CS 784: Computational Linguistics
Lecture 8: Neural Networks I
(for Text Classification)

Freda Shi

School of Computer Science, University of Waterloo
fhs@uwaterloo.ca

January 30, 2025

Check out https://pytorch.org/tutorials/intermediate/nlp_from_scratch_index.html if you aren't familiar with this topic!

Recap: A Unified View of Classification

$$\text{classify}(s) = \arg\max_{y} \mathit{score}(s, y; \boldsymbol{\Theta})$$

s: input text, y: class label, $\boldsymbol{\Theta}$: model parameters.

Modeling: define $\mathit{score}(s, y; \boldsymbol{\Theta})$.

Training: learn $\boldsymbol{\Theta}$ to maximize the likelihood of the training data.

Inference: find the best class label y for a given input text s.

This lecture and the next: score$(\cdot)$ with artificial neural networks.

Overview of This Lecture (and the Next)

Basics: Optimization

Basics: Perceptrons and multi-layer perceptrons (MLPs)

Convolutional neural networks (CNNs)

Recurrent and recursive neural networks (RNNs/RvNNs)

Attention

Transformers

Recall: Logistic Regression with Gradient Descent

$$\arg\min_{\mathbf{w}} \text{loss}\left(\mathbf{w}; D = \{\mathbf{x}_i, y_i\}_{i=1}^N\right)$$

For logistic regression,

$$\text{loss}(\mathbf{w}; D) = -\sum_{i=1}^{N} y_i \log \sigma\left(\mathbf{w}^T \mathbf{x}_i\right) - (1 - y_i) \log\left(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)\right)$$

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

Gradient descent: update $\mathbf{w}$ in the opposite direction of the gradient of the loss function, i.e., $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D)$.

$$\nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D) = \left[\frac{\partial \text{loss}(\mathbf{w}; D)}{\partial w_1}, \ldots, \frac{\partial \text{loss}(\mathbf{w}; D)}{\partial w_n}\right]$$

Applying the Chain Rule

$$\text{loss}(\mathbf{w}; D) = \sum_{i=1}^{N} y_i \log \sigma\left(\mathbf{w}^T \mathbf{x}_i\right) + (1 - y_i) \log\left(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)\right)$$

Let $z_i(\mathbf{w}; \mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i = \sum_j w_j x_{ij}$.

$$\text{loss}(\mathbf{w}; D) = \text{loss}(\mathbf{z}; \mathbf{y}) = \sum_{i=1}^{N} y_i \log \sigma(z_i) + (1 - y_i) \log\left(1 - \sigma(z_i)\right)$$

Apply the chain rule:

$$\frac{\partial \text{loss}}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial \text{loss}}{\partial z_i} \frac{\partial z_i}{\partial w_j}$$

D_i: the dataset that only consists of the i-th training example.

Stochastic Gradient Descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D)$$

$$\nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D) = \sum_{i=1}^{N} \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D_i)$$

Efficiency concerns: for each single update, we need to compute the gradient over the entire dataset.

Solution (**stochastic gradient descent**): use gradients computed over a small subset (also referred to as **mini-batch**) of examples.

$$\nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D) \approx \frac{N}{B} \sum_{i=1}^{B} \nabla_{\mathbf{w}} \text{loss}(\mathbf{w}; D_i)$$

Stochastic Gradient Descent: The Idea

Consider we are estimating the gradient over the entire N examples.

The quantity we aim to estimate is:

$$\begin{aligned}\nabla_{\mathbf{w}}\text{loss}(\mathbf{w}; D) &= \sum_{i=1}^{N} \nabla_{\mathbf{w}}\text{loss}(\mathbf{w}; D_i) \\ &= N \cdot \mathbb{E}_{D_i \sim D} \nabla_{\mathbf{w}}\text{loss}(\mathbf{w}; D_i)\end{aligned}$$

The expectation can be estimated by sampling $B (B << N)$ examples from D:

$$\mathbb{E}_{D_i \sim D} \nabla_{\mathbf{w}}\text{loss}(\mathbf{w}; D_i) \approx \frac{1}{B} \sum_{i=1}^{B} \nabla_{\mathbf{w}}\text{loss}(\mathbf{w}; D_i)$$

Optimizing Neural Networks

Theoretically, gradient based optimization only guarantees convergence to the global minimum for convex functions; for non-convex functions, it may converge to a local minimum.

