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).

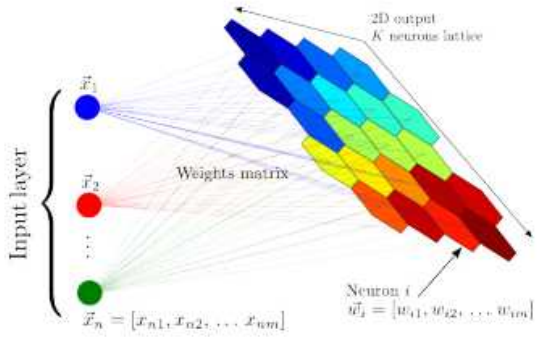

Figure 2: SOM architecture

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.

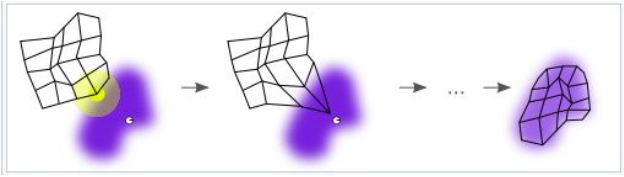

Figure 3: SOM training rocess

Initialization

Initialize all connection weights to random values. Initialize the weight of each node. The weights are set to standardized small random values.

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 4. 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'])
```

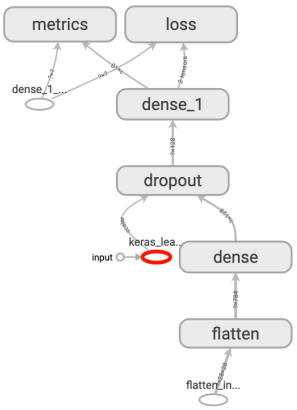

Figure 4: Multinomial logisticre gression network architecture

Training network

The following code training the neural network. We also applied tensorboard to visualize the trainging process, like Figure 5.

```
model.fit(x_train,
       y_train,
       epochs=5,
       validation_data=(x_test, y_test),
       callbacks=[tensorboard_callback])
```

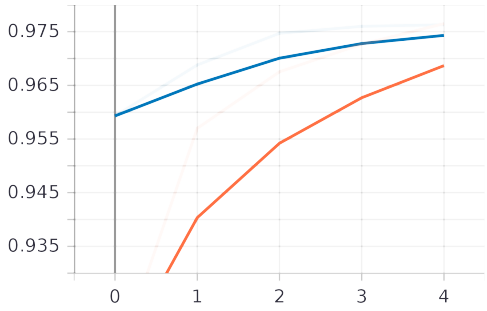

Figure 5: Epoch accuracy of training set and test set

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 6.

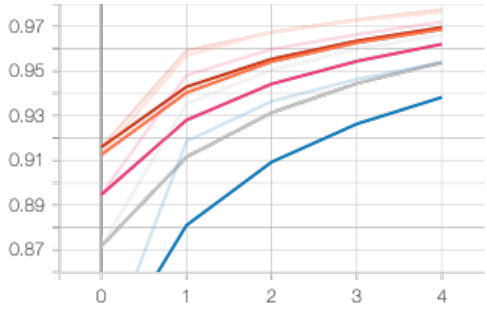

Figure 6: Epoch accuracy of different batch size

As we change optimizer and learning rate, the graphs vary like Figure 7.

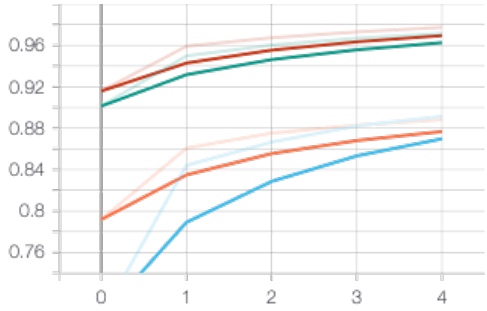

Figure 7: Epoch accuracy of different optimizer

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.

```
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

The following code define the training 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)
```