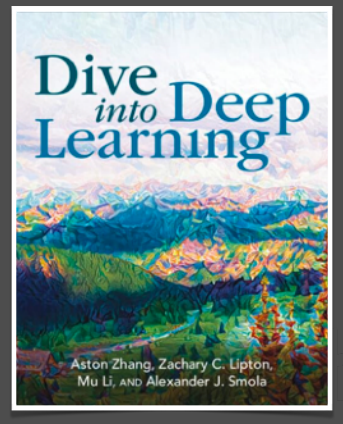
ROBOT INTELLIGENCE LAB

Introduction to Deep Learning

Multilayer Perceptron

Sungjoon Choi, Korea University



ROBOT INTELLIGENCE LAB

Multilayer Perceptron

Multilayer Perceptrons

ROBOT INTELLIGENCE LAB

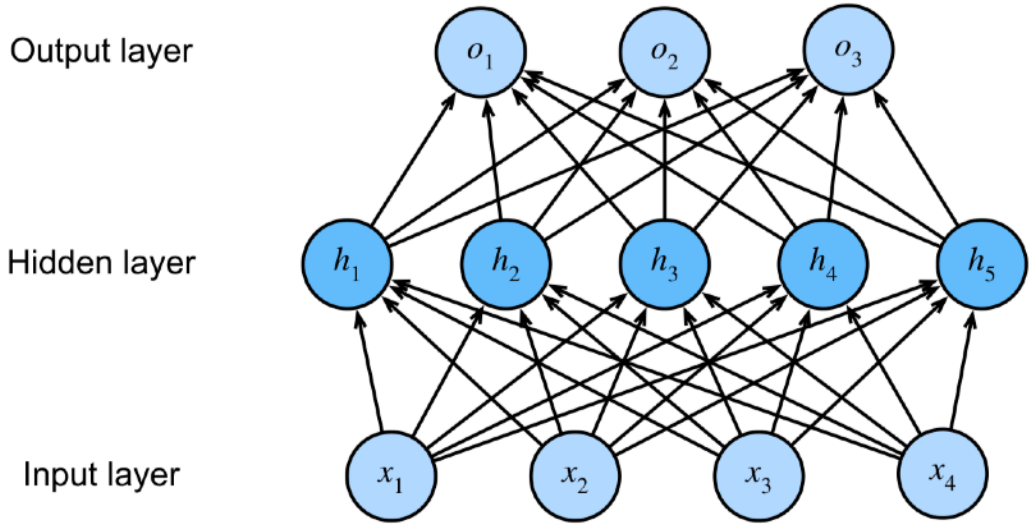
- We move on to truly (and the simplest) deep neural networks called **multilayer perceptrons** (MLPs).
  - They consist of multiple layers of neurons fully connected to those in the layer below and those above.
- Although automatic differentiation significantly simplifies the implementation, we will see how the gradients are calculated in deep networks.
- MLPs are high-capacity models and they are often prone to overfitting.
  - Thus, we will revisit regularization and generalization for deep networks.

Multilayer Perceptrons

ROBOT INTELLIGENCE LAB

- Our previous linear classification model simply maps inputs directly to outputs via a single affine transformation followed by a softmax operation.
- However, **linearity** (in affine transformation) is a strong assumption.
  - For example, linearity implies the weaker assumption of monotonicity.
  - However, most real-world problems are nonlinear (e.g., health assessments as a function of body temperature or dog classifier).
- One way to overcome this is to find a suitable **representation** of inputs, on top of which a single linear model would suffice.

Hidden Layers



- We can overcome the limitations of linear models by incorporating **one or more hidden layers**.
  - This architecture is commonly called a multilayer perceptron, aka **MLP**.

From Linear to Nonlinear

ROBOT INTELLIGENCE LAB

- Suppose we denote the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ a minibatch of n examples with d inputs (features).
- For a one-hidden-layer MLP whose hidden layer has h hidden units, we denote $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer (hidden representations).
- Then, the output of the MLP is calculated by:

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

- However, simply adding the hidden layer does not change anything!
  - Why?

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}$$

From Linear to Nonlinear

ROBOT INTELLIGENCE LAB

- Hence, we need a **nonlinear activation** function σ to be applied to each hidden unit following the affine transformation.
  - One popular choice is the ReLU (Rectified Linear Unit), $\sigma(x) = \max(0, x)$.
- Combining all together,

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$$

$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

- Of course, we can build more general MLPs by continuously stacking such hidden layers.

Universal Approximators

ROBOT INTELLIGENCE LAB

- It is worth asking just **how powerful** a deep network could be.
- A universal approximation property states that
- George Cybenko proved that an **MLP** with one hidden layer and sigmoid activations has the **universal approximation** property [1].
  - Polynomial functions also satisfy this property [2]. In fact, **polynomial functions** are the first function classes to have this property:

Fundamental Theorem of Approximation Theory

Let $f \in C[a, b]$, $-\infty < a < b < \infty$. Given $\epsilon > 0$, there exists an algebraic polynomial p for which

$$|f(x) - p(x)| < \epsilon$$

for all $x \in [a, b]$.

- In fact, if one can show that other families of functions (e.g., deep networks) behave like polynomials, then such families also have universal approximation properties.
- **Kernel methods** also satisfy this property [3].

