Machine Learning

Lecture 06: Deep Feedforward Networks

Nevin L. Zhang
lzhang@cse.ust.hk

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

This set of notes is based on various sources on the internet and Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT press. www.deeplearningbook.org

Introduction

- So far, probabilistic models for supervised learning

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow P(y|\mathbf{x})$$

Introduction

- So far, probabilistic models for supervised learning

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \to P(y|\mathbf{x})$$

- Next, **deep learning**:

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \to \mathbf{h} = f(\mathbf{x}), P(y|\mathbf{h})$$

Introduction

- So far, probabilistic models for supervised learning

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow P(y|\mathbf{x})$$

- Next, **deep learning**:

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow \mathbf{h} = f(\mathbf{x}), P(y|\mathbf{h})$$

  - $\mathbf{h} = f(\mathbf{x})$ is a feature transformation represented by a neural network,
  - $P(y|\mathbf{h})$ is a probabilistic model on the transformed features.

Introduction

- So far, probabilistic models for supervised learning

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow P(y|\mathbf{x})$$

- Next, **deep learning**:

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow \mathbf{h} = f(\mathbf{x}), P(y|\mathbf{h})$$

  - $\mathbf{h} = f(\mathbf{x})$ is a feature transformation represented by a neural network,
  - $P(y|\mathbf{h})$ is a probabilistic model on the transformed features.
  - Regarded as one whole model: $P(y|\mathbf{x})$.

Introduction

- So far, probabilistic models for supervised learning

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow P(y|\mathbf{x})$$

- Next, **deep learning**:

$$\{\mathbf{x}_i, y_i\}_{i=1}^N \rightarrow \mathbf{h} = f(\mathbf{x}), P(y|\mathbf{h})$$

  - $\mathbf{h} = f(\mathbf{x})$ is a feature transformation represented by a neural network,
  - $P(y|\mathbf{h})$ is a probabilistic model on the transformed features.
  - Regarded as one whole model: $P(y|\mathbf{x})$.

- This lecture: $\mathbf{h} = f(\mathbf{x})$ as a **feedforward neural network (FNN)**.

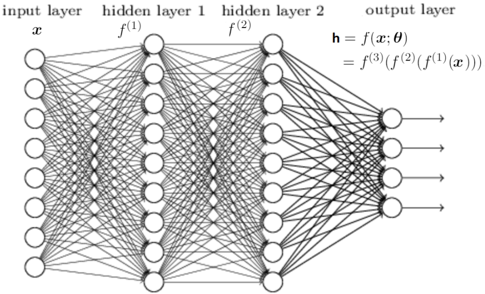
- Next lecture: $\mathbf{h} = f(\mathbf{x})$ as a **convolutional neural network (CNN or ConvNet)**.

Outline

Deep Feedforward Networks

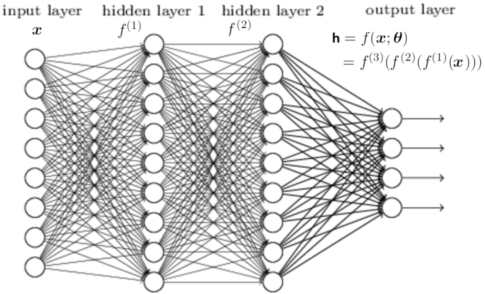
- **Deep feedforward networks**, also often called **feedforward neural networks (FNNs)**,or **multilayer perceptrons(MLPs)**, are the quintessential deep learning models
- A feedforward network defines a function $\mathbf{h} = f(\mathbf{x}, \boldsymbol{\theta})$.
- During learning, the parameters $\boldsymbol{\theta}$ are optimized so that $f(\mathbf{x}, \boldsymbol{\theta})$ approximates some target function $f^*(\mathbf{x})$

Feedforward Neural Networks



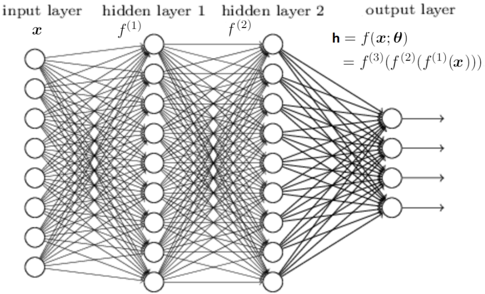
- Networks of simple computing elements (**units, neurons**).

Feedforward Neural Networks



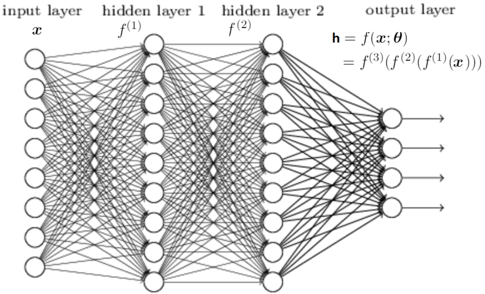
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer

Feedforward Neural Networks



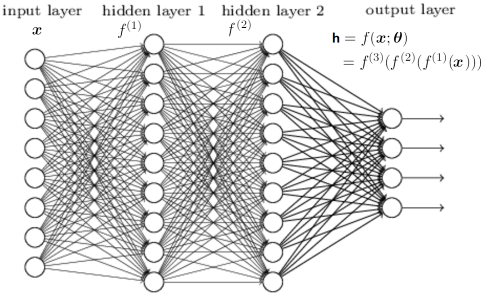
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer
  - Parameters include a **bias** for each unit and a **weight** for each link.

Feedforward Neural Networks



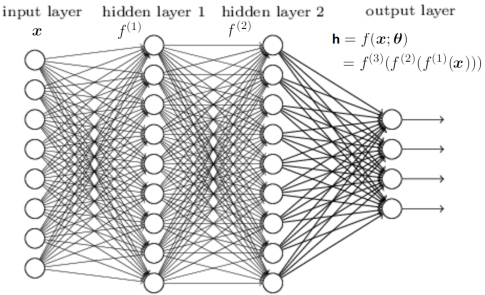
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer
  - Parameters include a **bias** for each unit and a **weight** for each link.

Feedforward Neural Networks



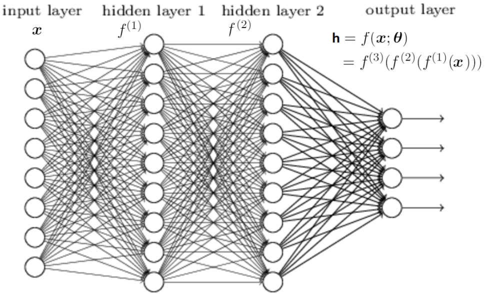
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer
  - Parameters include a **bias** for each unit and a **weight** for each link.
  - Units are divided into: **input units**, **hidden units**, and **output units**

Feedforward Neural Networks



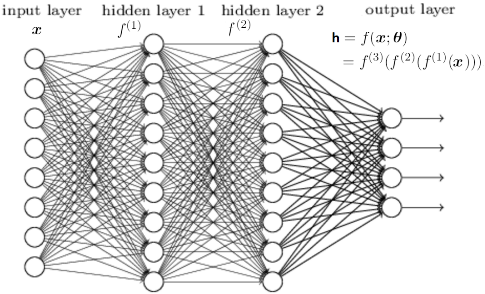
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer
  - Parameters include a **bias** for each unit and a **weight** for each link.
  - Units are divided into: **input units**, **hidden units**, and **output units**
- Feedforward networks:
  - Inputs enter the input units,

Feedforward Neural Networks



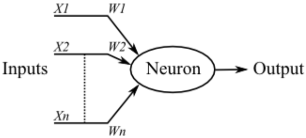
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer
  - Parameters include a **bias** for each unit and a **weight** for each link.
  - Units are divided into: **input units**, **hidden units**, and **output units**
- Feedforward networks:
  - Inputs enter the input units,
  - Propagate through the network to the output units.

Feedforward Neural Networks



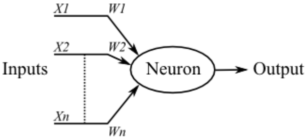
- Networks of simple computing elements (**units, neurons**).
  - Each unit is connected to units on the previous layer and units on the next layer
  - Parameters include a **bias** for each unit and a **weight** for each link.
  - Units are divided into: **input units**, **hidden units**, and **output units**
- Feedforward networks:
  - Inputs enter the input units,
  - Propagate through the network to the output units.

