

# Deep Learning (Fall 2023)

**Ikbeom Jang**

[ijang@hufs.ac.kr](mailto:ijang@hufs.ac.kr)

**CES HUFS**

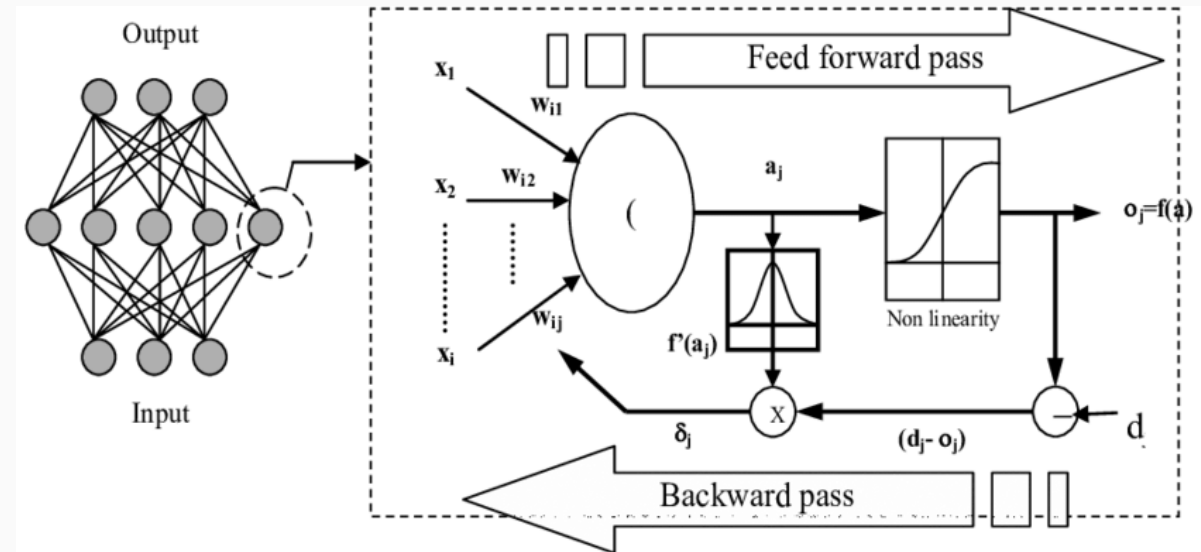
# **Multilayer Perceptrons**

- **Forward & Backward Propagation**
- **Vanishing and Exploding Gradients**
- **Initialization**
- **Generalization**
- **Dropout**

# Forward & Backward Propagation

# Forward & Backward Propagation

- So far, we have implemented **forward propagation** but invoked the **backpropagation** function provided by the DL framework.
- The automatic calculation of gradients profoundly simplifies the implementation of DL algorithms.
- Before **automatic differentiation**, even small changes to complicated models required recalculating complicated derivatives by hand.
- Let's take a deep dive into the details of backward propagation (= backpropagation).
- To convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs.
- To start, we focus our exposition on a one-hidden-layer MLP with weight decay ( $\ell_2$  regularization).



# Forward Propagation

- Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.
- We now work step-by-step through the mechanics of a neural network with one hidden layer.
- For simplicity, let's assume that the input example is  $\mathbf{x} \in \mathbb{R}^d$  and that our hidden layer does not include a bias term.

Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \quad (5.3.1)$$

where  $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  is the weight parameter of the hidden layer. After running the intermediate variable  $\mathbf{z} \in \mathbb{R}^h$  through the activation function  $\phi$  we obtain our hidden activation vector of length  $h$ :

$$\mathbf{h} = \phi(\mathbf{z}). \quad (5.3.2)$$

The hidden layer output  $\mathbf{h}$  is also an intermediate variable. Assuming that the parameters of the output layer possess only a weight of  $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ , we can obtain an output layer variable with a vector of length  $q$ :

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \quad (5.3.3)$$

# Forward Propagation

Assuming that the loss function is  $l$  and the example label is  $y$ , we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y). \quad (5.3.4)$$

As we will see the definition of  $\ell_2$  regularization to be introduced later, given the hyperparameter  $\lambda$ , the regularization term is

$$s = \frac{\lambda}{2} \left( \|\mathbf{W}^{(1)}\|_{\text{F}}^2 + \|\mathbf{W}^{(2)}\|_{\text{F}}^2 \right), \quad (5.3.5)$$

where the Frobenius norm of the matrix is simply the  $\ell_2$  norm applied after flattening the matrix into a vector. Finally, the model's regularized loss on a given data example is:

$$J = L + s. \quad (5.3.6)$$

We refer to  $J$  as the *objective function* in the following discussion.

