케라스를 사용한 신경망 구현

Keras 패키지는 Theano 또는 TensorFlow를 계산 엔진으로 사용하는 파이썬 패키지다. 신경망을 구성하기 위한 각 구성요소를 클래스로 제공하기 때문에 이를 간단히 연결하기만 하면 쉽게 신경망을 구현할 수 있다. 이 글에서는 버전 텐서플로우 2.0-beta에 추가된 케라스 2.2.4-tf 버전을 기준으로 설명한다.

In [23]:
```
%tensorflow_version 2.x
import tensorflow as tf
tf.__version__
```

Out[23]: '2.1.0'

In [24]:
```
from tensorflow import keras
keras.__version__
```

Out[24]: '2.2.4-tf'

예제 데이터

Keras는 다음과 같은 샘플데이터를 제공한다.

- CIFAR10 이미지
  - 10 종류의 카테고리 라벨을 가지는 50,000 개의 32x32 해상도 컬러 트레이닝 이미지와 10,000 개의 테스트 이미지
- CIFAR100 이미지
  - 100 종류의 카테고리 라벨을 가지는 50,000 개의 32x32 해상도 컬러 트레이닝 이미지와 10,000 개의 테스트 이미지
- IMDB 영화 감상
  - positive/negative 라벨을 가지는 25,000 영화 감상 데이터
  - 텍스트 단어는 숫자로 인코딩되어 있음
- 로이터 뉴스 토픽
  - 46 종류의 토픽 라벨을 가지는 11,228 개의 로이터 뉴스 텍스트
  - 텍스트 단어는 숫자로 인코딩되어 있음
- MNIST 숫자 이미지
  - 0부터 9까지의 숫자에 대한 28x28 단색 이미지
  - 트레이딩 데이터 60,000개. 테스트 이미지 10,000개
- MNIST 패션 이미지
  - 10 종류의 의류 대한 28x28 단색 이미지
  - 트레이딩 데이터 60,000개. 테스트 이미지 10,000개
- Boston housing price
  - 보스턴 주택 가격 데이터

Keras를 사용하는 방법은 어렵지 않기 때문에 바로 MNIST 데이터를 이용해 신경망을 구현하는 예를 보인다.

In [25]:
```
mnist = keras.datasets.mnist
(X_train0, y_train0), (X_test0, y_test0) = mnist.load_data()

import matplotlib.pylab as plt

plt.figure(figsize=(6, 1))
for i in range(36):
    plt.subplot(3, 12, i+1)
    plt.imshow(X_train0[i], cmap="gray")
    plt.axis("off")
plt.show()
```

Keras의 MNIST 이미지 데이터는 28x28로 scikit-learn보다 고해상도이다.

In [26]:
```
print(X_train0.shape, X_train0.dtype)
print(y_train0.shape, y_train0.dtype)
print(X_test0.shape, X_test0.dtype)
print(y_test0.shape, y_test0.dtype)
```

```
(60000, 28, 28) uint8
(60000,) uint8
(10000, 28, 28) uint8
(10000,) uint8
```

데이터를 `float` 타입으로 변환 후 스케일링한다. 이는 이미지를 전처리하는 보편적인 방법 중 하나이다.

In [27]:
```
X_train = X_train0.reshape(60000, 784).astype('float32') / 255.0
X_test = X_test0.reshape(10000, 784).astype('float32') / 255.0
print(X_train.shape, X_train.dtype)
```

```
(60000, 784) float32
```

정답데이터는 라벨에 해당하는 숫자로 되어 있다.

In [28]:
```
y_train0[:5]
```

Out[28]:
```
array([5, 0, 4, 1, 9], dtype=uint8)
```

이 값을 keras.np_utils.categorical() 을 사용하여 원핫인코딩(One-Hot-Encoding)로 변환한다.

In [29]:
```
from tensorflow.keras.utils import to_categorical

Y_train = to_categorical(y_train0, 10)
Y_test = to_categorical(y_test0, 10)
Y_train[:5]
```

Out[29]:
```
array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]], dtype=float32)
```

신경망 구현 순서

Keras 를 사용하면 다음과 같은 순서로 신경망을 구성할 수 있다.

