathbf{x}} |f(\mathbf{x})|^p \, d\mathbf{x} < \infty$
- including either energy-limited functions, or bounded functions on finite-domain
- e.g. all L^2 functions $(p = 2)$ form a Hilbert space, consisting of all functions arising from any physical process

Neural Networks: Universal Approximator (I)

multilayer perceptrons (MLP): a simple structure for neural nets, containing only one hidden layer between input and output



(c) MLP



$\Lambda_1 \subset \Lambda_2 \subset \Lambda_3 \cdots \subset \Lambda_N \subset \cdots \subset \Lambda_\infty \equiv C(\text{or } L^p)$

(d) nested function spaces

Neural Networks: Universal Approximator (II)

- MLPs are universal function approximators

Theorem 1

Denote all continuous functions on $\mathbb{R}^m$ as C. If the nonlinear activation function $\phi(\cdot)$ is continuous, bounded and non-constant, then Λ_N is dense in C as $N \to \infty$, i.e. $\lim_{N\to\infty} \Lambda_N = C$.

Theorem 2

Denote all L^p functions on $\mathbb{R}^m$ as L^p. If the ReLU function is used as the activation function $\phi(\cdot)$, then Λ_N is dense in L^p as $N \to \infty$, i.e. $\lim_{N\to\infty} \Lambda_N = L^p$.

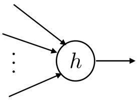
- applicable to many other neural network structures

Neural Network Structures

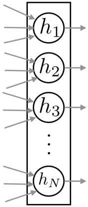
- Neurons vs. Layers of Neurons
- Building Blocks
  - full connection, convolution
  - nonlinear activation, softmax, max-pooling
  - normalization
  - time-delayed feedback
  - tapped delay line
  - attention
- Case Studies:
  1. fully-connected deep neural networks (DNNs)
  2. convolutional neural networks (CNNs)
  3. recurrent neural networks (RNNs)
  4. transformers

Neurons vs. Layers of Neurons

- a neuron: mathematically represents a variable in computation
- convenient to group relevant neurons as a layer
- a layer of neurons: represents a vector in computation
- neural nets are constructed by arranging layers of neurons



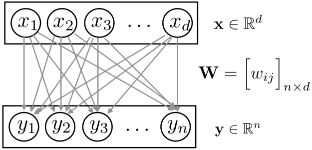
a neuron h



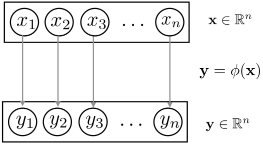
a layer
$\mathbf{h} = [h_1; h_2; \cdots; h_N]$

Building Blocks to Connect Layers (I)

- **full connection**: $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$
  - $\mathbf{W} \in \mathbb{R}^{n \times d}$ and $\mathbf{b} \in \mathbb{R}^n$ denote all parameters in a full connection
  - $n \times (d+1)$ parameters
  - computational complexity is $O(n \times d)$
  - mainly used for universal function approximation



- **nonlinear activation**: $\mathbf{y} = \phi(\mathbf{x})$
  - $\phi(\cdot)$: ReLU, sigmoid or tanh
  - no learnable parameter in this connection
  - used to introduce nonlinearity



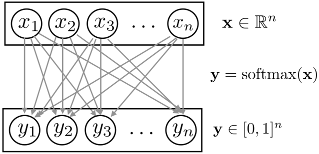
Building Blocks to Connect Layers (II)

- **softmax**

$$\mathbf{y} = \text{softmax}(\mathbf{x})$$

where $y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ for all i

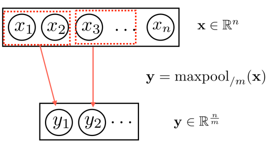
  - no learnable parameter in this connection
  - used to generate probability-like outputs



- **max-pooling**

$$\mathbf{y} = \text{maxpool}_{/m}(\mathbf{x}) \quad (\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^{\frac{n}{m}})$$

  - no learnable parameter in this connection
  - used to reduce the layer size
  - make the output less sensitive to small translation variations



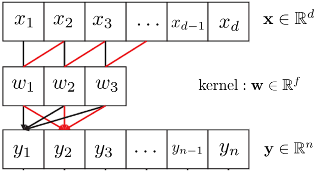
Building Blocks to Connect Layers (III)

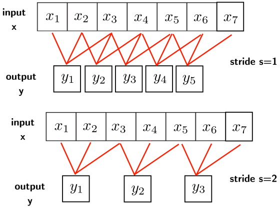
- **convolution**:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \quad (\mathbf{x} \in \mathbb{R}^d, \ \mathbf{w} \in \mathbb{R}^f, \ \mathbf{y} \in \mathbb{R}^n)$$

where $y_j = \sum_{i=1}^{f} w_i \times x_{j+i-1} \quad (\forall j)$

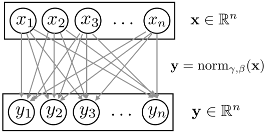
  - kernel $\mathbf{w}$ represents f learnable parameters
  - computational complexity: $O(d \times f)$
  - output neurons are $n = d - f + 1$ but can be adjusted by zero-padding and striding
  - convolution vs. full connection
    1. locality modelling: only capture a local feature
    2. weight sharing: $f (< d)$ weights (vs. $d \times n$ weights in full connection)





Building Blocks to Connect Layers (IV)

- **normalization**
  - normalize the dynamic ranges of neurons
  - smooth out the loss surface to facilitate optimization



1. *batch normalization*: $\mathbf{y} = \text{BN}_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\mathbf{x})$

   (1) normalize: $\hat{x}_i = \dfrac{x_i - \mu_{\text{B}}(i)}{\sqrt{\sigma^2_{\text{B}}(i) + \epsilon}}$ (2) re-scaling : $y_i = \gamma_i \hat{x}_i + \beta_i$

   where $\mu_{\text{B}}(i)$ and $\sigma^2_{\text{B}}(i)$ denote the sample mean and the sample variance over the current mini-batch

