DEEP LEARNING

HANDOUT: CLASSIFICATION

Hand Written Digital Picture Dataset, Build a model, Error Calculation, Non-Linear model, model complexity, Optimization Method

1.Hand Written Digital Picture Dataset:

We will take 0–9 digital picture recognition as an example to explore how to use machine learning to solve the classification problem.

Dataset: MNIST data set.

- It is handwritten digital dataset, generally scaled to a fixed size, such as 28x28 pixels. For simplicity grayscale information is retained.
- These pictures will be used as the input data ***x***. Also the data is labelled in nature.

1. *Handwritten digital pictures*

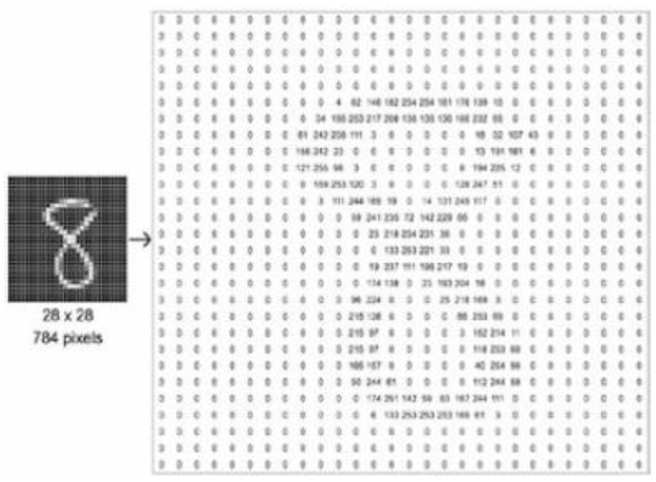
- MNIST data set contains real handwritten pictures of numbers 0–9. Each number has a total of 7,000 pictures, collected from different writing styles.
- Out of it 60k images are used for training and 10k for testing.

2. *MNIST dataset examples*

NOTE:

- Generally, pixel values are integers ranging from 0 to 255 to express color intensity information.
- For example, 0 represents the lowest intensity, and 255 indicates the highest intensity.

- If it is a color picture, each pixel contains the intensity information of the three channels R, G, and B, which, respectively, represent the color intensity of colors red, green, and blue.
- Therefore, each pixel of a color picture is represented by a one-dimensional vector with three elements, which represent the intensity of R, G, and B colors.
- As a result, a color image is saved as a tensor with dimension [h, w, 3].
- A grayscale picture only needs a two-dimensional matrix with shape [h, w] or a three- dimensional tensor with shape [h, w, 1] to represent its information.
- The matrix content of a picture for number 8. It can be seen that the black pixels in the picture are represented by 0 and the grayscale information is represented by 0–255.
- The whiter pixels in the picture correspond to the larger values in the matrix.



3. How a picture is represented[1]

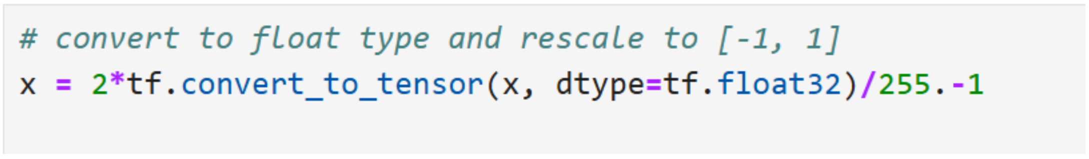
Steps to download, manage, and load the MNIST dataset:

Deep learning frameworks like TensorFlow and PyTorch can easily download, manage, and load the MNIST dataset through a few lines of code. Here we use TensorFlow to automatically download the MNIST dataset and convert it to a Numpy array format:

```python
import os
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, optimizers, datasets
```

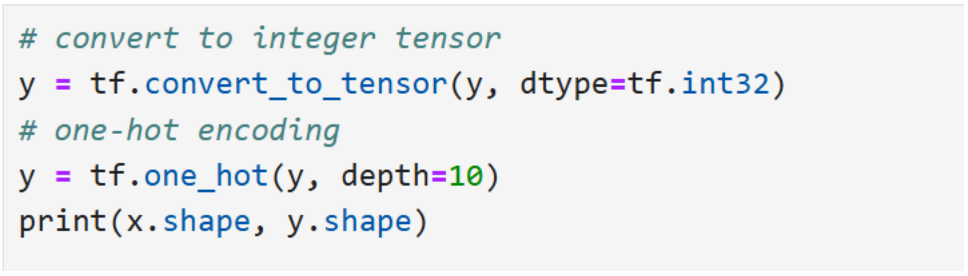
```python
# Load MNIST dataset
(x, y), (x_val, y_val) = datasets.mnist.load_data()
```