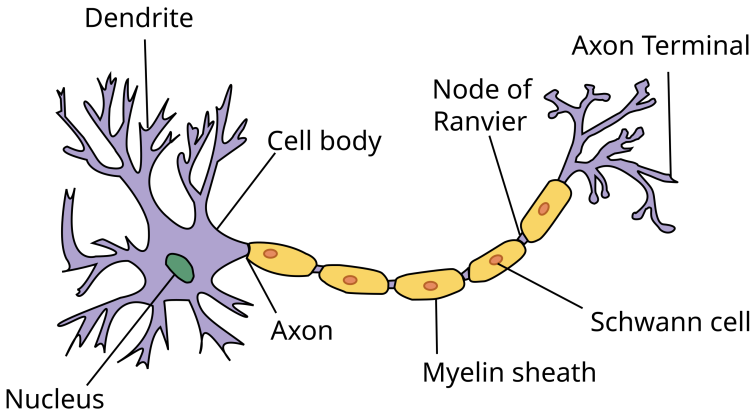
In (almost) every complicated real-world problems with neural network models, the loss function is non-convex.

However, empirical evidence suggests that gradient-based optimization works well in practice.

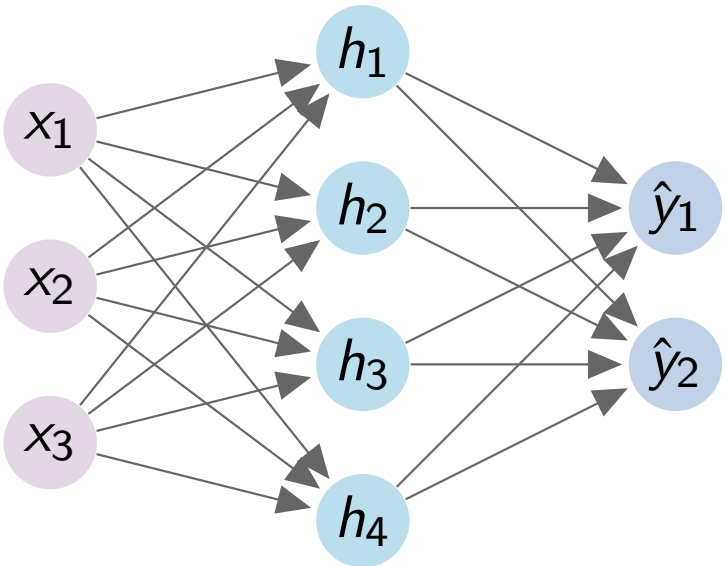
Adam (Kingma & Ba, 2015; momentum + adaptive learning rate) and AdamW (Loshchilov & Hutter, 2019; Adam + automatic weight decay) are popular choices now to optimize the loss function of neural networks.

`https://pytorch.org/docs/stable/optim.html#algorithms`

From Biological Neurons to Artificial Neurons



[Source: Wikipedia]



First computational model of a neuron (McCulloch & Pitts, 1943):

$$g(\mathbf{x}) = \sum_{i=1}^{n} x_i \qquad \widehat{y} = f_\theta(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{x}$: input vector, θ: threshold.

What is an artificial neural network?

An artificial neural network (ANN) is a function.

In machine learning context, the term **neural network** refers to artificial neural networks.

Two (not necessarily exclusive) views:

- View 1 (computer scientists): The idea of neural modeling is now better thought of as **dense representation learning**, although the design of ANNs was inspired by biological neurons.
- View 2 (neural scientists): The design of ANNs was inspired by biological neurons, so the study of biological neurons **may** draw insights from artificial neural network behaviors.