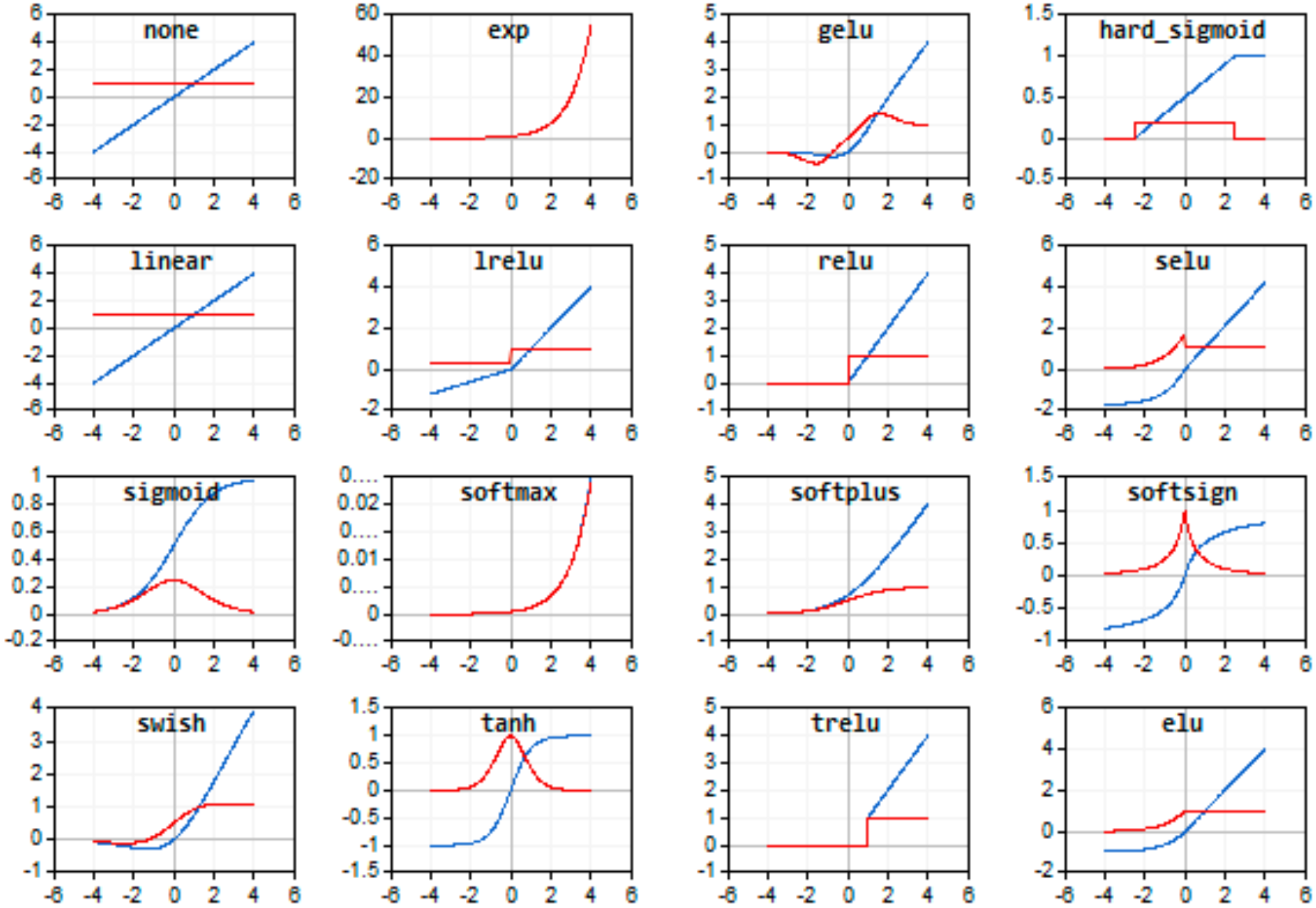
[1] "Approximation by superpositions of a sigmoidal function," 1989
[2] "A generalized Weierstrass approximation theorem," 1948
[3] "Universal Approximation Using Radial-Basis-Function Networks," 1991

Universal Approximators

ROBOT INTELLIGENCE LAB

- Note that these results only suggest that a single-hidden-layer network with enough nodes can model any continuous function.
  - Hence, just because a single-hidden-layer network can learn any function, it does not mean that you should try to solve all of your problems with this.

Activation Functions

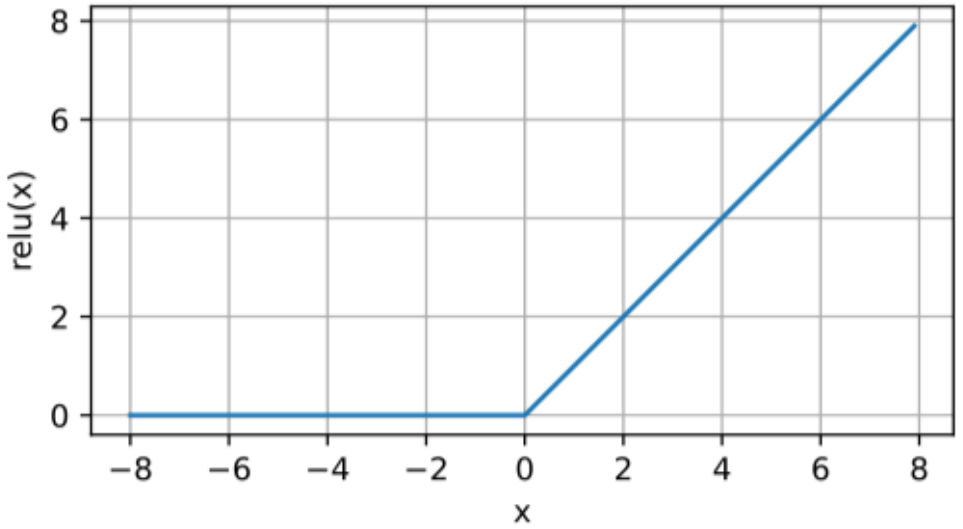


https://www.mql5.com/en/articles/12627

Activation Functions

ROBOT INTELLIGENCE LAB

- Activation functions are differentiable operators to transform input signals to outputs to add **non-linearity**.
- ReLU (rectified linear unit) Function: $\mathrm{ReLU}(x) = \max(x, 0)$

