Chapter 8
Neural Networks

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

August 2020
Outline

1. Artificial Neural Networks
2. Neural Network Structures
3. Learning Algorithms for Neural Networks
4. Heuristics and Tricks for Optimization
5. End-to-End Learning
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

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

\[
\text{sigmoid} : y = \frac{1}{1 + e^{-x}} \\
\text{tanh} : y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
\text{ReLU} : y = \max(0, x)
\]
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

- $f(x)$ is an $L^p$ function ($\forall p > 0$) iff $\int_x |f(x)|^p \, dx < \infty$

- 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

(d) nested function spaces

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

\[
h = [h_1; h_2; \ldots; h_N]
\]
**Building Blocks to Connect Layers (I)**

- **full connection**: \( y = Wx + b \)
  - \( W \in \mathbb{R}^{n \times d} \) and \( b \in \mathbb{R}^n \) 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**: \( y = \phi(x) \)
  - \( \phi(\cdot) \): ReLU, sigmoid or tanh
  - no learnable parameter in this connection
  - used to introduce nonlinearity
Building Blocks to Connect Layers (II)

- **softmax**
  
  \[ y = \text{softmax}(x) \]
  
  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**
  
  \[ y = \text{maxpool}_m(x) \quad (x \in \mathbb{R}^n, y \in \mathbb{R}^{n/m}) \]
  
  - 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:**
  \[ y = x \ast w \quad (x \in \mathbb{R}^d, \; w \in \mathbb{R}^f, \; y \in \mathbb{R}^n) \]
  where \( y_j = \sum_{i=1}^{f} w_i \times x_{j+i-1} \) (\( \forall j \))
  - kernel \( w \) 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**: $y = \text{BN}_{\gamma, \beta}(x)$

   (1) normalize: $\hat{x}_i = \frac{x_i - \mu_B(i)}{\sqrt{\sigma_B^2(i)} + \epsilon}$
   (2) re-scaling: $y_i = \gamma_i \hat{x}_i + \beta_i$

   where $\mu_B(i)$ and $\sigma_B^2(i)$ denote the sample mean and the sample variance over the current mini-batch

2. **layer normalization**: $y = \text{LN}_{\gamma, \beta}(x)$

   where local statistics are estimated over all dimensions in each input vector $x$
Building Blocks to Connect Layers (V)

- **time-delayed feedback**
  \[ y_{t-1} = z^{-1}(y_t) \]
  - \( z^{-1} \) 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
  \[ \hat{z}_t = \sum_{i=0}^{L-1} a_i \otimes y_{t-i} \]
  where \( \{a_i\} \) are learnable parameters
  - no feedback path \( \implies \) 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 $g()$
     \[
     g(q_t, k_t) \equiv \begin{bmatrix} c_0(t) & c_1(t) & \cdots & c_{L-1}(t) \end{bmatrix}^T
     \]
     - $q_t \in \mathbb{R}^l$: query vector at time $t$
     - $k_t \in \mathbb{R}^l$: key vector at time $t$
  3. normalize to one by softmax
     \[a_t = \text{softmax}(g(q_t, k_t))\]
  4. linearly combined at each time $t$
     \[
     \hat{z}_t = \sum_{i=0}^{L-1} a_i(t) y_{t-i} = \begin{bmatrix} y_t & y_{t-1} & \cdots & y_{t-L+1} \end{bmatrix} a_t
     \]
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:
  $$\hat{\mathbf{Z}} = \mathbf{V} \text{softmax} \left( g(\mathbf{Q}, \mathbf{K}) \right)$$
  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: $z_0 = x$

2. For each hidden layer $l = 1, 2, \cdots, L - 1$:

   $$a_l = W^{(l)} z_{l-1} + b^{(l)}$$

   $$z_l = \text{ReLU}(a_l)$$

3. For the output layer:

   $$a_L = W^{(L)} z_{L-1} + b^{(L)}$$

   $$y = z_L = \text{softmax}(a_L)$$
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.
extension #1: allow multiple feature plies in input $x$

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

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

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

- 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

\[
 y_{j_1,j_2} = \sum_{i_2=1}^{p} \sum_{i_1=1}^{f} w_{i_1,i_2,j_2} \times x_{j_1+i_1-1,i_2} \\
(\forall j_1 = 1, \cdots, n; \ j_2 = 1, \cdots, k) \\
y = x \ast w \quad (x \in \mathbb{R}^{p \times d}, \ w \in \mathbb{R}^{p \times f \times k}, \ y \in \mathbb{R}^{k \times n})
\]