# Forward Propagation

- Plotting **computational graphs** helps us visualize the dependencies of operators and variables within the calculation.
- Fig. 5.3.1 contains the graph associated with the simple network described above, where squares denote variables and circles denote operators.
- The lower-left corner signifies the input, and the upper-right corner is the output.

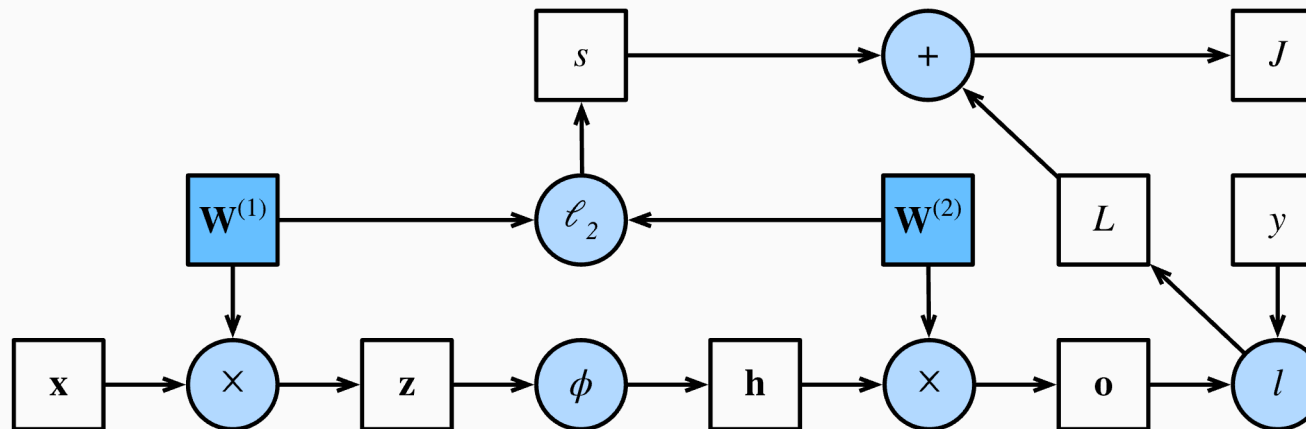


Fig. 5.3.1 Computational graph of forward propagation

# Backward Propagation

- **Backpropagation** refers to the method of **calculating the gradient of neural network parameters**.
- In short, the method traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus.
- The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters.
- Assume that we have functions  $Y = f(X)$  and  $Z = g(Y)$ , in which the input and the output  $X, Y, Z$  are tensors of arbitrary shapes.
- By using the chain rule, we can compute the derivative of  $Z$  with respect to  $X$  via

$$\frac{\partial Z}{\partial X} = \text{prod} \left( \frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right). \quad (5.3.7)$$



# Backward Propagation

- Here we use the *prod* operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions.
- For vectors, this is straightforward: it is simply matrix–matrix multiplication. (For higher dimensional tensors, we use the appropriate counterpart. The operator *prod* hides all the notational overhead.)
- Recall that the parameters of the simple network with one hidden layer are  $\mathbf{W}^{(1)}$  and  $\mathbf{W}^{(2)}$ .
- The objective of backpropagation is to calculate the gradients  $\partial J / \partial \mathbf{W}^{(1)}$  and  $\partial J / \partial \mathbf{W}^{(2)}$ .
- To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter.
- The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters.

# Backward Propagation

- The first step is to calculate the **gradients of the objective function  $J = L + S$  with respect to the loss term  $L$  and the regularization term  $s$** :

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \quad (5.3.8)$$

Next, we compute the gradient of the objective function with respect to variable of the output layer  $\mathbf{o}$  according to the chain rule:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left( \frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}} \right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \quad (5.3.9)$$

Next, we calculate the gradients of the regularization term with respect to both parameters:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \quad (5.3.10)$$

Now we are able to calculate the gradient  $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$  of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}} \right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \quad (5.3.11)$$

# Backward Propagation

To obtain the gradient with respect to  $\mathbf{W}^{(1)}$  we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer output  $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$  is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}. \quad (5.3.12)$$