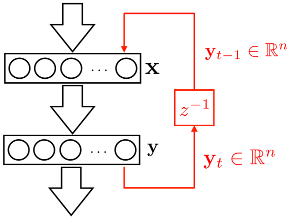
2. *layer normalization*: $\mathbf{y} = \text{LN}_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\mathbf{x})$
   where local statistics are estimated over all dimensions in each input vector $\mathbf{x}$

Building Blocks to Connect Layers (V)

- **time-delayed feedback**

$$\mathbf{y}_{t-1} = z^{-1}(\mathbf{y}_t)$$

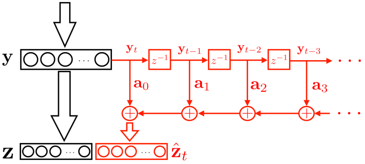
  - z^{-1} indicates a time-delay unit, which is physically implemented as a memory unit
  - recurrent neural networks (RNNs) use feedback to memorize the history
  - feedback paths introduce circles in nets



- **tapped delay line**
  - stored in a line of memory units
  - linearly combined to feed forward

$$\hat{\mathbf{z}}_t = \sum_{i=0}^{L-1} \mathbf{a}_i \otimes \mathbf{y}_{t-i}$$

    where $\{\mathbf{a}_i\}$ are learnable parameters
  - no feedback path $\Longrightarrow$ non-recurrent structures to memorize the history



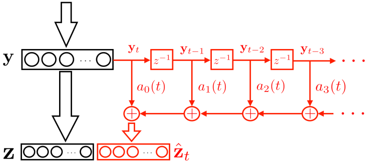
Building Blocks to Connect Layers (V): Attention (1)

- **attention**: use time-variant scalar coefficients in tapped delay lines
  1. tapped-delay-line is long enough to store entire sequence
  2. introduce an attention function $g(\,)$

     $$g(\mathbf{q}_t, \mathbf{k}_t) \triangleq \begin{bmatrix} c_0(t) & c_1(t) & \cdots & c_{L-1}(t) \end{bmatrix}^\intercal$$

     - $\mathbf{q}_t \in \mathbb{R}^l$: query vector at time t
     - $\mathbf{k}_t \in \mathbb{R}^l$: key vector at time t
  3. normalize to one by softmax

     $$\mathbf{a}_t = \text{softmax}\big(g(\mathbf{q}_t, \mathbf{k}_t)\big)$$
  4. linearly combined at each time t



$$\hat{\mathbf{z}}_t = \sum_{i=0}^{L-1} a_i(t)\mathbf{y}_{t-i} = \begin{bmatrix} \mathbf{y}_t \ \mathbf{y}_{t-1} & \cdots & \mathbf{y}_{t-L+1} \end{bmatrix} \mathbf{a}_t$$

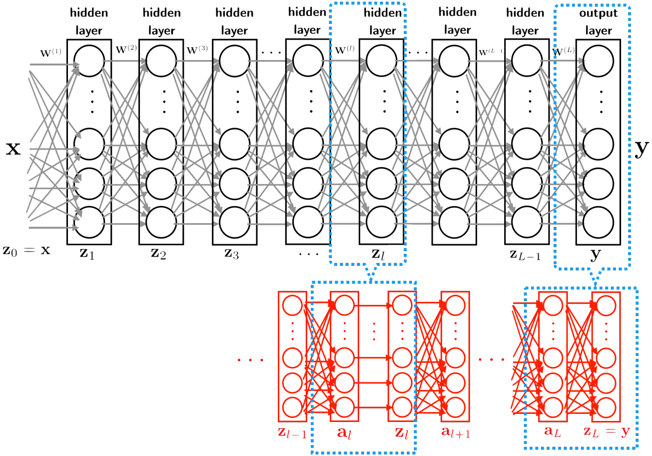
Building Blocks to Connect Layers (VI): Attention (2)

- use a matrix form to represent attention for all time instances
- value matrix: $\mathbf{V} = \begin{bmatrix} \mathbf{y}_T & \mathbf{y}_{T-1} & \cdots & \mathbf{y}_1 \end{bmatrix}_{n \times T}$
- query matrix: $\mathbf{Q} \triangleq \begin{bmatrix} \mathbf{q}_T & \mathbf{q}_{T-1} & \cdots & \mathbf{q}_1 \end{bmatrix}_{l \times T}$
- key matrix: $\mathbf{K} \triangleq \begin{bmatrix} \mathbf{k}_T & \mathbf{k}_{T-1} & \cdots & \mathbf{k}_1 \end{bmatrix}_{l \times T}$
- attention in a compact form:

$$\hat{\mathbf{Z}} = \mathbf{V}\ \text{softmax}\Big(g(\mathbf{Q}, \mathbf{K})\Big)$$

  where softmax is applied to $g(\mathbf{Q}, \mathbf{K}) \in \mathbb{R}^{T \times T}$ column-wise
- attention represents a very flexible and complex computation in neural networks, depending on how to choose the four elements: $\mathbf{V}$, $\mathbf{Q}$, $\mathbf{K}$ and $g(\cdot)$

Case Study (I): Fully-Connected Deep Neural Networks (1)



Case Study (I): Fully-Connected Deep Neural Networks (2)

Forward Pass of a fully-connected DNN

1. For the input layer: $\mathbf{z}_0 = \mathbf{x}$
2. For each hidden layer $l = 1, 2, \cdots, L-1$:

$$\mathbf{a}_l = \mathbf{W}^{(l)} \mathbf{z}_{l-1} + \mathbf{b}^{(l)}$$

$$\mathbf{z}_l = \text{ReLU}(\mathbf{a}_l)$$

3. For the output layer:

$$\mathbf{a}_L = \mathbf{W}^{(L)} \mathbf{z}_{L-1} + \mathbf{b}^{(L)}$$

$$\mathbf{y} = \mathbf{z}_L = \text{softmax}(\mathbf{a}_L)$$

Case Study (II): Convolutional Neural Networks

- convolutional neural networks (CNNs) are currently the dominant model for images/videos
- CNNs mainly rely on the basic convolution sum
- extension #1: allow multiple feature plies in input
- extension #2: allow multiple kernels
- extension #3: allow multiple input dimensions
- extension #4: stack many convolution layers
- typical CNN architectures:
  - AlexNet, VGG, ResNet, etc.