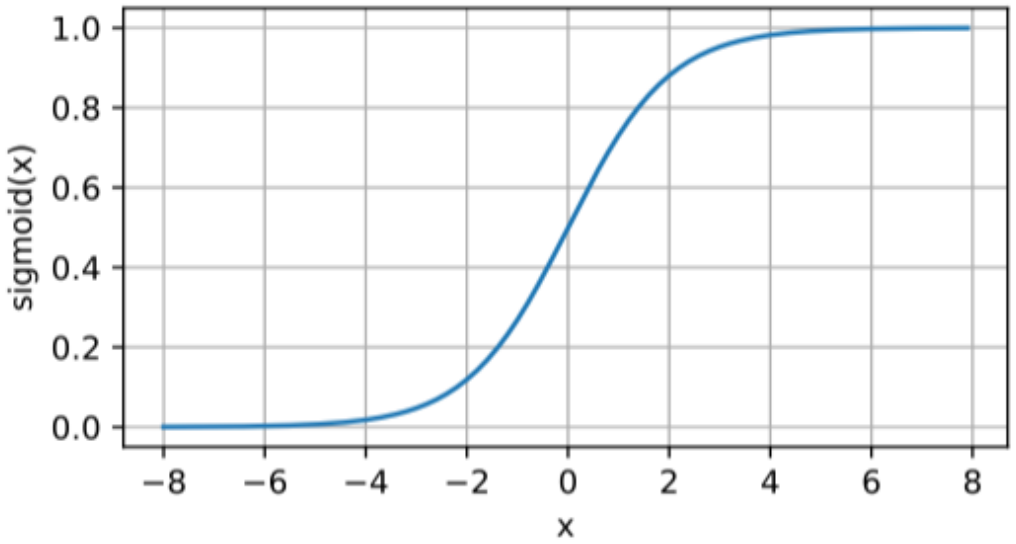


- A ReLU activation function is one of the most popular activation functions in deep learning and has its strength in handling the **vanishing gradient issue**.
- However, a dead relu problem might occur.

Activation Functions

ROBOT INTELLIGENCE LAB

- Sigmoid Function: $\text{sigmoid}(x) = \dfrac{1}{1+\exp(-x)}$

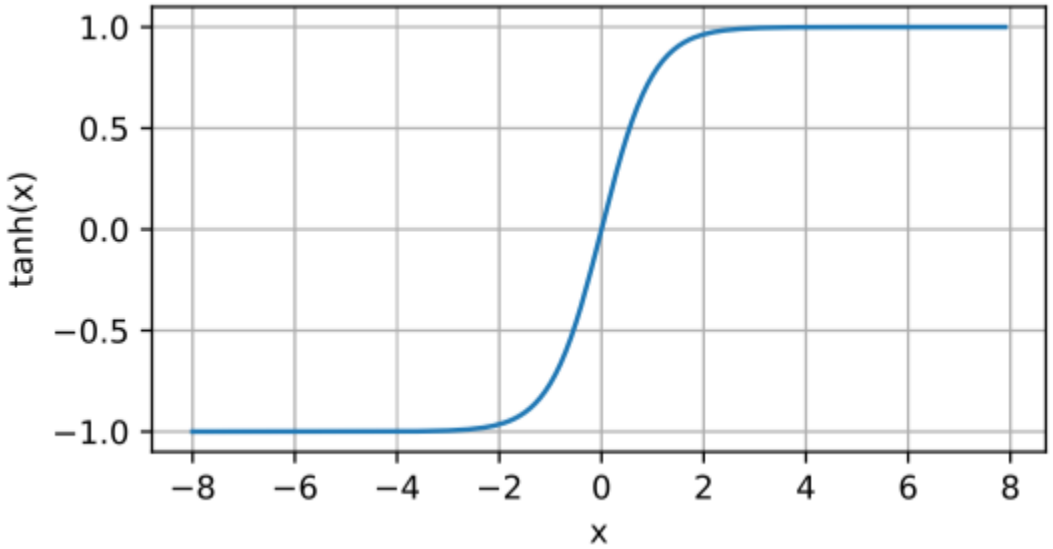


- The sigmoid function squashes the output to lie on the interval $(0,1)$.
- The gradient vanishing problem might happen, hindering its performance on deep networks.
- Also, the outputs are not zero-centered.
- One useful property of the sigmoid is that

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

Activation Functions

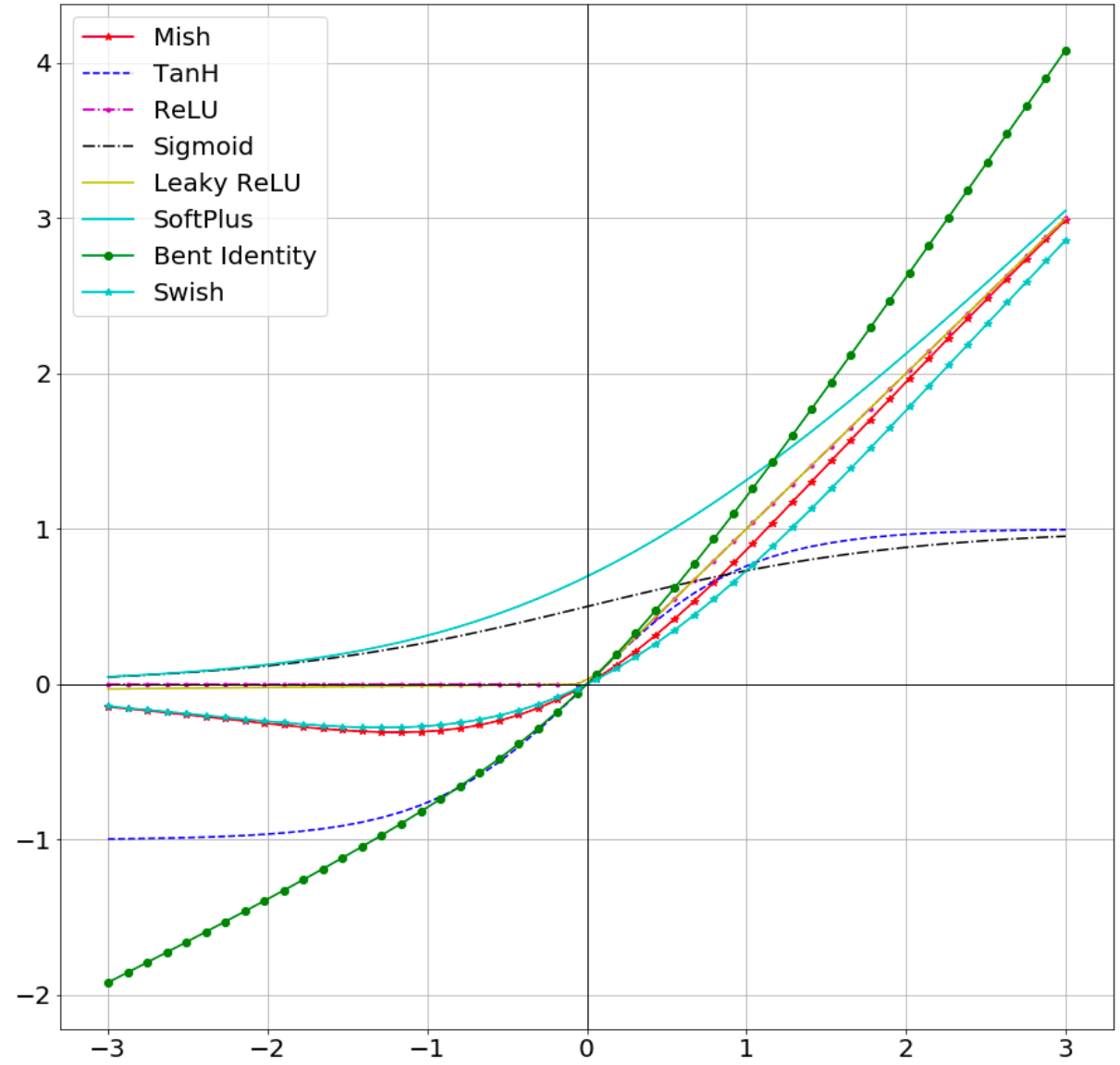
- Tanh Function: $\tanh(x) = \dfrac{1 - \exp(-2x)}{1 + \exp(-2x)}$



