CSCI964 Computational Intelligence: Assignment #3

Mei Wangzhihui 2019124044

Task 1

Introduction to MNIST

The MNIST dataset is a handwritten digital picture dataset, a very popular experimental data set in machine learning, almost becoming a model. It is available at website THE MNIST DATABASE, and it contains four parts:

- Training set images: train-images-idx3-ubyte.gz (9.9 MB, containing 60,000 samples)
- Training set labels: train-labels-idx1-ubyte.gz (29 KB, containing 60,000 labels)
- Test set images: t10k-images-idx3-ubyte.gz (1.6 MB, containing 10,000 samples)
- Test set labels: t10k-labels-idx1-ubyte.gz (5KB, containing 10,000 labels)

The MNIST dataset comes from the National Institute of Standards and Technology (NIST). Now do you know the origin of the name of this data set? M is the abbreviation of Modified. The training set is composed of numbers handwritten from 250 different people, of which 50% are high school students and 50% are from the staff of the Census Bureau. The test set is also the same proportion of handwritten digital data. Each picture is composed of 28x28 pixels, and each pixel is represented by a gray value. Here, the 28x28 pixels are expanded into a one-dimensional line vector (784 values per line). The picture label is one-hot code: 0-9. MNIST's original black and white (dual horizontal) image size is standardized to fit a 20x20 frame while retaining its aspect ratio. As a result of the anti-aliasing technique used by the normalization algorithm, the resulting image contains gray levels. The center of mass pixel image is shifted, positioning the image in the center of the 28x28 field, thereby centering it in the 28x28 image, just like Figure 1.

Figure 1: MNIST data sample

Introduction of SOM

SOM stands for Self-Organizing Map and is an unsupervised learning neural network for feature detection. It simulates different characteristics of the division of nerve cells in different regions of the human brain, that is, different regions have different response characteristics, and this process is automatically completed. SOM is used to generate a low-dimensional space of training samples. It can convert complex non-linear statistical relationships between high-dimensional data into simple geometric relationships and display them

in a low-dimensional manner. Therefore, it is usually used in dimensionality reduction problems. SOM is different from other artificial neural networks because they use competitive learning instead of error-related learning, while involving back propagation and gradient descent. In competitive learning, each will compete with each other to respond to a subset of input data(Figure 2).

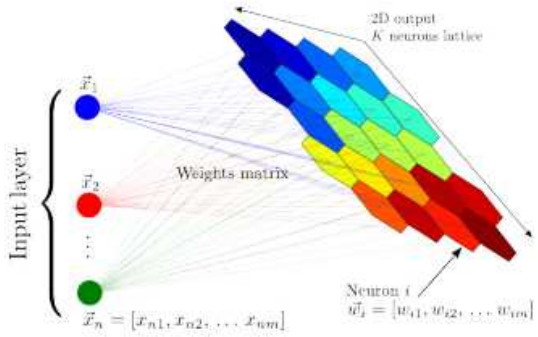

Figure 2: SOM architecture

The training process is like the following Figure 3. The purple area represents the distribution of training data, and the white grid represents the current training data extracted from the distribution. First, the SOM node is located anywhere in the data space. The node closest to the training data (highlighted in yellow) will be selected. It moves towards training data just like neighboring nodes in the grid. After many iterations, the grid tends to approximate this kind of data distribution.

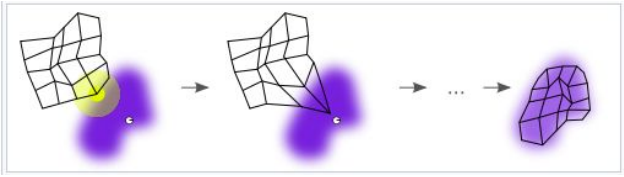

Figure 3: SOM training rocess

Ordering and convergence

If the parameters are correctly initialized, we can start from the completely disordered initial state, and the SOM algorithm will gradually make the activation patterns obtained from the input space represent ordered. (However, it may end up in a metastable state where the feature map has topological defects.)
There are two significant stages in this adaptive process: Ordering or self-organizing stage: During this period, the weight vectors are topologically sorted. Usually this will require up to 1000 iterations of the SOM algorithm, and careful selection of neighborhood and learning rate parameters needs to be considered. Convergence phase: During this period feature maps are fine tuned and provide accurate statistical quantization of the input space. Usually the number of iterations at this stage is at least 500 times the number of neurons in the network, and the parameters must be selected carefully. Here I set the number of ordering iterations as 1000, and the number of convergence iterations as 2000
In my implementation, the ordering epoch is set to 1000 and convergence epoch is set to 2000, and the learning rate is set to 0.1.

