Intelligent Systems Lab

19131A05B0

L.THRILOK NAIDU

CSE-2

WEEK-9

Aim: Write a program to implement the building blocks of CNN.

Description:

Convolutional neural networks are a specialized type of artificial neural networks that use a mathematical operation called convolution in place of general matrix multiplication in at least one of their layers.[13] They are specifically designed to process pixel data and are used in image recognition and processing.

A convolutional neural network consists of an input layer, hidden layers and an output layer. In any feed-forward neural network, any middle layers are called hidden because their inputs and outputs are masked by the activation function and final convolution. In a convolutional neural network, the hidden layers include layers that perform convolutions. Typically, this includes a layer that performs a dot product of the convolution kernel with the layer's input matrix. This product is usually the Frobenius inner product, and its activation function is commonly ReLU. As the convolution kernel slides along the input matrix for the layer, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers, fully connected layers, and normalization layers.

Convolutional layers

In a CNN, the input is a tensor with a shape: (number of inputs) × (input height) × (input width) × (input channels). After passing through a convolutional layer, the image becomes abstracted to a feature map, also called an activation map, with shape: (number of inputs) × (feature map height) × (feature map width) × (feature map channels).

Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus.[14] Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural networks can be used to learn features and classify data, this architecture is generally impractical for larger inputs such as high-resolution images. It would require a very high number of neurons, even in a shallow architecture, due to the large input size of images, where each pixel is a relevant input feature. For instance, a fully connected layer for a (small) image of size 100 × 100 has 10,000 weights for each neuron in the second layer. Instead, convolution reduces the number of free parameters, allowing the network to be deeper.[15] For example, regardless of image size, using a 5 × 5 tiling region, each with the same shared weights, requires only 25 learnable parameters. Using regularized weights over fewer parameters avoids the vanishing gradients and exploding gradients problems seen during backpropagation in traditional neural networks.[16][17] Furthermore, convolutional neural networks are ideal for data with a grid-like topology (such as images) as spatial relations between separate features are taken into account during convolution and/or pooling.

Pooling layers

Convolutional networks may include local and/or global pooling layers along with traditional convolutional layers. Pooling layers reduce the dimensions of data by combining the outputs of neuron clusters at one

layer into a single neuron in the next layer. Local pooling combines small clusters, tiling sizes such as 2 × 2 are commonly used. Global pooling acts on all the neurons of the feature map.[18][19] There are two common types of pooling in popular use: max and average. Max pooling uses the maximum value of each local cluster of neurons in the feature map, while average pooling takes the average value.

Fully connected layers

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is the same as a traditional multilayer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

Receptive field

In neural networks, each neuron receives input from some number of locations in the previous layer. In a convolutional layer, each neuron receives input from only a restricted area of the previous layer called the neuron's receptive field. Typically, the area is a square (e.g. 5 by 5 neurons). Whereas, in a fully connected layer, the receptive field is the entire previous layer. Thus, in each convolutional layer, each neuron takes input from a larger area in the input than previous layers. This is due to applying the convolution over and over, which takes into account the value of a pixel, as well as its surrounding pixels. When using dilated layers, the number of pixels in the receptive field remains constant, but the field is more sparsely populated as its dimensions grow when combining the effect of several layers.

Weights

Each neuron in a neural network computes an output value by applying a specific function to the input values received from the receptive field in the previous layer. The function that is applied to the input values is determined by a vector of weights and a bias (typically real numbers). Learning consists of iteratively adjusting these biases and weights.

The vectors of weights and biases are called filters and represent particular features of the input (e.g., a particular shape). A distinguishing feature of CNNs is that many neurons can share the same filter. This reduces the memory footprint because a single bias and a single vector of weights are used across all receptive fields that share that filter, as opposed to each receptive field having its own bias and vector weighting. Common types of pooling in popular use: max and average. *Max pooling* uses the maximum value of each local cluster of neurons in the feature map, while *average pooling* takes the average value.