Since the activation function  $\phi$  applies elementwise, calculating the gradient  $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$  of the intermediate variable  $\mathbf{z}$  requires that we use the elementwise multiplication operator, which we denote by  $\odot$ :

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}). \quad (5.3.13)$$

Finally, we can obtain the gradient  $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}} \right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \quad (5.3.14)$$

# Training Neural Networks

- When training neural networks, forward and backward propagation depend on each other.
- In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path.
- These are then used for backpropagation where the compute order on the graph is reversed.
- On the one hand, computing the regularization term (5.3.5) during forward propagation depends on the current values of model parameters  $\mathbf{W}^{(1)}$  and  $\mathbf{W}^{(2)}$ .
- They are given by the optimization algorithm according to backpropagation in the most recent iteration.
- On the other hand, the gradient calculation for the parameter (5.3.11) during backpropagation depends on the current value of the hidden layer output  $h$ , which is given by forward propagation.

# Backward Propagation

- Therefore, when training neural networks, once model parameters are initialized, **we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation.**
- Note that **backpropagation reuses the stored intermediate values from forward propagation** to avoid duplicate calculations.
- One of the consequences is that we need to retain the intermediate values until backpropagation is complete.
- This is also one of the reasons why **training requires significantly more memory than plain prediction.**
- Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size.
- Thus, **training deeper networks using larger batch sizes more easily leads to out-of-memory errors.**

# Summary

- **Forward propagation** sequentially calculates and stores intermediate variables within the computational graph defined by the neural network.
- It proceeds from the input to the output layer.
- **Backpropagation** sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- When training deep learning models, forward propagation and backpropagation are interdependent, and training requires significantly more memory than prediction.

# Vanishing and Exploding Gradients

```
%matplotlib inline  
import torch from d2l  
import torch as d2l
```

# Vanishing and Exploding Gradients

- Every model that we have implemented required that we initialize its parameters according to some pre-specified distribution.
- Until now, we took the initialization scheme for granted, glossing over the details of how these choices are made.
- The **choice of initialization scheme** plays a significant role in neural network learning, and it can be crucial for maintaining numerical stability.
- Moreover, these choices can be **tied up in interesting ways with the choice of the nonlinear activation function**.
- **Which function we choose and how we initialize parameters can determine how quickly our optimization algorithm converges.**
- Poor choices here can cause us to encounter **exploding or vanishing gradients** while training.
- In this section, we delve into these topics in greater detail and discuss some useful heuristics that you will find useful throughout your career in deep learning.



# Vanishing and Exploding Gradients

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:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \dots \circ f_1(\mathbf{x}). \quad (5.4.1)$$

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

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \dots \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}. \quad (5.4.2)$$

# Vanishing and Exploding Gradients

- In other words, this gradient is the product of  $L - 1$  matrices  $\mathbf{M}^{(L)} \dots \mathbf{M}^{(l+1)}$  and the gradient vector  $\mathbf{v}^{(l)}$ .
- Thus, we are susceptible to the same problems of numerical underflow that often crop up when multiplying together too many probabilities.
- When dealing with probabilities, a common trick is to switch into log-space
- Unfortunately, our problem above is more serious: initially the matrices  $\mathbf{M}^{(l)}$  may have a wide variety of eigenvalues.
- They might be small or large, and their product might be very large or very small.

# Vanishing and Exploding Gradients

- The risks posed by unstable gradients go beyond numerical representation.
- Gradients of unpredictable magnitude also threaten the stability of our optimization algorithms.
- We may be facing parameter updates that are either
  - (i) excessively large, destroying our model (the **exploding gradient** problem) or
  - (ii) excessively small (the **vanishing gradient** problem), rendering learning impossible as parameters hardly move on each update.

# Vanishing Gradients

- One frequent culprit causing the vanishing gradient problem is the choice of the activation function  $\sigma$  (that is appended following each layer's linear operations.)
- Historically, the sigmoid function  $1/(1 + \exp(-x))$  was popular because it resembles a thresholding function.
- Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either fully or not at all seemed appealing.
- Let's take a closer look at the sigmoid to see why it can cause vanishing gradients.