The Units



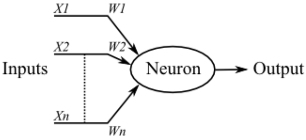
- A unit accepts vector of inputs $\mathbf{x}$,
- computes an affine transformation $z = \mathbf{W}^\top \mathbf{x} + b$, where $\mathbf{W} = (W_1, W_2, \ldots, W_n)^\top$ are the **link weights** and b is the **bias** of the unit.

The Units



- A unit accepts vector of inputs $\mathbf{x}$,
- computes an affine transformation $z = \mathbf{W}^\top \mathbf{x} + b$, where $\mathbf{W} = (W_1, W_2, \ldots, W_n)^\top$ are the **link weights** and b is the **bias** of the unit. z is sometimes called the **net input** of the unit.

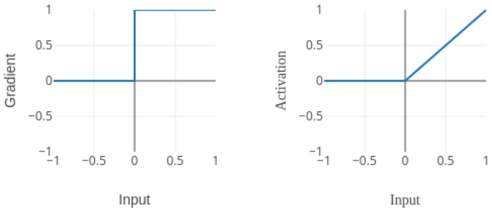
The Units



- A unit accepts vector of inputs $\mathbf{x}$,
- computes an affine transformation $z = \mathbf{W}^\top \mathbf{x} + b$, where $\mathbf{W} = (W_1, W_2, \ldots, W_n)^\top$ are the **link weights** and b is the **bias** of the unit. z is sometimes called the **net input** of the unit.
- applies nonlinear **activation function** $g(z)$, and
- gives output $g(z) = g(\mathbf{W}^\top \mathbf{x} + b)$

Different types of units have different activation functions.
Initialize all weights to small random values, and biases to zero or to small positive values.

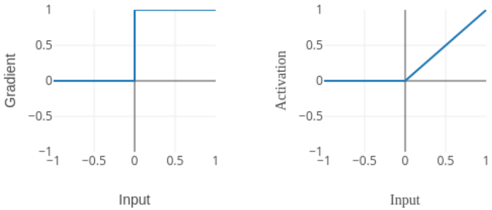
Rectified Linear Units (ReLU)



General form of activation function: $g(z) = g(\mathbf{W}^\top \mathbf{x} + b)$

- **ReLU**: $g(z) = \max\{0, z\}$.
- Constant gradient when $z > 0$, which leads to faster learning.

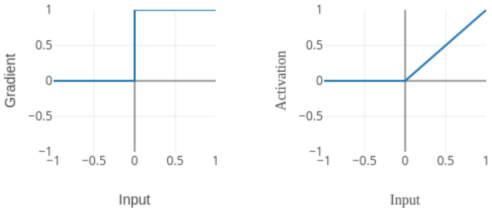
Rectified Linear Units (ReLU)



General form of activation function: $g(z) = g(\mathbf{W}^\top \mathbf{x} + b)$

- **ReLU**: $g(z) = \max\{0, z\}$.
- Constant gradient when $z > 0$, which leads to faster learning. Probably the most commonly used activation function.
- Zero gradient when $z < 0$.

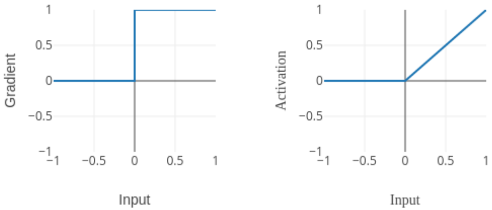
Rectified Linear Units (ReLU)



General form of activation function: $g(z) = g(\mathbf{W}^\top \mathbf{x} + b)$

- **ReLU**: $g(z) = \max\{0, z\}$.
- Constant gradient when $z > 0$, which leads to faster learning. Probably the most commonly used activation function.
- Zero gradient when $z < 0$. The neuron is **dead** if $z < 0$ for all training examples.

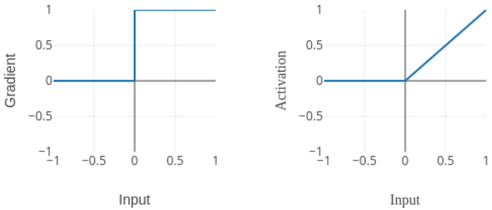
Rectified Linear Units (ReLU)



General form of activation function: $g(z) = g(\mathbf{W}^\top \mathbf{x} + b)$

- **ReLU**: $g(z) = \max\{0, z\}$.
- Constant gradient when $z > 0$, which leads to faster learning. Probably the most commonly used activation function.
- Zero gradient when $z < 0$. The neuron is **dead** if $z < 0$ for all training examples. Dead neurons cannot be revived.

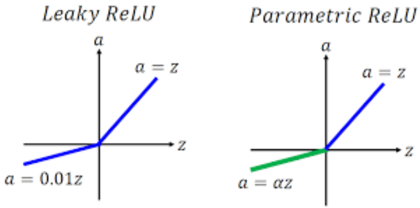
Rectified Linear Units (ReLU)



General form of activation function: $g(z) = g(\mathbf{W}^\top \mathbf{x} + b)$

- **ReLU**: $g(z) = \max\{0, z\}$.
- Constant gradient when $z > 0$, which leads to faster learning. Probably the most commonly used activation function.
- Zero gradient when $z < 0$. The neuron is **dead** if $z < 0$ for all training examples. Dead neurons cannot be revived.
  - To mitigate the problem somewhat, initialize b to be small positive value, e.g. 0.1, so that the unit is initially active ($z > 0$).

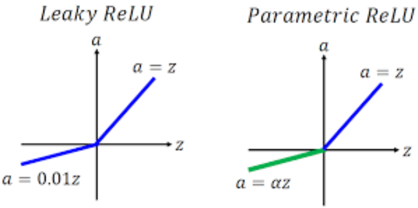
Variations of ReLU



$g(z, \alpha) = \max\{0, z\} + \alpha \min\{0, z\}$:

- **Absolute value rectification**: $\alpha = -1$,
- **Leaky ReLU**: α is small value like 0.01,
- **Parametric ReLU**: Learns α

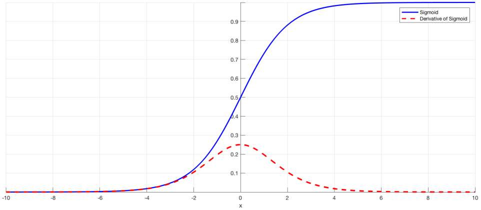
Variations of ReLU



$g(z, \alpha) = \max\{0, z\} + \alpha \min\{0, z\}$:

- **Absolute value rectification**: $\alpha = -1$,
- **Leaky ReLU**: α is small value like 0.01,
- **Parametric ReLU**: Learns α
- Non-zero gradient even when $z < 0$, which mitigates the dead neuron problem.

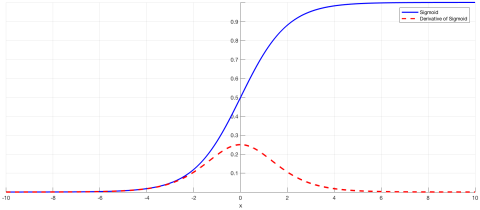
Sigmoid



- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

Sigmoid

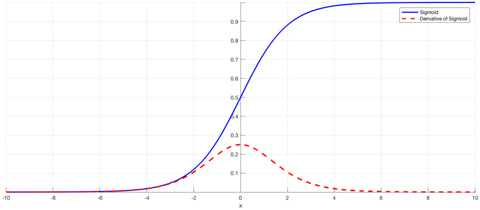


- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Sigmoidal unit has small gradients across most of its domain (**vanishing gradient**),

Sigmoid

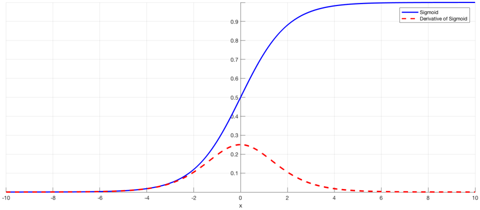


- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Sigmoidal unit has small gradients across most of its domain (**vanishing gradient**), and hence leads to slow learning,

Sigmoid

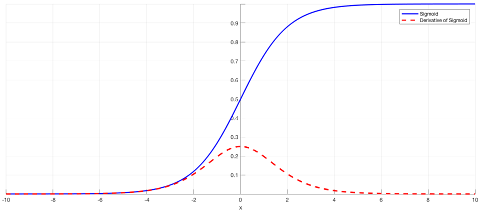


- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Sigmoidal unit has small gradients across most of its domain (**vanishing gradient**), and hence leads to slow learning, especially in deep models.