# convert to float type and rescale to [-1, 1]



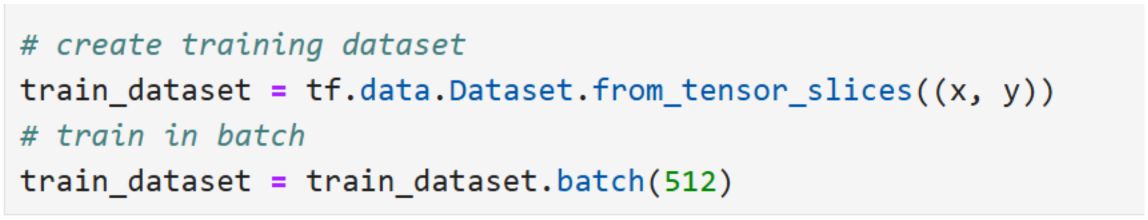
```
# convert to float type and rescale to [-1, 1]
x = 2*tf.convert_to_tensor(x, dtype=tf.float32)/255.-1
```

# convert to integer tensor

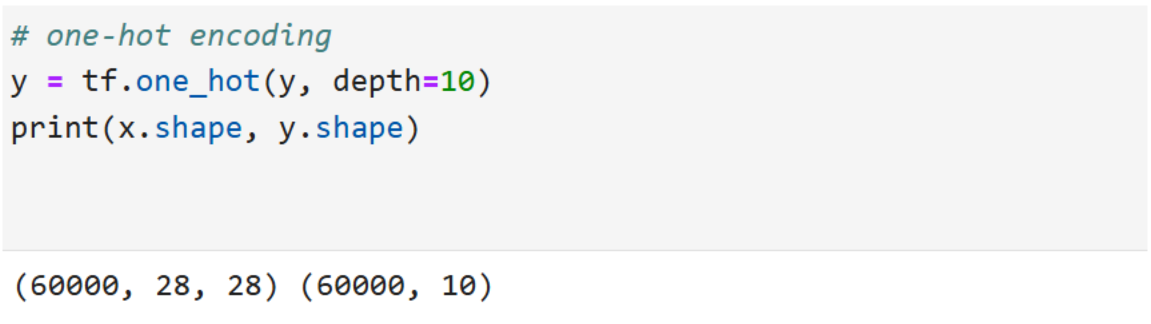


```
# convert to integer tensor
y = tf.convert_to_tensor(y, dtype=tf.int32)
# one-hot encoding
y = tf.one_hot(y, depth=10)
print(x.shape, y.shape)
```

# create training dataset:

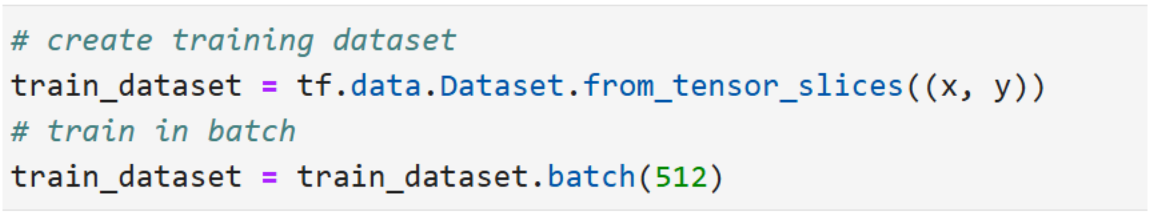


```
# create training dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x, y))
# train in batch
train_dataset = train_dataset.batch(512)
```



```
# one-hot encoding
y = tf.one_hot(y, depth=10)
print(x.shape, y.shape)
```