From Convolution Sum to CNNs (1)

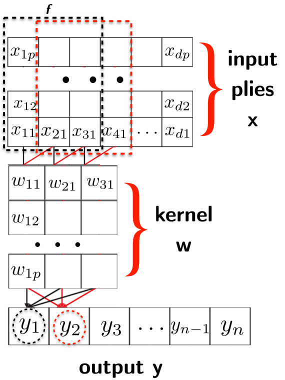
extension #1: allow multiple feature plies in input $\mathbf{x}$

- each input position contains p feature plies (e.g. R/G/B in color images)
- extend kernel to p plies ($p \times f$ weights)

$$y_j = \sum_{k=1}^{p} \sum_{i=1}^{f} w_{i,k} \times x_{j+i-1,k} \qquad (\forall j = 1, 2, \cdots, n)$$

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \qquad (\mathbf{x} \in \mathbb{R}^{p \times d},\ \mathbf{w} \in \mathbb{R}^{p \times f},\ \mathbf{y} \in \mathbb{R}^{n})$$

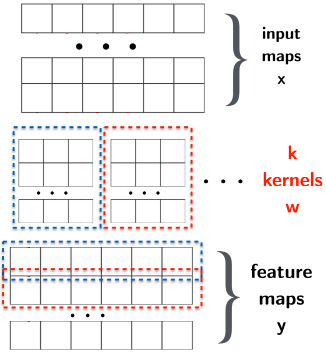
- computational complexity: $O(d \cdot f \cdot p)$
- zero-padding and striding
- locality modeling, weight sharing



From Convolution Sum to CNNs (2)

extension #2: allow multiple kernels for more local features

- a kernel captures only one local feature
- extend to k kernels ($p \times f \times k$ weights)
- output is a $k \times n$ feature map



$$y_{j_1,j_2} = \sum_{i_2=1}^{p} \sum_{i_1=1}^{f} w_{i_1,i_2,j_2} \times x_{j_1+i_1-1,i_2}$$

$$(\forall j_1 = 1, \cdots, n;\ j_2 = 1, \cdots, k)$$

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \quad (\mathbf{x} \in \mathbb{R}^{p\times d},\ \mathbf{w} \in \mathbb{R}^{p\times f\times k},\ \mathbf{y} \in \mathbb{R}^{k\times n})$$

- computational complexity: $O(d \cdot f \cdot p \cdot k)$
- zero-padding and striding
- locality modeling, weight sharing

From Convolution Sum to CNNs (3)

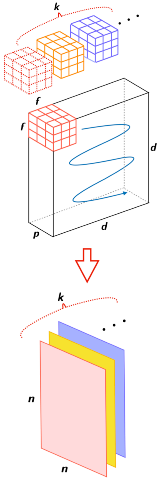
- **extension #3**: allow multiple input dimensions
- expand input dimension to handle multi-dim data, e.g. images (2D) and videos (3D)
- for 2D images, each input $\mathbf{x}$ is a $d \times d \times p$ tensor, extend each kernel into an $f \times f \times p$ tensor, output is an $n \times n \times k$ feature map

$$y_{j_1,j_2,j_3} = \sum_{i_3=1}^{p} \sum_{i_2=1}^{f} \sum_{i_1=1}^{f} w_{i_1,i_2,i_3,j_3} \times x_{j_1+i_1-1,j_2+i_2-1,i_3}$$

$$(j_1 = 1, \cdots, n;\; j_2 = 1, \cdots, n;\; j_3 = 1, \cdots, k)$$

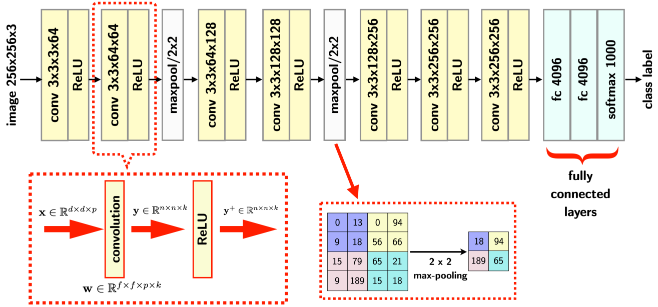
$$\mathbf{y} = \mathbf{x} * \mathbf{w} \quad (\mathbf{x} \in \mathbb{R}^{d \times d \times p},\; \mathbf{w} \in \mathbb{R}^{f \times f \times p \times k},\; \mathbf{y} \in \mathbb{R}^{n \times n \times k})$$

- computational complexity: $O(d^2 \cdot f^2 \cdot p \cdot k)$
- locality modeling: capture 2D local features



From Convolution Sum to CNNs (4)

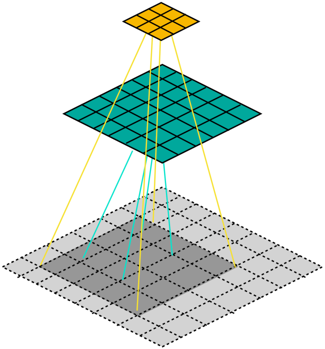
extension #4: stack many convolution layers to form CNNs