1. `Sequential` 모형 클래스 객체 생성
2. `add` 메서드로 레이어 추가.
    - 입력단부터 순차적으로 추가한다.
    - 레이어는 출력 뉴런 갯수를 첫번째 인수로 받는다.
    - 최초의 레이어는 `input_dim` 인수로 입력 크기를 설정해야 한다.
    - `activation` 인수로 활성화함수 설정
3. `compile` 메서드로 모형 완성.
    - `loss` 인수로 비용함수 설정
    - `optimizer` 인수로 최적화 알고리즘 설정
    - `metrics` 인수로 트레이닝 단계에서 기록할 성능 기준 설정
4. `fit` 메서드로 트레이닝
    - `nb_epoch` 로 에포크(epoch) 횟수 설정
    - `batch_size` 로 배치크기(batch size) 설정
    - `verbose` 는 학습 중 출력되는 문구를 설정하는 것으로, 주피터노트북(Jupyter Notebook)을 사용할 때는 `verbose=2` 로 설정하여 진행 막대(progress bar)가 나오지 않도록 설정한다.

다음은 간단한 신경망 모형을 방금 설명한 방법으로 구현한 것이다.

In [0]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD

tf.random.set_seed(0)

model = Sequential()
model.add(Dense(15, input_dim=784, activation="sigmoid"))
model.add(Dense(10, activation="sigmoid"))
model.compile(optimizer=SGD(lr=0.2), loss='mean_squared_error', metrics=["accuracy"])
```

만들어진 모형은 summary 명령으로 모델 내부의 layers 리스트를 살펴봄으로써 내부 구조를 확인할 수 있다.

In [31]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 15)                11775
_________________________________________________________________
dense_3 (Dense)              (None, 10)                160
=================================================================
Total params: 11,935
Trainable params: 11,935
Non-trainable params: 0
_________________________________________________________________
```

layers 속성으로 각 레이어의 특성을 살펴볼 수도 있다.

In [32]:
```
l1 = model.layers[0]
l2 = model.layers[1]

print(l1.name, type(l1), l1.output_shape, l1.activation.__name__, l1.count_params())
print(l2.name, type(l1), l2.output_shape, l2.activation.__name__, l2.count_params())
```

```
dense_2 <class 'tensorflow.python.keras.layers.core.Dense'> (None, 15) sigmoid 11775
dense_3 <class 'tensorflow.python.keras.layers.core.Dense'> (None, 10) sigmoid 160
```

모델을 완성했다면 fit 메서드로 트레이닝을 시작한다.

In [33]:
```
%%time
hist = model.fit(X_train, Y_train,
                 epochs=50, batch_size=100,
                 validation_data=(X_test, Y_test),
                 verbose=2)
```

