In [28]:
```python
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense,Conv2D,MaxPool2D,Dropout,Flatten
```

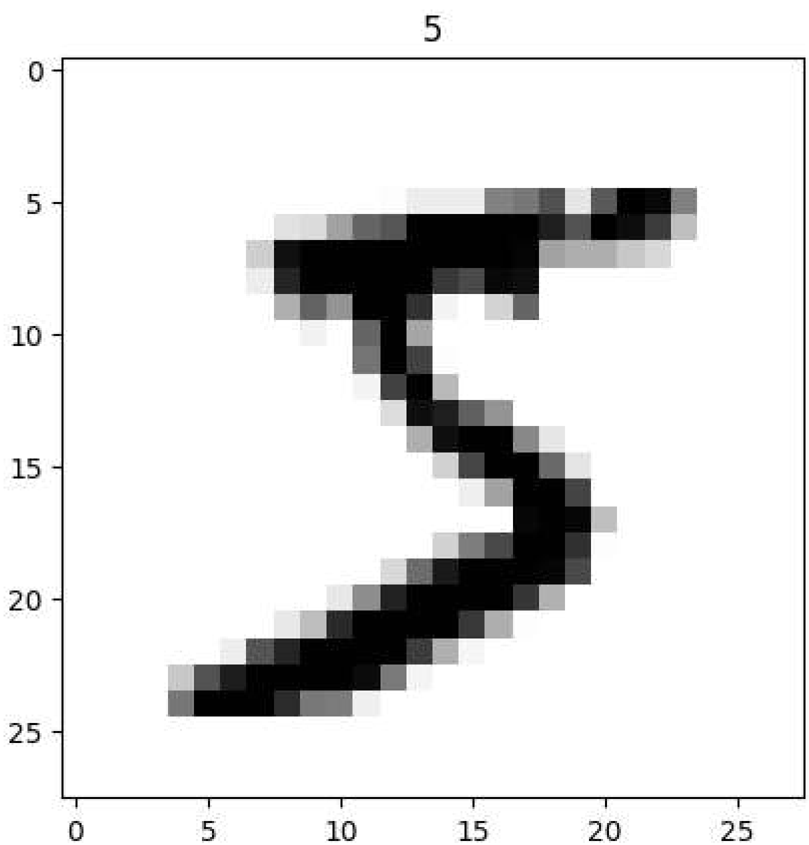
get the data and preprocess it

In [29]:
```python
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print('The current size of our dataset is : \nX_train -> ',X_train.shape,'\ny_train -
```

```
The current size of our dataset is :
X_train ->  (60000, 28, 28)
y_train ->  (60000,)
X_test ->  (10000, 28, 28)
y_test (10000,)
```

In [30]:
```python
def plot_img(i):
    plt.imshow(X_train[i],cmap='binary')
    plt.title(y_train[i])
    plt.show()
```

In [31]:
```python
for i in range(1):
    plot_img(i)
```



preprocessing images

In [32]:
```
# Normalizing the image
'''
Converts pixel values from 0-255 (uint8) to 0.0-1.0 (float32).
This helps neural networks train faster and more accurately.
'''
X_train = X_train.astype(np.float32)/255
X_test = X_test.astype(np.float32)/255

# expand the dimension to (28,28,1)
X_train = np.expand_dims(X_train,-1)
X_test = np.expand_dims(X_test,-1)
```

In [33]:
```
# we have converted our class to one hot vector
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)
```

- Let's say your original `y_train` = `[3, 5, 1]`
- After this line, it becomes:

```python
[[0 0 0 1 0 0 0 0 0 0],  # class 3
 [0 0 0 0 0 1 0 0 0 0],  # class 5
 [0 1 0 0 0 0 0 0 0 0]]  # class 1
```

This is **one-hot encoding** — turning class labels into vectors where only the index of the correct class is `1`, everything else is `0`.