- stacked convolution layers: hierarchical visual feature extraction
- fully-connected layers: a universal function approximator to map these features to the target labels

Convolutional Neural Networks (CNNs)

- locality modelling $\Longrightarrow$ hierarchical modeling
  - recursively combine local features
  - **receptive fields** in CNN: broaden in upper layers
- CNNs are dominant in image classification, segmentation, generation
- typical CNN architectures:
  - AlexNet, VGG, ResNet, etc.
  - *ResNet*: a very deep structure with shortcut paths



Case Study (III): Recurrent Neural Network (RNN)

- use a simple RNN to process a sequence of input vectors: $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_T\}$
- for all $t = 1, 2, \cdots, T$

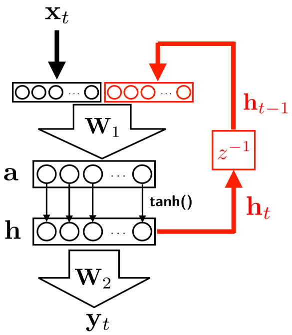
$$\mathbf{a}_t = \mathbf{W}_1 \left[\mathbf{x}_t;\ \mathbf{h}_{t-1}\right] + \mathbf{b}_1$$

$$\mathbf{h}_t = \tanh(\mathbf{a}_t)$$

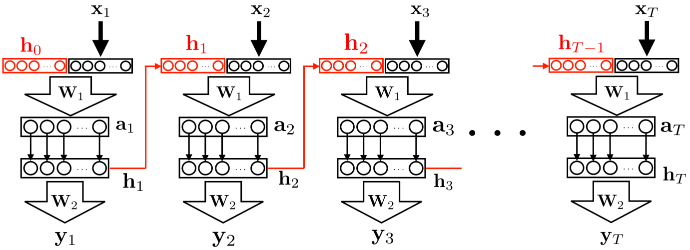
$$\mathbf{y}_t = \mathbf{W}_2 \mathbf{h}_t + \mathbf{b}_2$$

where $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$ and $\mathbf{b}_2$ are all RNN parameters

- RNN generates an output sequence: $\{\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_T\}$



Case Study (III): Recurrent Neural Network (RNN)



- an RNN can be unfolded into a non-recurrent structure
- RNNs fail to capture long-term dependency due to long traversal paths in the deep structures
- more effective RNN structures, e.g. LSTMs, GRUs, HORNNs

Case Study (IV): Transformer (1)

use a particular attention mechanism to directly map an input sequence to an output sequence

$$\mathbf{X} = \begin{bmatrix}\mathbf{x}_T & \cdots & \mathbf{x}_2 \; \mathbf{x}_1\end{bmatrix} \quad \longmapsto \quad \mathbf{Z} = \begin{bmatrix}\mathbf{z}_T & \cdots & \mathbf{z}_2 \; \mathbf{z}_1\end{bmatrix}$$

1. choose query matrix $\mathbf{Q}$, key matrix $\mathbf{K}$ and value matrix $\mathbf{V}$ as:

$$\mathbf{Q} = \mathbf{A}\mathbf{X} \qquad \mathbf{K} = \mathbf{B}\mathbf{X} \qquad \mathbf{V} = \mathbf{C}\mathbf{X}$$

   where $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{l \times d}$; $\mathbf{C} \in \mathbb{R}^{o \times d}$; $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{l \times T}$ and $\mathbf{V} \in \mathbb{R}^{o \times T}$

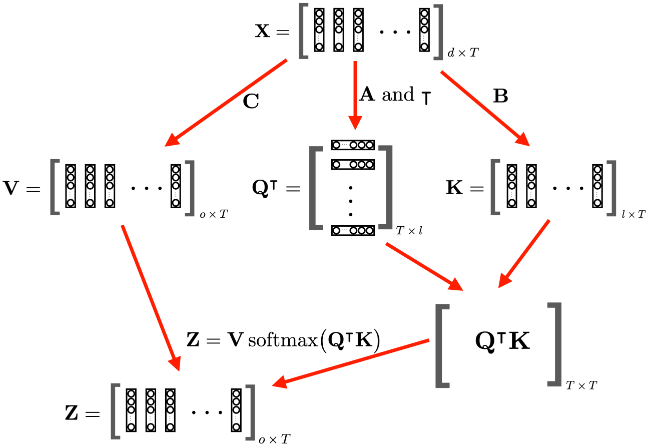
2. define the attention function as a bilinear function:

$$g(\mathbf{Q}, \mathbf{K}) = \mathbf{Q}^\intercal \mathbf{K} \quad (\in \mathbb{R}^{T \times T})$$

3. transformer as attention:

$$\mathbf{Z} = (\mathbf{C}\mathbf{X}) \; \mathrm{softmax}\Big((\mathbf{A}\mathbf{X})^\intercal (\mathbf{B}\mathbf{X}) \Big)$$

Case Study (IV): Transformer (2)



two enhancements:

1. use multiple heads in each transformer
2. stack more transformer layers to form a deep structure

Case Study (IV): Transformer (3)

Multi-head Transformer

Choose $d = 512$, $o = 64$, a multi-head transformer will transform an input sequence $\mathbf{X} \in \mathbb{R}^{512 \times T}$ into $\mathbf{Y} \in \mathbb{R}^{n \times T}$:

- multi-head transformer: use 8 sets of parameters $\mathbf{A}^{(j)}, \mathbf{B}^{(j)} \in \mathbb{R}^{l \times 512}, \mathbf{C}^{(j)} \in \mathbb{R}^{64 \times 512} \quad (j = 1, 2, \cdots, 8)$
- for $j = 1, 2, \cdots, 8$:

$$\mathbf{Z}^{(j)} \in \mathbb{R}^{64 \times T} = \left(\mathbf{C}^{(j)}\mathbf{X}\right) \text{softmax}\Big(\left(\mathbf{A}^{(j)}\mathbf{X}\right)^{\intercal}\left(\mathbf{B}^{(j)}\mathbf{X}\right)\Big)$$