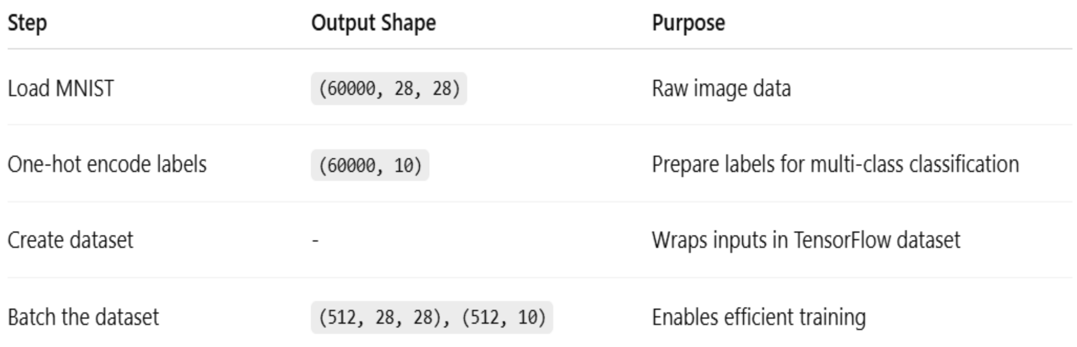
```
(60000, 28, 28) (60000, 10)
```



```
# create training dataset
train_dataset = tf.data.Dataset.from_tensor_slices((x, y))
# train in batch
train_dataset = train_dataset.batch(512)
```

After batching: 60000 samples → ≈ 118 batches of size 512

Summary:



Step	Output Shape	Purpose
Load MNIST	`(60000, 28, 28)`	Raw image data
One-hot encode labels	`(60000, 10)`	Prepare labels for multi-class classification
Create dataset	-	Wraps inputs in TensorFlow dataset
Batch the dataset	`(512, 28, 28), (512, 10)`	Enables efficient training

The load_data () function returns two tuple objects: the first is the training set, and the second is the test set. The first element of the first tuple is the training picture data *X*, and the second element is the corresponding category number *Y*. Each image (Figure 3) in the training set *X* consists of 28×28 pixels, and there are 60,000 images in the training set *X*, so the final dimension of *X* is (60000,28,28). The size of *Y* is (60,000), representing the 60,000 digital numbers ranging from 0–9.
Similarly, the test set contains 10,000 test pictures and corresponding digital numbers with dimensions (10000,28,28) and (10,000) separately. The MNIST dataset loaded from TensorFlow contains images with values from 0 to 255. In machine learning, it is generally desired that the range of data is distributed in a small range around 0. Therefore, we rescale the pixel range to interval [−1, 1], which will benefit the model optimization process.

NOTE:

- We use a matrix of shape [*h*, *w*] to represent a picture.
- For multiple pictures, we can add one more dimension in front and use a tensor of shape [*b*, *h*, *w*] to represent them.
- Here *b* represents the batch size.
- Color pictures can be represented by a tensor with the shape of [*b*, *h*, *w*, *c*], where *c* represents the number of channels, which is 3 for color pictures.
- TensorFlow's Dataset object can be used to conveniently convert a dataset into batches using the batch ( ) function.

Q1: Why do we rescale pixel values to the range [-1, 1]?
Answer: Neural networks train faster and more effectively when the input data is centered around 0. The original MNIST pixel values range from 0 to 255, so we scale them to [-1, 1] to ensure:

- Better convergence
- Stable gradients
- Improved optimization

This normalization improves learning efficiency and is a common best practice.

Q2: What is the shape of the data after loading and converting it using TensorFlow?

Answer: After loading:
(x, y), (x_val, y_val) = datasets.mnist.load_data()

- x shape: (60000, 28, 28) → 60,000 grayscale images
- y shape: (60000,) → 60,000 labels

After one-hot encoding:
y = tf.one_hot(y, depth=10)
y shape: (60000, 10) → labels converted to one-hot vectors (0–9 classes)

Q3: What is the purpose of train_dataset.batch(512)?
Answer: The batch() function splits the dataset into smaller groups of size 512. This is called **batch training**.
Benefits:

- Reduces memory usage
- Increases training speed (vectorized computation)
- Enables stable updates in gradient descent

train_dataset = train_dataset.batch(512)
Each batch now contains 512 images and their corresponding labels.

Q4: Why do we use one-hot encoding for labels?
Answer: One-hot encoding transforms categorical labels into binary vectors. For classification tasks:

- It allows the model to assign probabilities to each class.
- It works well with loss functions like **categorical cross-entropy**.