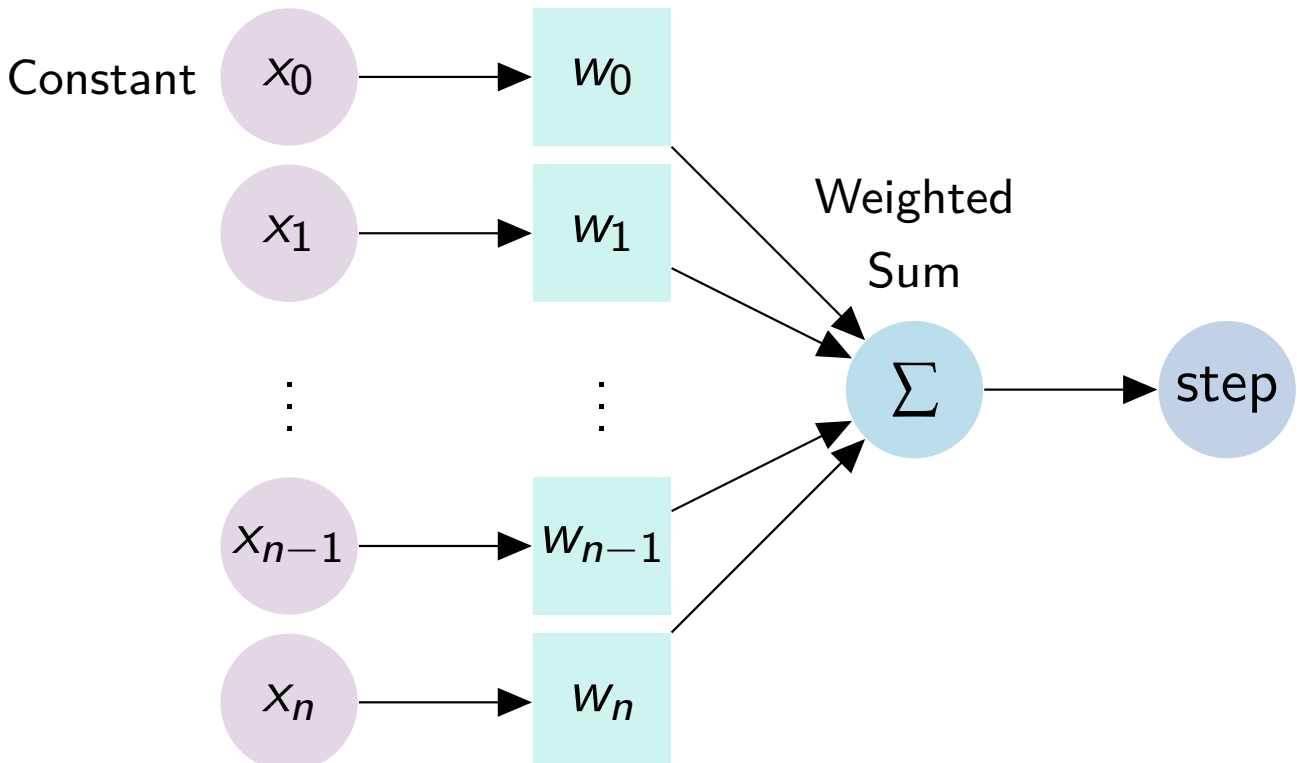
Notations

- $\mathbf{u}, \mathbf{v}$: vectors
- u_i, v_i: entry i in the vector
- $\mathbf{W}$: a matrix
- W_{ij}: entry (i, j) in the matrix
- (Future lectures) $\mathcal{Y}$: a structured object (e.g., a sequence, a tree, etc.)
- (Future lectures) y_i: entry i in the structured object

Perceptron (Minsky & Papert, 1969)

$$\text{Perceptron}(\mathbf{x}; \mathbf{w}, b) = \text{step}(\mathbf{w}^T\mathbf{x} + b), \quad \text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{o.w.} \end{cases}$$

Can be written as $\text{step}(\mathbf{w}^T\mathbf{x})$ if one entry of $\mathbf{x}$ is always constant.



Learning Perceptrons

$$\text{perceptron}(\mathbf{x}) = \text{step}(\mathbf{w}^T\mathbf{x})$$

For each training example $(\mathbf{x}_i, y_i)$:

$$\begin{aligned} \text{Predict the label :} \quad & \widehat{y}_i = \text{perceptron}(\mathbf{x}_i) \\ \text{Update the weights :} \quad & \mathbf{w} = \mathbf{w} + \eta(y_i - \widehat{y}_i)\mathbf{x}_i \end{aligned}$$

η: learning rate, y_i: ground-truth label, $\widehat{y}_i$: predicted label.

- A perceptron is a binary classifier.
- $\mathbf{w}$: weights that define canonical positive class, and $-\mathbf{w}$ is the canonical negative class.
- If the prediction is incorrect, adjust $\mathbf{w}$.

Perceptron Update as Stochastic Gradient Descent

$$y_i = \text{perceptron}(\mathbf{x}_i) = \text{step}(\mathbf{w}^T \mathbf{x}_i)$$
$$\mathbf{w} = \mathbf{w} + \eta \left(y_i - \widehat{y}_i\right) \mathbf{x}_i$$

If we set the loss function for example i as

$$\text{loss}(\mathbf{w}; \mathbf{x}_i, y_i) = \left(\widehat{y}_i - y_i\right) \mathbf{w}^T \mathbf{x}_i$$

and **view $\widehat{y}_i$ as a constant**, then the perceptron update rule is equivalent to stochastic gradient descent on the loss function.