\* culprit: 범인

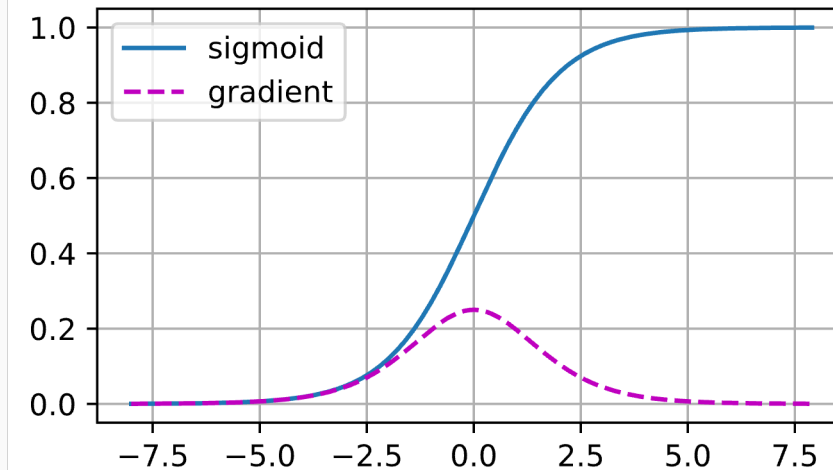
# Vanishing Gradients

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
```

```
y = torch.sigmoid(x)
```

```
y.backward(torch.ones_like(x))
```

```
d2l.plot(x.detach().numpy(), [y.detach().numpy(), x.grad.numpy()], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



# Vanishing Gradients

- As you can see, the **sigmoid's gradient vanishes** both **when its inputs are large and when they are small**.
- Moreover, when backpropagating through many layers, unless we are in the Goldilocks zone, where the inputs to many of the sigmoids are close to zero, **the gradients of the overall product may vanish**.
- When our network boasts **many layers**, unless we are careful, **the gradient will likely be cut off at some layer**.
- This problem used to plague deep network training.
- ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice for practitioners.

\* plague: 괴롭히다, 성가시게 하다

# Exploding Gradients

- The opposite problem, when **gradients explode**, can be similarly vexing.
- To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix.
- For the scale that we picked (the choice of the **variance  $\sigma^2 = 1$** ), the **matrix product explodes**.
- When this happens because of the initialization of a deep network, we **have no chance of getting a gradient descent optimizer to converge**.

\* vex: 성가시게 하다

# Breaking the Symmetry

- **Symmetry Issue in Neural Networks:**
  - Neural networks can suffer from symmetry in parametrization, where permutations of weights in certain layers result in the same function.
- **Permutation Symmetry in Hidden Units:**
  - In a multi-layer perceptron (MLP) with one hidden layer and two units, permutation symmetry exists among the hidden units, making them indistinguishable.
- **Practical Consequences of Symmetry:**
  - Symmetry can lead to issues in network expressiveness, especially illustrated in scenarios where initialization of hidden layer parameters is uniform, hindering gradient-based iterations.
- **Example of Symmetry Impact:**
  - Using a constant initialization for all hidden layer parameters may result in both hidden units producing the same activations during forward propagation, impeding gradient-based symmetry breaking.



# Breaking the Symmetry

- **Gradient-Based Iteration Challenges:**
  - Gradient-based iterations, like minibatch stochastic gradient descent, may struggle to break symmetry, leaving the hidden layer to behave as if it has only a single unit.
- **Overcoming Symmetry with Dropout:**
  - While minibatch stochastic gradient descent may not naturally break symmetry, dropout regularization can effectively address symmetry issues and enhance the network's expressive power.
- **Note on Dropout's Role:**
  - Dropout regularization, introduced later, plays a crucial role in breaking symmetry and preventing the hidden layer from behaving as if it has only a single unit.

# Exploding Gradients

```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))
print('after multiplying 100 matrices\n', M)
```

```
a single matrix
tensor([[ -0.8755, -1.2171, 1.3316, 0.1357],
        [ 0.4399, 1.4073, -1.9131, -0.4608],
        [-2.1420, 0.3643, -0.5267, 1.0277],
        [-0.1734, -0.7549, 2.3024, 1.3085]])

after multiplying 100 matrices
tensor([[ -2.9185e+23, 1.3915e+25, -1.1865e+25, 1.4354e+24],
        [ 4.9142e+23, -2.3430e+25, 1.9979e+25, -2.4169e+24],
        [ 2.6578e+23, -1.2672e+25, 1.0805e+25, -1.3072e+24],
        [-5.2223e+23, 2.4899e+25, -2.1231e+25, 2.5684e+24]])
```

# Parameter Initialization

- One way of addressing—or at least **mitigating—the issues** raised above is through **careful initialization**.
- As we will see later, additional care during optimization and **suitable regularization** can further enhance stability.
- **Default Initialization**
  - In the previous sections, we used a normal distribution to initialize the values of our weights.
  - If we do not specify the initialization method, the framework will use a default random initialization method, which often works well in practice for moderate problem sizes.