Distance each epoch

The euclidean distance between its values in the t-th and (t+1)-th iterations convergence in a few epochs and then decrease slowly with fluctuation.

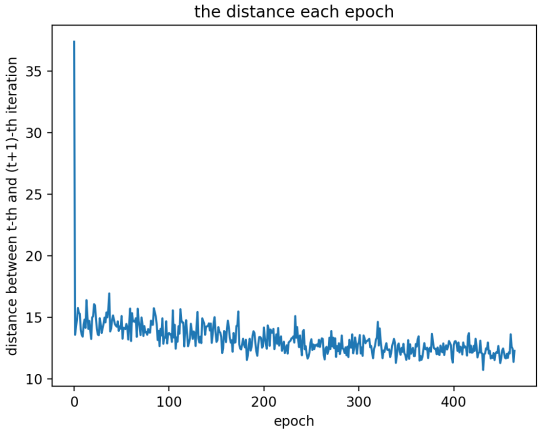

Figure 4: Distance each epoch

2D lattices of three stages

Initialization stage

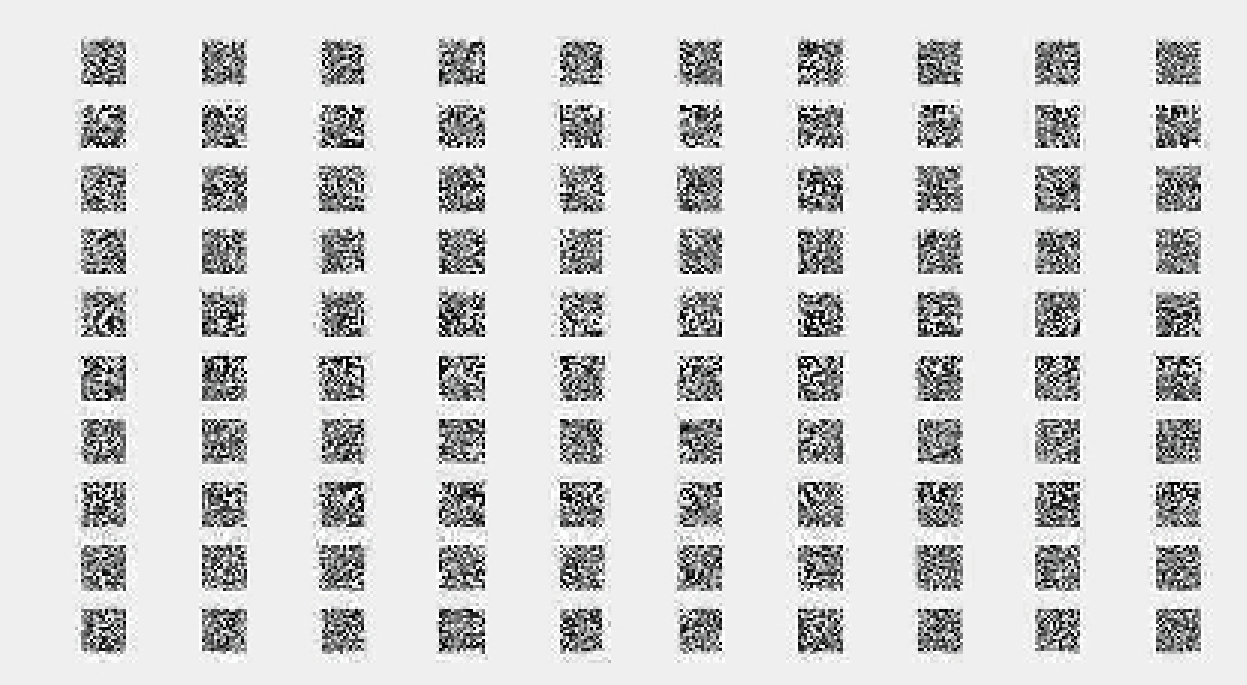

Figure 5: Initialization stage

Between Initialization stage and Convergence stage

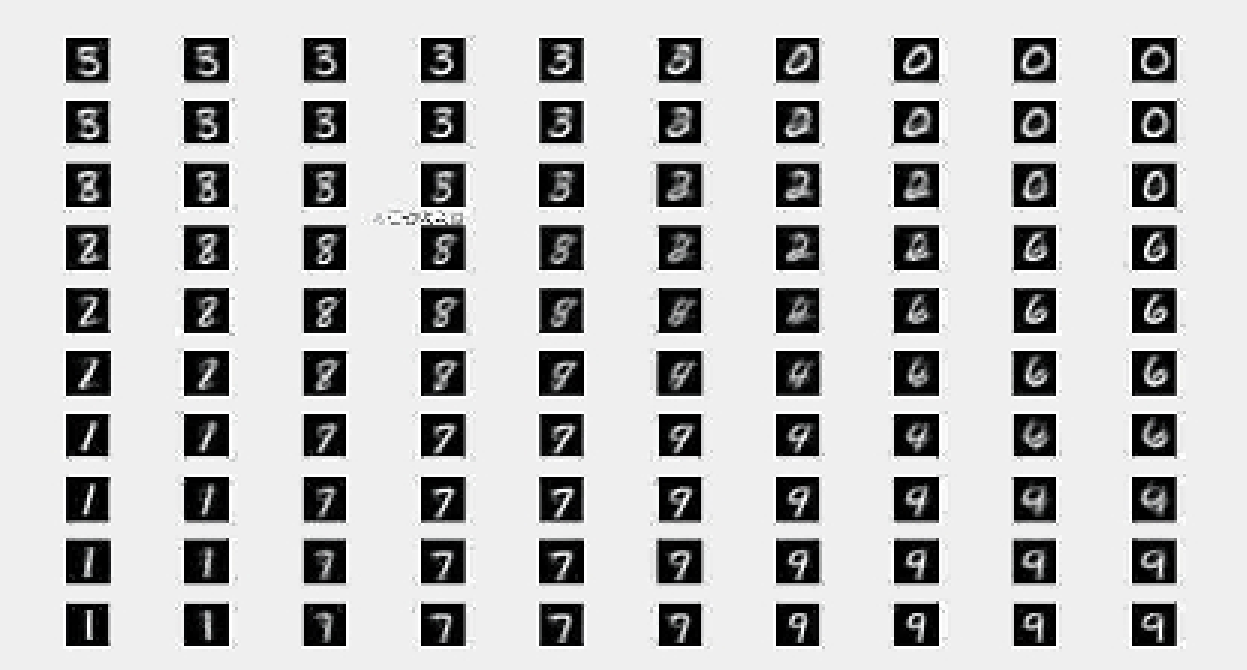

Figure 6: Between Initialization stage and Convergence stage

Convergence stage

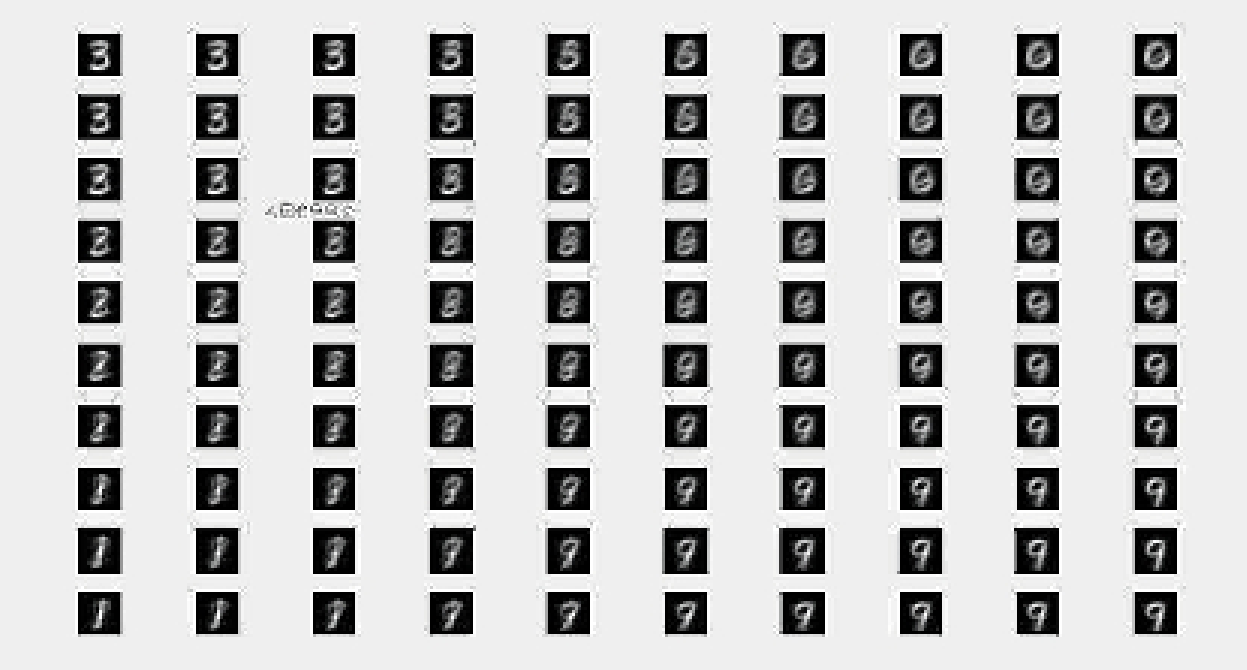

Figure 7: Convergence stage

Task 2

Introduction to MNIST

The MNIST dataset is a handwritten digital picture dataset, a very popular experimental data set in machine learning, almost becoming a model. It is available at website THE MNIST DATABASE, and it contains four parts:

- Training set images: train-images-idx3-ubyte.gz (9.9 MB, containing 60,000 samples)
- Training set labels: train-labels-idx1-ubyte.gz (29 KB, containing 60,000 labels)
- Test set images: t10k-images-idx3-ubyte.gz (1.6 MB, containing 10,000 samples)