$\widehat{y}_i$	y_i	$\text{loss}(\mathbf{w}; \mathbf{x}_i, y_i)$
0	0	0
1	1	0
0	1	+
1	0	+

Neural Layer: Generalized Perceptron

From a machine-learning perspective,
a neural layer = affine transformation + nonlinearity.

$$\text{perceptron}(\mathbf{x}) = \text{step}(\mathbf{w}^T\mathbf{x} + b) \in \{0, 1\}$$

$$\text{neural-layer}(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^{d_{\text{out}}}$$
$$\text{neural-layer}(\mathbf{x})_i = g(\mathbf{w}_i\mathbf{x} + b_i) \in \mathbb{R}$$

g: (nonlinear) activation function.

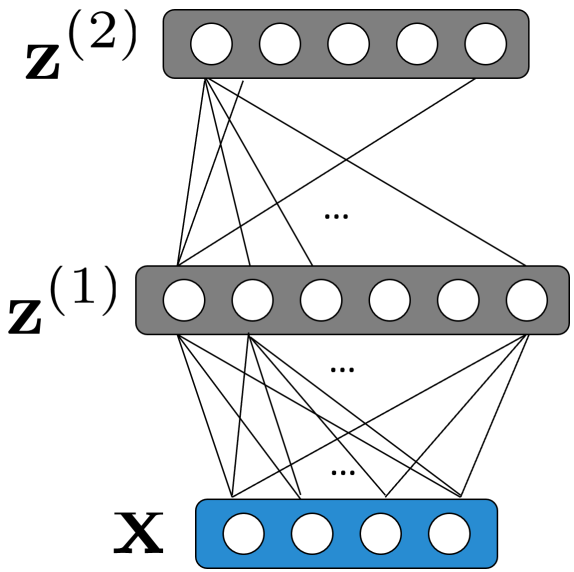
$\mathbf{W} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, $\mathbf{b} \in \mathbb{R}^{d_{\text{out}}}$: parameter of the affine transformation.

$\mathbf{w}_i$: i-th row vector of $\mathbf{W}$.

d_{in}: input dimension, d_{out}: output dimension.

Stacking Neural Layers

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$
$$\mathbf{z}^{(2)} = g\left(\mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}\right)$$
$$\dots$$



- Use the output of one layer as the input of the next layer.
- Feed-Forward Neural Network
- Fully Connected Neural Network
- Multi-Layer Perceptron (MLP)

Nonlinearities: Activation Functions

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

- g is applied to each entry of the vector in an element-wise manner.
- Common activation functions: sigmoid, tanh, ReLU, Leaky ReLU, etc.

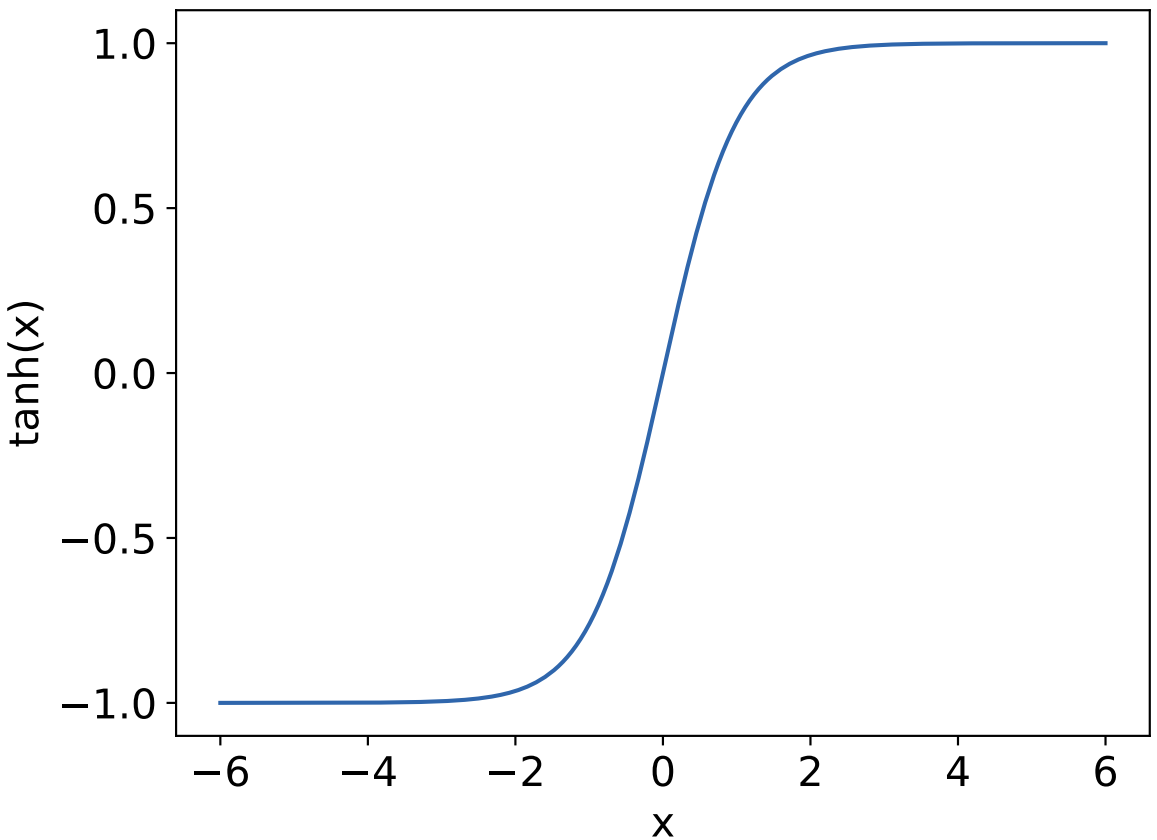
Why do we need nonlinearity?