Sigmoid

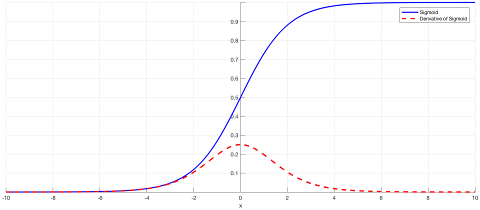


- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Sigmoidal unit has small gradients across most of its domain (**vanishing gradient**), and hence leads to slow learning, especially in deep models.
    - It is said to saturate easily, where saturation refers to the state in which a neuron predominantly outputs values close to the asymptotic ends of the bounded activation function.

Sigmoid

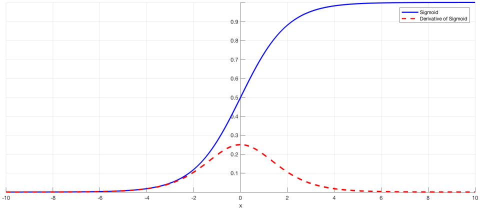


- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Sigmoidal unit has small gradients across most of its domain (**vanishing gradient**), and hence leads to slow learning, especially in deep models.
  - It is said to saturate easily, where saturation refers to the state in which a neuron predominantly outputs values close to the asymptotic ends of the bounded activation function. Saturation implies slow learning.

Sigmoid

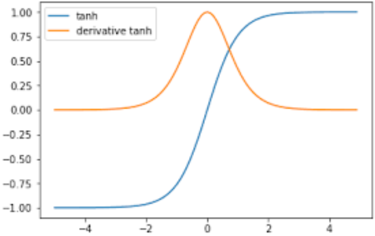


- **Sigmoid activation function**:

$$g(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Sigmoidal unit has small gradients across most of its domain (**vanishing gradient**), and hence leads to slow learning, especially in deep models.
  - It is said to saturate easily, where saturation refers to the state in which a neuron predominantly outputs values close to the asymptotic ends of the bounded activation function. Saturation implies slow learning.
- Not recommended as internal unit.

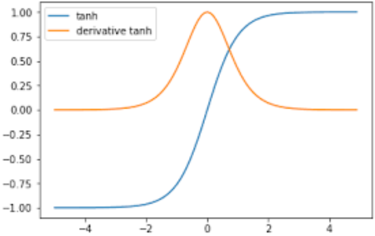
Hyperbolic Tagent



- **Hyperbolic tagent activation function**:

$$g(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

Hyperbolic Tagent

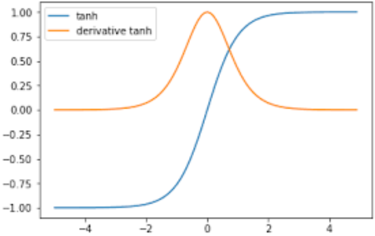


- **Hyperbolic tagent activation function**:

$$g(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

- Tanh has larger gradients than sigmoid, and smoother than ReLU.

Hyperbolic Tagent

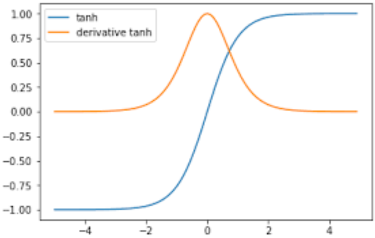


- **Hyperbolic tagent activation function**:

$$g(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

- Tanh has larger gradients than sigmoid, and smoother than ReLU. However, it can still saturate.

Hyperbolic Tagent

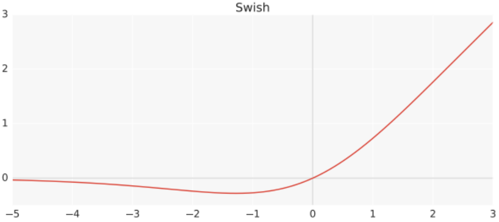


- **Hyperbolic tagent activation function**:

$$g(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

- Tanh has larger gradients than sigmoid, and smoother than ReLU. However, it can still saturate.
- A popular choice in practice.

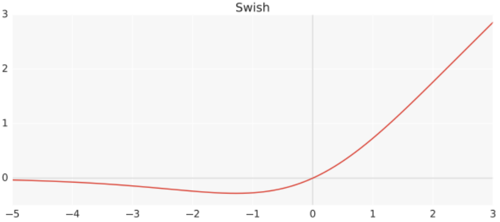
Swish



- **Swish activation** was proposed in 2017:

$$g(z) = z\sigma(\beta z), \ \beta \text{ is parameter fixed or learned}$$

Swish

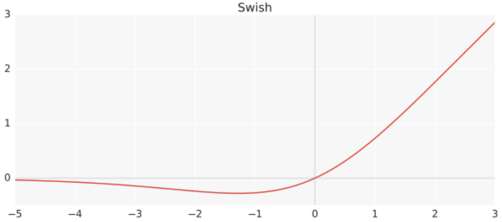


- **Swish activation** was proposed in 2017:

$$g(z) = z\sigma(\beta z), \ \beta \text{ is parameter fixed or learned}$$

- Non-zero gradients when $z < 0$,

Swish

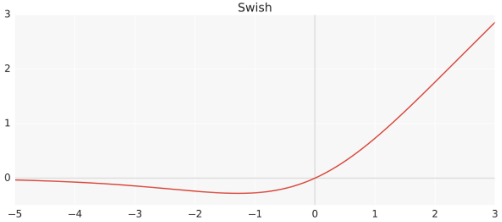


- **Swish activation** was proposed in 2017:

$$g(z) = z\sigma(\beta z), \ \beta \text{ is parameter fixed or learned}$$

- Non-zero gradients when $z < 0$, smooth and non-monotonic.

Swish

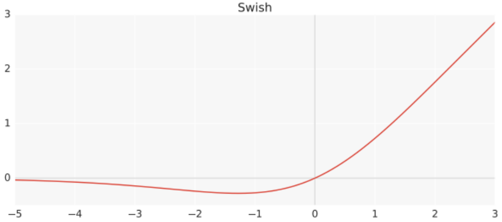


- **Swish activation** was proposed in 2017:

$$g(z) = z\sigma(\beta z), \ \beta \text{ is parameter fixed or learned}$$

- Non-zero gradients when $z < 0$, smooth and non-monotonic.
- Unbounded above, which avoids saturation.

Swish

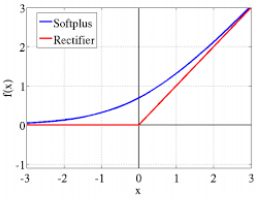


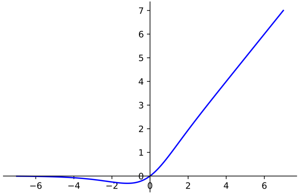
- **Swish activation** was proposed in 2017:

$$g(z) = z\sigma(\beta z), \;\; \beta \text{ is parameter fixed or learned}$$

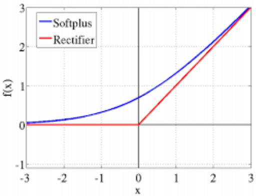
- Non-zero gradients when $z < 0$, smooth and non-monotonic.
- Unbounded above, which avoids saturation.
- Outperforms ReLU in deep networks.