- The tanh function squashes the output to lie on the interval $(-1,1)$, hence symmetric.
- The vanishing gradient problem still exists.
- One useful property of the tanh is that

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

Activation Functions



- The **relu** is notorious for the dead relu problem.
- To handle this, the **elu** function was proposed. However, it introduces a longer computation time due to the exponential operation included.

$$\text{elu}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

- The **leaky relu** function also avoids the dead relu problem and is fast. However, we have to tune the parameter α.

$$\text{lrelu}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & x < 0 \end{cases}$$

- The **gelu** function works well in NLP, specifically Transformer models, as it is fast.

$$\text{gelu}(x) = 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3)))$$

- The **swish** function is continuous and differentiable at all points. And it works well on standard image datasets (CIFAR or ImageNet) compared to others (relu, lrelu, elu, gelu).

$$\text{swish}(x) = x(1 + e^{-x})^{-1}$$

- The **mish** function is C^∞-continuous and approximates identity near the origin. In some experiments, the **mish** works better than **swish** activations.

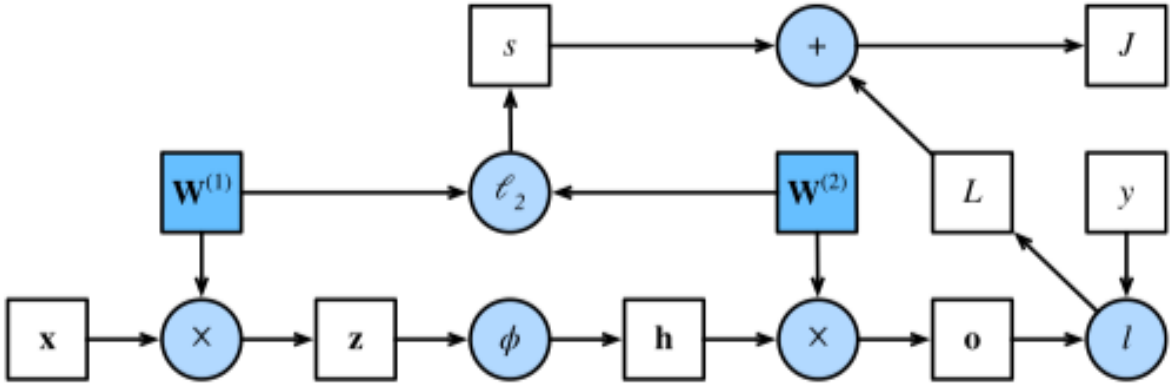
$$\text{mish}(x) = x \tanh(\text{softplus}(x))$$

https://data-newbie.tistory.com/297

ROBOT INTELLIGENCE LAB

Backpropagation

Forward Propagation



- For the sake of simplicity, an input example is $\mathbf{x} \in \mathbb{R}^d$ and no bias term exists.
- Then, the intermediate variable is $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} \in \mathbb{R}^h$ where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer.
- Our hidden activation vector is $\mathbf{h} = \phi(\mathbf{z})$.
- The hidden layer output becomes $\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h} \in \mathbb{R}^q$.
- The loss term for a single data becomes $L = l(\mathbf{o}, y)$.
- Also, the regularization term becomes $s = \frac{\lambda}{2}(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2)$.
- Finally, the model's regularized loss on a given data example is: $J = L + s$.

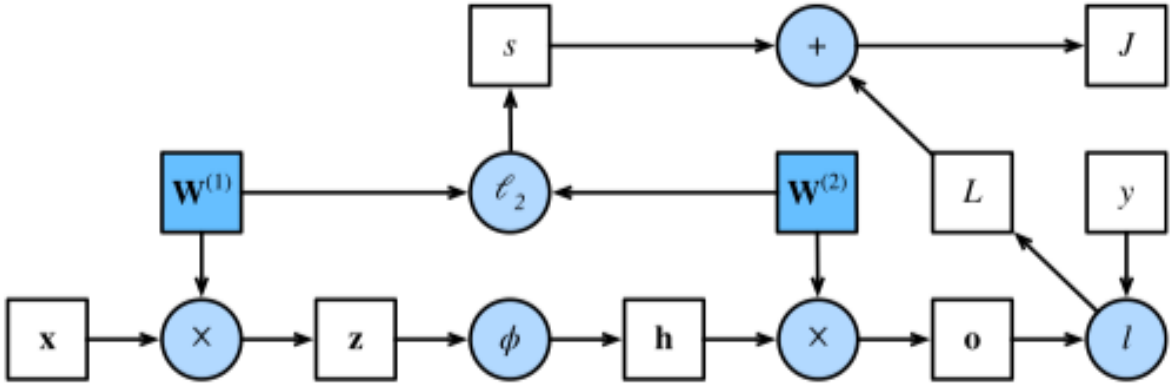
Backpropagation

ROBOT INTELLIGENCE LAB

- **Backpropagation** refers to the method of calculating the gradient of neural network parameters.
- In short, this method traverses the network in reverse order, from the output to the input layer, according to the **chain rule** from calculus.
  - Assume that we have $Y = f(X)$ and $Z = g(Y)$.
  - By using the chain rule:

$$\frac{\partial Z}{\partial X} = \text{prod}\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right)$$