# Parameter Initialization

- **Xavier Initialization**

- The **goal** is to initialize the weights such that the **variance of the activations are the same across every layer**.
- This constant variance **helps prevent the gradient from exploding or vanishing**.
- Typically, this method **samples weights from a Gaussian distribution with zero mean and variance  $\sigma^2 = \frac{2}{n_{in} + n_{out}}$** .
- We can also adapt this to choose the variance when **sampling weights from a uniform distribution**.
- The uniform distribution  $U(-a, a)$  has variance  $\frac{a^2}{3}$ . Plugging  $\frac{a^2}{3}$  into our condition on  $\sigma^2$  prompts us to initialize according to

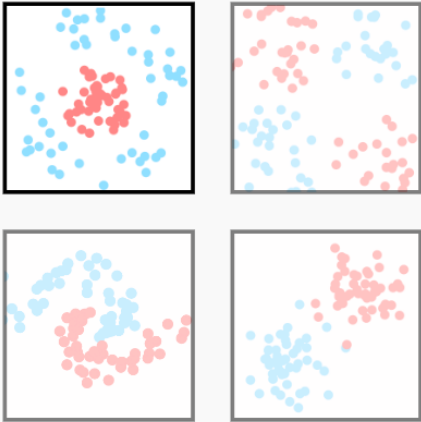
$$U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

- Though the assumption for nonexistence of nonlinearities in the above mathematical reasoning can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice.

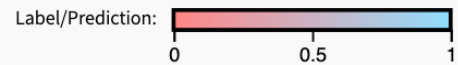
# Parameter Initialization – Exercise 1




## 1. Choose input dataset

Select a training dataset.



This legend details the color scheme for labels, and the values of the weights/gradients.



Node Type:  Input  Relu  Sigmoid

## 2. Choose initialization method

Select an initialization method for the values of your neural network parameters<sup>1</sup>.

 Zero  Too small  Appropriate  Too large

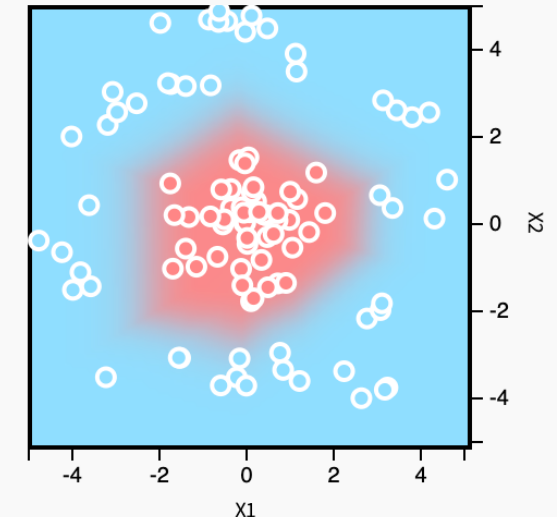
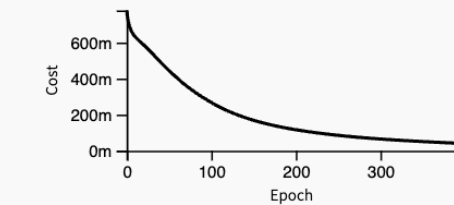


Select whether to visualize the weights or gradients of the network above.

 Weight  Gradient

## 3. Train the network.

Observe the cost function and the decision boundary.



# Parameter Initialization – Exercise 2

## 1. Load your dataset

Load 10,000 handwritten digits images ([MNIST](#)).

Load MNIST (100%)