Without nonlinearity, the composition of multiple layers of affine transformations is still an affine transformation.

See also the Pytorch documentation.

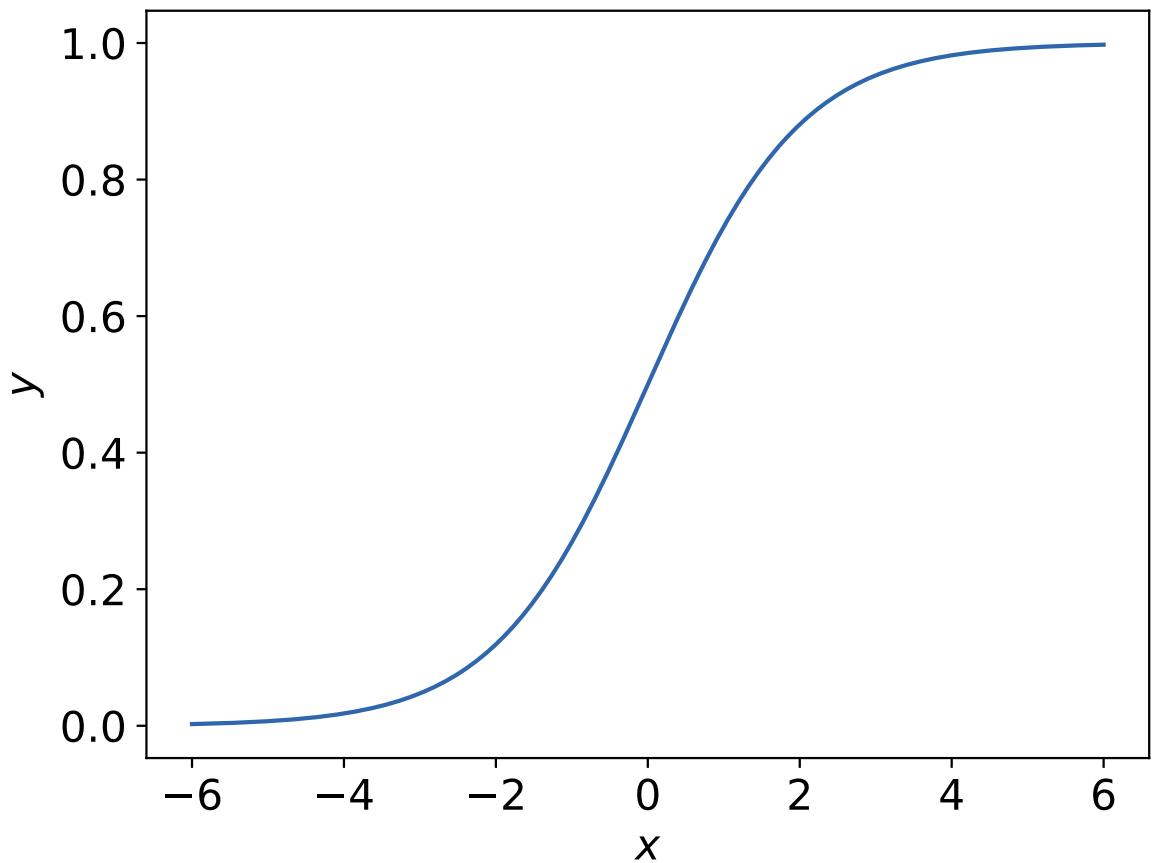
Activation Function: Tanh

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



Activation Function: Sigmoid

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$



Tanh and Sigmoid

$$
\begin{aligned}
\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
&= \frac{1 - e^{-2x}}{1 + e^{-2x}} && \text{divide by } e^x \textit{ onbothsides} \\
&= \frac{1}{1 + e^{-2x}} - \frac{e^{-2x}}{1 + e^{-2x}} \\
&= \sigma(2x) - (1 - \sigma(2x)) \\
&= 2\sigma(2x) - 1
\end{aligned}
$$