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

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

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

computational complexity: $O(d^2 \cdot f^2 \cdot p \cdot k)$

locality modeling: capture 2D local features
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 → 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: \( \{x_1, x_2, \cdots, x_T\} \)
- for all \( t = 1, 2, \cdots, T \)
  \[
  a_t = W_1 [x_t; h_{t-1}] + b_1
  \]
  \[
  h_t = \tanh(a_t)
  \]
  \[
  y_t = W_2 h_t + b_2
  \]
where \( W_1, b_1, W_2 \) and \( b_2 \) are all RNN parameters
- RNN generates an output sequence: \( \{y_1, y_2, \cdots, y_T\} \)
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

\[ X = [x_T \cdots x_2 x_1] \quad \mapsto \quad Z = [z_T \cdots z_2 z_1] \]

1. choose query matrix \( Q \), key matrix \( K \) and value matrix \( V \) as:

\[ Q = AX \quad K = BX \quad V = CX \]

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

2. define the attention function as a bilinear function:

\[ g(Q, K) = Q^T K \quad (\in \mathbb{R}^{T \times T}) \]

3. transformer as attention:

\[ Z = (CX) \text{ softmax} \left( (AX)^T (BX) \right) \]
Case Study (IV): Transformer (2)

two enhancements:

1. use multiple heads in each transformer

2. stack more transformer layers to form a deep structure
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 $A^{(j)}, B^{(j)} \in \mathbb{R}^{l \times 512}, C^{(j)} \in \mathbb{R}^{64 \times 512}$ ($j = 1, 2, \ldots, 8$)
- for $j = 1, 2, \ldots, 8$:
  \[
  \mathbf{Z}^{(j)} \in \mathbb{R}^{64 \times T} = (C^{(j)} \mathbf{X}) \text{ softmax}\left((A^{(j)} \mathbf{X})^T(B^{(j)} \mathbf{X})\right)
  \]
- concatenate all heads: $\mathbf{Z} \in \mathbb{R}^{512 \times T} = \text{concat}(\mathbf{Z}^{(1)}, \mathbf{Z}^{(2)}, \ldots, \mathbf{Z}^{(8)})$
- apply nonlinearity: $\mathbf{Y} = \text{feedforward}\left(\text{LN}_{\gamma,\beta}(\mathbf{X} + \mathbf{Z})\right)$
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:

\[ y = f(x; W) \]

where \( W \) to denote all network parameters

- Learn \( W \) from a training set of input-output pairs:

\[ \mathcal{D}_N = \{(x_1, r_1), (x_2, r_2), \ldots, (x_N, r_N)\} \]

- Mean square error (MSE) for regression problems

\[ Q_{\text{MSE}}(W; \mathcal{D}_N) = \sum_{i=1}^{N} \| f(x_i; W) - r_i \|^2 \]

- Cross-entropy (CE) error for classification problems

\[ Q_{\text{CE}}(W; \mathcal{D}_N) = - \sum_{i=1}^{N} \ln [y_i]_{r_i} = - \sum_{i=1}^{N} \ln \left[ f(x_i; W) \right]_{r_i} \]
Optimization Method: mini-batch SGD

mini-batch SGD to learn neural networks

randomly initialize $\mathbf{W}^{(0)}$; set $\eta_0$, $n = 0$ and $t = 0$

while not converged do

randomly shuffle training data into mini-batches

for each mini-batch $B$ do

for each $x \in B$ do

compute the gradient: $\frac{\partial Q(\mathbf{W}^{(n)}; x)}{\partial \mathbf{W}}$

end for

update model: $\mathbf{W}^{(n+1)} = \mathbf{W}^{(n)} - \frac{\eta_t}{|B|} \sum_{x \in B} \frac{\partial Q(\mathbf{W}^{(n)}; x)}{\partial \mathbf{W}}$

$n = n + 1$

end for

adjust $\eta_t \rightarrow \eta_{t+1}$

$t = t + 1$

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:

\[ y = f_w(x) \]

1. define the error signal: \( e = \frac{\partial Q}{\partial y} \)
2. derive the gradient by local computations:
   \[
   \frac{\partial Q}{\partial w} = \frac{\partial Q}{\partial y} \frac{\partial y}{\partial w} = e \frac{\partial f_w(x)}{\partial w}
   \]
3. back-propagate the error signal:
   \[
   \frac{\partial Q}{\partial x} = \frac{\partial Q}{\partial y} \frac{\partial y}{\partial x} = e \frac{df_w(x)}{dx}
   \]
extend AD to a vector-input and vector-output module:

\[ f_w(x) \]

compute two Jacobian matrices:

\[
J_w = \begin{bmatrix}
\frac{\partial y_j}{\partial w_i} \\
\vdots \\
\frac{\partial y_j}{\partial w_k}
\end{bmatrix}_{k \times n}
\]

\[
J_x = \begin{bmatrix}
\frac{\partial y_j}{\partial x_i} \\
\vdots \\
\frac{\partial y_j}{\partial x_m}
\end{bmatrix}_{m \times n}
\]

given the error signal \( e = \frac{\partial Q}{\partial y} \) (\( \in \mathbb{R}^n \))