- Test set labels: t10k-labels-idx1-ubyte.gz (5KB, containing 10,000 labels)

The MNIST dataset can be imported by TensorFlow API directly.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models


(train_images, train_labels), (test_images,test_labels) = datasets.mnist.load_data()
```

Multinomial logistic regression

Defining network

The following code define a 3-layer network like Figure 8. The input dimention was flattened from 28×28 to 784×1 in Flatten layer. The Dense layer was full-connection layer. The Dropout layer was adopted to preventing overfitting by keeping neutrons hidden and unchanged in one training epoch with a certain probability. The tf.nn.softmax function converts these logits to "probabilities" for each class. The losses.SparseCategoricalCrossentropy loss takes a vector of logits and a True index and returns a scalar loss for each example. Finally, set the model's optimizer to Adam (an optimizer that can adjust the learning rate adaptively), set loss function to SparseCategoricalCrossentropy and metrics to accuracy.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])
```

Training network

The following code training the neural network. We also applied tensorboard to visualize the trainging process, like Figure 9.

```
model.fit(x_train,
      y_train,
      epochs=5,
      validation_data=(x_test, y_test),
```

Problem 2 continued on next page...

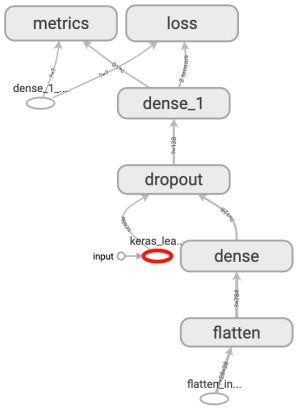

Figure 8: Multinomial logisticre gression network architecture

```
callbacks=[tensorboard_callback])
```

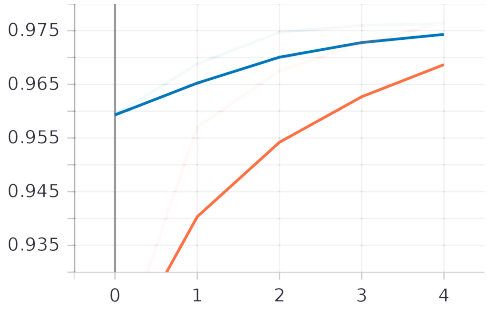

Figure 9: Epoch accuracy of training set and test set

Test

The following code test the neural network's accuracy.

```
model.evaluate(x_test, y_test, verbose=2)
```

Tuning

As we change batch size, the graph down pan down like Figure 10.
As we change optimizer and learning rate, the graphs vary like Figure 11.

Multi-layer neural network

Defining network

The following code define a basic neural network model. The first layer is a convolutional layer with 32 3x3 convolution kernel and RELU activation function. The second layer is a Flatten layer to flatten multi-dimensional input to a 1d-vector. The third is a hidden full-connection layer with RELU as activation function. The final layer is a output layer to output a one-hot encoding vector.