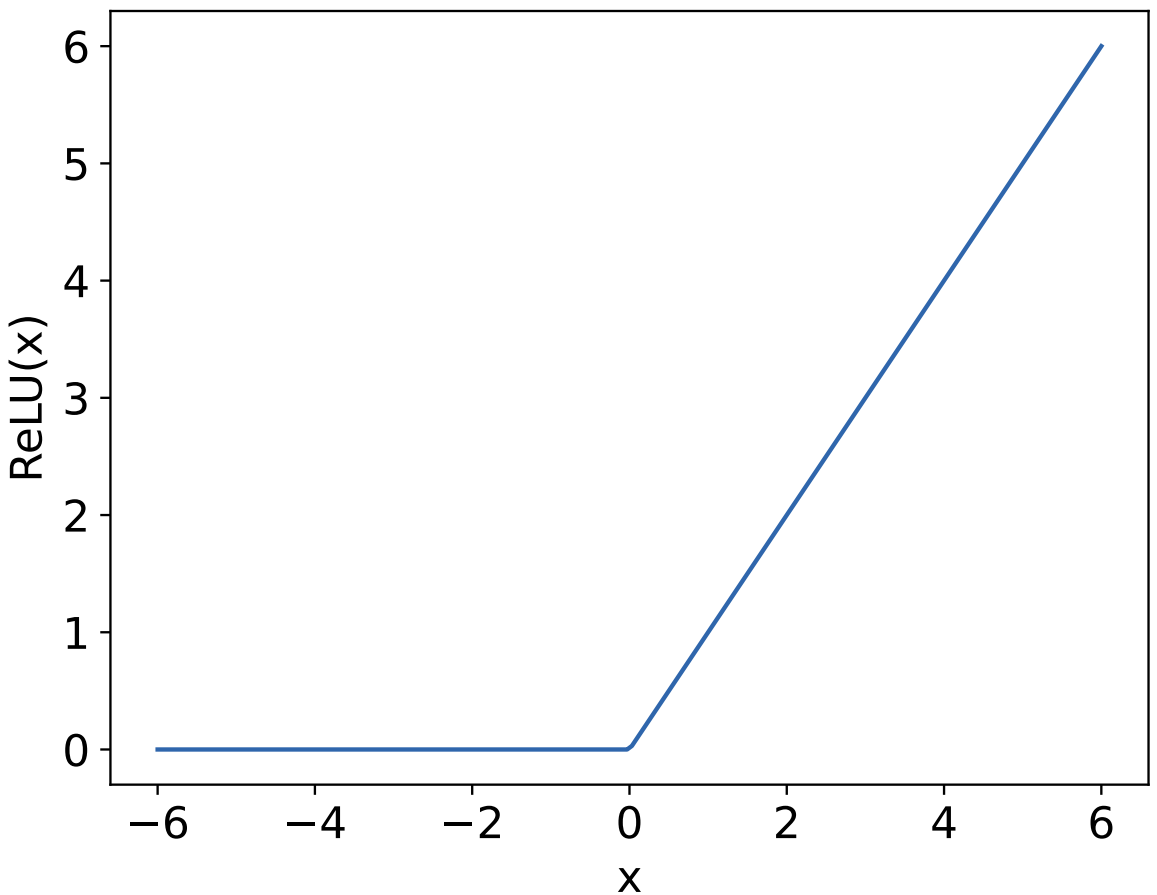
The derivatives of both tanh and sigmoid can be expressed in terms of the function itself.

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

Activation Function: Rectified Linear Unit (ReLU)

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



The Dying ReLU Problem

ReLU was once the most popular activation function in deep learning.

However, it can be fragile during training.