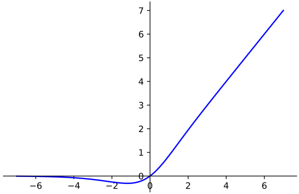
Softplus and Mish





Softplus and Mish





- **Softplus activation function**:

$$g(z) = \zeta(z) = \log(1 + \exp(z))$$

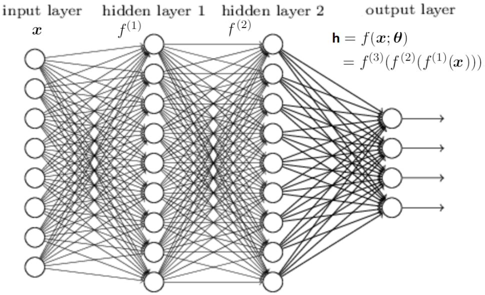
  - Softplus is a smooth version of ReLU, empirically not as good as ReLU.

- **Mish activation function**:

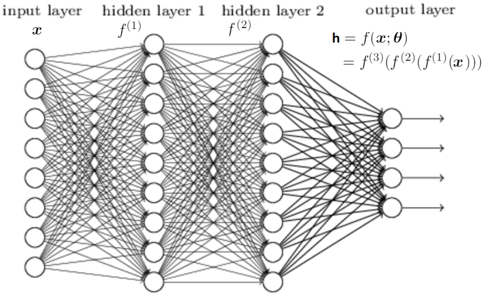
$$g(z) = z \tanh(\log(1 + \exp(z)) = z \tanh(\zeta(z))$$

  - Recently proposed.Similar to Swish, and better.

Computation by Feedfoward Neural Network



Computation by Feedfoward Neural Network



- $\mathbf{h}^{(i)}$ — Column vector for units on layer i; $\mathbf{b}^{(i)}$ — Biases for units on layer i; $g^{(i)}$ — Activation functions for units on layer i.
- $\mathbf{W}^{(i)}$ — Matrix of weights for units on layer i, with weight for unit j at the j-th column.

$$\begin{aligned}
\mathbf{h}^{(1)} &= g^{(1)}(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)}) \\
\mathbf{h}^{(2)} &= g^{(2)}(\mathbf{W}^{(2)\top}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \\
\mathbf{h} &= g^{(3)}(\mathbf{W}^{(3)\top}\mathbf{h}^{(2)} + \mathbf{b}^{(3)})
\end{aligned}$$

Universal Approximation

Theorem

Only one layer of sigmoid hidden units suffices to approximate any well-behaved function (e.g., bounded continuous functions) to arbitrary precision.

Universal Approximation

Theorem

Only one layer of sigmoid hidden units suffices to approximate any well-behaved function (e.g., bounded continuous functions) to arbitrary precision.

Deep learning useful when you:

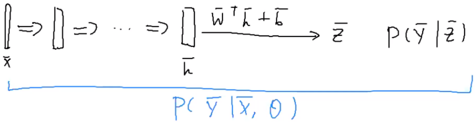
- Need a complex function, and
- Have abundant data.

Outline

An FNN can be used to define a probabilistic model

$$\underbrace{\underset{\bar{x}}{[\;]} \Rightarrow [\;] \Rightarrow \cdots \Rightarrow \underset{\bar{h}}{[\;]} \xrightarrow{\bar{W}^T \bar{h} + \bar{b}} \bar{z} \qquad p(\bar{y} \mid \bar{z})}_{P(\bar{y} \mid \bar{x}, \theta)}$$

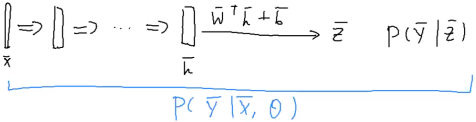
An FNN can be used to define a probabilistic model



- The first-second last layers define a feature transformation:

$$\mathbf{h} = f(\mathbf{x})$$

An FNN can be used to define a probabilistic model



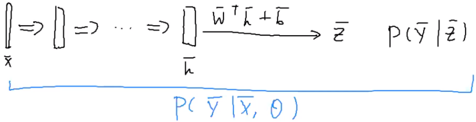
- The first-second last layers define a feature transformation:

$$\mathbf{h} = f(\mathbf{x})$$

- The last layer defines a probabilistic model on the features:

$$P(\mathbf{y}|\mathbf{h})$$

An FNN can be used to define a probabilistic model



- The first-second last layers define a feature transformation:

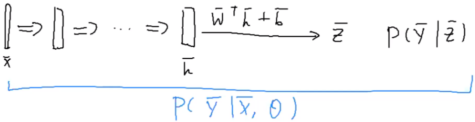
$$\mathbf{h} = f(\mathbf{x})$$

- The last layer defines a probabilistic model on the features:

$$P(\mathbf{y}|\mathbf{h})$$

Note: Here $\mathbf{y}$ is a vector of output variables, while so far we have been talking about only one output variable y.

An FNN can be used to define a probabilistic model



- The first-second last layers define a feature transformation:

$$\mathbf{h} = f(\mathbf{x})$$

- The last layer defines a probabilistic model on the features:

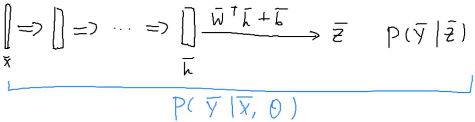
$$P(\mathbf{y}|\mathbf{h})$$

  Note: Here $\mathbf{y}$ is a vector of output variables, while so far we have been talking about only one output variable y.

- The whole network defines a probabilistic model:

$$P(\mathbf{y}|\mathbf{x}, \theta)$$

An FNN can be used to define a probabilistic model



- The first-second last layers define a feature transformation:

$$\mathbf{h} = f(\mathbf{x})$$

- The last layer defines a probabilistic model on the features:

$$P(\mathbf{y}|\mathbf{h})$$

  Note: Here $\mathbf{y}$ is a vector of output variables, while so far we have been talking about only one output variable y.

- The whole network defines a probabilistic model:

$$P(\mathbf{y}|\mathbf{x}, \theta),$$

  where θ consists of weights in f and parameters in $P(\mathbf{y}|\mathbf{h})$.

Logits

- To define a probabilistic model on the features $\mathbf{h}$, first define a **logit vector**

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b},$$

where $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector.

Logits

- To define a probabilistic model on the features $\mathbf{h}$, first define a **logit vector**

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b},$$

where $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector. They are the parameters of the last layer.

Logits

- To define a probabilistic model on the features **h**, first define a **logit vector**

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b},$$

  where **W** is a weight matrix and **b** is a bias vector. They are the parameters of the last layer. $\mathbf{z} = (z_1, z_2, \ldots)^\top$.

- Then, we can define various probabilistic models using **z**,

Logits

- To define a probabilistic model on the features **h**, first define a **logit vector**

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b},$$

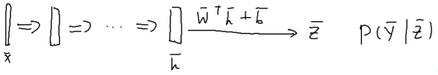
  where **W** is a weight matrix and **b** is a bias vector. They are the parameters of the last layer. $\mathbf{z} = (z_1, z_2, \ldots)^\top$.

- Then, we can define various probabilistic models using **z**, which are viewed as units for the last layer.

