

## Chapter 15 Customized Dataset

---

Spending a year on artificial intelligence is enough to  
make people believe in the existence of God.

–Alan Paley

---

Deep learning has been widely used in various industries such as medicine, biology, and finance, and has been deployed on various platforms such as internet and mobile terminals. When we introduced the algorithm earlier, most of the datasets were commonly used classic datasets. The downloading, loading and preprocessing of the dataset can be completed with a few lines of TensorFlow code, which greatly improves the research efficiency. In actual applications, the datasets are different for different application scenarios. For customized datasets, using TensorFlow to complete data loading, designing excellent network model training process, and deploying the trained model to platforms such as mobile and internet network is an indispensable link for the implementation of deep learning algorithms.

In this chapter, we will take a specific application scenario of image classification as an example to introduce a series of practical technologies such as downloading of customized datasets, data processing, network model design, and transfer learning.

### 15.1 Pokémon Go Dataset

Pokémon Go is a mobile game that uses Augmented Reality (AR) technology to capture and train Pokémon elves outdoors, and use them to fight. The game was launched on Android and IOS in July 2016. Once released, it was sought after by players all over the world. At one time, the server was paralyzed due to too many players. As shown in Figure 15.1, a player scanned the real environment with his mobile phone and collected the virtual Pokémon "Pikachu".




Figure 0.1 Pokémon game screen

We use the Pokémon dataset crawled from the web to demonstrate how to use customized dataset. The Pokémon data set collects a total of 5 eleven creatures: Pikachu, Mewtwo, Squirtle,

Charmander, and Bulbasaur. The information of each elven is shown in Table 15.1, a total of 1168 pictures. There are incorrectly labeled samples in these pictures, so the wrongly labeled samples were artificially eliminated, and a total of 1,122 valid pictures were obtained.

Table 0.1 Pokémon dataset information

Elven	Pikachu	Mewtwo	Squirtle	Charmander	Bulbasaur
Amount	226	239	209	224	224
Sample picture					

Readers can download the provided dataset file by themselves, and after decompression, we can get the root directory named pokemon, which contains 5 subfolders, the file name of each subfolder represents the category name of the pictures, and the corresponding category is stored under each subfolder as shown in Figure 15.2.

Name	Date modified	Type	Size
bulbasaur	5/25/2019 10:11 AM	File folder	
charmander	5/25/2019 10:11 AM	File folder	
mewtwo	5/25/2019 10:11 AM	File folder	
pikachu	5/25/2019 10:11 AM	File folder	
squirtle	5/25/2019 10:11 AM	File folder	

Figure 0.2 Pokémon dataset storage directory

## 15.2 Customized dataset loading

In practical applications, the storage methods of samples and sample labels may vary. For example, in some occasions, all pictures are stored in the same directory, and the category name can be derived from the picture name, such as a picture with a file name of "pikachu\_asxes0132.png". The category information can be extracted from the file name pikachu. The label information of some data samples is saved in a text file in JSON format, and the label of each sample needs to be queried in JSON format. No matter how the dataset is stored, we can always use logic rules to obtain the path and label information of all samples.

We abstract the loading process of customized data into the following steps.

### 15.2.1 Create Code Table

The category of the sample is generally marked with the category name of the string type, but for the neural network, the category name needs to be digitally encoded, and then converted into one-hot encoding or other encoding formats when appropriate. Considering a dataset of  $n$  categories, we randomly code each category into a number  $l \in [0, n - 1]$ . The mapping relationship between category names and numbers is called a coding table. Once created, it generally cannot be changed.

For the storage format of the Pokémon dataset, we create a coding table in the following way.