Example: Label: 3 → One-hot: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

Q5: How does the datasets.mnist.load_data() function simplify dataset preparation?
Answer:
This function:

- Automatically downloads the MNIST dataset
- Returns training and test data as NumPy arrays
- Ensures data is pre-structured as (images, labels)

(x, y), (x_val, y_val) = datasets.mnist.load_data()
This eliminates manual downloading, file parsing, or reshaping, making it ideal for rapid experimentation and teaching.

Q6: Why do we flatten the image?
Answer: Neural networks expect a vector input. So, we reshape [28, 28] into a vector of length 784.

Task: You trained a neural network on MNIST images without rescaling the pixel values (left them between 0–255). You notice the model converges slowly and shows unstable training behaviour.

Question:

Why is this happening, and how can you fix it?

Answer:

Pixel values between 0–255 are too large for most optimizers to handle efficiently. This causes **unstable gradients and poor convergence**.

Fix: Normalize input to a smaller range (e.g., [-1, 1]) using:

x = 2 * tf.convert_to_tensor(x, dtype=tf.float32)/255. - 1

Task: You forgot to use the .batch() method while training with TensorFlow's Dataset object. Your training is slow and inefficient.

Question:

Why is batching important, and how do you apply it?

Answer:

Batching groups multiple samples together to:

- Improve training speed (vectorized processing)
- Use memory efficiently
- Stabilize gradient updates

Apply batching like this:

```
train_dataset = train_dataset.batch(512)
```

2. BUILDING A MODEL:

Linear Model:

- For a single input scalar: We reduce the input vector

$x = [x_1, x_2, \ldots, x_{d_{in}}]^T$, to a single input scalar x, and the model can be expressed as $y = xw + b$.

- For multi-input, single output:

$$y = w^T x + b = [w_1, w_2, w_3, \ldots, w_{d_{in}}] \cdot [x_1\ x_2\ x_3 \vdots x_{d_{in}}] + b$$

- For multi-input, multi-output: y = Wx + b

  where $x \in R^{d_{in}}$, $b \in R^{d_{out}}$, $y \in R^{d_{out}}$, and $W \in R^{d_{out} \times d_{in}}$.

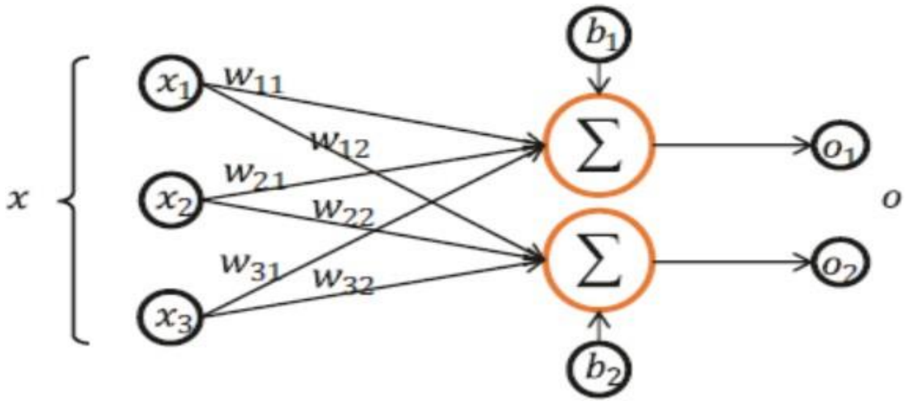
- In batch form: Y = X @ W + b

  where $X \in R^{b \times d_{in}}$, $b \in R^{d_{out}}$, $Y \in R^{b \times d_{out}}$, $W \in R^{d_{in} \times d_{out}}$

Where:

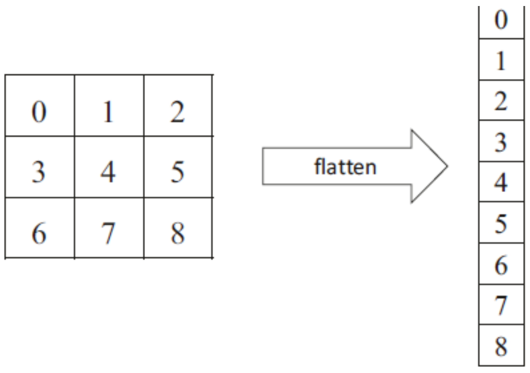
- X: Input matrix [batch_size, input_dim]
- W: Weight matrix [input_dim, output_dim]
- b: Bias vector [output_dim]
- @ represents matrix multiplication
- *din* represents input dimension, and *dout* indicates output dimension.
- X has shape $[b, d_{in}]$, b is the number of samples and *din* is the length of each sample
- W has shape $[d_{in}, d_{out}]$, containing $d_{in} * d_{out}$ parameters.
- Bias vector b has shape d_{out}.
- Since the result of the operation X @ W is a matrix of shape $[b, d_{out}]$, it cannot be directly added to the vector b.
- Therefore, the + sign in batch form needs to support **broadcasting**, that is, expand the vector b into a matrix of shape [*b*, *dout*] by replicating b.