Input batch of 100 images

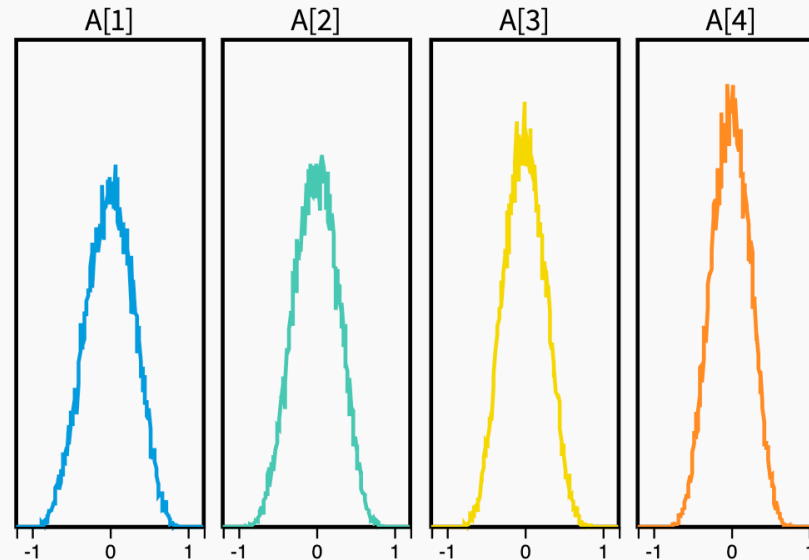


Batch: 58 Epoch: 1

## 2. Select an initialization method

Among the below distributions, select the one to use to initialize your parameters<sup>3</sup>.

☐ Zero ☐ Uniform ☒ Xavier ☐ Standard Normal

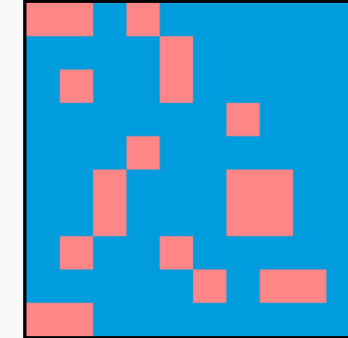


## 3. Train the network and observe

The grid below refers to the input images, **Blue** squares represent correctly classified images. **Red** squares represent misclassified images.

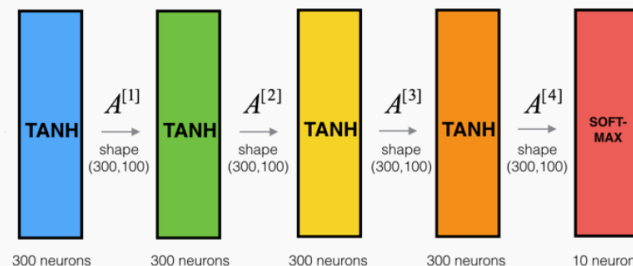


Output predictions of 100 images



Misclassified: 21/100 Cost: 0.94

$X = A^{[0]}$   
Batch of 100 grayscale images of shape 28x28  
X.shape = (784, 100) because 784 = 28x28



$\hat{y} = A^{[5]}$   
output probability over 10  
classes for a batch of  
100 images  
 $\hat{y}.shape = (10, 100)$

# Summary

- Common Issues in Deep Networks:
  - **Vanishing and exploding gradients** are prevalent challenges in deep neural networks.
- Importance of Parameter Initialization:
  - **Careful parameter initialization** is crucial to maintain control over gradients and parameters in deep networks.
- Initialization Heuristics for Gradient Control:
  - **Initialization heuristics are necessary** to ensure that initial gradients are neither too large nor too small.
- Symmetry Breaking with Random Initialization:
  - **Random initialization is a key** strategy to **break symmetry** before optimization in deep networks.
- **Xavier Initialization for Variance Control:**
  - Xavier initialization recommends maintaining the variance of outputs and gradients independent of the number of inputs and outputs in each layer.
- ReLU Activation for Vanishing Gradient Mitigation:
  - **ReLU activation** functions are effective in **mitigating the vanishing gradient** problem, contributing to **faster convergence** in deep neural networks.

# Generalization in Deep Learning



# Generalization in Deep Learning

- Regression and Classification with Linear Models:
  - Sections 3 and 4 covered linear model fitting for regression and classification, optimizing parameters for likelihood maximization of training labels.
- Machine Learning's Core Objective:
  - The primary goal is not just fitting data but discovering general patterns for accurate predictions on new examples from the same population.
- Optimization and Statistical Principles:
  - Optimization in machine learning is a means to an end; the focus is on statistical principles guiding models to generalize beyond the training set.
- Deep Neural Networks' Success:
  - Deep neural networks, trained with stochastic gradient descent, excel in generalization across diverse domains.

# Generalization in Deep Learning

- Challenges in Deep Learning Understanding:
  - Understanding why neural networks are optimizable and generalize well remains a complex, evolving field in both theory and practice.
- Rapid Evolution in Theory and Practice:
  - Both deep learning theory and practice evolve rapidly, with theorists and practitioners contributing to strategies, heuristics, and folk knowledge.
- Generalization Challenge:
  - While fitting parameters to training data is usually manageable, understanding how models generalize effectively is a significant challenge in deep learning.
- Practitioner's Toolkit:
  - Despite a lack of a comprehensive theory, practitioners have developed techniques for effective model generalization in practice, shaping the ongoing state of deep learning research.