---

First, traverse all sub-directories under the pokemon root directory in order. For each sub-target, use the category name as the key of the code table dictionary object name2label, and the number of existing key-value pairs in the code table as the label mapping number of the category, and save it into name2label dictionary object. The implementation is as follows:

```
def load_pokemon(root, mode='train'):  
    # Create digital dictionary table  
    name2label = {} # Coding dictionary, "sq...":0  
    # Traverse the subfolders under the root directory and sort them to  
    ensure that the mapping relationship is fixed  
    for name in sorted(os.listdir(os.path.join(root))):  
        # Skip non-folder objects  
        if not os.path.isdir(os.path.join(root, name)):  
            continue  
        # Code a number for each category  
        name2label[name] = len(name2label.keys())  
    ...
```

### 15.2.2 Create sample and label form

After the coding table is determined, we need to obtain the storage path of each sample and its label number according to the actual data storage method, which are represented as two List objects, images and labels, respectively. The images List stores the path string of each sample, and the labels List stores the category number of the sample. The two have the same length, and the elements at the corresponding positions are related to each other.

We store the images and labels information in a csv format file, where the csv file format is a plain text file format with data separated by commas, which can be opened with Notepad or MS Excel software. There are many advantages by storing all sample information in a csv file, such as direct dataset division, batch sampling, etc. The csv file can save the information of all samples in the dataset, or you can create 3 csv files based on the training set, validation set and test set. The content of the resulting csv file is shown in Figure 15.3. The first element of each row stores the storage path of the current sample, and the second element stores the category number of the sample.

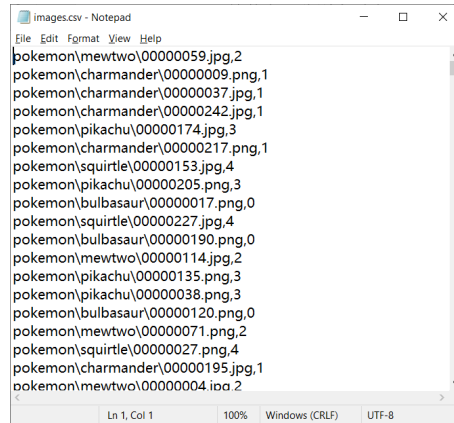


Figure 0.3 Path and label saved in CSV file

The process of creating a csv file is: traverse all pictures in the root directory of pokemon, record the path of the picture, and obtain the code number according to the coding table, and write it into the csv file as a line. The code is as follows:

```
def load_csv(root, filename, name2label):
    # Return images,labels Lists from csv file
    # root: root directory, filename:csv file name, name2label:category
    # coding table
    if not os.path.exists(os.path.join(root, filename)):
        # Create csv file if not exist.
        images = []
        for name in name2label.keys(): # Traverse all subdirectories to get
all pictures
            # Only consider image files with suffix png,jpg,jpeg:
            'pokemon\\mewtwo\\00001.png'
            images += glob.glob(os.path.join(root, name, '*.png'))
            images += glob.glob(os.path.join(root, name, '*.jpg'))
            images += glob.glob(os.path.join(root, name, '*.jpeg'))
            # Print data info: 1167, 'pokemon\\bulbasaur\\00000000.png'
            print(len(images), images)
            random.shuffle(images) # Randomly shuffle
            # Create csv file, and store image path and corresponding label info
            with open(os.path.join(root, filename), mode='w', newline='') as f:
                writer = csv.writer(f)
                for img in images: # 'pokemon\\bulbasaur\\00000000.png'
                    name = img.split(os.sep)[-2]
                    label = name2label[name]
                    # 'pokemon\\bulbasaur\\00000000.png', 0
                    writer.writerow([img, label])
                print('written into csv file:', filename)
    ...
```

After creating the csv file, you only need to read the sample path and label information from

---

the csv file next time, instead of generating the csv file every time, which improves the calculation efficiency. The code is as follows:

```
def load_csv(root, filename, name2label):
    ...
    # At this time there is already a csv file on the file system, read
    directly
    images, labels = [], []
    with open(os.path.join(root, filename)) as f:
        reader = csv.reader(f)
        for row in reader:
            # 'pokemon\\bulbasaur\\00000000.png', 0
            img, label = row
            label = int(label)
            images.append(img)
            labels.append(label)
    # Return image path list and tag list
    return images, labels
```

