# mnist_with_keras

December 14, 2020

## 1 MNIST digits classification with Keras

We don't expect you to code anything here because you've already solved it with TensorFlow.
But you can appreciate how simpler it is with Keras.
We'll be happy if you play around with the architecture though, there're some tips at the end.

```
In [1]: import numpy as np
        from sklearn.metrics import accuracy_score
        from matplotlib import pyplot as plt
        %matplotlib inline
        import tensorflow as tf
        print("We're using TF", tf.__version__)
        import keras
        print("We are using Keras", keras.__version__)

        import sys
        sys.path.append("../..")
        import keras_utils
        from keras_utils import reset_tf_session
```

```
We're using TF 1.2.1


Using TensorFlow backend.


We are using Keras 2.0.6
```

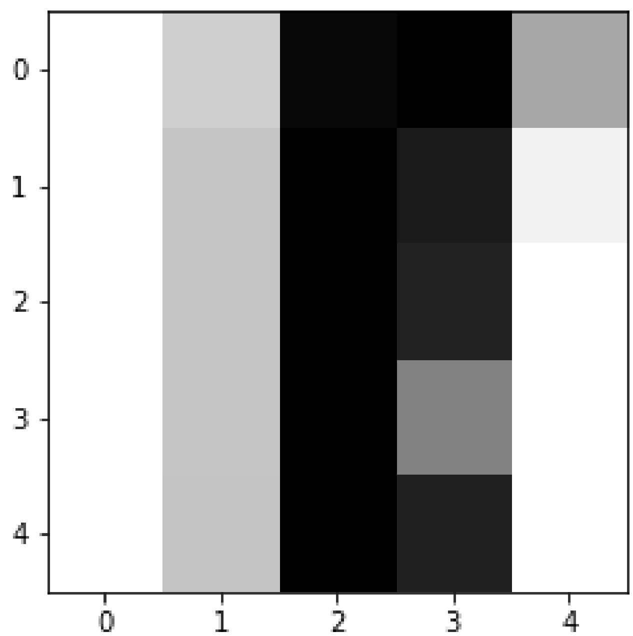
## 2 Look at the data

In this task we have 50000 28x28 images of digits from 0 to 9. We will train a classifier on this data.

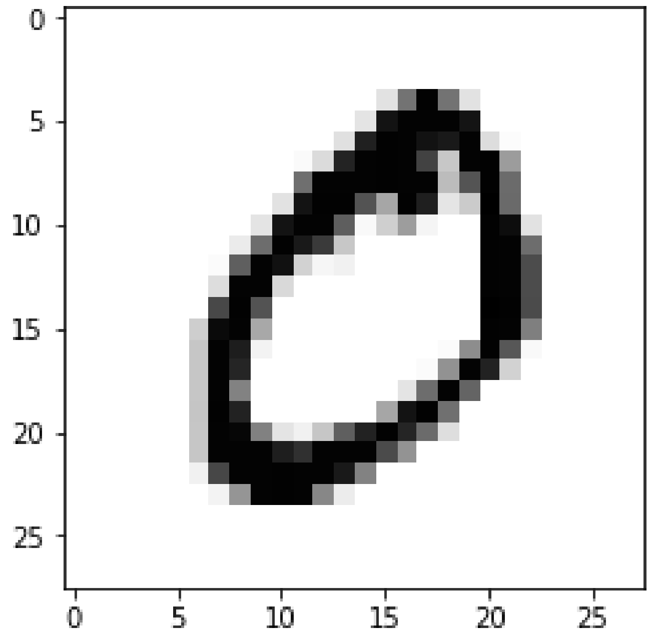
```
In [2]: import preprocessed_mnist
        X_train, y_train, X_val, y_val, X_test, y_test = preprocessed_mnist.load_dataset_from_fi
```

```
In [3]: # X contains rgb values divided by 255
        print("X_train [shape %s] sample patch:\n" % (str(X_train.shape)), X_train[1, 15:20, 5:1
        print("A closeup of a sample patch:")
        plt.imshow(X_train[1, 15:20, 5:10], cmap="Greys")
        plt.show()
        print("And the whole sample:")
        plt.imshow(X_train[1], cmap="Greys")
        plt.show()
        print("y_train [shape %s] 10 samples:\n" % (str(y_train.shape)), y_train[:10])
```

```
X_train [shape (50000, 28, 28)] sample patch:
 [[ 0.          0.29803922  0.96470588  0.98823529  0.43921569]
 [ 0.          0.33333333  0.98823529  0.90196078  0.09803922]
 [ 0.          0.33333333  0.98823529  0.8745098   0.        ]
 [ 0.          0.33333333  0.98823529  0.56862745  0.        ]
 [ 0.          0.3372549   0.99215686  0.88235294  0.        ]]
A closeup of a sample patch:
```




```
And the whole sample:
```




```
y_train [shape (50000,)] 10 samples:
 [5 0 4 1 9 2 1 3 1 4]
```

```
In [4]: # flatten images
        X_train_flat = X_train.reshape((X_train.shape[0], -1))
        print(X_train_flat.shape)

        X_val_flat = X_val.reshape((X_val.shape[0], -1))
        print(X_val_flat.shape)
```

```
(50000, 784)
(10000, 784)
```

```
In [5]: # one-hot encode the target
        y_train_oh = keras.utils.to_categorical(y_train, 10)
        y_val_oh = keras.utils.to_categorical(y_val, 10)

        print(y_train_oh.shape)
        print(y_train_oh[:3], y_train[:3])
```

```
(50000, 10)
[[ 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.]] [5 0 4]
```