Linear-Gaussian Output Unit

$$\bar{x} \Rightarrow [\;] \Rightarrow \cdots \Rightarrow \bar{h} \xrightarrow{\bar{W}^T \bar{h} + \bar{b}} \bar{z} \qquad p(\bar{y} \mid \bar{z})$$

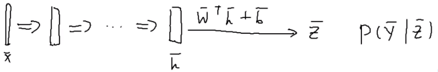
Linear-Gaussian Output Unit



- When $\mathbf{y}$ is real-valued vector, we can assume that $\mathbf{y}$ follows a Gaussian distribution with mean $\mathbf{z}$ and identity covariance matrix $\mathbf{I}$:

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{z}, \mathbf{I})$$

Linear-Gaussian Output Unit



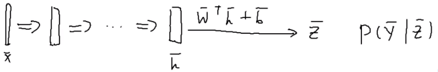
- When **y** is real-valued vector, we can assume that **y** follows a Gaussian distribution with mean **z** and identity covariance matrix **I**:

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{z}, \mathbf{I})$$

- In this case, the **per-sample loss** is:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -\log P(\mathbf{y}|\mathbf{x}, \theta) \propto \frac{1}{2}||\mathbf{y} - \mathbf{z}||_2^2 \qquad \text{(See p12 of L02)}$$

Linear-Gaussian Output Unit



- When **y** is real-valued vector, we can assume that **y** follows a Gaussian distribution with mean **z** and identity covariance matrix **I**:

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{z}, \mathbf{I})$$

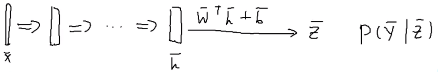
- In this case, the **per-sample loss** is:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -\log P(\mathbf{y}|\mathbf{x}, \theta) \propto \frac{1}{2}||\mathbf{y} - \mathbf{z}||_2^2 \qquad \text{(See p12 of L02)}$$

- Partial derivative of per-sample loss with respect to logit z_k:

$$\frac{\partial L}{\partial z_k} = z_k - y_k.$$

Linear-Gaussian Output Unit



- When **y** is real-valued vector, we can assume that **y** follows a Gaussian distribution with mean **z** and identity covariance matrix **I**:

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{z}, \mathbf{I})$$

- In this case, the **per-sample loss** is:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -\log P(\mathbf{y}|\mathbf{x}, \theta) \propto \frac{1}{2}||\mathbf{y} - \mathbf{z}||_2^2 \qquad \text{(See p12 of L02)}$$

- Partial derivative of per-sample loss with respect to logit z_k:

$$\frac{\partial L}{\partial z_k} = z_k - y_k.$$

It is error.

Sigmoid Output Unit

- When there is only one binary output variable $y \in \{0, 1\}$, we can define a distribution $p(y|\mathbf{x})$ using a sigmoid unit:

$$P(y|\mathbf{x}) = Ber(y|\sigma(z)),$$

where z is a scalar.

Sigmoid Output Unit

- When there is only one binary output variable $y \in \{0, 1\}$, we can define a distribution $p(y|\mathbf{x})$ using a sigmoid unit:

$$P(y|\mathbf{x}) = Ber(y|\sigma(z)),$$

  where z is a scalar.

- In this case, the **per-sample loss** is:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -[y \log \sigma(z) + (1 - y) \log(1 - \sigma(z))] \qquad \text{(See p10 of L03)}$$

Sigmoid Output Unit

- When there is only one binary output variable $y \in \{0, 1\}$, we can define a distribution $p(y|\mathbf{x})$ using a sigmoid unit:

$$P(y|\mathbf{x}) = Ber(y|\sigma(z)),$$

where z is a scalar.

- In this case, the **per-sample loss** is:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -[y \log \sigma(z) + (1-y) \log(1 - \sigma(z))] \qquad \text{(See p10 of L03)}$$

- Partial derivative of per-sample loss with respect to logit z:

$$\frac{\partial L}{\partial z} = \sigma(z) - y. \qquad \text{(See p21 of L03)}$$

Sigmoid Output Unit

- When there is only one binary output variable $y \in \{0, 1\}$, we can define a distribution $p(y|\mathbf{x})$ using a sigmoid unit:

$$P(y|\mathbf{x}) = Ber(y|\sigma(z)),$$

  where z is a scalar.

- In this case, the **per-sample loss** is:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -[y \log \sigma(z) + (1 - y) \log(1 - \sigma(z))] \qquad \text{(See p10 of L03)}$$

- Partial derivative of per-sample loss with respect to logit z:

$$\frac{\partial L}{\partial z} = \sigma(z) - y. \qquad \text{(See p21 of L03)}$$

  It is error.

Sigmoid Output Unit

- We said earlier that sigmoid units are not recommended for hidden units because they saturates across most of their domains.
- They are fine as output units because the negative log-likelihood in the cost function helps to avoid the problem.
- In fact, $\sigma(z) - y \approx 0$ means:
    - $z \gg 0, y = 1$, or
    - $z \ll 0, y = 0$

Sigmoid Output Unit

- We said earlier that sigmoid units are not recommended for hidden units because they saturates across most of their domains.
- They are fine as output units because the negative log-likelihood in the cost function helps to avoid the problem.
- In fact, $\sigma(z) - y \approx 0$ means:
  - $z \gg 0, y = 1$, or
  - $z \ll 0, y = 0$

  In words, saturation occurs only when the model already has the right answer: When $y = 1$ and z is very positive, or $y = 0$ and z is very negative.

Softmax Output Unit

- When y is a variable with C possible values $\{1, 2, \ldots, C\}$, we can define a distribution $p(y|\mathbf{x})$

Softmax Output Unit

- When y is a variable with C possible values $\{1, 2, \ldots, C\}$, we can define a distribution $p(y|\mathbf{x})$ using a **softmax unit**.

Softmax Output Unit

- When y is a variable with C possible values $\{1, 2, \ldots, C\}$, we can define a distribution $p(y|\mathbf{x})$ using a **softmax unit**.
- Here $\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$ is a vector of n components: $\mathbf{z} = (z_1, z_2, \ldots, z_C)^\top$.
- The follow formula gives a distribution over the domain of y:

$$P(y = k|\mathbf{x}) = \frac{\exp(z_k)}{\sum_{j=1}^{C} \exp(z_j)}$$

Softmax Output Unit

- When y is a variable with C possible values $\{1, 2, \ldots, C\}$, we can define a distribution $p(y|\mathbf{x})$ using a **softmax unit**.
- Here $\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$ is a vector of n components: $\mathbf{z} = (z_1, z_2, \ldots, z_C)^\top$.
- The follow formula gives a distribution over the domain of y:

$$P(y = k|\mathbf{x}) = \frac{\exp(z_k)}{\sum_{j=1}^{C} \exp(z_j)}$$

- Partial derivative of per-sample cross entroyp loss with respect to logit z_k:

$$\frac{\partial L}{\partial z_k} = P(y = k) - \mathbf{1}(y = k). \qquad \text{(See p40 of L03)}$$

Softmax Output Unit

- When y is a variable with C possible values $\{1, 2, \ldots, C\}$, we can define a distribution $p(y|\mathbf{x})$ using a **softmax unit**.
- Here $\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$ is a vector of n components: $\mathbf{z} = (z_1, z_2, \ldots, z_C)^\top$.
- The follow formula gives a distribution over the domain of y:

$$P(y = k|\mathbf{x}) = \frac{\exp(z_k)}{\sum_{j=1}^{C} \exp(z_j)}$$

- Partial derivative of per-sample cross entroyp loss with respect to logit z_k:

$$\frac{\partial L}{\partial z_k} = P(y = k) - \mathbf{1}(y = k). \qquad \text{(See p40 of L03)}$$

It is error.

Outline

Training Feedforward Neural Networks

- Given data $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$, we want to learn the parameters θ by minimizing the cross-entropy loss:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}_i, y_i, \theta) = \frac{1}{N}\sum_{i=1}^{N}(-\log P(\mathbf{y}_i|\mathbf{x}_i, \theta))$$

Training Feedforward Neural Networks

- Given data $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$, we want to learn the parameters θ by minimizing the cross-entropy loss:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}_i, y_i, \theta) = \frac{1}{N}\sum_{i=1}^{N}(-\log P(\mathbf{y}_i|\mathbf{x}_i, \theta))$$

- Algorithm: Stochastic gradient descent (SGD)
  - Initialize θ
  - Repeat for a predetermined number of epochs
    - For each minibatch B,

$$\theta \leftarrow \theta - \alpha \frac{1}{|B|}\sum_{i \in B} \nabla L(\mathbf{x}_i, y_i, \theta)$$

Training Feedforward Neural Networks

- Given data $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$, we want to learn the parameters θ by minimizing the cross-entropy loss:

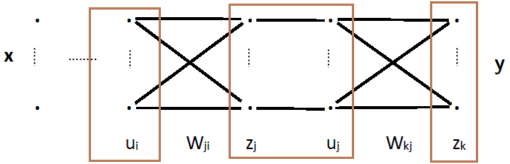
$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}_i, y_i, \theta) = \frac{1}{N}\sum_{i=1}^{N}(-\log P(\mathbf{y}_i|\mathbf{x}_i, \theta))$$

- Algorithm: Stochastic gradient descent (SGD)
  - Initialize θ
  - Repeat for a predetermined number of epochs
    - For each minibatch B,

$$\theta \leftarrow \theta - \alpha \frac{1}{|B|}\sum_{i\in B}\nabla L(\mathbf{x}_i, y_i, \theta)$$

- Question: How to compute the gradient $\nabla L(\mathbf{x}_i, y_i, \theta)$ for each training example?
- Answer: Backpropagation.