buidling the model

In [34]:
```
model = Sequential()

model.add(Conv2D(32,(3,3),input_shape = (28,28,1),activation = 'relu' ))
model.add(MaxPool2D((2,2)))

model.add(Conv2D(64,(3,3),activation = 'relu' ))
model.add(MaxPool2D((2,2)))

model.add(Flatten())

model.add(Dropout(0.25))

model.add(Dense(10,activation='softmax'))
```

```
C:\Users\anshb\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.12_qbz5n2kfra8p0\LocalCache\local-packages\Python312\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

`model.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='relu'))`

- Conv2D: This is a 2D **convolutional layer**.
- **32**: Number of filters (i.e., different pattern detectors).
- **(3,3)**: Each filter is 3x3 pixels.
- **input_shape=(28,28,1)**: The input image is 28x28 pixels with 1 channel (grayscale).
- **activation='relu'**: ReLU function adds non-linearity and turns all negative values to zero. It helps the model learn complex patterns.

👉 After this layer, you'll get 32 feature maps of size 26x26 (because convolution with (3,3) reduces size a bit unless you use padding).

2. `Conv2D(32, (3, 3), input_shape=(28, 28, 1), activation='relu')`

This adds a **2D Convolution Layer**:

- It uses **32 filters** (or feature detectors)
- Each filter is **3×3** pixels in size
- `input_shape=(28, 28, 1)` tells the model:
  - Height = 28
  - Width = 28
  - Channels = 1 (grayscale)
- `activation='relu'` introduces non-linearity and keeps only positive values

📌 **Purpose:** To extract low-level features like edges, lines, corners

3. `MaxPool2D((2,2))`

This adds a **Max Pooling layer**:

- It takes a **2×2 block** of values and keeps **only the maximum**
- Reduces the image size → helps with **speed & overfitting**

📌 **Purpose:** Downsample the feature maps (make them smaller)

---

4. `Conv2D(64, (3, 3), activation='relu')`

Another convolutional layer, but now with:

- **64 filters**
- Still 3×3 in size

This layer works on the already reduced feature maps from the previous layer.

📌 **Purpose:** Learn more complex patterns (shapes, curves, textures)

6. `Flatten()`

This **flattens the 2D output** from the previous layer into a **1D vector**.

For example, if the output was a 7×7×64 feature map, it becomes a vector of size `7×7×64 = 3136`.

📌 **Purpose:** Prepare data to pass into a Dense (fully connected) layer.

---

7. `Dropout(0.25)`

Dropout randomly **"turns off" 25% of the neurons** during training.

📌 **Purpose:** Prevent **overfitting** by forcing the model to not rely too much on specific neurons.

---

8. `Dense(10, activation='softmax')`

Final **fully connected output layer**:

- `10` neurons → one for each digit class (0 through 9)
- `softmax` activation → converts raw scores into **probabilities** that sum to 1

📌 **Purpose:** Output the predicted class for each image.

Final Flow Summary:

csharp

```csharp
[Input Image 28x28x1]
↓
[Conv2D → feature maps]
↓
[MaxPool → reduced maps]
↓
[Conv2D → deeper features]
↓
[MaxPool → more reduction]
↓
[Flatten → long vector]
↓
[Dropout → regularization]
↓
[Dense → 10 outputs (digit classes)]
```

Let me know if you want a diagram to visualize this flow — happy to generate one!

In [51]:
```python
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dropout (Dropout)	(None, 1600)	0
dense (Dense)	(None, 10)	16,010

Total params: 104,480 (408.13 KB)

Trainable params: 34,826 (136.04 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 69,654 (272.09 KB)

🖼️ What is an image, really?

An image is just a **grid of tiny squares** called **pixels**.

Each pixel has a **value** that represents **how bright or dark** it is.

For **grayscale images** (like handwritten digits):

- Each pixel has **one number** between:
  - `0` (black)
  - `255` (white)
  - Anything in between is a shade of gray.

Example of a small 3×3 image (grayscale):

```lua
[[  0, 128, 255],
 [255, 128,   0],
 [ 64, 192,  64]]
```

This means:

- Top-left pixel is black (0)
- Top-right pixel is white (255)
- Others are shades of gray

Now your case: `28x28` image

That means the image is a **grid of 28 rows and 28 columns** = **784 pixels total**.

Each one has a value from 0–255.

You can think of it like a 2D matrix:

```csharp
[
 [  0,  34, 255, ...,  90],
 [128, 255,  67, ...,  12],
 ...
]
```

After `expand_dims`:

Now the shape becomes:

```python
(60000, 28, 28, 1)
```

That `1` at the end tells the model:

- This is a **grayscale** image
- It has **1 channel** (not 3 like RGB color images)

🔁 Summary:

Before	Shape	Meaning
`X_train`	`(60000, 28, 28)`	60k grayscale images, 2D only
`expand_dims`	→ `(60000, 28, 28, 1)`	Adds channel dimension for CNN

🧱 The 4 Dimensions Explained

Dimension	Example Value	What it means
`batch_size`	60000	How many images you're feeding at once
`height`	28	Height of each image (in pixels)
`width`	28	Width of each image (in pixels)
`channels`	1 (grayscale)	Number of color channels (1 = gray, 3 = RGB)

✅ Why the model *needs* this:

A convolutional layer (`Conv2D`) applies filters across:

- height
- width
- and channels

So the model needs to know:

- Where the image starts and ends (height & width)
- Whether it's color or grayscale (channels)

Without the 4th dimension, the model doesn't know how to apply filters correctly.

⚠️ What happens if you skip it?

If you try to train a CNN with shape `(60000, 28, 28)`:

- It'll throw an error like:

  "Expected 4D input, got 3D instead"

Because it's missing the channel info.

🎨 Color vs Grayscale

Type	Shape of 1 image
Grayscale	`(28, 28, 1)`
RGB Color	`(28, 28, 3)`

So grayscale still needs that `1` to say: "Hey, this is single-channel data."

In [40]:
```
model.compile(optimizer='adam',
              loss=keras.losses.categorical_crossentropy,
              metrics = ['accuracy'])
```

🔧 `optimizer='adam'`

- **Adam** is an optimization algorithm that adjusts learning rates during training.
- It combines the benefits of **AdaGrad** and **RMSProp**.
- Helps the model converge faster and more reliably.