1. local gradients:

\[
\frac{\partial Q}{\partial w} = J_w e
\]

2. back-propagation:

\[
\frac{\partial Q}{\partial x} = J_x e
\]
Automatic Differentiation (III)

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

\[
y = Wx + b
\]

where \( W \in \mathbb{R}^{n \times d} \) and \( b \in \mathbb{R}^n \)

  - back-propagation:

\[
J_x = \begin{bmatrix}
\frac{\partial y_j}{\partial x_i}
\end{bmatrix}_{d \times n} = W^\top \implies \frac{\partial Q}{\partial x} = W^\top e
\]

  - local gradients:

\[
\frac{\partial Q}{\partial W} = \begin{bmatrix}
\frac{\partial Q}{\partial y_1} \\
\vdots \\
\frac{\partial Q}{\partial y_n}
\end{bmatrix} x^\top = e x^\top \\
\frac{\partial Q}{\partial b} = e
\]
Automatic Differentiation (III)

- **nonlinear activation** from \( x \in \mathbb{R}^n \) to \( y \in \mathbb{R}^n \):
  \[
  y = \phi(x)
  \]

- no learnable parameters \( \implies \) no local gradients

- back-propagation:
  \[
  \frac{\partial Q}{\partial x} = J_x e = \phi'(x) \odot e
  \]

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

- **softmax**: mapping an \(n\)-dimensional vector \(x \in \mathbb{R}^n\) into another \(n\)-dimensional vector \(y\) 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 \(\implies\) no local gradients

- the Jacobian matrix

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

- back-propagation:

\[
\frac{\partial Q}{\partial x} = J_x e
\]
Automatic Differentiation (V): Convolution (1)

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

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

- the Jacobian matrix \( J_x \):

\[
J_x = \left[ \frac{\partial y_j}{\partial x_i} \right]_{d \times n} = \begin{bmatrix}
w_1 & w_1 & \cdots & \cdots & \cdots & w_1 \\
w_2 & w_{f-1} & \cdots & \cdots & \cdots & w_2 \\
\vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\
w_f & w_{f-1} & \cdots & w_f & w_f & \cdots \\
w_f & w_{f-1} & \cdots & w_f & w_f & \cdots \\
\end{bmatrix}_{d \times n}
\]
Automatic Differentiation (V): Convolution (2)

- back-propagation by convolution:
  \[
  \frac{\partial Q}{\partial x} = J_x e = \begin{bmatrix}
  w_1 \frac{\partial Q}{\partial y_1} \\
  w_2 \frac{\partial Q}{\partial y_1} + w_1 \frac{\partial Q}{\partial y_2} \\
  \vdots \\
  w_f \frac{\partial Q}{\partial y_n}
\end{bmatrix}
\]
  \[\Delta = e^{(\emptyset)} \ast w\]

- computing local gradients by convolution:
  \[J_w = \left[ \frac{\partial y_j}{\partial w_i} \right]_{f \times n} = \begin{bmatrix}
  x_1 & x_2 & \cdots & x_n \\
  x_2 & x_3 & \cdots & x_{n+1} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_f & x_{f+1} & \cdots & x_{n+f-1}
\end{bmatrix}_{f \times n} \Rightarrow \frac{\partial Q}{\partial w} = J_w e = \Delta = x \ast e\]
Automatic Differentiation (V): Convolution (3)

- extend to 2D convolutions
- back-propagation by convolution:
  \[
  \frac{\partial Q}{\partial x_i} = \sum_{j=1}^{k} e_j(\emptyset) \ast \frac{\nabla Q}{\nabla w_{ij}} \quad (i = 1, 2 \cdots p)
  \]
- computing local gradient by convolution:
  \[
  \frac{\partial Q}{\partial w_{ij}} = x_i \ast e_j \quad (i = 1, 2 \cdots p; \quad j = 1, 2 \cdots k)
  \]

where \( x_i \in \mathbb{R}^{d \times d} \) and \( e_j \in \mathbb{R}^{n \times n} \)
Automatic Differentiation (VI)