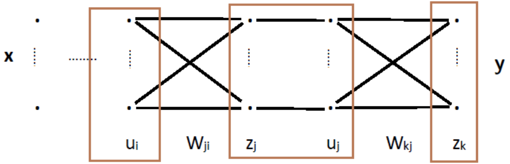
Backpropagation: Output Layer



- Let y_k, u_j and u_i each be the output of a unit at the last three layers, and z_k, z_j, z_i be the output before applying the activation function:

$$z_k = \sum_j u_j W_{kj}, \quad u_j = g(z_j), \quad z_j = \sum_i u_i W_{ji}$$

Backpropagation: Output Layer

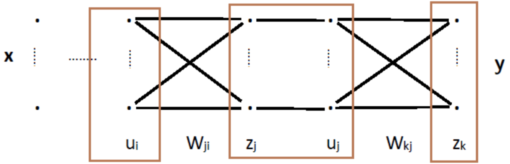


- Let y_k, u_j and u_i each be the output of a unit at the last three layers, and z_k, z_j, z_i be the output before applying the activation function:

$$z_k = \sum_j u_j W_{kj}, \quad u_j = g(z_j), \quad z_j = \sum_i u_i W_{ji}$$

(Bias ignored for simplicity.)

Backpropagation: Output Layer



- Let y_k, u_j and u_i each be the output of a unit at the last three layers, and z_k, z_j, z_i be the output before applying the activation function:

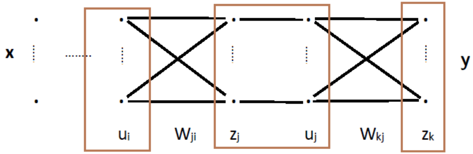
$$z_k = \sum_j u_j W_{kj}, \quad u_j = g(z_j), \quad z_j = \sum_i u_i W_{ji}$$

  (Bias ignored for simplicity.)

- Let $L = L(\mathbf{x}, \mathbf{y}, \theta)$ for a particular training example $(\mathbf{x}, \mathbf{y})$. The partial derivative of L w.r.t a weigh in the last layer:

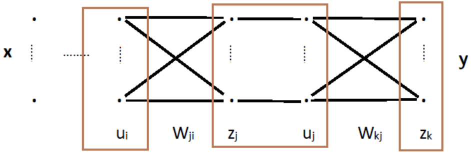
$$\frac{\partial L}{\partial W_{kj}} = \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial W_{kj}} = u_j \delta_k, \text{ where } \delta_k = \frac{\partial L}{\partial z_k}$$

Backpropagation: Hidden Layer



$$z_k = \sum_j u_j W_{kj}, \quad u_j = g(z_j), \quad z_j = \sum_i u_i W_{ji}$$

Backpropagation: Hidden Layer



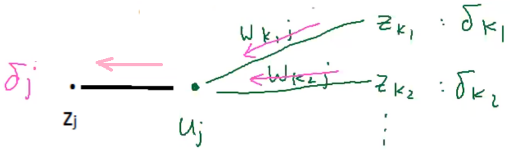
$$z_k = \sum_j u_j W_{kj}, \quad u_j = g(z_j), \quad z_j = \sum_i u_i W_{ji}$$

- The partial derivative of L w.r.t a weigh in the second last layer (similar for weight in other hidden layers):

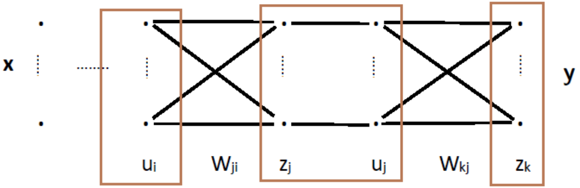
$$\begin{aligned}\frac{\partial L}{\partial W_{ji}} &= \sum_k \frac{\partial L}{\partial z_k}\frac{\partial z_k}{\partial W_{ji}} = \sum_k \delta_k \frac{\partial z_k}{\partial u_j}\frac{\partial u_j}{\partial z_j}\frac{\partial z_j}{\partial W_{ji}} \\ &= \sum_k \delta_k W_{kj}\frac{\partial u_j}{\partial z_j} u_i = u_i \delta_j, \text{ where } \delta_j = \frac{\partial u_j}{\partial z_j}\sum_k W_{kj}\delta_k\end{aligned}$$

Backpropagation of error

$$\delta_j = \frac{\partial u_j}{\partial z_j} \sum_k W_{kj} \delta_k$$



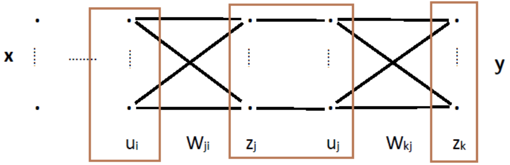
Backpropagation



Summary:

- For a weight W_{kj} the output layer: $\frac{\partial L}{\partial W_{kj}} = u_j \delta_k$, where $\delta_k = \frac{\partial L}{\partial z_k}$
  - u_j is the output of unit j from input $\mathbf{x}$
  - δ_k is the "error value" for unit k obtained by comparing the model output $\mathbf{z} = f(\mathbf{x})$ and the observed output $\mathbf{y}$ in L.
- For a weight W_{ji} in a hidden layer: $\frac{\partial L}{\partial W_{ji}} = u_i \delta_j$, where $\delta_j = \frac{\partial u_j}{\partial z_j} \sum_k W_{kj} \delta_k$
  - u_i is the output of unit i from input $\mathbf{x}$
  - δ_j is the "error value" for unit j obtained by backpropagating the errors (δ_k) from the next layer .

The Backpropagation Algorithm



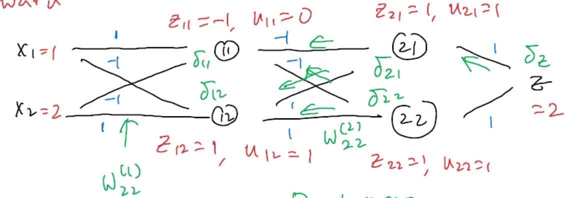
- Objective: Compute $\frac{\partial L}{\partial W_{ji}}$ for all weights (and biases) for each training example:

1. **Propagate forward** to compute the net input and output of each unit (i.e, u_i, u_j, $\ldots$).
2. **Propagate "error" backward**
   - For each output unit k: $\delta_k \leftarrow \frac{\partial L}{\partial z_k}$
   - For each hidden unit j: $\delta_j \leftarrow \frac{\partial u_j}{\partial z_j} \sum_k W_{kj}\delta_k$
3. Get the gradient for each weight: $\frac{\partial L}{\partial W_{ji}} \leftarrow u_i\delta_j$, $\frac{\partial L}{\partial b_j} \leftarrow \delta_j$

Backprop Example

Backprop Example

Forward



hidden unit: ReLU

$$P(Y|z) = \begin{cases} \sigma(z) & Y=1 \\ 1-\sigma(z) & Y=0 \end{cases}$$

Example

x_1	x_2	Y
1	2	0

$$\delta_{11} = 0$$

Back prop

$$\delta_{21} = \frac{\partial u_{21}}{\partial z_{21}} \, 1 \cdot \delta_z = 0.88$$

$$\delta_z = \sigma(2) - 0 \approx 0.88$$

$$\delta_{12} = \frac{\partial u_{12}}{\partial z_{12}} \cdot (1 \cdot \delta_{21} + 1 \cdot \delta_{22}) = 1.76$$

$$\delta_{22} = 0.88$$

$$\frac{\partial L}{\partial W^{(1)}_{22}} = 2 \times 1.76$$

$$\frac{\partial L}{\partial W^{(2)}_{22}} = 1 \times 0.88$$

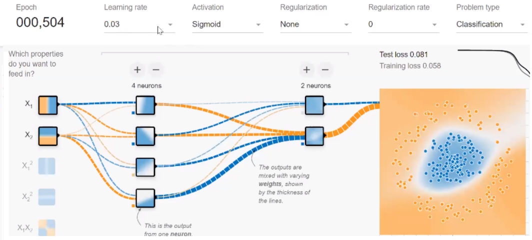
The Backpropagation Algorithm

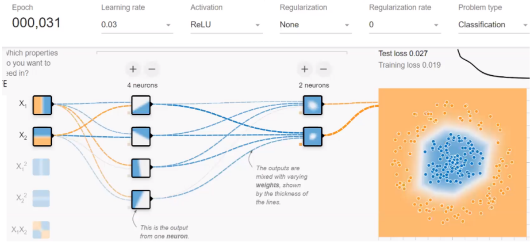
- The backpropagation algorithm and SGD are implemented in deep learning packages such as Tensorflow.
- Weights are packed into **tensors** ($[a_{ijk}]$) for efficiency
  - A 1-D tensor is a vector
  - A 2-D tensor is a matrix
  - A 3-D tensor is several matrices stacked on top of each other.
- Although you do not need to know everything about backpropagation and SGD, some basic understanding will be very helpful when you work with deep learning.