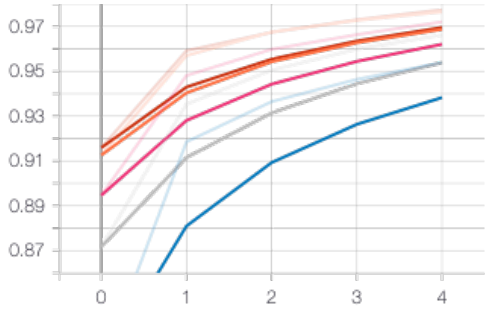

Figure 10: Epoch accuracy of different batch size

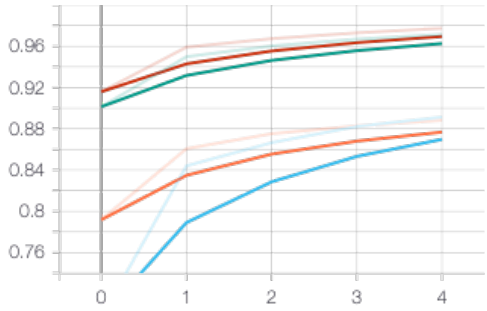

Figure 11: Epoch accuracy of different optimizer

```
class MyModel(Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.conv1 = Conv2D(32, 3, activation='relu')
        self.flatten = Flatten()
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(10)

    def call(self, x):
        x = self.conv1(x)
        x = self.flatten(x)
        x = self.d1(x)
        return self.d2(x)


model = MyModel()

loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

optimizer = tf.keras.optimizers.Adam()

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
```

```
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='test_accuracy')
```

Training and testing

The following code applied the training process and testing process.

```
@tf.function
def train_step(images, labels):
    with tf.GradientTape() as tape:
        # training=True is only needed if there are layers with different
        # behavior during training versus inference (e.g. Dropout).
        predictions = model(images, training=True)
        loss = loss_object(labels, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)
    train_accuracy(labels, predictions)


@tf.function
def test_step(images, labels):
    # training=False is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    predictions = model(images, training=False)
    t_loss = loss_object(labels, predictions)

    test_loss(t_loss)
    test_accuracy(labels, predictions)

    EPOCHS = 5

for epoch in range(EPOCHS):
    # Reset the metrics at the start of the next epoch
    train_loss.reset_states()
    train_accuracy.reset_states()
    test_loss.reset_states()
    test_accuracy.reset_states()

    for images, labels in train_ds:
        train_step(images, labels)

    for test_images, test_labels in test_ds:
        test_step(test_images, test_labels)

    template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'
    print(
        template.format(epoch + 1, train_loss.result(),
                        train_accuracy.result() * 100, test_loss.result(),
                        test_accuracy.result() * 100))
```

Problem 2 continued on next page...

Tuning

Tune the size of hidden layer, the accuracy and loss change like this Figure 12.

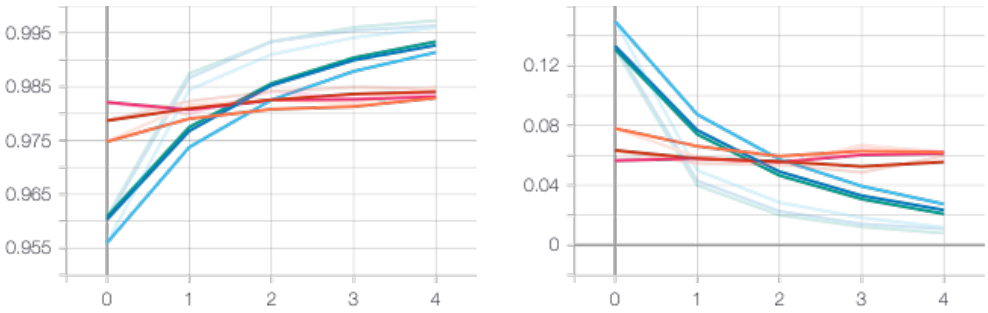

Figure 12: Epoch accuracy/loss of different hidden layer size

Tune the size of convolution kernel, the test accuracy increase and test loss decrease. See Figure 13.