Fully connected layers

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is the same as a traditional multilayer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

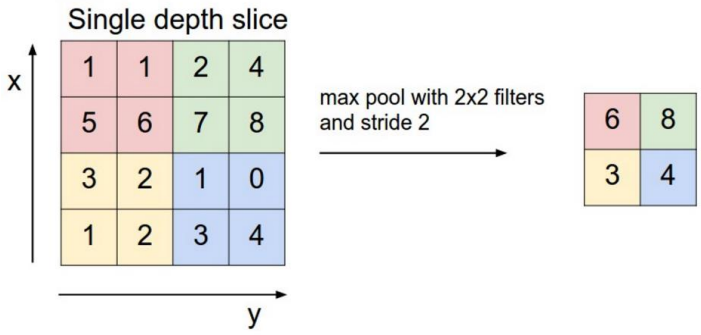
Receptive field

In neural networks, each neuron receives input from some number of locations in the previous layer. In a convolutional layer, each neuron receives input from only a restricted area of the previous layer called the neuron's *receptive field*. Typically, the area is a square (e.g. 5 by 5 neurons). Whereas, in a fully connected layer, the receptive field is the *entire previous layer*. Thus, in each convolutional layer, each neuron takes input from a larger area in the input than previous layers. This is due to applying the convolution over and over, which takes into account the value of a pixel, as well as its surrounding pixels. When using dilated layers, the number of pixels in the receptive field remains constant, but the field is more sparsely populated as its dimensions grow when combining the effect of several layers.

Weights

Each neuron in a neural network computes an output value by applying a specific function to the input values received from the receptive field in the previous layer. The function that is applied to the input values is determined by a vector of weights and a bias (typically real numbers). Learning consists of iteratively adjusting these biases and weights.

The vectors of weights and biases are called *filters* and represent particular features of the input (e.g., a particular shape). A distinguishing feature of CNNs is that many neurons can share the same filter. This reduces the memory footprint because a single bias and a single vector of weights are used across all receptive fields that share that filter, as opposed to each receptive field having its own bias and vector weighting.



Code:

```
import tensorflow as tf

import matplotlib.pyplot as plt

import numpy as np


mnist=tf.keras.datasets.mnist

(x_train,y_train),(x_test,y_test)=mnist.load_data()

print(y_train[5])
```

Input:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

mnist=tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)=mnist.load_data()
print(y_train[5])
```

```
2
```

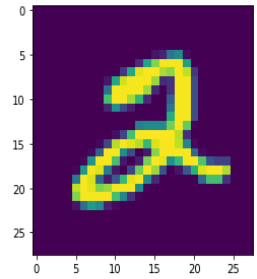
Code:

```
plt.imshow(x_train[5])

plt.show()
```

Input:

```
plt.imshow(x_train[5])
plt.show()
```



Code:

```
model=tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128,activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128,activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(10,activation=tf.nn.softmax))
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.fit(x_train,y_train,epochs=3)
```

Input:

```
Epoch 1/3
1875/1875 [==============================] - 7s 4ms/step - loss: 1.7110 - accuracy: 0.8739
Epoch 2/3
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3724 - accuracy: 0.9252
Epoch 3/3
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2285 - accuracy: 0.9435
<keras.callbacks.History at 0x7fbbd55f4c50>
```

Code:

```
predictions=model.predict(x_test)
print(predictions[3])
```

Input:

```
predictions=model.predict(x_test)
print(predictions[3])
```

```
313/313 [==============================] - 1s 2ms/step
[9.9995595e-01 9.7355951e-12 3.6737600e-08 2.2135872e-08 1.8755960e-05
 3.2397147e-06 1.9047271e-05 9.7340687e-09 2.9214143e-06 3.9205634e-08]
```

Code:

```
print(np.argmax(predictions[3]))
```

Input:

```
print(np.argmax(predictions[3]))
```

```
0
```

Code:

plt.imshow(x_test[3])

plt.show()

Input:

```
plt.imshow(x_test[3])
plt.show()
```