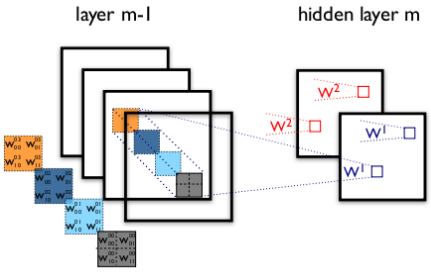
Convolutional Neural Networks

on MNIST dataset
Group:LGD
Members:
Xiaodi Sun(xs2315)
Wenyi Lyu(wl2618)
Tiancheng Zheng(tz2349)

1. Explanation of how CNN works

Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs. According to work of biologist, 'visual cortex', which contains a complex arrangement of cells, is sensitive to small sub-regions of the visual field, called a receptive field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation in images.



To simulate the functionality of visual cortex, CNN utilizes the following two concepts:

a) Sparse Connectivity

CNNs exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. As shown in the figure, each neuron in hidden layer just accept the local regions from former layer as inputs, instead of a whole image.

However, stacking many such layers leads to (non-linear) filters that become 'global' (i.e. responsive to a larger region of pixel space).

b) Shared Weights

In addition, in CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map. As shown in the figure, a feature map is corresponding to the same parameters (shown in same color).

Replicating units in this way allows for features to be detected regardless of their position in the visual field. Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt.

The formal formulation of a single filter in CNN is:

$$h_{ij}^k = \tanh((W^k * x)_{ij} + b_k).$$

in which *k* is the index of kernel, *i* and *j* are position along the two axis, tanh() is activate function (note that it can be many other choices, e.g. ReLU(), etc.)

Sliding the filter along two axis of an image to get feature map. After several turns, the feature map is vectorized and fed into fully connected layers.

2. Explanation of each parameters with its functions

Keras is a popular deep learning library, using tensorflow or theano as backends. Here I choose tensorflow.

Key functions or class used are explained here:

Sequential(): class to add each layer as a block onto the network

Parameters: **optimizer**: String (name of optimizer) or optimizer object
loss: String (name of objective function) or objective function
metrics: List of metrics to be evaluated by the model during training and testing
sample_weight_mode: If you need to do timestep-wise sample weighting (2D weights), set this to "temporal".
weighted_metrics: List of metrics to be evaluated and weighted by sample_weight or class_weight during training and testing.

Conv2D(): assign the number, the size of filters, also activate function can be assigned together

Parameters: **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
kernel_size: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window.
activation: Activation function to use.

MaxPooling2D(): assign the size of pooling area

Parameters: **pool_size**: Integer, size of the max pooling windows.

Dense(): assign the number of neurons in MLP, also activate function can be assigned together

Parameters: **units**: Positive integer, dimensionality of the output space.
activation: Activation function to use.

ImageDataGenerator(): generate new data images based on datasets, operations include shifting, flipping, and rotation and so on

Parameters: **featurewise_center**: Boolean. Set input mean to 0 over the dataset, feature-wise.
samplewise_center: Boolean. Set each sample mean to 0.
featurewise_std_normalization: Boolean. Divide inputs by std of the dataset, feature-wise.
samplewise_std_normalization: Boolean. Divide each input by its std.
zca_whitening: Boolean. Apply ZCA whitening.
rotation_range: Int. Degree range for random rotations.
width_shift_range: randomly shift images horizontally.
height_shift_range: randomly shift images vertically.
horizontal_flip: randomly flip images horizontally.
vertical_flip: randomly flip images vertically.

3. Code logic

Final codes in **xs2315_XiaodiSun_GroupProject.py**. Note that as changing into any stage below is easy, I do not hand in them as individual files.

The codes can be divided into main parts:

i. Preparation: set the parameters, read in the data sets and transform into suitable form;

```
batch_size = 128
num_classes = 10
epochs = 100
data_augmentation = False

# input image dimensions
img_rows, img_cols = 28, 28

# The data, split between train and test sets:
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

ii. Model definition: define the structure of CNN;

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

iii. Training and testing: prepare a data augmentation step, then train on training set, finally, evaluate on testing set.

```
datagen = ImageDataGenerator(featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=False,
    vertical_flip=False)

model.fit_generator(datagen.flow(x_train, y_train,
                                 batch_size=batch_size),
                    samples_per_epoch=len(x_train),
                    epochs=epochs,
                    workers=4)

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

4. **Improve the model (including starting model and how we chose that and explanation of how we reached final model from starting model with intermediate stages as well)**
   a) **Stage One**: The starting model is from the website: https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py
   And fix the bug in

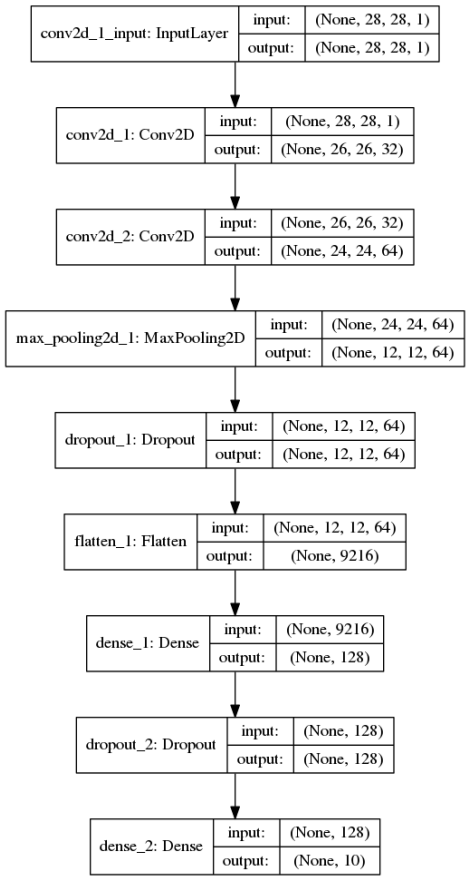
```
model.fit(x_train, y_train,
          batch_size=batch_size,
```

```
                epochs=epochs,
                verbose=1,
                validation_data=(x_test, y_test))
```

with:

```
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              verbose=1,
              validation_split=0.05)
```