### 15.2.3 Dataset division

The division of the data set needs to be flexibly adjusted according to the actual situation. When the number of samples in the dataset is large, you can choose a ratio of 80%-10%-10% to allocate to the training set, validation set and test set; when the number of samples is small, for example, the total number of pictures in the Pokémon dataset here is only 1000, if the ratio of the validation set and test set is only 10%, the number of pictures is about 100, so the validation accuracy and test accuracy may fluctuate greatly. For small data sets, although the sample size is small, it is necessary to appropriately increase the ratio of the validation set and test set to ensure accurate test results. Here we set the ratio of validation set and test set to 20%, that is, there are about 200 pictures for validation and testing.

First, call the load\_csv function to load the images and labels list, and load the corresponding pictures and labels according to the current model parameters. Specifically, if the model parameter is train, the first 60% data of images and labels are taken as the training set; if the model parameter is val, the 60% to 80% area data of images and labels are taken as the validation set; if the model parameter is test, the last 20% of images and labels are taken as the test set. The code is implemented as follows:

```
def load_pokemon(root, mode='train'):
    ...
    # Read Label info
    # [file1,file2,], [3,1]
    images, labels = load_csv(root, 'images.csv', name2label)
    # Dataset division
    if mode == 'train': # 60%
        images = images[:int(0.6 * len(images))]
```

---

```

        labels = labels[:int(0.6 * len(labels))]
elif mode == 'val': # 20% = 60%→80%
    images = images[int(0.6 * len(images)):int(0.8 * len(images))]
    labels = labels[int(0.6 * len(labels)):int(0.8 * len(labels))]
else: # 20% = 80%→100%
    images = images[int(0.8 * len(images)):]
    labels = labels[int(0.8 * len(labels)):]

return images, labels, name2label

```

It should be noted that the dataset division scheme for each run needs to be fixed to prevent the use of test set for training, resulting in inaccurate model generalization performance.

## 15.3 Hands-on Pokémon dataset

After introducing the loading process of the custom dataset, let's load and train the Pokémon data set.

### 15.3.1 Create Dataset Object

First, return the images, labels and coding table information through the `load_pokemon` function as follows:

```

# Load the pokemon dataset, specify to load the training set
# Return the sample path list of the training set, the label number list
and the coding table dictionary
images, labels, table = load_pokemon('pokemon', 'train')
print('images:', len(images), images)
print('labels:', len(labels), labels)
print('table:', table)

```

Construct a Dataset object, and complete the random break up, preprocessing and batch operation of the dataset. The code is as follows:

```

# images: string path
# labels: number
db = tf.data.Dataset.from_tensor_slices((images, labels))
db = db.shuffle(1000).map(preprocess).batch(32)

```

When we use `tf.data.Dataset.from_tensor_slices` to construct the dataset, the passed-in parameter is a tuple composed of images and labels, so when the `db` object is iterated, the tuple object of  $(X_i, Y_i)$  is returned, where  $X_i$  is the image tensor of the  $i$ th batch,  $Y_i$  is the image label data of the  $i$ th batch. We can view the image samples of each traversal through TensorBoard visualization as follows:

```

# Create TensorBoard summary object
writer = tf.summary.create_file_writer('logs')
for step, (x,y) in enumerate(db):
    # x: [32, 224, 224, 3]

```

---

```

# y: [32]
with writer.as_default():
    x = denormalize(x) # Denormalize
    # Write in image data
    tf.summary.image('img',x,step=step,max_outputs=9)
    time.sleep(5) # Delay 5s

```

### 15.3.2 Data preprocessing

We complete the preprocessing of the data by calling the `.map(preprocess)` function when constructing the data set. Since our images list currently only saves the path information of all images, not the content tensor of the image, it is necessary to complete the image reading and tensor conversion in the preprocessing function.