- concatenate all heads: $\mathbf{Z} \in \mathbb{R}^{512 \times T} = \text{concat}\left(\mathbf{Z}^{(1)}, \mathbf{Z}^{(2)}, \cdots, \mathbf{Z}^{(8)}\right)$
- apply nonlinearity: $\mathbf{Y} = \text{feedforward}\Big(\text{LN}_{\gamma,\boldsymbol{\beta}}\left(\mathbf{X} + \mathbf{Z}\right)\Big)$

Learning Neural Networks

- Loss Function
- Optimization Method: SGD
- Automatic Differentiation
  - full connection
  - nonlinear activation
  - softmax
  - max-pooling
  - convolution
  - normalization
- Error Backpropagation Examples:
  - fully-connected deep neural networks

Loss Function

- once network structure is determined, a neural network can be viewed as a multivariate and vector-valued function as:

$$\mathbf{y} = f(\mathbf{x}; \mathbb{W})$$

  where $\mathbb{W}$ to denote all network parameters

- learn $\mathbb{W}$ from a training set of input-output pairs: $\mathcal{D}_N = \Big\{(\mathbf{x}_1, \mathbf{r}_1), (\mathbf{x}_2, \mathbf{r}_2), \cdots, (\mathbf{x}_N, \mathbf{r}_N)\Big\}$

- mean square error (MSE) for regression problems

$$Q_{\text{MSE}}(\mathbb{W}; \mathcal{D}_N) = \sum_{i=1}^{N} \|f(\mathbf{x}_i; \mathbb{W}) - \mathbf{r}_i\|^2$$

- cross-entropy (CE) error for classification problems

$$Q_{\text{CE}}(\mathbb{W}; \mathcal{D}_N) = -\sum_{i=1}^{N} \ln \Big[\mathbf{y}_i\Big]_{r_i} = -\sum_{i=1}^{N} \ln \Big[f(\mathbf{x}_i; \mathbb{W})\Big]_{r_i}$$

Optimization Method: mini-batch SGD

mini-batch SGD to learn neural networks

randomly initialize $\mathbb{W}^{(0)}$; set η_0, $n = 0$ and $t = 0$
while not converged **do**
  randomly shuffle training data into mini-batches
  for each mini-batch B **do**
    for each $\mathbf{x} \in B$ **do**
      compute the gradient: $\frac{\partial Q(\mathbb{W}^{(n)}; \mathbf{x})}{\partial \mathbb{W}}$
    end for
    update model: $\mathbb{W}^{(n+1)} = \mathbb{W}^{(n)} - \frac{\eta_t}{|B|} \sum_{\mathbf{x} \in B} \frac{\partial Q(\mathbb{W}^{(n)}; \mathbf{x})}{\partial \mathbb{W}}$
    $n = n + 1$
  end for
  adjust $\eta_t \to \eta_{t+1}$
  $t = t + 1$
end while

Automatic Differentiation (I)

- how to efficiently compute gradients for arbitrary networks?
- automatic differentiation (AD), a.k.a. error back-propagation:
  - the most efficient for any network structure by systematically applying the chain rule

a simple example:

$$\cdots \xrightarrow{x} \boxed{y = f_{\mathbf{w}}(x)} \xrightarrow{y} \cdots$$

given any objective function $Q(\cdot)$

1. define the error signal: $e = \frac{\partial Q}{\partial y}$
2. derive the gradient by local computations:

$$\frac{\partial Q}{\partial \mathbf{w}} = \frac{\partial Q}{\partial y} \frac{\partial y}{\partial \mathbf{w}} = e \frac{\partial f_{\mathbf{w}}(x)}{\partial \mathbf{w}}$$

3. back-propagate the error signal:

$$\frac{\partial Q}{\partial x} = \frac{\partial Q}{\partial y} \frac{\partial y}{\partial x} = e \frac{d f_{\mathbf{w}}(x)}{dx}$$

Automatic Differentiation (II)

extend AD to a vector-input and vector-output module:

$$\cdots \xrightarrow{\mathbf{x} \in \mathbb{R}^m} \boxed{\mathbf{y} = f_{\mathbf{w}}(\mathbf{x})} \xrightarrow{\mathbf{y} \in \mathbb{R}^n} \cdots$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{bmatrix}$$

compute two Jacobian matrices:

$$J_{\mathbf{w}} = \left[\frac{\partial y_j}{\partial w_i} \right]_{k \times n}$$

$$J_{\mathbf{x}} = \left[\frac{\partial y_j}{\partial x_i} \right]_{m \times n}$$

- given the error signal $\mathbf{e} \triangleq \frac{\partial Q}{\partial \mathbf{y}} \quad (\in \mathbb{R}^n)$

  1. **local gradients:**

  $$\frac{\partial Q}{\partial \mathbf{w}} = J_{\mathbf{w}} \; \mathbf{e}$$

  2. **back-propagation:**

  $$\frac{\partial Q}{\partial \mathbf{x}} = J_{\mathbf{x}} \; \mathbf{e}$$

Automatic Differentiation (III)

- **full connection** from $\mathbf{x} \in \mathbb{R}^d$ to output $\mathbf{y} \in \mathbb{R}^n$:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where $\mathbf{W} \in \mathbb{R}^{n\times d}$ and $\mathbf{b} \in \mathbb{R}^n$

- back-propagation:

$$J_{\mathbf{x}} = \left[\frac{\partial y_j}{\partial x_i}\right]_{d\times n} = \mathbf{W}^\intercal \qquad \Longrightarrow \qquad \frac{\partial Q}{\partial \mathbf{x}} = \mathbf{W}^\intercal \mathbf{e}$$