Backpropagation



• The first step is to calculate the gradients of the objective $J = L + s$ w.r.t. the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1$$

• Next, we compute the gradient of J w.r.t. the output layer $\mathbf{o}$:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$$

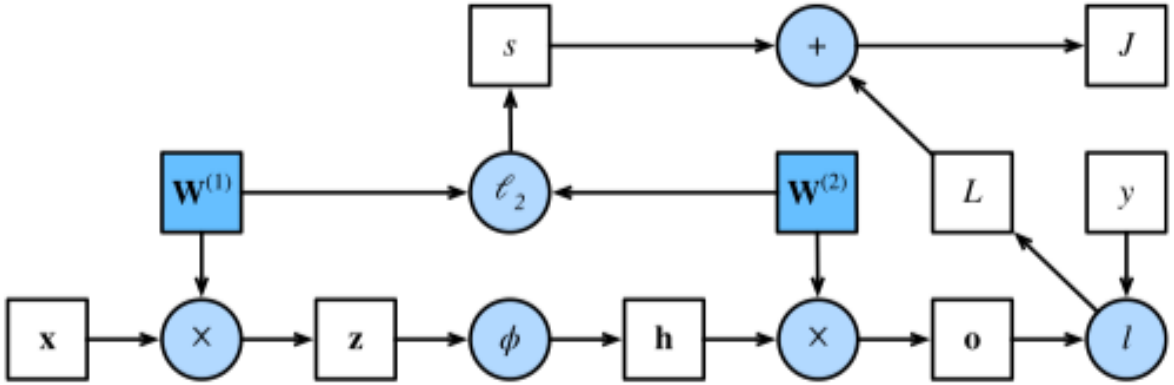
• Next, we calculate the gradients of s w.r.t. parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$$

• Now, we are able to calculate the gradient $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^T + \gamma \mathbf{W}^{(2)}$$

Backpropagation



• To obtain the gradient w.r.t. $\mathbf{W}^{(1)}$, we need to continue backpropagation along the output layer to the hidden layer.

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = \mathbf{W}^{(2)^T}\frac{\partial J}{\partial \mathbf{o}}$$

• Since the activation function ϕ applies elementwise:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$$

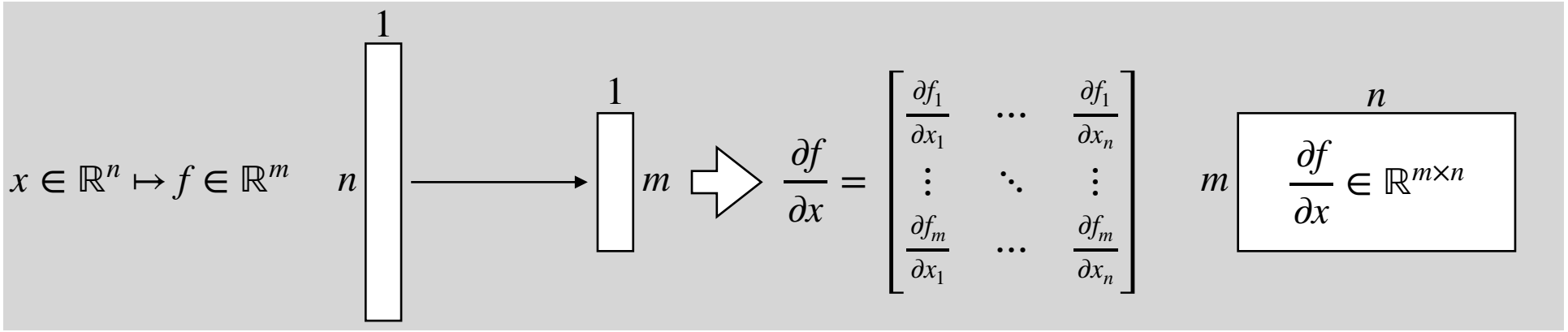
• Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$:

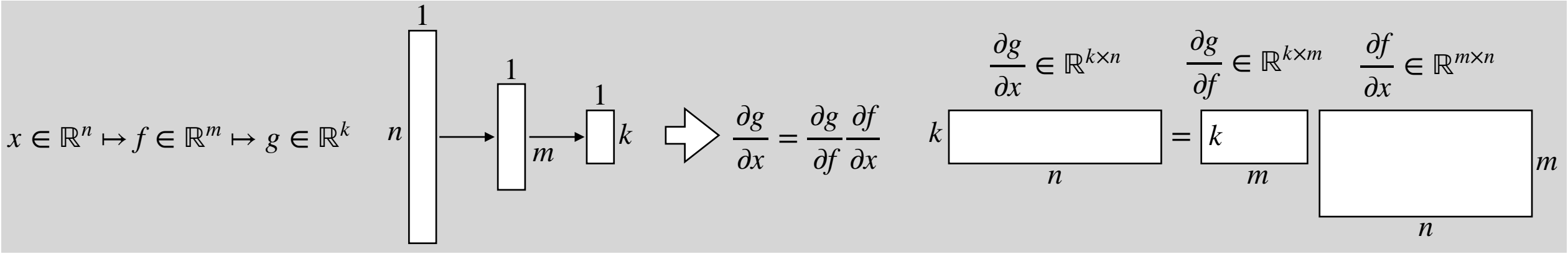
$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^T + \lambda \mathbf{W}^{(1)}$$