For the preprocess function  $(x,y) = \text{preprocess}(x,y)$ , its incoming parameters need to be saved in the same format as the parameters given when creating the Dataset, and the return parameters need to be saved in the same format as the incoming parameters. In particular, we pass in the  $(\mathbf{x}, \mathbf{y})$  tuple object when constructing the dataset, where  $\mathbf{x}$  is the path list of all pictures and  $\mathbf{y}$  is the label number list of all pictures. Considering that the location of the map function is `db = db.shuffle(1000).map(preprocess).batch(32)`, then the incoming parameters of preprocess are  $(x_i, y_i)$ , where  $x_i$  and  $y_i$  are respectively the  $i$ -th picture path string and label number. If the location of the map function is `db = db.shuffle(1000).batch(32).map(preprocess)`, then the incoming parameters of preprocess are  $(\mathbf{x}_i, \mathbf{y}_i)$ , where  $\mathbf{x}_i$  and  $\mathbf{y}_i$  are the path and tag list of the  $i$ -th batch respectively. The code is as below:

```

def preprocess(x,y): # preprocess function
    # x: image path, y: image coding number
    x = tf.io.read_file(x) # Read image
    x = tf.image.decode_jpeg(x, channels=3) # Decode image
    x = tf.image.resize(x, [244, 244]) # Resize to 244x244

    # Data agumentation
    # x = tf.image.random_flip_up_down(x)
    x = tf.image.random_flip_left_right(x) # flip left and right
    x = tf.image.random_crop(x, [224, 224, 3]) # Crop to 224x224
    # Convert to tensor and [0, 1] range
    # x: [0,255]=> 0~1
    x = tf.cast(x, dtype=tf.float32) / 255.
    # 0~1 => D(0,1)
    x = normalize(x) # Normalize
    y = tf.convert_to_tensor(y) # To tensor

    return x, y

```

Considering that the scale of our dataset is very small, in order to prevent overfitting, we

---

have done a small amount of data enhancement transformation to obtain more data. Finally, we scale the pixel values in the range of 0~255 to the range of 0~1, and normalize the data, and map the pixels to the distribution around 0, which is beneficial to the optimization of the network. Finally, the data is converted to tensor data and returned. At this time, the data returned will be the tensor data in batch form when iterating over the db object.

The standardized data is suitable for network training and prediction, but when visualizing, the data needs to be mapped back to the range of 0~1. The reverse process of standardization and standardization is as follows:

```
# The mean and std here are calculated based on real data, such as ImageNet
img_mean = tf.constant([0.485, 0.456, 0.406])
img_std = tf.constant([0.229, 0.224, 0.225])
def normalize(x, mean=img_mean, std=img_std):
    # Normalization function
    # x: [224, 224, 3]
    # mean: [224, 224, 3], std: [3]
    x = (x - mean)/std
    return x

def denormalize(x, mean=img_mean, std=img_std):
    # Denormalization function
    x = x * std + mean
    return x
```

Using the above method, distribute the Dataset objects that create the training set, validation set, and test set. Generally speaking, the validation set and test set do not directly participate in the optimization of network parameters, and there is no need to randomly break the order of samples.

```
batchsz = 128
# Create training dataset
images, labels, table = load_pokemon('pokemon',mode='train')
db_train = tf.data.Dataset.from_tensor_slices((images, labels))
db_train = db_train.shuffle(1000).map(preprocess).batch(batchsz)
# Create validation dataset
images2, labels2, table = load_pokemon('pokemon',mode='val')
db_val = tf.data.Dataset.from_tensor_slices((images2, labels2))
db_val = db_val.map(preprocess).batch(batchsz)
# Create testing dataset
images3, labels3, table = load_pokemon('pokemon',mode='test')
db_test = tf.data.Dataset.from_tensor_slices((images3, labels3))
db_test = db_test.map(preprocess).batch(batchsz)
```