Let us build a Neural network of the same with 3 inputs and 2 outputs:



A grayscale image is stored using a matrix with shape [*h*, *w*], and *b* pictures are stored using a tensor with shape [*b*, *h*, *w*].

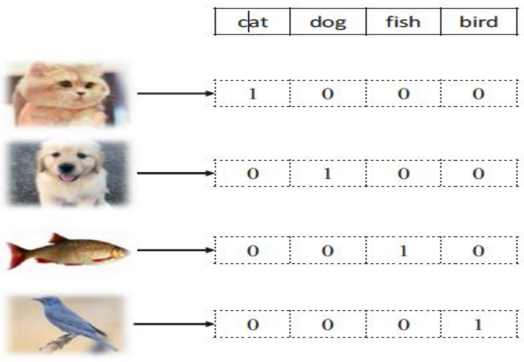
- However, our model can only accept vectors, so we need to **flatten** the [*h*, *w*] matrix into a vector of length [*h* · *w*]. Thus the length of the input features *din* = *h* · *w*.



5. *Flatten a matrix*

- The output can be set to a set of vectors with length d_{out}, where d_{out} is the same as the number of categories.

  For example, if the output belongs to the first category, then the corresponding index is set to 1, and the other positions are set to 0.



	cat	dog	fish	bird
	1	0	0	0
	0	1	0	0
	0	0	1	0
	0	0	0	1

6. *One-hot encoding example*

This encoding method is called **one-hot encoding.:**

One-hot encoding is very sparse. Compared with digital encoding, it needs more storage, so digital encoding is generally used for storage. During calculation, digital encoding is converted to one-hot encoding, which can be achieved through the tf.one_hot() function as follows:

```
y = tf.constant([0,1,2,3]) # digits 0-3
y = tf.one_hot(y, depth=10) # one-hot encoding with length 10
print(y)
```

```
tf.Tensor(
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]], shape=(4, 10), dtype=float32)
```

Task: You are designing a neural network for a classification task with 10 classes. Your final weight matrix W has a shape of [784, 1], and your model outputs shape [b, 1].

Question:

Why is this wrong, and how should you fix the shape of W?

Answer:

A [b, 1] output means the model is producing a single value per sample, which is suitable for regression or binary classification.

But for multi-class classification with 10 classes, you need:

W.shape = [784, 10] $\rightarrow$ Output shape = [b, 10]

Task: While implementing Y = X @ W + b, you encounter a shape mismatch error because b has shape [10], and X @ W returns shape [64, 10].

Question:

What causes this, and how is it resolved?

Answer:

The issue is that b is a 1D vector, but it’s being added to a 2D matrix. TensorFlow uses **broadcasting** to expand b into shape [64, 10] automatically (replicating it across the batch). Ensure that broadcasting is supported. If needed, reshape b as:

b = tf.reshape(b, (1, 10)) # shape becomes broadcastable

Q1: What is the shape of X @ W when:

- **X.shape = [32, 784] (32 samples, 784 features each)**
- **W.shape = [784, 10] (weights for 10 output classes)**

Answer: Matrix multiplication results in:

X @ W $\rightarrow$ shape = [32, 10]

This gives 10 outputs (logits) for each of the 32 input samples.

Q2: Why is broadcasting needed when adding the bias vector b to X @ W?

Answer: After computing X @ W, the result has shape [batch_size, output_dim] (e.g., [32, 10]). But:

b.shape = [10] (1D vector)

To add them:

- TensorFlow **broadcasts** the bias vector b to match shape [32, 10] by replicating it across all samples.

This enables element-wise addition:

```
output = X @ W + b  # broadcasting b
```

Q3: What is the purpose of one-hot encoding in classification tasks?