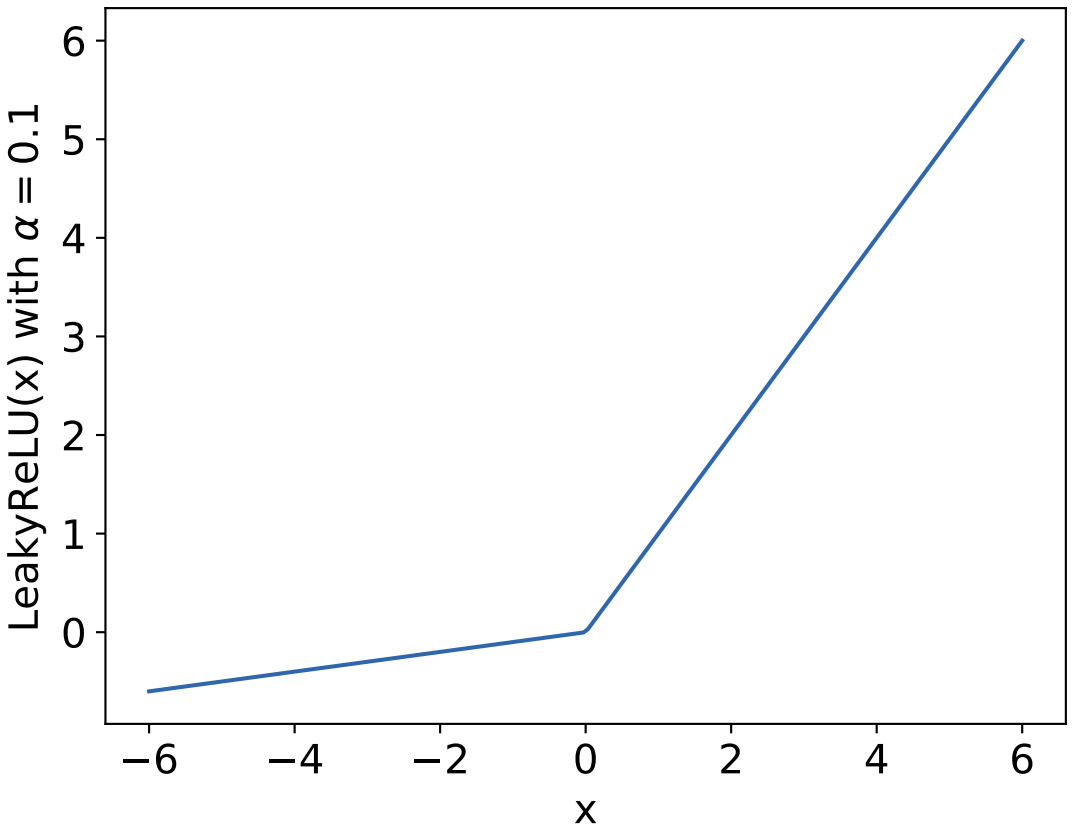
Due to problematic **weight initialization**, or **imbalanced data**, the input of a ReLU unit can become negative for all training examples.

These neurons will never activate again.

Solution: Leaky ReLU, Parametric ReLU, GELU, SELU, etc.

Leaky ReLU

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x (\alpha < 1) & \text{otherwise} \end{cases}$$



Text Classification with MLP

A 2-layer MLP is a simple and general text classification model:

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$
$$\mathbf{s} = \mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

$\mathbf{x}$: text features, $\mathbf{s} \in \mathbb{R}^{|\mathcal{Y}|}$: score$(\mathbf{x}, \cdot)$.

We empirically do not add nonlinear activation to the output layer—we'll see the reason soon!

Remaining Questions:

- How can we obtain the text representation $\mathbf{x}$?
  Anyhow we can convert text into a fixed-dimensional vector.
- How do we train the model (update model parameters)?
  By minimizing the objective function (e.g., cross-entropy loss) over the training data, using gradient-based optimization methods.

The `softmax` Operator

The `softmax` operator converts **a list of scores** into probabilities:

$$\texttt{softmax}: \mathbb{R}^d \to \mathbb{R}^d$$

$$\widehat{P}(Y = y_i) = \texttt{softmax}(\mathbf{s})_i = \frac{e^{s_i}}{\sum_{j=1}^{|Y|} e^{s_j}}$$

`softmax` is a differentiable operator so that we can compute the gradient of its output with respect to input.

In some tasks, particularly language model inference, `softmax` is usually coupled with a **temperature** τ.

$$\texttt{softmax}_\tau(\mathbf{s})_i = \frac{e^{s_i/\tau}}{\sum_j e^{s_j/\tau}}$$

When $\tau \to 0$ (temperature annealing), `softmax` becomes **argmax**.

The `softmax` Operator

Many activation functions restrict the range of the output to some narrow interval.

Some probability distributions can not be represented by `softmax`, if passing the output of these functions to `softmax`.

Example:

$$\mathbf{s} = \tanh(\mathbf{a}) \in \mathbb{R}^2$$

$\texttt{softmax}(\mathbf{s})$ cannot represent a distribution sharper than

$$\left[\frac{e^{-1}}{e + e^{-1}}, \frac{e}{e + e^{-1}}\right] = [0.12, 0.88].$$

This explains why we drop nonlinear activation for the output layer.

Softmax vs. Sigmoid

Softmax is a generalization of sigmoid to multiple classes.