### 15.3.3 Create Model

The mainstream network models such as VGG13 and ResNet18 have been introduced and



---

implemented before, and we will not repeat the specific implementation details of the model here. Commonly used network models are implemented in the `keras.applications` module, such as VGG series, ResNet series, DenseNet series, MobileNet series, etc., and these model networks can be created with only one line of code. E.g:

```
# Load the DenseNet network model, remove the last fully connected layer,
and set the last pooling layer to max pooling
net = keras.applications.DenseNet121(include_top=False, pooling='max')
# Set trainable to True, i.e. DenseNet's parameters will be updated.
net.trainable = True
newnet = keras.Sequential([
    net, # Remove last layer of DenseNet121
    layers.Dense(1024, activation='relu'), # Add fully connected layer
    layers.BatchNormalization(), # Add BN layer
    layers.Dropout(rate=0.5), # Add Dropout layer
    layers.Dense(5) # Set last layer node to 5 according to output
categories
])
newnet.build(input_shape=(4, 224, 224, 3))
newnet.summary()
```

The DenseNet121 model is used to create the network. Since the output node of the last layer of DenseNet121 is designed to be 1000, we remove the last layer of DenseNet121 and add a fully connected layer with the number of output nodes of 5 according to the number of categories of the customized dataset. The whole setup is repackaged into a new network model through Sequential containers, where `include_top=False` indicates that the last fully connected layer is removed, and `pooling='max'` indicates that the last Pooling layer of DenseNet121 is designed as Max Pooling. The network model structure is shown in Figure 15.4.

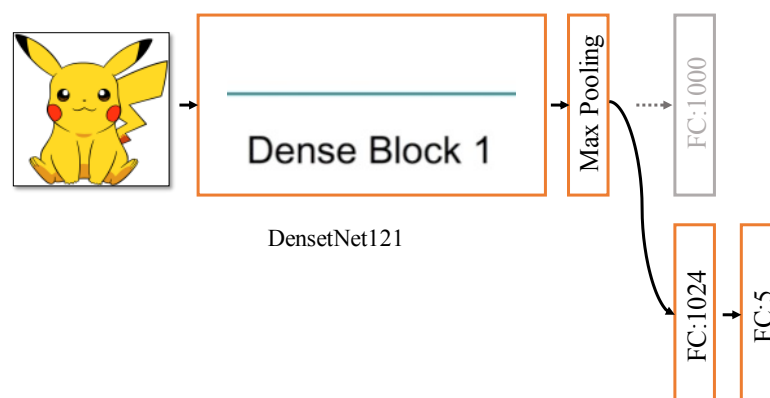


Figure 0.4 Model structure diagram

### 15.3.4 Network training and testing

We directly use the `Compile&Fit` method provided by Keras to compile and train the

---

network. The optimizer uses the most commonly used Adam optimizer, the error function uses the cross-entropy loss function, and sets `from_logits=True`. The measurement index that we pay attention to during the training process is the accuracy rate. The network model compile code is as follows:

```
# Compile model
newnet.compile(optimizer=optimizers.Adam(lr=1e-3),
               loss=losses.CategoricalCrossentropy(from_logits=True),
               metrics=['accuracy'])
```

Use the fit function to train the model on the training set. Each iteration of Epoch tests a validation set. The maximum number of training Epochs is 100. In order to prevent overfitting, we use Early Stopping technology, and pass Early Stopping into the callbacks parameter of the fit function as below:

```
# Model training, support early stopping
history = newnet.fit(db_train, validation_data=db_val, validation_freq=1, epochs=100,
                    callbacks=[early_stopping])
```

where `early_stopping` is the standard `EarlyStopping` class. The indicator it monitors is the accuracy of the validation set. If the measurement result of the validation set does not increase by 0.001 for three consecutive times, the `EarlyStopping` condition is triggered and the training ends.

```
# Create Early Stopping class
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    min_delta=0.001,
    patience=3
)
```

We draw the training accuracy rate, validation accuracy rate and the accuracy rate obtained on the final test set in the training process as a curve, as shown in Figure 15.5. It can be seen that the training accuracy rate has increased rapidly and maintained at a high state, but the validation accuracy rate is relatively poor, and at the same time, it has not been greatly improved. The Early Stopping condition is triggered, and the training process is quickly terminated. The network has a very serious overfitting problem.