- **max-pooling:**
  - no parameters $\implies$ 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:** $y = \text{BN}_{\gamma,\beta}(x)$
  - back-propagation:
    \[
    \frac{\partial Q}{\partial x^{(m)}} = \frac{M\gamma \odot e^{(m)} - \sum_{k=1}^{M} \gamma \odot e^{(k)} - \gamma \odot \hat{x}^{(m)} \odot \left(\sum_{k=1}^{M} e^{(k)} \odot \hat{x}^{(k)}\right)}{M \sqrt{\sigma_B^2(i) + \epsilon}}
    \]
  - local gradients:
    \[
    \frac{\partial Q}{\partial \gamma} = \sum_{k=1}^{M} \hat{x}^{(k)} \odot e^{(k)} \\
    \frac{\partial Q}{\partial \beta} = \sum_{k=1}^{M} e^{(k)}
    \]
Error Backpropagation Example: Fully-Connected DNNs

- all parameters:
  \[ W = \{ W^{(l)}, b^{(l)} \mid l = 1, 2 \cdots L \} \]

- the cross-entropy error:
  \[ Q(W; x) = -\ln [y]_r \quad \implies \]
  \[ \frac{\partial Q(W; x)}{\partial y} = \begin{bmatrix} 0 & \cdots & 0 & -\frac{1}{y_r} & 0 & \cdots & 0 \end{bmatrix}^T \]

- define error signals \( e^{(l)} = \frac{\partial Q(W; x)}{\partial a_l} \) 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 \((x, r)\)

1. for the output layer \(L\):
   \[
   e^{(L)} = [y_1 \quad y_2 \quad \cdots \quad y_r - 1 \quad \cdots \quad y_n]^T
   \]

2. for each hidden layer \(l = L - 1, \cdots, 2, 1\):
   \[
   e^{(l)} = \left( (W^{(l+1)})^T e^{(l+1)} \right) \odot H(z_l)
   \]

3. for all layers \(l = L, \cdots, 2, 1\):
   \[
   \frac{\partial Q(W;x)}{\partial W^{(l)}} = e^{(l)} (z_{l-1})^T
   \]
   \[
   \frac{\partial Q(W;x)}{\partial b^{(l)}} = e^{(l)}
   \]

where \(y\) and \(z_l\) \((l = 0, 1, \cdots, L - 1)\) are computed in the forward pass.
Heuristics and Tricks for Optimization

- Hyperparameters
- Optimization Method: ADAM
- Regularization
- Fine-tuning Tricks
Hyperparameters of Learning Neural Networks

- initial parameters
- epoch number
- mini-batch size
- learning rate
  - a good initial learning rate $\eta_0$
  - 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 $W^{(0)}$, and set $\eta$, $t = 0$, $n = 0$ and $u_0 = v_0 = 0$

while not converged do

randomly shuffle training data into mini-batches

for each mini-batch $B$ do

for each $x \in B$ do

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

end for

$g_n = \frac{1}{|B|} \sum_{x \in B} \frac{\partial Q(W^{(n)}; x)}{\partial W}$

$u_{n+1} = \alpha u_n + (1 - \alpha) g_n$ and $v_{n+1} = \beta v_n + (1 - \beta) g_n \odot g_n$

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

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

$n = n + 1$

end for

t = t + 1

end while
Self-adjusting Mechanism in ADAM

- use exponential average to accumulate 1st-order and 2nd-order moments ($u_n$ and $v_n$) of the gradient ($g_n$)
- normalize to yield unbiased estimates:
  \[
  \mathbb{E}[\hat{u}_{n+1}(i)] = \mathbb{E}[g_n(i)] \quad \mathbb{E}[\hat{v}_{n+1}(i)] = \mathbb{E}[g_n^2(i)]
  \]
- model update formula:
  \[
  W_i^{(n+1)} = W_i^{(n)} - \eta \frac{\hat{u}_{n+1}(i)}{\sqrt{\hat{v}_{n+1}(i)} + \epsilon^2}
  \]
- self-adjusting model updates $\Delta W_i^{(n)}$:
  \[
  \|\Delta W_i^{(n)}\|^2 \approx \eta^2 \frac{\mathbb{E}[\hat{u}_{n+1}(i)]^2}{\mathbb{E}[\hat{v}_{n+1}(i)]} = \frac{\eta^2 (\mathbb{E}[g_n(i)])^2}{(\mathbb{E}[g_n(i)])^2 + \text{var}[g_n(i)]}
  \]
Regularization in Neural Networks

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

\[
Q(\mathbf{W}) + \frac{\lambda}{2} \cdot \|\mathbf{W}\|^2 \implies \mathbf{W}^{(n+1)} = \mathbf{W}^{(n)} - \eta \frac{\partial Q(\mathbf{W}^{(n)})}{\partial \mathbf{W}} - \lambda \cdot \mathbf{W}^{(n)}
\]

- **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

![Diagram showing learning curves with different learning rates: very high, high, and low. Each curve represents a different learning rate, with the loss function decreasing over epochs.]
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.
encoder and decoder are powerful neural networks that can handle sequences, e.g. RNNs, LSTMs, or transformers