• We can further express this with:

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \left(\left(\mathbf{W}^{(2)^T}\frac{\partial L}{\partial \mathbf{o}}\right) \odot \phi'\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)\mathbf{x}^T + \lambda \mathbf{W}^{(1)}$$

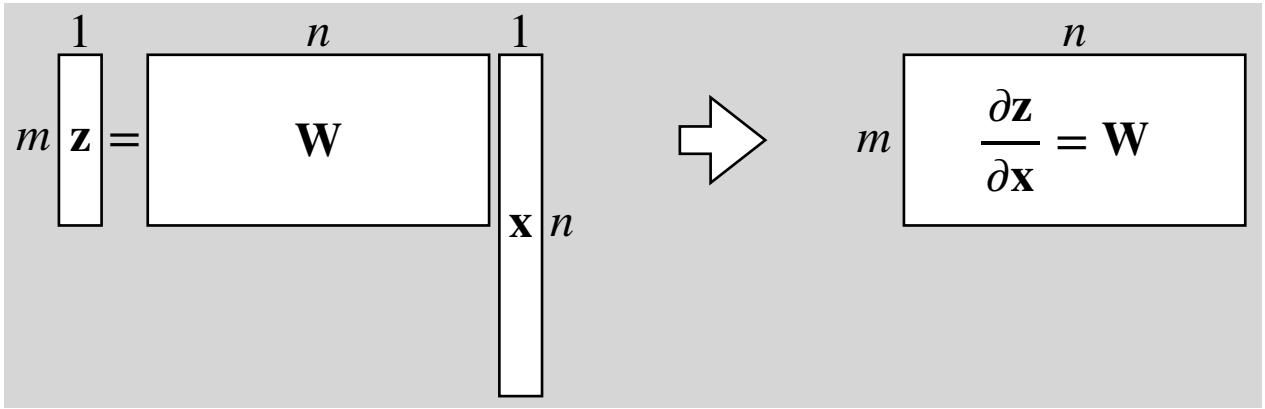
Matrix Calculus



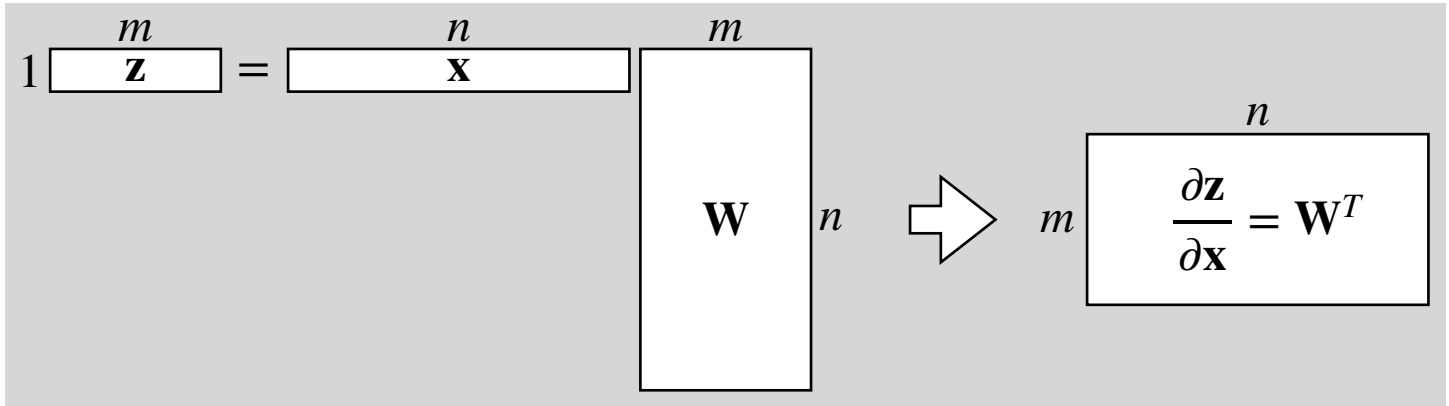


Matrix Calculus

- Matrix times column vector: $\mathbf{z} = \mathbf{W}\mathbf{x}$ where $\mathbf{z} \in \mathbb{R}^{m\times 1}$, $\mathbf{x} \in \mathbb{R}^{n\times 1}$, and $\mathbf{W} \in \mathbb{R}^{m\times n}$. Then, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W} \in \mathbb{R}^{m\times n}$.

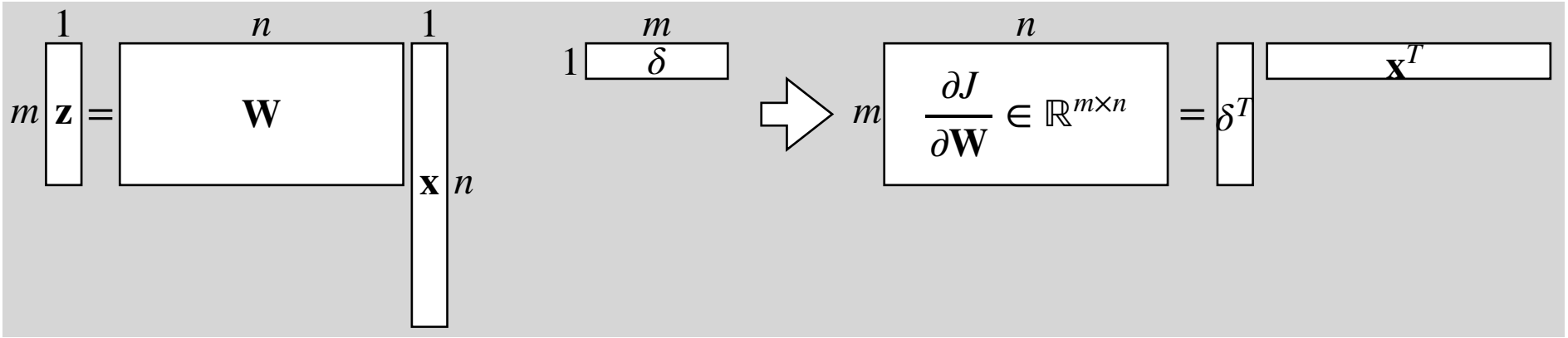


- Row vector times matrix: $\mathbf{z} = \mathbf{x}\mathbf{W}$ where $\mathbf{z} \in \mathbb{R}^{1\times m}$, $\mathbf{x} \in \mathbb{R}^{1\times n}$, and $\mathbf{W} \in \mathbb{R}^{n\times m}$. Then, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W}^T \in \mathbb{R}^{m\times n}$.