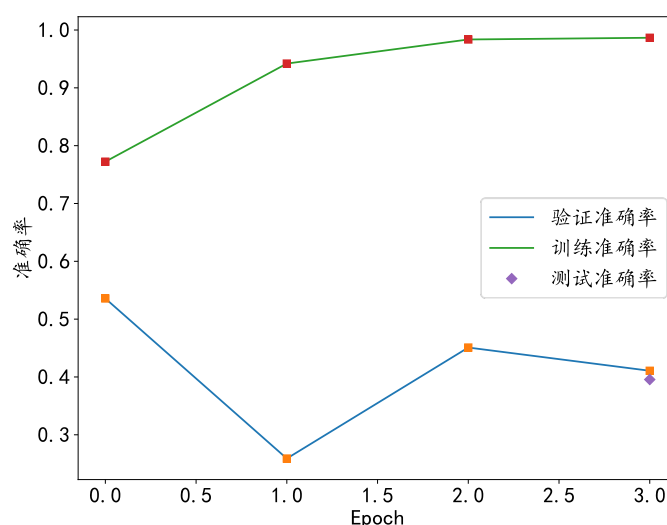


Figure 0.5 Train DenseNet

So why does overfitting occur? The number of layers of the DenseNet121 model has reached 121, and the number of parameters has reached 7 million, which is a large network model, while our dataset has only about 1,000 samples. According to experience, this is far from enough to train such a large-scale network model, and it is extremely prone to overfitting. In order to reduce overfitting, a network model with a shallower number of layers and fewer parameters can be used, or regularization items can be added, or even the size of the data set can be increased. In addition to these methods, another effective method is transfer learning technology.

## 15.4 Transfer learning

### 15.4.1 Principles of Transfer Learning

Transfer Learning is a research direction of machine learning. It mainly studies how to transfer the knowledge learned on task A to task B to improve the generalization performance on task B. For example, task A is a cat and dog classification problem, and a classifier needs to be trained to better distinguish pictures of cats and dogs, and task B is a cattle and sheep classification problem. It can be found that there is a lot of shared knowledge in task A and task B. For example, these animals can be distinguished from the aspects of hair, body shape, shape, and hair color. Therefore, the classifier obtained in task A has mastered this part of knowledge. When training the classifier of task B, you don't need to start training from scratch, instead you can train or fine-tune the knowledge obtained on task A, which is very similar to the idea of "standing on the shoulders of giants". By transferring the knowledge learned on task A, training the classifier on task B can use fewer samples and lower training costs, and obtain good performance.

We introduce a relatively simple, but very commonly used transfer learning method: network fine-tuning technology. For convolutional neural networks, it is generally believed that it can extract features layer by layer. The abstract feature extraction ability of the network at the end of the layer is stronger. The output layer generally uses the fully connected layer with the same

number of output nodes as the classification network as the probability distribution prediction . For similar tasks A and B, if their feature extraction methods are similar, the previous layers of the network can be reused, and the following layers can be trained from scratch according to specific task settings.

As shown in Figure 15.6, the network on the left is trained on task A to learn the knowledge of task A. When migrating to task B, the parameters of the early layers of the network model can be reused, and the later layers can be replaced with new networks and start training from scratch. We call the model trained on task A a pre-trained model. For image classification, the model pre-trained on the ImageNet dataset is a better choice.