- local gradients:

$$\frac{\partial Q}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial Q}{\partial y_1} \\ \vdots \\ \frac{\partial Q}{\partial y_n} \end{bmatrix} \mathbf{x}^\intercal = \mathbf{e}\,\mathbf{x}^\intercal \qquad \frac{\partial Q}{\partial \mathbf{b}} = \mathbf{e}$$

Automatic Differentiation (III)

- **nonlinear activation** from $\mathbf{x}$ ($\in \mathbb{R}^n$) to $\mathbf{y}$ ($\in \mathbb{R}^n$):

$$\mathbf{y} = \phi(\mathbf{x})$$

- no learnable parameters $\implies$ no local gradients
- back-propagation:

$$\frac{\partial Q}{\partial \mathbf{x}} = \mathbf{J}_{\mathbf{x}}\mathbf{e} = \phi'(\mathbf{x}) \odot \mathbf{e}$$

  where $\odot$ denotes element-wise multiplication
  - for ReLU activation: $\frac{\partial Q}{\partial \mathbf{x}} = H(\mathbf{x}) \odot \mathbf{e}$
  - for sigmoid activation: $\frac{\partial Q}{\partial \mathbf{x}} = l(\mathbf{x}) \odot \big(\mathbf{1} - l(\mathbf{x})\big) \odot \mathbf{e}$

Automatic Differentiation (IV)

- **softmax**: mapping an n-dimensional vector $\mathbf{x}$ $(\in \mathbb{R}^n)$ into another n-dimensional vector $\mathbf{y}$ inside the hypercube $[0,1]^n$, with $y_j = \frac{e^{x_j}}{\sum_{i=1}^n e^{x_i}}$ for all $i = 1, 2, \cdots, n$
- no learnable parameters $\Longrightarrow$ no local gradients
- the Jacobian matrix

$$\mathbf{J_x} = \left[\frac{\partial y_j}{\partial x_i}\right]_{n\times n} = \begin{bmatrix} y_1(1-y_1) & -y_1y_2 & \cdots & -y_1y_n \\ -y_1y_2 & y_2(1-y_2) & \cdots & -y_2y_n \\ \vdots & \vdots & \vdots & \vdots \\ -y_1y_n & -y_2y_n & \cdots & y_n(1-y_n) \end{bmatrix}_{n\times n}$$

- back-propagation:

$$\frac{\partial Q}{\partial \mathbf{x}} = \mathbf{J_x}\mathbf{e}$$

Automatic Differentiation (V): Convolution (1)

- **convolution**: mapping an input vector $\mathbf{x} \in \mathbb{R}^d$ to an output vector $\mathbf{y} \in \mathbb{R}^n$ by $\mathbf{y} = \mathbf{x} * \mathbf{w}$ with $\mathbf{w} \in \mathbb{R}^f$, with

$$y_j = \sum_{i=1}^{f} w_i \times x_{j+i-1} \qquad j = 1, 2 \cdots, n$$

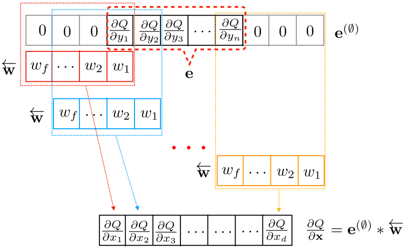
- the Jacobian matrix $\mathbf{J_x}$:

$$\mathbf{J_x} = \left[\frac{\partial y_j}{\partial x_i} \right]_{d \times n} = \begin{bmatrix} w_1 & & & \\ w_2 & w_1 & & \\ \vdots & \vdots & \ddots & \\ w_f & w_{f-1} & \ddots & w_1 \\ & w_f & & w_2 \\ & & \ddots & \vdots \\ & & & w_f \end{bmatrix}_{d \times n}$$

Automatic Differentiation (V): Convolution (2)

- back-propagation by convolution:

$$
\begin{aligned}
\frac{\partial Q}{\partial \mathbf{x}} &= \mathbf{J_x e} = \begin{bmatrix} w_1 \frac{\partial Q}{\partial y_1} \\ w_2 \frac{\partial Q}{\partial y_1} + w_1 \frac{\partial Q}{\partial y_2} \\ \vdots \\ w_f \frac{\partial Q}{\partial y_n} \end{bmatrix} \\
&\triangleq \mathbf{e}^{(\emptyset)} * \overleftarrow{\mathbf{w}}
\end{aligned}
$$



- computing local gradients by convolution:

$$
\mathbf{J_w} = \left[\frac{\partial y_j}{\partial w_i}\right]_{f\times n} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_f & x_{f+1} & \cdots & x_{n+f-1} \end{bmatrix}_{f\times n} \implies \frac{\partial Q}{\partial \mathbf{w}} = \mathbf{J_w e} \triangleq \mathbf{x} * \mathbf{e}
$$

Automatic Differentiation (V): Convolution (3)

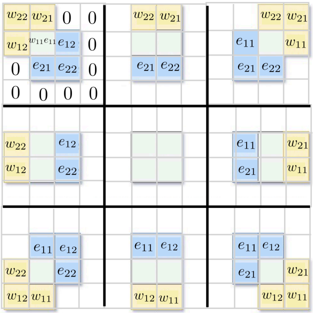
- extend to 2D convolutions
- back-propagation by convolution:

$$\frac{\partial Q}{\partial \mathbf{x}_i} = \sum_{j=1}^{k} \mathbf{e_j}^{(\emptyset)} * \overleftarrow{\mathbf{w}}_{ij} \quad (i = 1, 2 \cdots p)$$

- computing local gradient by convolution:

$$\frac{\partial Q}{\partial \mathbf{w}_{ij}} = \mathbf{x}_i * \mathbf{e}_j \quad (i = 1, 2 \cdots p; \;\; j = 1, 2 \cdots k)$$