Demonstrations `http://playground.tensorflow.org/`

Demonstrations http://playground.tensorflow.org/

- ReLU units are easier to learn than Sigmoid and Tanh units

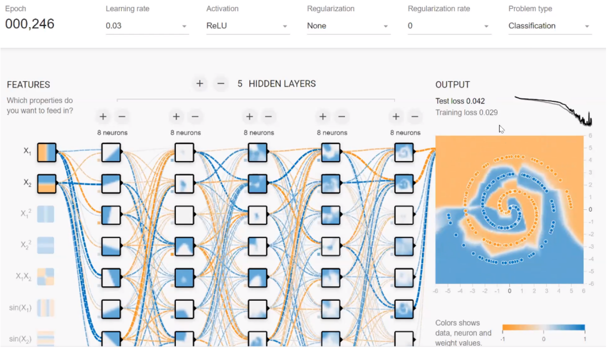




Demonstrations `http://playground.tensorflow.org/`

Demonstrations http://playground.tensorflow.org/

- Deep structure helps



- Need to adjust learning rate during training.

Outline

Introduction

- DL models are complex. Overfitting is an important issue.

Introduction

- DL models are complex. Overfitting is an important issue.
- One way to do that is to regularize (weight decay):

$$\tilde{J}(\theta) = J(\theta) + \lambda||\theta||_2^2$$

Introduction

- DL models are complex. Overfitting is an important issue.
- One way to do that is to regularize (weight decay):

$$\tilde{J}(\theta) = J(\theta) + \lambda ||\theta||_2^2$$

Almost always do this.

Introduction

- DL models are complex. Overfitting is an important issue.
- One way to do that is to regularize (weight decay):

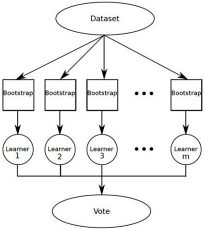
$$\tilde{J}(\theta) = J(\theta) + \lambda ||\theta||_2^2$$

  Almost always do this.

- Dropout is another technique for avoiding overfitting specifically proposed for deep learning.

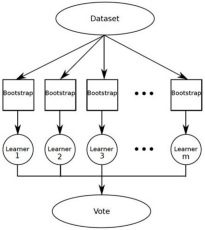
Bagging (Bootstrap Aggregating)

- A technique for reducing generalization error by combining several models
  - Train several different models separately on different randomly selected subsets of data called **bootstrap samples**,
  - Classification: Have all of the models vote on the output for test examples.



Bagging (Bootstrap Aggregating)

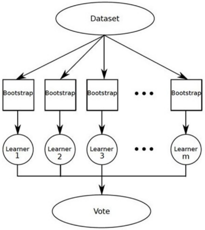
- A technique for reducing generalization error by combining several models
  - Train several different models separately on different randomly selected subsets of data called **bootstrap samples**,
  - Classification: Have all of the models vote on the output for test examples.



- Bagging reduces variance:

Bagging (Bootstrap Aggregating)

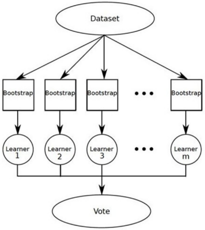
- A technique for reducing generalization error by combining several models
  - Train several different models separately on different randomly selected subsets of data called **bootstrap samples**,
  - Classification: Have all of the models vote on the output for test examples.



- Bagging reduces variance: Variance is error due to randomness.

Bagging (Bootstrap Aggregating)

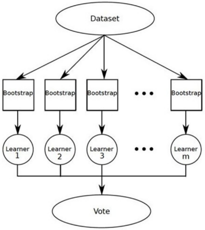
- A technique for reducing generalization error by combining several models
  - Train several different models separately on different randomly selected subsets of data called **bootstrap samples**,
  - Classification: Have all of the models vote on the output for test examples.



- Bagging reduces variance: Variance is error due to randomness. Different bootstrap samples contain different randomness, and hence unlikely to **most** make the same errors on the test set.

Bagging (Bootstrap Aggregating)

- A technique for reducing generalization error by combining several models
  - Train several different models separately on different randomly selected subsets of data called **bootstrap samples**,
  - Classification: Have all of the models vote on the output for test examples.



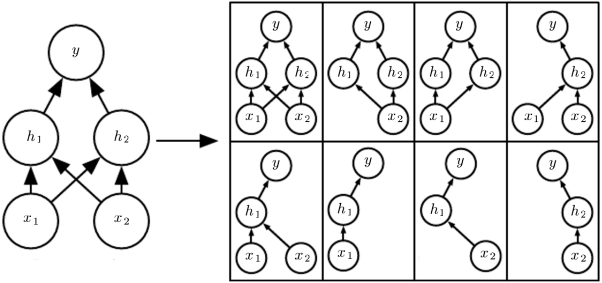
- Bagging reduces variance: Variance is error due to randomness. Different bootstrap samples contain different randomness, and hence unlikely to **most** make the same errors on the test set.

One Way to Use Bagging for Feedforward Networks

- Start with a large network.
- Obtain subnetworks and train them separately on different randomly selected subsets of data.
- To classify future examples, let the subnetworks vote.

One Way to Use Bagging for Feedforward Networks

- Start with a large network.
- Obtain subnetworks and train them separately on different randomly selected subsets of data.
- To classify future examples, let the subnetworks vote.
- **Problem**: Too expensive.
- **Dropout** is an inexpensive approximation of the idea.

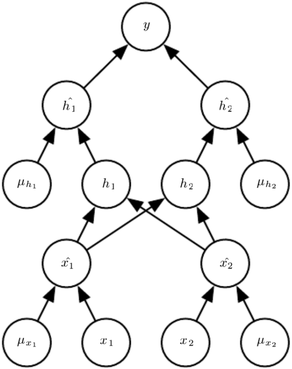


Base network

Ensemble of subnetworks

Subnetwork Selection via Masking

- Associate a **binary mask variable** with each input and hidden unit.
- Sample their values randomly and independently (e.g., with probability 0.5 for hidden units and 0.8 for input units).
- Multiply the output of each unit by its mask value before passing it to the next layer.
- Units with 0 mask values are effectively removed from the network.



Dropout

- To be used with minibatch-based algorithms such as SGD.
- For each minibatch,
  - Randomly sample values for mask variables
  - Carry out one step of gradient descent.
    - (Only parameters for the units with mask value 1 are updated. The others have gradient zero and hence not updated.)
- There is only **one network**, no subnetworks. At each step, training is conducted in a part of the network (a subnetwork).

Dropout

- To be used with minibatch-based algorithms such as SGD.
- For each minibatch,
    - Randomly sample values for mask variables
    - Carry out one step of gradient descent.
        - (Only parameters for the units with mask value 1 are updated. The others have gradient zero and hence not updated.)
- There is only **one network**, no subnetworks. At each step, training is conducted in a part of the network (a subnetwork).
- Dropout is a widely used regularization method for deep learning.