$\sigma(x)$ gives the distribution

$$\left[\frac{1}{1+e^{-x}}, \frac{e^{-x}}{1+e^{-x}}\right] = \left[\frac{e^{0}}{e^{0}+e^{-x}}, \frac{e^{-x}}{e^{0}+e^{-x}}\right]$$
$$= \texttt{softmax}([0, -x])$$

Logistic regression can be considered as a single-layer neural network with sigmoid activation for 2-way classification.

Training Objective: The Cross-Entropy Loss

Recall the cross-entropy loss between the population distribution $Pop(\cdot \mid \mathbf{x})$ and the predicted distribution $\widehat{P}(\cdot \mid \mathbf{x})$:

$$H(Pop, \widehat{P}) = \mathbb{E}_{\mathbf{x}, y \sim Pop} \left[-\log \widehat{P}(y \mid \mathbf{x}) \right]$$

In practice, we **estimate** $H(Pop, \widehat{P})$ using the training data, with the assumption that they are i.i.d. samples from the population distribution.

For single-label classification, the cross-entropy loss then becomes the negative log-likelihood loss:

$$\text{loss}(\mathbf{x}_i, y_i) = -\log \widehat{P}(y_i \mid \mathbf{x}_i) = -\log \texttt{softmax}(\mathbf{s}(\mathbf{x}))_{y_i}$$

Training Neural Classifiers

Assume $\mathbf{x}_i$ is a text feature vector (obtained by bag-of-words over pretrained word embeddings).

For a mini-batch of examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{B}$:

- Compute the scores $\mathbf{s} = NN_{\mathbf{\Theta}}(\mathbf{x}_i)$ and $\text{loss}(\mathbf{x}_i, y_i)$ for each example.
- Compute the average loss over the mini-batch.
- Compute the gradient of the average loss with respect to the model parameters $\mathbf{\Theta}$ – using back propagation.
- Update the model parameters using the gradients.

Back Propagation

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$
$$\mathbf{s} = \mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$
$$\text{loss}(\mathbf{x}, y) = -\log \texttt{softmax}(\mathbf{s})_y$$

The gradient for the parameters $\mathbf{W}^{(0)}, \mathbf{b}^{(0)}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}$ can be computed using the chain rule.

$$\frac{\partial \text{loss}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \text{loss}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{W}^{(1)}} \quad \frac{\partial \text{loss}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \text{loss}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{b}^{(1)}} \quad \frac{\partial \text{loss}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \text{loss}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{z}^{(1)}}$$

$$\frac{\partial \text{loss}}{\partial \mathbf{W}^{(0)}} = \frac{\partial \text{loss}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(0)}} \qquad \frac{\partial \text{loss}}{\partial \mathbf{b}^{(0)}} = \frac{\partial \text{loss}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{b}^{(0)}}$$

SGD and advanced optimizers can be applied as long as the gradients are computable!

Some Philosophy: KL Divergence in Classification

[Source: David McAllester]

Recall: The Kullback–Leibler (KL) divergence serves as a natural measurement of the difference between two distributions.

$$KL(P \parallel Q) = \mathbb{E}_{y \sim P} \log \frac{P(y)}{Q(y)}$$

How many more bits are needed to encode samples from P using the optimal code of Q, compared to the optimal code of P.

What we really aim to optimize is $KL(Pop \parallel \widehat{P})$.

$$KL(Pop \parallel \widehat{P}) = \mathbb{E}_{\mathbf{x}, y \sim Pop} \log \frac{Pop(y \mid \mathbf{x})}{\widehat{P}(y \mid \mathbf{x})}$$

However, we don't have a good estimation of $Pop(y \mid \mathbf{x})$ due to lack of data.

Some Philosophy: KL Divergence vs. Cross Entropy

$$KL(Pop \,\|\, \widehat{P}) = \mathbb{E}_{\mathbf{x},y\sim Pop} \log \frac{Pop(y \mid \mathbf{x})}{\widehat{P}(y \mid \mathbf{x})}$$
$$= H(Pop, \widehat{P}) - H(Pop)$$
$$H(Pop, \widehat{P}) = \mathbb{E}_{\mathbf{x},y\sim Pop} - \log \widehat{P}(y \mid \mathbf{x})$$

Although the optimal $\widehat{P}$ are the same, we can use the training data to estimate $H(Pop, \widehat{P})$, but not $KL(Pop \,\|\, \widehat{P})$.

We are never simply interested in minimizing the cross-entropy loss between the training data distribution and the model predictions.

We are interested in doing so because training data is the only information we have about the **population distribution**.

Next

Advanced neural network architectures.