Answer: One-hot encoding converts class labels (e.g., 0–9) into binary vectors so that:

- The model outputs **a probability for each class**
- A proper **loss function** (like categorical cross-entropy) can compare the predicted and true distributions

Example:

Label: 2 → One-hot: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

Q4: What will be the output of the following code?

```
import tensorflow as tf
y = tf.constant([0, 1, 2, 3])
y_onehot = tf.one_hot(y, depth=10)
print(y_onehot)
```

Answer: The output will be a tensor of shape (4, 10) where each row is a one-hot encoded vector of a digit from 0 to 3:

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]]
```

These represent one-hot encoded values of digits 0 to 3.

3. ERROR CALCULATION

- For classification problems, our goal is to maximize a certain performance metric, such as **accuracy**.
- But when accuracy is used as a loss function, it is in fact indifferentiable.
- As a result, the gradient descent algorithm cannot be used to optimize the model parameters.
- For the error calculation of a classification problem, it is more common to use the **cross- entropy loss function** instead of the mean squared error loss function introduced in the regression problem.

MAJOR ISSUES in Handwritten digital picture recognition problems are:

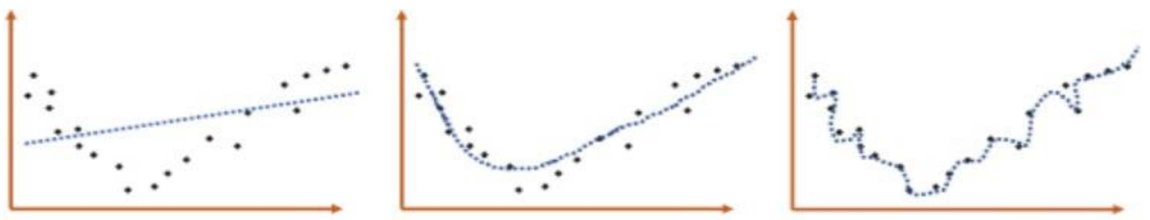
1. A **linear model** is not enough because:
   - It is one of the simplest models in machine learning.
   - It has only a few parameters
   - It can only express linear relationships.

The perception and decision-making of complex brains are far more complex than a linear model.

2. **Complexity:**
   - It is the model ability to approximate complex distributions.
   - The preceding solution only uses a one-layer neural network model composed of a small number of neurons.
   - Compared with the 100 billion neuron interconnection structure in the human brain, its generalization ability is obviously weaker.

Example of model complexity and data distribution:

- The distribution of sampling points with observation errors is plotted. The actual distribution may be a quadratic parabolic model.
- If you use a linear model to fit the data, it is difficult to learn a good model.
- If you use a suitable polynomial function model to learn, such as a quadratic polynomial, you can learn a suitable model.
- But when the model is too complex, such as a ten-degree polynomial, it is likely to overfit and hurt the generalization ability of the model



(a) Linear model (b) Matching model (c) Complex model

Task:
You are training a neural network to recognize handwritten digits. You observe three different models as shown in the diagram:

- (a) Linear model (b) Matching model (c) Complex model

Question:
Which model is likely underfitting, which one fits well, and which one is overfitting? Justify your answer.

Answer:

- (a) **Underfitting** – Model is too simple, can't capture the data pattern.
- (b) **Good fit** – Just enough complexity to capture the data trend.
- (c) **Overfitting** – Too complex; learns noise, not just patterns.

1. What is underfitting in machine learning?

Answer:

Underfitting happens when the model is too simple to learn the underlying pattern of the data. It performs poorly on both training and test data.