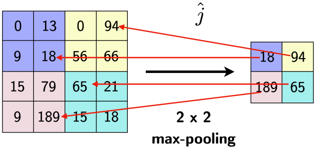
where $\mathbf{x}_i \in \mathbb{R}^{d \times d}$ and $\mathbf{e}_j \in \mathbb{R}^{n \times n}$



Automatic Differentiation (VI)

- **max-pooling**:
  - no parameters $\Longrightarrow$ no local gradients
  - back-propagation:

$$\frac{\partial Q}{\partial x_i} = \begin{cases} \frac{\partial Q}{\partial y_j} & \text{if } i = \hat{j} \\ 0 & \text{otherwise} \end{cases}$$



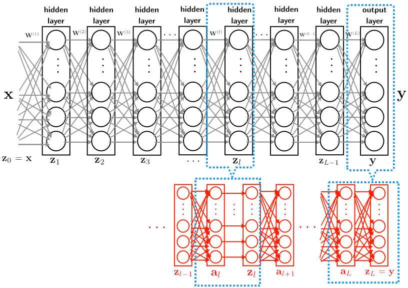
- **batch normalization**: $\mathbf{y} = \text{BN}_{\boldsymbol{\gamma},\boldsymbol{\beta}}(\mathbf{x})$
  - back-propagation:

$$\frac{\partial Q}{\partial \mathbf{x}^{(m)}} = \frac{M\boldsymbol{\gamma}\odot\mathbf{e}^{(m)} - \sum_{k=1}^{M}\boldsymbol{\gamma}\odot\mathbf{e}^{(k)} - \boldsymbol{\gamma}\odot\hat{\mathbf{x}}^{(m)}\odot\left(\sum_{k=1}^{M}\mathbf{e}^{(k)}\odot\hat{\mathbf{x}}^{(k)}\right)}{M\sqrt{\sigma_{\text{B}}^2(i)+\epsilon}}$$

  - local gradients:

$$\frac{\partial Q}{\partial \boldsymbol{\gamma}} = \sum_{k=1}^{M}\hat{\mathbf{x}}^{(k)}\odot\mathbf{e}^{(k)} \qquad \frac{\partial Q}{\partial \boldsymbol{\beta}} = \sum_{k=1}^{M}\mathbf{e}^{(k)}$$

Error Backpropagation Example: Fully-Connected DNNs



- all parameters: $\mathbb{W} = \left\{ \mathbf{W}^{(l)}, \mathbf{b}^{(l)} \mid l = 1, 2 \cdots L \right\}$
- the cross-entropy error: $Q(\mathbb{W}; \mathbf{x}) = -\ln \left[\mathbf{y} \right]_r \quad \Longrightarrow$

$$\frac{\partial Q(\mathbb{W}; \mathbf{x})}{\partial \mathbf{y}} = \left[0 \ \cdots \ 0 \ \ -\frac{1}{y_r} \ \ 0 \ \cdots \ 0 \right]^{\intercal}$$

- define error signals $\mathbf{e}^{(l)} = \frac{\partial Q(\mathbb{W};\mathbf{x})}{\partial \mathbf{a}_l}$ for all $l = L, \cdots, 2, 1$
- apply AD to the softmax, nonlinear activation and full connection modules to back-propagate error signals

Error Backpropagation Example: Fully-Connected DNNs

backward pass of fully-connected DNNs

for the cross-entropy error of any input-output pair $(\mathbf{x}, \mathbf{r})$

1. for the output layer L:
$$\mathbf{e}^{(L)} = \begin{bmatrix} y_1 \; y_2 \; \cdots \; y_r - 1 \; \cdots \; y_n \end{bmatrix}^\intercal$$
2. for each hidden layer $l = L-1, \cdots, 2, 1$:
$$\mathbf{e}^{(l)} = \left(\left(\mathbf{W}^{(l+1)}\right)^\intercal \mathbf{e}^{(l+1)} \right) \odot H(\mathbf{z}_l)$$
3. for all layers $l = L, \cdots, 2, 1$:
$$\frac{\partial Q(\mathbb{W};\mathbf{x})}{\partial \mathbf{W}^{(l)}} = \mathbf{e}^{(l)} \left(\mathbf{z}_{l-1}\right)^\intercal$$
$$\frac{\partial Q(\mathbb{W};\mathbf{x})}{\partial \mathbf{b}^{(l)}} = \mathbf{e}^{(l)}$$

where $\mathbf{y}$ and $\mathbf{z}_l$ $(l = 0, 1, \cdots, L-1)$ are computed in the forward pass

Heuristics and Tricks for Optimization

- Hyperparmeters
- Optimization Method: ADAM
- Regularization
- Fine-tuning Tricks

Hyperparmeters of Learning Neural Networks

- initial parameters
- epoch number
- mini-batch size
- learning rate
  - a good initial learning rate η_0
  - an annealing schedule to adjust $\eta_t \rightarrow \eta_{t+1}$
  - call for some self-adjusting mechanisms, e.g. Adagrad, Adadelta, ADAM, AdaMax, etc.