Dropout

- To be used with minibatch-based algorithms such as SGD.
- For each minibatch,
  - Randomly sample values for mask variables
  - Carry out one step of gradient descent.
    - (Only parameters for the units with mask value 1 are updated. The others have gradient zero and hence not updated.)
- There is only **one network**, no subnetworks. At each step, training is conducted in a part of the network (a subnetwork).
- Dropout is a widely used regularization method for deep learning. It reduces overfitting by preventing complex co-adaptations of parameters on training data.

Dropout

- To be used with minibatch-based algorithms such as SGD.
- For each minibatch,
  - Randomly sample values for mask variables
  - Carry out one step of gradient descent.
    - (Only parameters for the units with mask value 1 are updated. The others have gradient zero and hence not updated.)
- There is only **one network**, no subnetworks. At each step, training is conducted in a part of the network (a subnetwork).
- Dropout is a widely used regularization method for deep learning. It reduces overfitting by preventing complex co-adaptations of parameters on training data.

See example in CNN code.

Outline

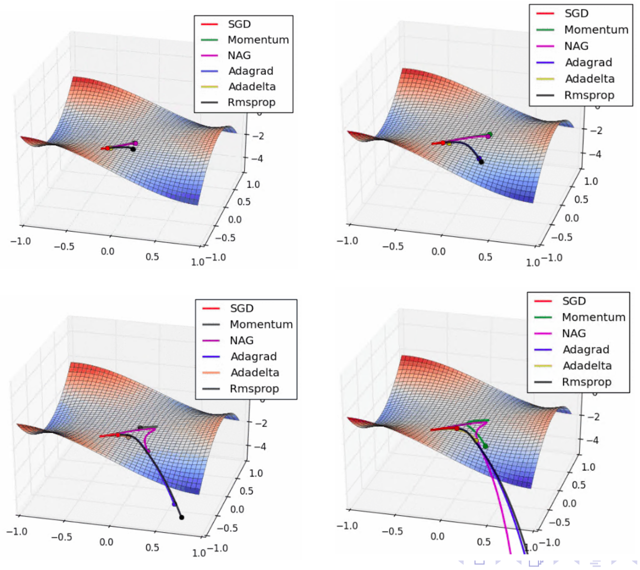
- Our task is: Find θ to minimize the loss function $J(\theta)$.
- There are multiple algorithms that we can use. Stochastic gradient descent is one of them.

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while

- There are others ....

Performances of Different Optimizers Vary a Lot



Momentum

Momentum

SGD

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$

SGD with Momentum

Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

Momentum

SGD

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$

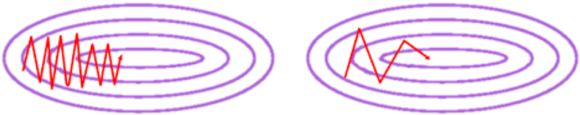
SGD with Momentum

Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

- **Momentum** is a technique to accelerate SGD.
- View θ as a "particle" that travels in the space of parameter values.
- In SGD, its movement is determined by gradient and learning rate.
- In SGD with momentum, the movement is additionally affected by its **velocity** v
  - v is an exponentially decaying average of past negative gradients.
  - The hyperparameter α is usually set at 0.5, 0.9 or 0.99.

Momentum

- The incorporation of momentum into stochastic gradient descent reduces the variation in overall gradient directions and speeds up learning.



Online demo: https://medium.com/@ramrajchandradevan/the-evolution-of-gradient-descend-optimization-algorithm-4106a6702d39

Algorithms with Adaptive Learning Rates

- Learning rate should be reduced gradually.
- Several common algorithms:
    - AdaGrad
    - RMSProp
    - Adam
- Key idea: Parameters that have changed a lot should be allowed to change less in future.

AdaGrad (Adaptive Gradient Algorithm)

AdaGrad (Adaptive Gradient Algorithm)

- AdaGrad scales the learning rates of all model parameters by inversely proportional to the square root of the sum of all of their historical squared gradients.
- The net effect is greater progress in the more gently sloped directions of parameter space.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

RMSProp (Root Mean Square Propagation)

RMSProp (Root Mean Square Propagation)

- RMSProp uses an exponentially decaying average to discard history from the extreme past

---

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while

AdaGrad vs RMSProp

AdaGrad vs RMSProp

$$\bar{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \end{bmatrix} \qquad \bar{g} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \end{bmatrix}$$

"sum of past changes": $\bar{r} = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \end{bmatrix}$

AdaGrad

$$\bar{r} \leftarrow \bar{r} + \bar{g} \odot \bar{g}$$

t	r_i	
1	g_1^2	g_{i1}
2	$g_1^2 + g_2^2$	
3	$g_1^2 + g_2^2 + g_3^2$	

No decaying

RMSProp

$$\bar{r} \leftarrow \rho \bar{r} + (1-\rho)\, \bar{g} \odot \bar{g} \qquad 0 < \rho < 1$$

t	r_i	
1	$(1-\rho) g_1^2$	(g_{i1})
2	$\rho(1-\rho) g_1^2 + (1-\rho) g_2^2$ $= (1-\rho)(\rho g_1^2 + g_2^2)$	
3	$(1-\rho)(\underline{\rho^2} g_1^2 + \underline{\rho} g_2^2 + g_3)$	

Exponentially decaying weights

Adam (Adaptive Moments)

- Roughly a combination of RMSProp and momentum, with bias correction.
- "Insofar, Adam might be the best overall choice."

Momentum

$$\bar{v} \leftarrow \alpha \bar{v} - \epsilon \bar{g} \qquad \Delta\bar{\theta} = \bar{v}$$

$$\bar{\mu} = -\bar{v}$$

$$\bar{\mu} \leftarrow \alpha \bar{\mu} + \epsilon \bar{g} \qquad \Delta\bar{\theta} = -\bar{\mu}$$

RMS Prop

$$\bar{r} \leftarrow \rho \bar{r} + (1-\rho)\, \bar{g} \odot \bar{g}$$

$$\Delta\theta = -\frac{\epsilon}{\delta + \sqrt{r}} \odot \bar{g}$$

Adam

$$\bar{s} \leftarrow \rho_1 \bar{s} + (1-\rho_1)\bar{g} \qquad \bar{r} \leftarrow \rho_2 \bar{r} + (1-\rho_2)\, \bar{g} \odot \bar{g}$$

Bias correction

t: iteration number

$$\bar{s} \leftarrow \frac{\bar{s}}{1-\rho_1^t} \qquad \bar{r} \leftarrow \frac{\bar{r}}{1-\rho_2^t}$$

$$\Delta\bar{\theta} = -\frac{\epsilon}{\delta + \sqrt{\bar{r}}} \odot \bar{s}$$

Adam: Bias Correction

Adam: Bias Correction

$$\bar{r} \leftarrow \rho_2 \bar{r} + (1-\rho_2)\, \bar{g} \odot \bar{g} \qquad \hat{r} \leftarrow \frac{r}{1-\rho_2^t}$$

t	$\bar{r}$
3	$(1-\rho_2)(\rho_2^2 g_1^2 + \rho_2 g_2^2 + g_3^2)$
t	$(1-\rho_2)(\rho_2^{t-1} g_1^2 + \rho_2^{t-2} g_2^2 + \dots + g_t^2)$

r_t

$$\hat{r}_t = \frac{r_t}{1-\rho_2^t} = \frac{\cancel{(1-\rho_2)}(\rho_2^{t-1} g_1^2 + \rho_2^{t-2} g_2^2 + \dots + g_t^2)}{\cancel{(1-\rho_2)}(\rho_2^{t-1} + \rho_2^{t-2} + \dots + 1)}$$

$$= \frac{\rho_2^{t-1}}{Z} g_1^2 + \frac{\rho_2^{t-2}}{Z} g_2^2 + \dots + \frac{1}{Z} g_t^2 \qquad Z$$

Expectation

$\int x^2 p(x)\, dx$

2nd order moment

Prob distribution

Adam (Adaptive Moments)

Adam (Adaptive Moments)

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$ Momentum
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$ RMSProp
    <u>Correct bias</u> in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    <u>Correct bias</u> in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while

Adam (Adaptive Moments)

Results of different optimizers