2. What is overfitting in machine learning?

Answer:

Overfitting occurs when a model learns the training data too well, including its noise. It performs well on training data but poorly on new, unseen data.

3. Why is a linear model not suitable for handwritten digit recognition?

Answer:

Because a linear model cannot capture complex patterns in images. Handwritten digits require models that can learn non-linear relationships.

4. What is the role of model complexity in machine learning?

Answer:

Model complexity refers to a model's ability to fit complex patterns. Higher complexity can fit more detailed trends, but too much can lead to overfitting.

5. Why is accuracy not used as a loss function in classification tasks?

Answer:

Because accuracy is not differentiable, so gradient-based optimization (like gradient descent) cannot use it. Cross-entropy loss is used instead.

6. What is a good alternative to Mean Squared Error for classification problems?

Answer:

Cross-entropy loss is a better alternative because it is differentiable and works well for classification tasks.

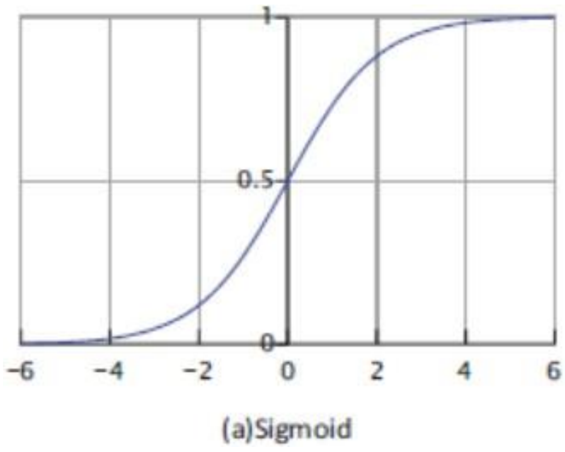
NON-LINEAR MODEL

- Since a linear model is not feasible, we can embed a nonlinear function in the linear model and convert it to a nonlinear model.
- We call this nonlinear function the **activation function**, which is represented by σ:

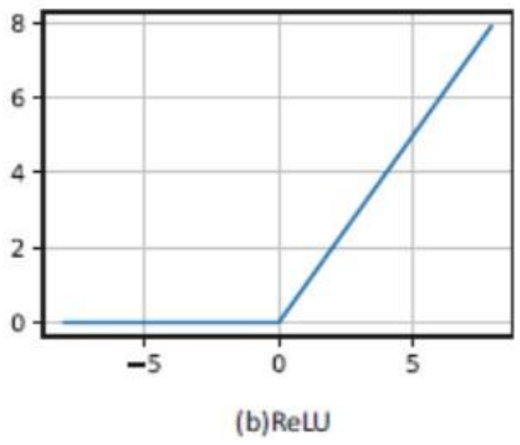
$$o = \sigma(Wx+b)$$

Activation Function:

- Activation functions introduce non-linearities into the network, allowing it to learn complex patterns in the data.
- Common activation functions are ReLU, Sigmoid, Tanh etc.



(a)Sigmoid



(b)ReLU

- The ReLU function only retains the positive part of function $y = x$ and sets the negative part to be zeros.
- It has a unilateral suppression characteristic. Although simple, the ReLU function has excellent nonlinear characteristics, easy gradient calculation, and stable training process.
- It is one of the most widely used activation functions for deep learning models.
- we convert the model to a nonlinear model by embedding the ReLU function:

$$o = ReLU(Wx+ b)$$

Model Complexity:

- To increase the model complexity, we can repeatedly stack multiple transformations such as:

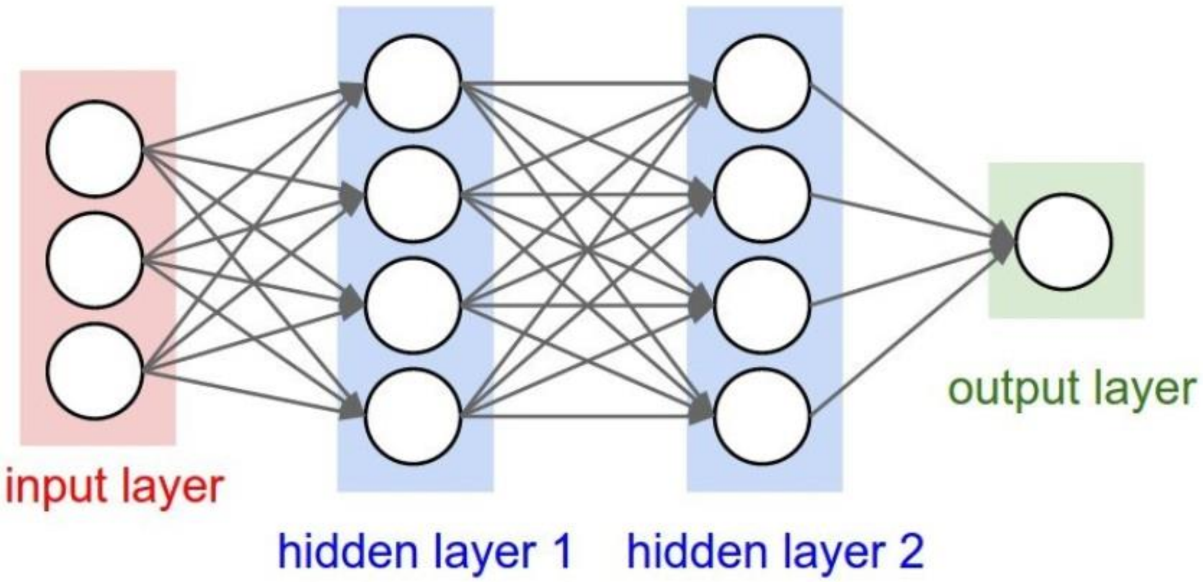
$$h_1 = ReLU(W_1 x + b_1)$$

$$h_2 = ReLU(W_2 h_1 + b_2)$$

$$o = W_3 h_2 + b_3$$

In the preceding equations, we take:

- the output value *h*1 of the first-layer neuron as the input of the second-layer neuron .
- Then take the output *h*2 of the second-layer neuron as the input of the third-layer neuron.
- The output of the last-layer neuron is the model output.



- We call the layer where the input node *x* is located the **input layer**.
- The output of each nonlinear module *hi* along with its parameters *Wi* and *bi* is called a **network layer**.
- In particular, the layer in the middle of the network is called the **hidden layer**, and the last layer is called the **output layer**.
- This network structure formed by the connection of a large number of neurons is called a **neural network.**
- The number of nodes in each layer and the number of layers determine the **complexity** of the neural network.

Task:
You are building a deep neural network model to classify images. You decide to use two hidden layers. Each layer applies a non-linear activation to its input. The architecture and equations are as follows:

$$h_1 = ReLU(W_1 x + b_1)$$
$$h_2 = ReLU(W_2 h_1 + b_2)$$
$$o = W_3 h_2 + b_3$$

Question:

1. Why is it necessary to use a non-linear activation function (like ReLU) in hidden layers?
2. What would happen if all activation functions were removed and only linear operations remained?

Answer:

1. Without non-linearity, a neural network—regardless of how many layers it has—behaves like a single linear transformation. Non-linear activation functions like **ReLU** allow the model to learn **complex, non-linear patterns** in data, which is essential for tasks like image recognition.
2. If activation functions were removed, the model would collapse into a **linear model**, limiting its expressiveness and making it unable to capture the complexity in the data.

1. What is the ReLU activation function?

Answer:

ReLU (Rectified Linear Unit) outputs the input directly if it is positive; otherwise, it outputs zero. It's defined as:

$$ReLU(x) = \max(0, x)$$

2. What is the main difference between ReLU and Sigmoid?

Answer:

- **Sigmoid** outputs values between 0 and 1 and is non-linear but can cause vanishing gradients.
- **ReLU** is faster, does not saturate for large values, and is more commonly used in deep networks.
-

3. What is a hidden layer in a neural network?

Answer:

A layer between the input and output layers where intermediate computations and feature extraction happen.

4. Why do we stack multiple layers in a neural network?

Answer:

To increase **model complexity** and allow the network to learn **hierarchical features** from data.

5. What determines the complexity of a neural network?

Answer:

- Number of **layers**
- Number of **neurons per layer**
- Type of **activation functions** used

OPTIMISATION METHOD: CLASSIFICATION

- Optimization methods similar to regression can also be used to solve classification problems.

For a network model with **single layer:**

- we can directly derive the partial derivative expression of $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$
- and then calculate the gradient for each step and update the parameters *w* and *b* using the gradient descent algorithm.
- As complex nonlinear functions are embedded, the number of network layers and the length of data features also increase.
- The model becomes very complicated, and it is difficult to manually derive the gradient expressions.
- once the network structure changes, the model function and corresponding gradient expressions also change.
- Therefore, it is obviously not feasible to rely on the manual calculation of the gradient.

SOLUTION TO THIS PROBLEM:

- Invention of **deep learning frameworks**.
- With the help of **auto differentiation technology**, deep learning frameworks can build the neural network's computational graph during the calculation of each layer's output corresponding loss function and then **automatically calculate the gradient** of

$$\frac{\partial L}{\partial \theta} \text{ of any parameter } \theta$$

- Users only need to set up the **network structure**, and the gradient will automatically be calculated and updated, which is very convenient and efficient to use.