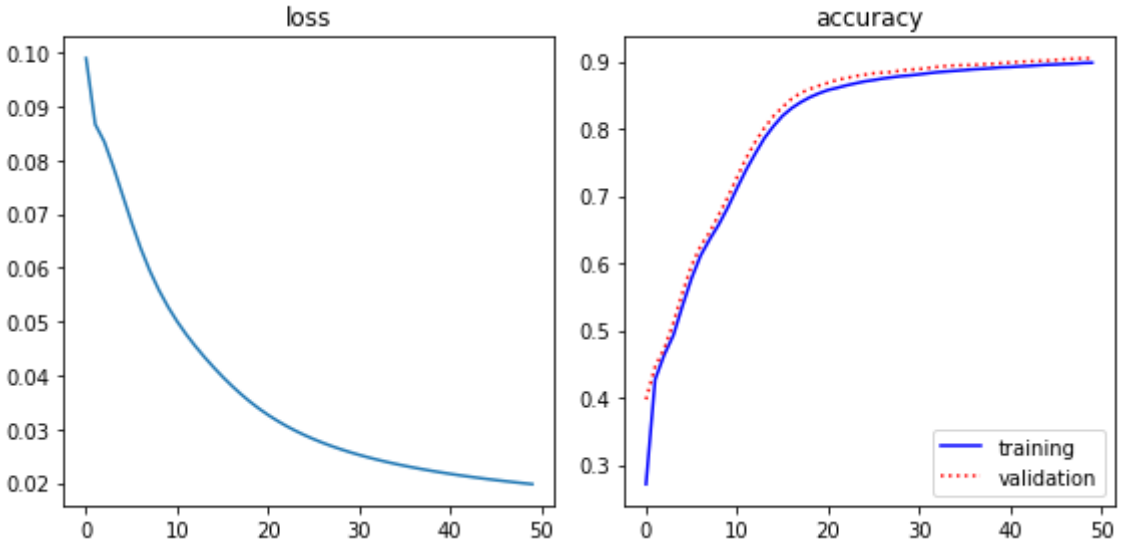
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 - 2s - loss: 0.0990 - accuracy: 0.2716 - val_loss: 0.0878 - val_accuracy: 0.3975
Epoch 2/50
60000/60000 - 2s - loss: 0.0867 - accuracy: 0.4272 - val_loss: 0.0853 - val_accuracy: 0.4462
Epoch 3/50
60000/60000 - 1s - loss: 0.0835 - accuracy: 0.4638 - val_loss: 0.0811 - val_accuracy: 0.4729
Epoch 4/50
60000/60000 - 2s - loss: 0.0787 - accuracy: 0.4927 - val_loss: 0.0760 - val_accuracy: 0.5112
Epoch 5/50
60000/60000 - 2s - loss: 0.0736 - accuracy: 0.5364 - val_loss: 0.0707 - val_accuracy: 0.5560
Epoch 6/50
60000/60000 - 1s - loss: 0.0684 - accuracy: 0.5776 - val_loss: 0.0657 - val_accuracy: 0.5965
Epoch 7/50
60000/60000 - 1s - loss: 0.0636 - accuracy: 0.6115 - val_loss: 0.0612 - val_accuracy: 0.6259
Epoch 8/50
60000/60000 - 2s - loss: 0.0594 - accuracy: 0.6357 - val_loss: 0.0572 - val_accuracy: 0.6461
Epoch 9/50
60000/60000 - 2s - loss: 0.0558 - accuracy: 0.6577 - val_loss: 0.0539 - val_accuracy: 0.6724
Epoch 10/50
60000/60000 - 2s - loss: 0.0527 - accuracy: 0.6834 - val_loss: 0.0510 - val_accuracy: 0.6986
Epoch 11/50
60000/60000 - 2s - loss: 0.0500 - accuracy: 0.7117 - val_loss: 0.0484 - val_accuracy: 0.7279
Epoch 12/50
60000/60000 - 2s - loss: 0.0476 - accuracy: 0.7390 - val_loss: 0.0461 - val_accuracy: 0.7563
Epoch 13/50
60000/60000 - 2s - loss: 0.0454 - accuracy: 0.7631 - val_loss: 0.0440 - val_accuracy: 0.7805
Epoch 14/50
60000/60000 - 2s - loss: 0.0434 - accuracy: 0.7859 - val_loss: 0.0420 - val_accuracy: 0.8012
Epoch 15/50
60000/60000 - 2s - loss: 0.0415 - accuracy: 0.8039 - val_loss: 0.0401 - val_accuracy: 0.8187
Epoch 16/50
60000/60000 - 2s - loss: 0.0397 - accuracy: 0.8192 - val_loss: 0.0384 - val_accuracy: 0.8325
Epoch 17/50
60000/60000 - 2s - loss: 0.0381 - accuracy: 0.8306 - val_loss: 0.0367 - val_accuracy: 0.8445
Epoch 18/50
60000/60000 - 1s - loss: 0.0365 - accuracy: 0.8395 - val_loss: 0.0352 - val_accuracy: 0.8534
Epoch 19/50
60000/60000 - 2s - loss: 0.0351 - accuracy: 0.8468 - val_loss: 0.0338 - val_accuracy: 0.8591
Epoch 20/50
60000/60000 - 2s - loss: 0.0338 - accuracy: 0.8527 - val_loss: 0.0326 - val_accuracy: 0.8640
```