# Overfitting and Regularization

- **"No Free Lunch" Theorem:**
  - Wolpert and Macready's "no free lunch" theorem states that learning algorithms generalize better on certain data distributions and worse on others, emphasizing the importance of assumptions and inductive biases.
- **Inductive Biases in Training:**
  - To achieve human-level performance, identifying inductive biases that reflect human thinking about the world is crucial.
  - E.g., a deep MLP has an inductive bias towards composing simpler functions to build a complex function.
- **Two Phases in Training Approach:**
  - Training machine learning models involves fitting the training data and estimating generalization error by evaluating the model on holdout data.
  - The difference between training and test data fits is the generalization gap, indicating overfitting if large.
- **Overfitting Challenges in Deep Learning:**
  - Deep learning introduces counterintuitive challenges.
  - In classification problems, models can perfectly fit large training datasets.
  - Strangely, increasing model complexity, such as adding layers, may reduce generalization error despite perfect training data fit.

# Overfitting and Regularization

- **Non-Monotonic Patterns** in Generalization:
  - The relationship between generalization gap and model complexity can be non-monotonic in deep learning, following a "double-descent" pattern, where greater complexity initially hurts but later helps.
- **Bag of Tricks** for Overfitting Mitigation:
  - Deep learning practitioners use a variety of techniques to mitigate overfitting, some seemingly restrictive and others making the model more expressive, navigating the complexity of deep learning tasks.
- **Limitations** of Classical Learning Theory:
  - Classical learning theory guarantees may be conservative for classical models and fail to explain why deep neural networks generalize.
  - Traditional complexity-based generalization bounds cannot elucidate why neural networks generalize despite fitting arbitrary labels for large datasets.

# Inspiration from Nonparametrics

- Deep Learning as **Nonparametric Models**:
  - Although deep neural networks have millions of parameters, it can be more fruitful to conceptualize them as behaving like nonparametric models.
- Characteristics of Nonparametric Models:
  - Nonparametric models, including neural networks, exhibit complexity that grows with available data, challenging the traditional parametric view.
- **1-Nearest Neighbor** as Example:
  - The 1-nearest neighbor algorithm is a simple example of a nonparametric model, memorizing the dataset during training and achieving zero training error, demonstrating consistency under mild conditions.
- Deep Neural Networks and Nonparametric Connections:
  - Over-parametrized deep neural networks, possessing more parameters than needed, tend to interpolate training data, behaving akin to nonparametric models.
  - The neural tangent kernel, a specific choice of the kernel function, establishes a connection between large neural networks and nonparametric methods, aiding in the understanding of their behavior.

# Classical Regularization Methods for Deep Networks

- Classical **Regularization** Techniques:
  - To **constrain model complexity**, we can introduce weight decay with **L2 (ridge)** or **L1 (lasso)** penalties on weight norms.
- **Weight Decay** in Deep Learning:
  - In deep learning, weight decay is commonly used, but its traditional strengths may be insufficient to prevent networks from interpolating data, and its benefits might rely on early stopping.
- **Inductive Biases** and Generalization:
  - Classical regularizers in deep learning, like weight decay, may contribute to better generalization not by significantly constraining the network's power but by encoding inductive biases compatible with dataset patterns.
- Innovation in DL:
  - DL researchers extend classical regularization techniques, incorporating methods such as **adding noise** to model inputs.
  - **Dropout**, a technique introduced by Srivastava et al. (2014), is a prominent example, even though its theoretical basis remains somewhat mysterious.

# Dropout

```
import torch
from torch import nn
from d2l import torch as d2l
```

# Dropout

- Expectations from a Predictive Model:
  - A good predictive model should perform well on unseen data, aiming to minimize the gap between training and test performance.
- Classical Generalization Theory:
  - Classical generalization theory suggests simplicity as a key factor, including a small number of dimensions, low (inverse) norm of parameters, and smoothness, where the function is not sensitive to small input changes.
- Tikhonov Regularization and Input Noise:
  - Bishop (1995) formalized the idea that training with input noise is equivalent to Tikhonov regularization, connecting smoothness and resilience to input perturbations.
- Dropout Technique for Internal Layers:
  - Srivastava et al. (2014) introduced dropout, a technique involving injecting noise during forward propagation in internal layers of a neural network by zeroing out some fraction of nodes, breaking up co-adaptation.