❌ `loss='categorical_crossentropy'`

- This is the **loss function** the model tries to minimize.
- **Categorical cross-entropy** is used for **multi-class classification** where the labels are **one-hot encoded**.
- It measures how far the predicted probabilities are from the true ones.

📊 `metrics=['accuracy']`

- This tells Keras to track **accuracy** during training and evaluation.
- Accuracy = % of predictions that were correct.

In [46]:

```
# calllbacks

from keras.callbacks import EarlyStopping, ModelCheckpoint

# earlystopping

es = EarlyStopping(monitor = 'val_accuracy',min_delta = 0.01 , patience = 4 ,verbose

# model checkpoint

mc = ModelCheckpoint(r'F:\AMRITA ALL SEMESTER\SEMESTER-4\\ML\\project\\image_recog_mr

cb = [es,mc]
```

🔁 `EarlyStopping`

python

```python
es = EarlyStopping(
    monitor='val_accuracy',       # What to track – here, validation accuracy.
    min_delta=0.01,               # Minimum change in the monitored metric to qualify as an improveme
    patience=4,                   # How many epochs to wait for improvement before stopping.
    verbose=1                     # Print a message when training stops early.
)
```

In short:

> *If validation accuracy doesn't improve by at least 1% for 4 epochs in a row, stop training.*

💾 `ModelCheckpoint`

python

```python
mc = ModelCheckpoint(
    './bestmodel.h5',            # File path to save the model.
    monitor='val_accuracy',      # Metric to track – again, validation accuracy.
    verbose=1,                   # Print message each time a new best model is saved.
    save_best_only=True          # Only save the model when it improves (not every epoch).
)
```

In short:

> *Keeps saving the best version of your model based on validation accuracy.*

Model Training

In [47]:
```python
his = model.fit(X_train,y_train,epochs=5,validation_split=0.3,callbacks=cb)
```

```
Epoch 1/5
1310/1313 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9955 - loss: 0.0122
Epoch 1: val_accuracy improved from -inf to 0.99028, saving model to F:\AMRITA ALL SEMESTER\SEMESTER-4\\ML\\project\\image_recog_mnist\bestmodel.h5
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 14s 11ms/step - accuracy: 0.9955 - loss: 0.0122 - val_a
ccuracy: 0.9903 - val_loss: 0.0432
Epoch 2/5
1312/1313 ━━━━━━━━━━━━━━━━━━━╸ 0s 11ms/step - accuracy: 0.9954 - loss: 0.0117
Epoch 2: val_accuracy did not improve from 0.99028
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 16s 12ms/step - accuracy: 0.9954 - loss: 0.0117 - val_a
ccuracy: 0.9903 - val_loss: 0.0413
Epoch 3/5
1312/1313 ━━━━━━━━━━━━━━━━━━━╸ 0s 12ms/step - accuracy: 0.9955 - loss: 0.0127
Epoch 3: val_accuracy did not improve from 0.99028
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 18s 14ms/step - accuracy: 0.9955 - loss: 0.0127 - val_a
ccuracy: 0.9892 - val_loss: 0.0439
Epoch 4/5
1312/1313 ━━━━━━━━━━━━━━━━━━━╸ 0s 12ms/step - accuracy: 0.9960 - loss: 0.0122
Epoch 4: val_accuracy did not improve from 0.99028
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 17s 13ms/step - accuracy: 0.9960 - loss: 0.0122 - val_a
ccuracy: 0.9882 - val_loss: 0.0482
Epoch 5/5
1309/1313 ━━━━━━━━━━━━━━━━━━━╸ 0s 12ms/step - accuracy: 0.9966 - loss: 0.0103
Epoch 5: val_accuracy improved from 0.99028 to 0.99044, saving model to F:\AMRITA ALL
SEMESTER\SEMESTER-4\\ML\\project\\image_recog_mnist\bestmodel.h5
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.sa
ving.save_model(model)`. This file format is considered legacy. We recommend using ins
tead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.sav
e_model(model, 'my_model.keras')`.
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 18s 14ms/step - accuracy: 0.9966 - loss: 0.0103 - val_a
ccuracy: 0.9904 - val_loss: 0.0376
Epoch 5: early stopping
```

🔍 What it does:

1. `model.fit(...)`
   - Starts training the model using your training data (`X_train`, `y_train`).
2. `epochs=5`
   - The model will go through **the entire training data 5 times**.
3. `validation_split=0.3`
   - 30% of the training data will be used for **validation** (to check performance during training, but not used for learning).
   - So, 70% is used to train, and 30% is used to validate after each epoch.
4. `his = ...`
   - Stores the training history (loss, accuracy, etc.) in the variable `his`.

In [49]:
```python
model_S = keras.models.load_model(r'F:\AMRITA ALL SEMESTER\SEMESTER-4\\ML\\project\\i
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built.
`model.compile_metrics` will be empty until you train or evaluate the model.
```

In [50]:
```python
score = model_S.evaluate(X_test,y_test)
# 0 returns the loss
```

```
print(f'mode accuracy is {score[1]}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9907 - loss: 0.0324
mode accuracy is 0.9922000169754028
```