```
Epoch 21/50
60000/60000 - 2s - loss: 0.0326 - accuracy: 0.8576 - val_loss: 0.0315 - val_accuracy: 0.8683
Epoch 22/50
60000/60000 - 1s - loss: 0.0315 - accuracy: 0.8608 - val_loss: 0.0304 - val_accuracy: 0.8727
Epoch 23/50
60000/60000 - 2s - loss: 0.0306 - accuracy: 0.8645 - val_loss: 0.0295 - val_accuracy: 0.8753
Epoch 24/50
60000/60000 - 1s - loss: 0.0297 - accuracy: 0.8675 - val_loss: 0.0286 - val_accuracy: 0.8783
Epoch 25/50
60000/60000 - 1s - loss: 0.0289 - accuracy: 0.8704 - val_loss: 0.0278 - val_accuracy: 0.8803
Epoch 26/50
60000/60000 - 2s - loss: 0.0281 - accuracy: 0.8724 - val_loss: 0.0271 - val_accuracy: 0.8824
Epoch 27/50
60000/60000 - 2s - loss: 0.0274 - accuracy: 0.8746 - val_loss: 0.0264 - val_accuracy: 0.8835
Epoch 28/50
60000/60000 - 2s - loss: 0.0268 - accuracy: 0.8765 - val_loss: 0.0258 - val_accuracy: 0.8842
Epoch 29/50
60000/60000 - 1s - loss: 0.0262 - accuracy: 0.8782 - val_loss: 0.0253 - val_accuracy: 0.8866
Epoch 30/50
60000/60000 - 1s - loss: 0.0257 - accuracy: 0.8794 - val_loss: 0.0248 - val_accuracy: 0.8879
Epoch 31/50
60000/60000 - 1s - loss: 0.0252 - accuracy: 0.8809 - val_loss: 0.0243 - val_accuracy: 0.8887
Epoch 32/50
60000/60000 - 2s - loss: 0.0247 - accuracy: 0.8826 - val_loss: 0.0238 - val_accuracy: 0.8903
Epoch 33/50
60000/60000 - 2s - loss: 0.0243 - accuracy: 0.8841 - val_loss: 0.0234 - val_accuracy: 0.8920
Epoch 34/50
60000/60000 - 2s - loss: 0.0239 - accuracy: 0.8852 - val_loss: 0.0230 - val_accuracy: 0.8933
Epoch 35/50
60000/60000 - 1s - loss: 0.0235 - accuracy: 0.8862 - val_loss: 0.0227 - val_accuracy: 0.8942
Epoch 36/50
60000/60000 - 1s - loss: 0.0231 - accuracy: 0.8872 - val_loss: 0.0223 - val_accuracy: 0.8946
Epoch 37/50
60000/60000 - 1s - loss: 0.0228 - accuracy: 0.8884 - val_loss: 0.0220 - val_accuracy: 0.8949
Epoch 38/50
60000/60000 - 1s - loss: 0.0225 - accuracy: 0.8891 - val_loss: 0.0217 - val_accuracy: 0.8959
Epoch 39/50
60000/60000 - 1s - loss: 0.0222 - accuracy: 0.8901 - val_loss: 0.0214 - val_accuracy: 0.8964
Epoch 40/50
60000/60000 - 1s - loss: 0.0219 - accuracy: 0.8913 - val_loss: 0.0212 - val_accuracy: 0.8976
Epoch 41/50
60000/60000 - 1s - loss: 0.0216 - accuracy: 0.8917 - val_loss: 0.0209 - val_accuracy: 0.8985
```