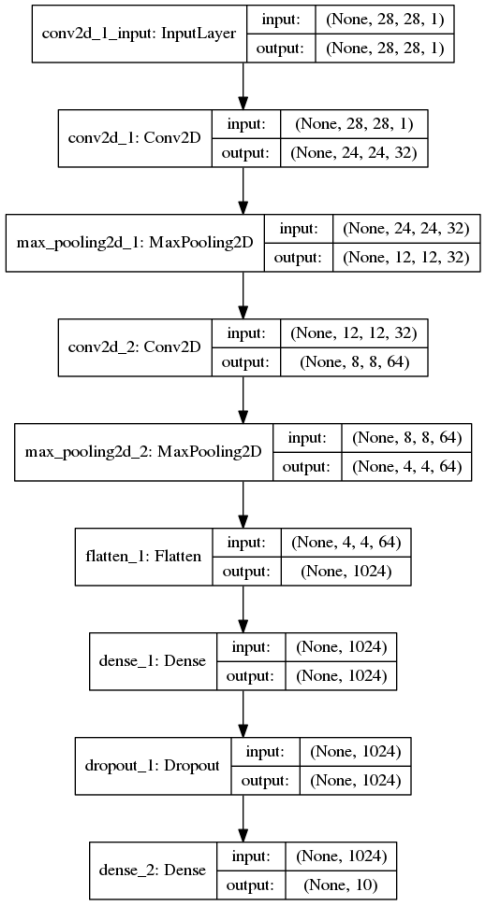
i.e., test data cannot be used during training, so I use a 5% split ratio from training data to validate the model.
The starting model can be visualized as:



After 100 epochs training, the final testing accuracy is: 0.9907.

```
Test loss: 0.03534448798
Test accuracy: 0.9907
```

b) **Stage Two**: replace the convolutional structure with that of LeNet[1].
First, LeNet uses Max-pooling after each convolutional layer;
Second, LeNet uses convolutional filter of size 5*5 (because the images in MNIST are black background around hand-written digits, there is no need to use small filter);
Thirdly, dropout is just used at final prediction.
Forth, more neurons are used in fully connected layer (increase from 128 to 1024). Note the I don't use two fc layers as what LeNet did because there is no need to do a non-linear transformation twice.
The final model can be visualized as:



After 100 epochs training, the final testing accuracy is: 0.9948.

```
Epoch 96/100
57000/57000 [==============================] - 5s 87us/step - loss: 1.5456e-04 - acc: 0.9999 - val_loss: 0.0458 - val_acc: 0.9923
Epoch 97/100
57000/57000 [==============================] - 5s 86us/step - loss: 1.3844e-04 - acc: 0.9999 - val_loss: 0.0485 - val_acc: 0.9930
Epoch 98/100
57000/57000 [==============================] - 5s 87us/step - loss: 7.4850e-05 - acc: 1.0000 - val_loss: 0.0497 - val_acc: 0.9927
Epoch 99/100
57000/57000 [==============================] - 5s 88us/step - loss: 8.5868e-05 - acc: 1.0000 - val_loss: 0.0463 - val_acc: 0.9933
Epoch 100/100
57000/57000 [==============================] - 5s 86us/step - loss: 5.5599e-05 - acc: 1.0000 - val_loss: 0.0484 - val_acc: 0.9927
Test loss: 0.0288629979312
Test accuracy: 0.9948
```

c) **Stage Three**: data augmentation

Note that in the former one, training accuracy achieves 1.0, however, the final testing accuracy is: 0.9948. This indicates the over-fitting of the model.

To make our model more robust, notice that hand-written digits always have a slight deformation. So generate more images by transformation from original images will help improve the model.

After 100 epochs training, the final testing accuracy is: **0.9954**.

As the training accuracy is nearly the same with testing, the model is likely to get rid of over-fitting.

```
Epoch 96/100
469/468 [==============================] - 16s 34ms/step - loss: 0.0164 - acc: 0.9950
Epoch 97/100
469/468 [==============================] - 16s 34ms/step - loss: 0.0166 - acc: 0.9949
Epoch 98/100
469/468 [==============================] - 16s 33ms/step - loss: 0.0168 - acc: 0.9948
Epoch 99/100
469/468 [==============================] - 16s 34ms/step - loss: 0.0170 - acc: 0.9947
Epoch 100/100
469/468 [==============================] - 16s 35ms/step - loss: 0.0160 - acc: 0.9949
Test loss: 0.0155878566469
Test accuracy: 0.9954
```

5. **Difficulties**
   In my work, I think it is somehow hard to think of data augmentation;

6. **How to improve the network further**
   I think of the two following ways to improve the model further:
   a) More regularization techniques. For example, norm-2 regularization for the parameters in fully connected layer. Norm regularization usually can overcome the problem of over-fitting[2];
   b) More advanced convolutional filter. For example, as Google Inception network[3] did, we can use 1X1 filter to reduce the number of feature maps, or replace a nXn filter by connecting a nX1 and 1Xn filter. These improvements will not only speed up the calculation but also adding non-linear transformations in the networks, and so can obtain a better performance.

Reference

[1] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition [J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.
[2] http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/
[3] Szegedy C, Ioffe S, Vanhoucke V, et al. Inception-v4, inception-resnet and the impact of residual connections on learning[C]//AAAI. 2017, 4: 12.