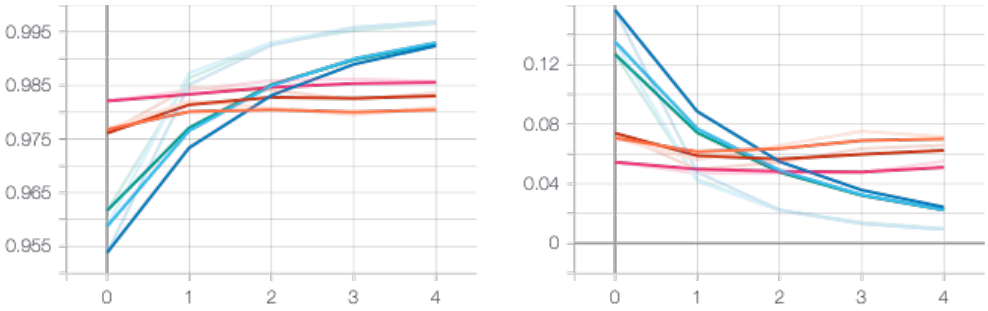

Figure 13: Epoch accuracy/loss of different convolution kernel size

Cell image classification

a

The classification technique is deep Convolutional Neural Network(CNN).
The toolboxes is TensorFlow and TensorBoard.
The language is Python.

b

I applied CNN to this task. The network is a 22-layer-network with 15 convolutional layers, 4 pooling layer(MaxPooling) and 3 fully connected layers. Like Figure 14. The parameters of k, s and p denote the kernel size, stride, and padding size, respectively.

c

I applied gray level stretch to normalize the image intensities like Eq.1

$$g = \frac{(f - \min(f)) \times 255}{\max(f) - \min(f)} \tag{1}$$

where f is the input image pixel gray level and g is the output pixel gray level.

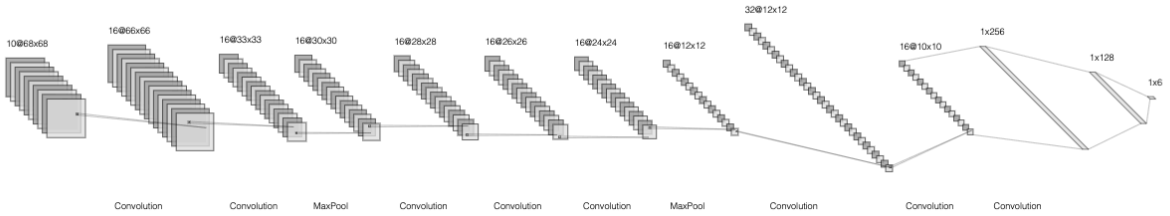

Figure 14: The architecture of the network

Then, I applied data augmentation to the dataset. Data augmentation is a useful method for training CNN on small datasets. The images of different cell patterns in training set were rotated with different angles like Table 1.

Patterns	No. of Images	Training set for CNN	Rotation degree 1	No. After Rotation	Rotation degree 2	No. After Rotation	Rotation degree 3	No. After Rotation
Golgi	721	463	24	6945	12	13890	6	27780
Homogeneous	2494	1596		7980		15960		31920
Speckled	2831	1812		9060		18120		36240
Nucleolar	2598	1663	72	8315	36	16630	18	33260
Centromere	2741	1754		8770		17540		35080
NuMem	2208	1413		7065		14130		28260
Total	13596	8701		48135		96270		192540

Table 1: Data augmentation to dataset

d

We choose convolutional kernel like 1×1 and 3×3 as they require less parameters and computational loss. Besides, 1×1 kernel can reduce or add the dimension of feature map and integrate channel information of feature map. Each convolution layer is followed by a nonlinear activation layer with the activation function PreLU as Eq.2, where θ is a parameter in learning process. On the side, 2×2 max pooling is employed to decrease the dimension of feature maps. At the end of the proposed network, two fully-connected layers and one softmax layer are employed to classify cell images into six categories. To prevent the network from overfitting, I applied 0.5 dropout layer to it.

$$F = max(\theta x, x) \tag{2}$$

e

The accuracy performance is like Table 2.

	Rotation degree 1	Rotation degree 2	Rotation degree 3
Training	98.69	99.36	96.33
Validation	96.33	98.73	99.34

Table 2: The training accuracy

f

The final accuracy performance is like Table 3.

Rotation degree 1	Rotation degree 2	Rotation degree 3
95.33	98.02	98.17

Table 3: The final accuracy performance

g

We can generate the output of each layer and get the following results Figure 15 and 16.

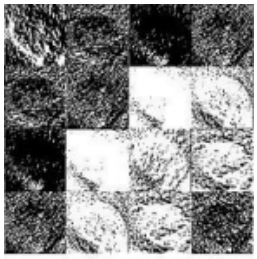

Figure 15: Feature maps from first convolution layer

Figure 16: Feature maps from last convolution layer

It seems to be that the first convolution layer reveal the edge of initial image, and features learned by the last layer is abstract and clear.
We can also draw the conclusion that more data generate better accuracy from the final accuracy, as the number of data increase with degree.