```
Epoch 42/50
60000/60000 - 1s - loss: 0.0214 - accuracy: 0.8926 - val_loss: 0.0207 - val_accuracy: 0.8995
Epoch 43/50
60000/60000 - 1s - loss: 0.0212 - accuracy: 0.8933 - val_loss: 0.0204 - val_accuracy: 0.9001
Epoch 44/50
60000/60000 - 2s - loss: 0.0209 - accuracy: 0.8943 - val_loss: 0.0202 - val_accuracy: 0.9008
Epoch 45/50
60000/60000 - 1s - loss: 0.0207 - accuracy: 0.8951 - val_loss: 0.0200 - val_accuracy: 0.9013
Epoch 46/50
60000/60000 - 1s - loss: 0.0205 - accuracy: 0.8956 - val_loss: 0.0198 - val_accuracy: 0.9022
Epoch 47/50
60000/60000 - 2s - loss: 0.0203 - accuracy: 0.8963 - val_loss: 0.0196 - val_accuracy: 0.9032
Epoch 48/50
60000/60000 - 2s - loss: 0.0201 - accuracy: 0.8968 - val_loss: 0.0195 - val_accuracy: 0.9041
Epoch 49/50
60000/60000 - 2s - loss: 0.0199 - accuracy: 0.8978 - val_loss: 0.0193 - val_accuracy: 0.9047
Epoch 50/50
60000/60000 - 1s - loss: 0.0197 - accuracy: 0.8981 - val_loss: 0.0191 - val_accuracy: 0.9048
CPU times: user 1min 32s, sys: 7.96 s, total: 1min 40s
Wall time: 1min 15s
```

학습이 끝나면 기록된 변수를 확인한다. 다음 두 그래프는 방금 학습 시킨 모델의 비용함수와 성능지표에 대한 것이다.

In [34]:
```
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.plot(hist.history['loss'])
plt.title("loss")
plt.subplot(1, 2, 2)
plt.title("accuracy")
plt.plot(hist.history['accuracy'], 'b-', label="training")
plt.plot(hist.history['val_accuracy'], 'r:', label="validation")
plt.legend()
plt.tight_layout()
plt.show()
```



가중치 정보

트레이닝이 끝난 모형의 가중치 정보는 `get_weights` 메서드로 구할 수 있다. 이 메서드는 신경망 모형에서 사용된 가중치 w 값과 b 값을 출력한다.

In [13]:
```
# 첫번째 레이어
w1 = l1.get_weights()
w1[0].shape, w1[1].shape
```

Out[13]:
```
((784, 15), (15,))
```

In [14]:
```
# 두번째 레이어
w2 = l2.get_weights()
w2[0].shape, w2[1].shape
```

Out[14]:
```
((15, 10), (10,))
```

모형의 사용

트레이닝이 끝난 모형은 `predict` 메서드로 y 값을 출력하거나 출력된 y값을 각 클래스에 대한 판별함수로 가정하고 `predict_classes` 메서드로 분류를 수행할 수 있다. 예로 테스트 데이터셋의 첫번째 이미지를 예측하면 다음과 같다.

In [15]:
```
model.predict(X_test[:1, :])
```

Out[15]:
```
array([[0.12997532, 0.04451276, 0.06393065, 0.10513153, 0.1064776 ,
        0.1350903 , 0.00947816, 0.69169044, 0.13726085, 0.21310748]],
      dtype=float32)
```

In [16]:
```
model.predict_classes(X_test[:1, :], verbose=0)
```

Out[16]:
```
array([7])
```

테스트 데이터셋의 첫번째 이미지를 출력해보면 다음처럼 실제로 7이 나온다

In [17]:
```
plt.figure(figsize=(1, 1))
plt.imshow(X_test0[0], cmap=plt.cm.bone_r)
plt.grid(False)
plt.axis("off")
plt.show()
```

모형의 저장

트레이닝이 끝난 모형은 save 메서드로 가중치와 함께 "hdf5" 형식으로 저장하였다가 나중에 load 명령으로 불러 사용할 수 있다.

In [18]:
```
print(model.predict(X_test[:1, :]))
```

```
[[0.12997532 0.04451276 0.06393065 0.10513153 0.1064776  0.1350903
  0.00947816 0.69169044 0.13726085 0.21310748]]
```

In [19]:
```
print(model.predict_classes(X_test[:1, :], verbose=0))
```

```
[7]
```

In [0]:
```
model.save('my_model.hdf5')
del model
```

In [21]:
```
from tensorflow.keras.models import load_model

model2 = load_model('my_model.hdf5')
print(model2.predict_classes(X_test[:1, :], verbose=0))
```

```
[7]
```

연습 문제 1

Keras를 사용하여 iris 분류 문제를 해결하는 신경망을 구현하라.

연습 문제 2

Keras를 사용하여 olivetti_faces 분류 문제를 해결하는 신경망을 구현하라.