# Dropout

- Dropout's Link to Sexual Reproduction Analogy:
  - While the original dropout paper links the technique to a sexual reproduction analogy, the enduring success of dropout in training neural networks has made it a standard despite debates about its theoretical justification.
- Challenges in Noise Injection:
  - The challenge in noise injection lies in unbiased application, ensuring that the expected value of each layer, when fixing others, equals its value without noise.
  - Techniques like adding Gaussian noise or zeroing out nodes with subsequent debiasing are common in dropout regularization.
- Standard dropout regularization
  - We usually zeros out some fraction of the nodes in each layer and then debiases each layer by normalizing by the fraction of nodes that were retained (not dropped out).
  - In other words, with dropout probability  $p$ , each intermediate activation  $h$  is replaced by a random variable  $h'$ . By design, the expectation remains unchanged.

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

# Dropout

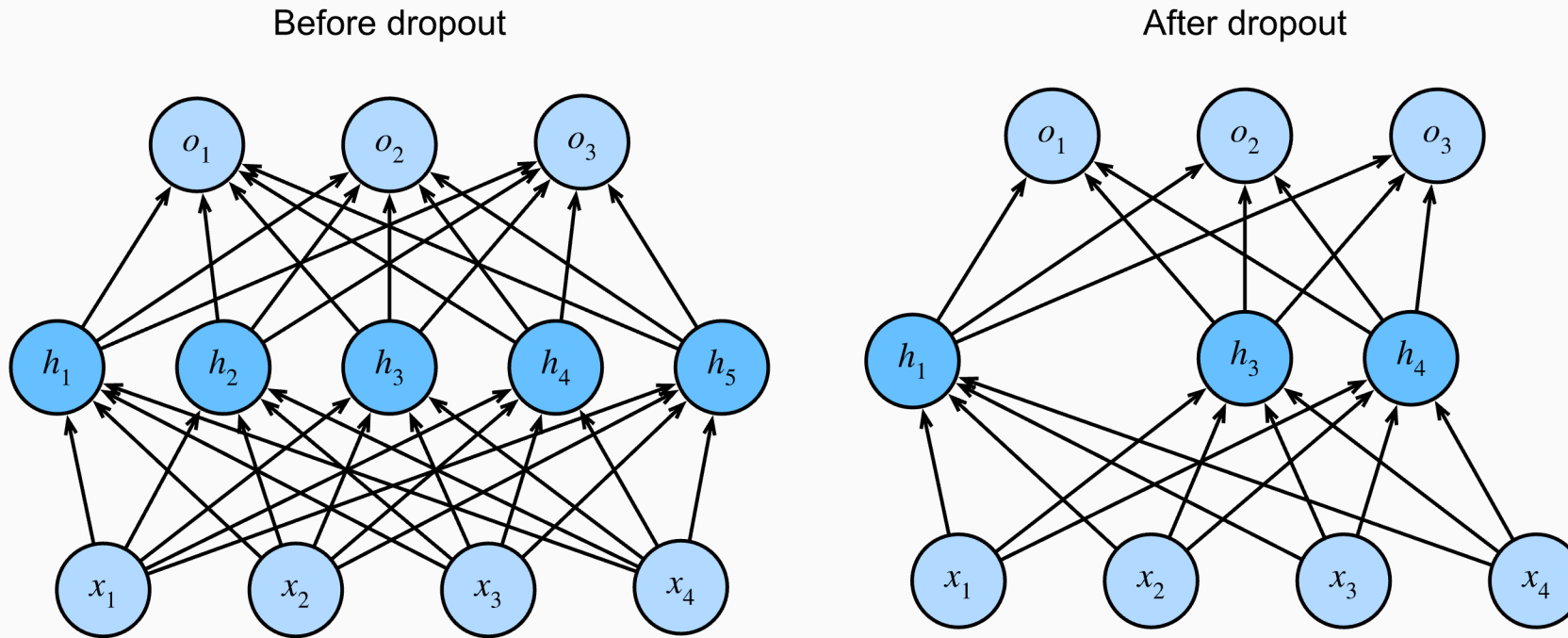


Fig. 5.6.1 MLP before and after dropout

```
import torch
from torch import nn
from d2l import torch as d2l
```

# Lab

- *Lab time!*
- Implementation from Scratch
- Concise Implementation