Optimization method: ADAM

ADAM to learn neural networks

randomly initialize $\mathbb{W}^{(0)}$, and set η, $t = 0$, $n = 0$ and $\mathbf{u}_0 = \mathbf{v}_0 = 0$

while not converged **do**

  randomly shuffle training data into mini-batches

  for each mini-batch B **do**

    for each $\mathbf{x} \in B$ **do**

      compute $\frac{\partial Q(\mathbb{W}^{(n)}\,;\mathbf{x})}{\partial \mathbb{W}}$

    end for

    $\mathbf{g}_n = \frac{1}{|B|} \sum_{\mathbf{x}\in B} \frac{\partial Q(\mathbb{W}^{(n)}\,;\mathbf{x})}{\partial \mathbb{W}}$

    $\mathbf{u}_{n+1} = \alpha\,\mathbf{u}_n + (1-\alpha)\,\mathbf{g}_n$ and $\mathbf{v}_{n+1} = \beta\,\mathbf{v}_n + (1-\beta)\,\mathbf{g}_n \odot \mathbf{g}_n$

    $\hat{\mathbf{u}}_{n+1} = \frac{\mathbf{u}_{n+1}}{1-\alpha^{n+1}}$ and $\hat{\mathbf{v}}_{n+1} = \frac{\mathbf{v}_{n+1}}{1-\beta^{n+1}}$

    update model: $\mathbb{W}^{(n+1)} = \mathbb{W}^{(n)} - \eta \cdot \hat{\mathbf{u}}_{n+1} \odot \left((\hat{\mathbf{v}}_{n+1} + \boldsymbol{\epsilon}^2)^{-\frac{1}{2}} \right)$

    $n = n + 1$

  end for

  $t = t + 1$

end while

Self-adjusting Mechanism in ADAM

- use exponential average to accumulate 1st-order and 2nd-order moments ($\mathbf{u}_n$ and $\mathbf{v}_n$) of the gradient ($\mathbf{g}_n$)
- normalize to yield unbiased estimates:

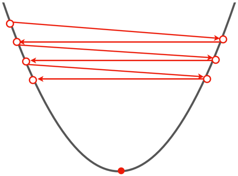
$$\mathbb{E}\big[\hat{u}_{n+1}(i)\big] = \mathbb{E}\big[g_n(i)\big] \quad \mathbb{E}\big[\hat{v}_{n+1}(i)\big] = \mathbb{E}\big[g_n^2(i)\big]$$

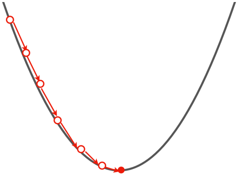
- model update formula:

$$\mathbb{W}_i^{(n+1)} = \mathbb{W}_i^{(n)} - \eta \frac{\hat{u}_{n+1}(i)}{\sqrt{\hat{v}_{n+1}(i) + \epsilon^2}}$$

- self-adjusting model updates $\Delta\mathbb{W}_i^{(n)}$:

$$\big\|\Delta\mathbb{W}_i^{(n)}\big\|^2 \simeq \eta^2 \frac{\big(\mathbb{E}\big[\hat{u}_{n+1}(i)\big]\big)^2}{\mathbb{E}\big[\hat{v}_{n+1}(i)\big]} = \frac{\eta^2\big(\mathbb{E}\big[g_n(i)\big]\big)^2}{\big(\mathbb{E}\big[g_n(i)\big]\big)^2 + \mathsf{var}\big[g_n(i)\big]}$$





Regularization in Neural Networks

- **weight decay**: use L_2 norm regularization

$$Q(\mathbb{W}) + \frac{\lambda}{2} \cdot \left\| \mathbb{W} \right\|^2 \implies \mathbb{W}^{(n+1)} = \mathbb{W}^{(n)} - \eta \frac{\partial Q(\mathbb{W}^{(n)})}{\partial \mathbb{W}} - \lambda \cdot \mathbb{W}^{(n)}$$

- **weight normalization**: normalize weight vectors to facilitate optimization
  1. tied-scalar reparameterization:
  $$\mathbf{w} = \gamma \cdot \mathbf{v} \quad \text{s.t.} \quad ||\mathbf{v}|| \leq 1$$
  2. normalizing reparameterization:
  $$\mathbf{w} = \frac{\gamma}{||\mathbf{v}||}\mathbf{v}$$

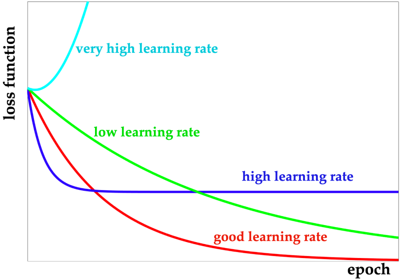
- **dropout**

- **data augmentation**

Fine-tuning Tricks

critical to monitor three learning curves:

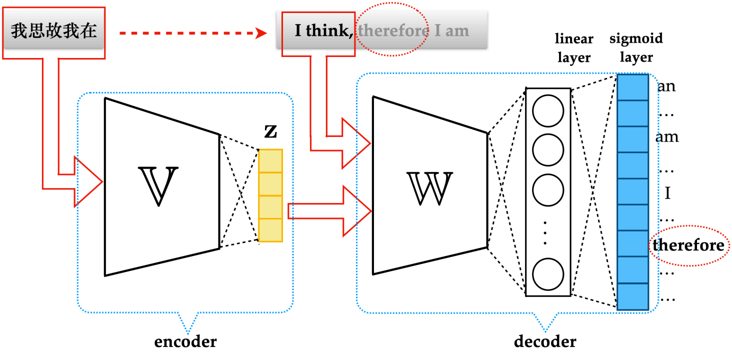
- the objective function (a.k.a. loss function)
- performance on training data
- performance on development data



End-to-End Learning

- **end-to-end learning**: train a single model to map directly from raw data to final targets
- neural networks are suitable for end-to-end learning
  - flexible architectures to accommodate a variety of raw data
  - powerful enough to approximate potentially complex mapping
  - arrange output structures to generate real data, e.g. *deconvolution* layers for images, *WaveNet* for audio/speech
- the popular **encoder-decoder** structure
- **sequence-to-sequence learning**: learn deep neural networks to map from one input sequence to an output sequence
  - suitable for many NLP tasks, e.g. machine translation, question-answering, etc.

Sequence-to-Sequence Learning



- encoder and decoder are powerful neural networks that can handle sequences, e.g. RNNs, LSTMs, or transformers