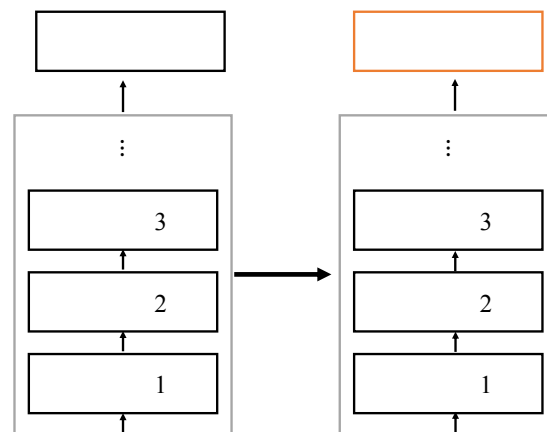


Figure 0.6 Diagram of neural network transfer learning

## 15.4.2 Hands-on Transfer Learning

Based on DenseNet121, we initialize the network with the model parameters pre-trained on the ImageNet dataset, remove the last fully connected layer, add a new classification sub-network, and set the number of output nodes in the last layer to 5.

```
# Load DenseNet model, remove last layer, set last pooling layer as
max pooling
# Initilize with pre-trained parameters
net = keras.applications.DenseNet121(weights='imagenet', include_top=False,
pooling='max')
# Set trainable to False, i.e. fix the DenseNet parameters
net.trainable = False
newnet = keras.Sequential([
    net, # DenseNet121 with last layer
    layers.Dense(1024, activation='relu'), # Add fully connected layer
    layers.BatchNormalization(), # Add BN layer
    layers.Dropout(rate=0.5), # Add Dropout layer
    layers.Dense(5) # Set the nodes of last layer to 5
])
newnet.build(input_shape=(4,224,224,3))
newnet.summary()
```

---

When the above code creates DenseNet121, the pre-trained DenseNet121 model object can be returned by setting the `weights='imagenet'` parameter, and the reused network layer and the new sub-classification network are repackaged into a new model `newnet` through the `Sequential` container. In the fine-tuning stage, the parameters of the DenseNet121 part can be fixed by setting `net.trainable = False`, that is, the DenseNet121 part of the network does not need to update the parameters, so only the newly added sub-classification network needs to be trained, which greatly reduces the amount of parameters actually involved in training. Of course, you can also train all parameters like a normal network by setting `net.trainable = True`. Even so, because the reused part of the network has initialized with a good parameter state, the network can still quickly converge and achieve better performance.

Based on the pre-trained DenseNet121 model, we plot the training accuracy, validation accuracy, and test accuracy in Figure 15.7. Compared with training from scratch approach, with the help of transfer learning, the network only needs a few samples to achieve better performance, and the improvement is very significant.

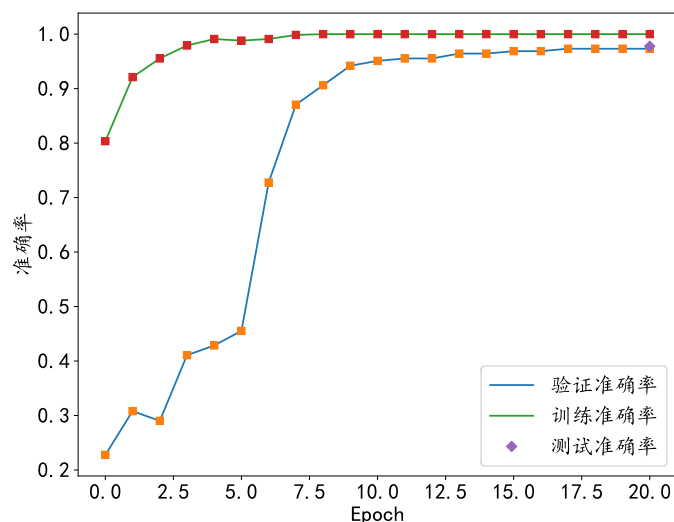


Figure 0.7 Pre-trained DenseNet performance