Matrix Calculus

- A vector with itself: $\mathbf{z} = \mathbf{x}$. Then, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{I}$.
- An elementwise function applied to a vector: $\mathbf{z} = f(\mathbf{x})$ where $z_i = f(x_i)$. Then, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \text{diag}(f'(\mathbf{x}))$.
  - We can also write $\odot f'(\mathbf{x})$ when applying the chain rule!
- Matrix times column vector: $\mathbf{z} = \mathbf{W}\mathbf{x}$ and $\delta = \frac{\partial J}{\partial \mathbf{z}} \in \mathbb{R}^{1 \times m}$. Then, $\frac{\partial J}{\partial \mathbf{W}} = \frac{\partial J}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \delta \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \delta^T \mathbf{x}^T \in \mathbb{R}^{m \times n}$



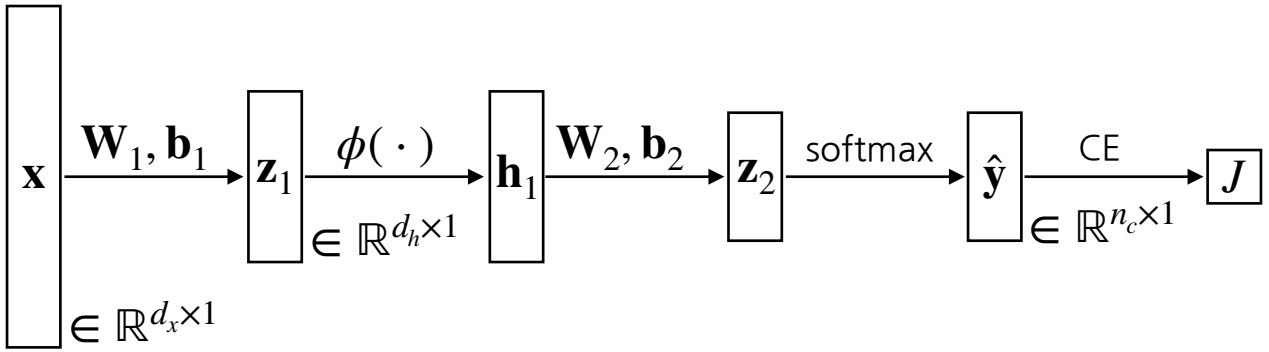
- Row vector times Matrix: $\mathbf{x} \in \mathbb{R}^{1 \times n}$, $\mathbf{W} \in \mathbb{R}^{n \times m}$, $\mathbf{z} = \mathbf{x}\mathbf{W} \in \mathbb{R}^{1 \times m}$, and $\delta = \frac{\partial J}{\partial \mathbf{z}} \in \mathbb{R}^{1 \times m}$.
  - Then, $\frac{\partial J}{\partial \mathbf{W}} = \frac{\partial J}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \delta \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \mathbf{x}^T \delta \in \mathbb{R}^{n \times m}$

Matrix Calculus

ROBOT INTELLIGENCE LAB

- (Recall) Cross-entropy loss w.r.t. logits:
  - Suppose $J = \text{CE}(\mathbf{y}, \hat{\mathbf{y}}) \in \mathbb{R}$ and $\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}) \in \mathbb{R}^{1\times k}$.
    - Then, $\dfrac{\partial J}{\partial \mathbf{z}} = \hat{\mathbf{y}} - \mathbf{y} \in \mathbb{R}^{1\times k}$ where k is the number of classes.

Quiz: One-Layer Neural Network

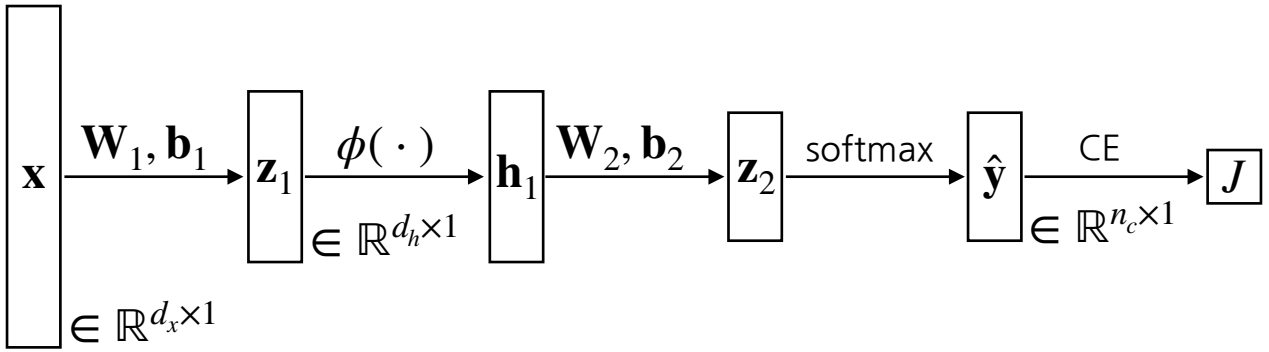


- Let's compute $\frac{\partial J}{\partial \mathbf{W}_2}$, $\frac{\partial J}{\partial \mathbf{b}_2}$, $\frac{\partial J}{\partial \mathbf{W}_1}$, $\frac{\partial J}{\partial \mathbf{b}_1}$, and $\frac{\partial J}{\partial \mathbf{x}}$.
- We introduce two intermediate derivatives:

$$\delta_1 = \frac{\partial J}{\partial \mathbf{z}_2}$$

$$\delta_2 = \frac{\partial J}{\partial \mathbf{z}_1}$$

Quiz: One-Layer Neural Network



- Let's compute $\frac{\partial J}{\partial \mathbf{W}_2}$, $\frac{\partial J}{\partial \mathbf{b}_2}$, $\frac{\partial J}{\partial \mathbf{W}_1}$, $\frac{\partial J}{\partial \mathbf{b}_1}$, and $\frac{\partial J}{\partial \mathbf{x}}$.
- We introduce two intermediate derivatives:

$$\delta_1 = \frac{\partial J}{\partial \mathbf{z}_2} = (\hat{\mathbf{y}} - \mathbf{y})^T \in \mathbb{R}^{1 \times n_c}$$

$$\delta_2 = \frac{\partial J}{\partial \mathbf{z}_1} = \frac{\partial J}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial \mathbf{z}_1} = (\hat{\mathbf{y}} - \mathbf{y})^T \mathbf{W}_2 \odot \phi'(\mathbf{z}_1) = (\hat{\mathbf{y}} - \mathbf{y})^T \mathbf{W}_2 \odot \phi'(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \in \mathbb{R}^{1 \times d_h}$$

- Then, we can compute our gradients:
  - $\frac{\partial J}{\partial \mathbf{W}_2} = \frac{\partial J}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{W}_2} = \delta_1 \frac{\partial \mathbf{z}_2}{\partial \mathbf{W}_2} = \delta_1^T \mathbf{h}_1^T \in \mathbb{R}^{n_c \times d_h}$
  - $\frac{\partial J}{\partial \mathbf{b}_2} = \frac{\partial J}{\partial \mathbf{z}_2}\frac{\partial \mathbf{z}_2}{\partial \mathbf{b}_2} = \delta_1 \frac{\partial \mathbf{z}_2}{\partial \mathbf{b}_2} = \delta_1^T \in \mathbb{R}^{n_c \times 1}$
  - $\frac{\partial J}{\partial \mathbf{W}_1} = \frac{\partial J}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial \mathbf{W}_1} = \delta_2 \frac{\partial \mathbf{z}_1}{\partial \mathbf{W}_1} = \delta_2^T \mathbf{x}^T \in \mathbb{R}^{d_h \times d_x}$
  - $\frac{\partial J}{\partial \mathbf{b}_1} = \frac{\partial J}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial \mathbf{b}_1} = \delta_2 \frac{\partial \mathbf{z}_1}{\partial \mathbf{b}_1} = \delta_2^T \mathbf{I}^T = \delta_2^T \in \mathbb{R}^{d_h \times 1}$
  - $\frac{\partial J}{\partial \mathbf{x}} = \frac{\partial J}{\partial \mathbf{z}_1}\frac{\partial \mathbf{z}_1}{\partial \mathbf{x}} = \mathbf{W}^T \delta_2^T \in \mathbb{R}^{d_x \times 1}$

Vanishing and Exploding Gradients

ROBOT INTELLIGENCE LAB

- Consider a deep network with L layers, input $\mathbf{x}$, and output $\mathbf{o}$.
- With each layer l defined by a transformation f_l parametrized by weights $\mathbf{W}^{(l)}$, whose hidden layer output is $\mathbf{h}^{(l)}$ (let $\mathbf{h}^{(0)} = \mathbf{x}$), our network can be expressed as:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \cdots \circ f_1(\mathbf{x})$$

- If all the hidden layer output and the input are vectors, we can write the gradient of $\mathbf{o}$ with respect to $\mathbf{W}^{(l)}$:

$$\partial_{\mathbf{W}^{(l)}}\mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}}\mathbf{h}^{(L)}}_{=M^{(L)}} \cdots \underbrace{\partial_{\mathbf{h}^{(l)}}\mathbf{h}^{(l+1)}}_{=M^{(l+1)}} \underbrace{\partial_{\mathbf{W}^{(l)}}\mathbf{h}^{(l)}}_{=\mathbf{v}^{(l)}}$$

- In other words, this gradient is the **product of $L-1$ matrices** and the gradient vector.

ROBOT INTELLIGENCE LAB

Other Issues

Nonparametrics?

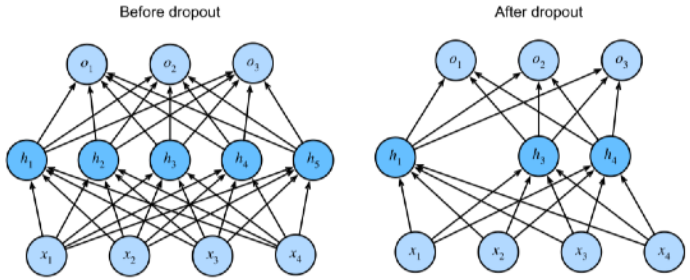
ROBOT INTELLIGENCE LAB

- Are MLPs parametric models?
  - The models do have millions of parameters.
- While neural networks clearly have parameters, in some ways, it can be more fruitful to think of them as behaving like nonparametric models.
- So what precisely makes a model nonparametric?
  - While the name covers a diverse set of approaches, one common theme is that nonparametric methods tend to have a level of complexity that grows as the amount of available data grows.
  - Nonparametric methods include 1) k-nearest neighbor algorithm and 2) kernel-based methods.
- In a sense, because neural networks are over-parametrized, they tend to interpolate the training data (fitting it perfectly) having the same property as nonparametric models.

Regularization in Neural Networks

ROBOT INTELLIGENCE LAB

- Early stopping
  - While deep neural networks are capable of fitting arbitrary labels, early stopping becomes an efficient method for regularizing networks.
  - In other words, whenever a model has fitted the cleanly labeled data but not randomly labeled examples, it has, in fact, been generalized.
- Weight decay
  - Depending on which weight norm is penalized, it is known either as ridge regularization (for l_2-norm) or lass regularization (for l_1-norm).
  - While it still remains a popular tool, researchers have noted that typical strengths of weight decay are insufficient for generalization.
- Dropout
  - Randomly replace some portion of nodes into 0.



Title Text