In [6]:
```
# building a model with keras
from keras.layers import Dense, Activation
from keras.models import Sequential

# we still need to clear a graph though
s = reset_tf_session()

model = Sequential()  # it is a feed-forward network without loops like in RNN
model.add(Dense(256, input_shape=(784,)))  # the first layer must specify the input shap
model.add(Activation('sigmoid'))
model.add(Dense(256))
model.add(Activation('sigmoid'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

In [7]:
```
# you can look at all layers and parameter count
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               200960
_________________________________________________________________
activation_1 (Activation)    (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               65792
_________________________________________________________________
activation_2 (Activation)    (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                2570
_________________________________________________________________
activation_3 (Activation)    (None, 10)                0
=================================================================
Total params: 269,322
Trainable params: 269,322
Non-trainable params: 0
_________________________________________________________________
```

In [8]:
```
# now we "compile" the model specifying the loss and optimizer
model.compile(
    loss='categorical_crossentropy', # this is our cross-entropy
    optimizer='adam',
    metrics=['accuracy']  # report accuracy during training
)
```

In [9]:
```
# and now we can fit the model with model.fit()
# and we don't have to write loops and batching manually as in TensorFlow
model.fit(
```

```
            X_train_flat,
            y_train_oh,
            batch_size=512,
            epochs=40,
            validation_data=(X_val_flat, y_val_oh),
            callbacks=[keras_utils.TqdmProgressCallback()],
            verbose=0
        )
```

```
Epoch 1/40
```

```
A Jupyter Widget
```

```
Epoch 2/40
```

```
A Jupyter Widget
```

```
Epoch 3/40
```

```
A Jupyter Widget
```

```
Epoch 4/40
```

```
A Jupyter Widget
```

```
Epoch 5/40
```

```
A Jupyter Widget
```

```
Epoch 6/40
```

A Jupyter Widget

Epoch 7/40

A Jupyter Widget

Epoch 8/40

A Jupyter Widget

Epoch 9/40

A Jupyter Widget

Epoch 10/40

A Jupyter Widget

Epoch 11/40

A Jupyter Widget

Epoch 12/40

A Jupyter Widget

```
Epoch 13/40
```

A Jupyter Widget

```
Epoch 14/40
```

A Jupyter Widget

```
Epoch 15/40
```

A Jupyter Widget

```
Epoch 16/40
```

A Jupyter Widget

```
Epoch 17/40
```

A Jupyter Widget

```
Epoch 18/40
```

A Jupyter Widget

```
Epoch 19/40
```

A Jupyter Widget

Epoch 20/40

A Jupyter Widget

Epoch 21/40

A Jupyter Widget

Epoch 22/40

A Jupyter Widget

Epoch 23/40

A Jupyter Widget

Epoch 24/40

A Jupyter Widget

Epoch 25/40

A Jupyter Widget

```
Epoch 26/40

A Jupyter Widget

Epoch 27/40

A Jupyter Widget

Epoch 28/40

A Jupyter Widget

Epoch 29/40

A Jupyter Widget

Epoch 30/40

A Jupyter Widget

Epoch 31/40

A Jupyter Widget

Epoch 32/40
```

A Jupyter Widget

Epoch 33/40

A Jupyter Widget

Epoch 34/40

A Jupyter Widget

Epoch 35/40

A Jupyter Widget

Epoch 36/40

A Jupyter Widget

Epoch 37/40

A Jupyter Widget

Epoch 38/40

A Jupyter Widget

```
Epoch 39/40
```

```
A Jupyter Widget
```

```
Epoch 40/40
```

```
A Jupyter Widget
```

```
Out[9]: <keras.callbacks.History at 0x7faa436d6550>
```

## 3 Here're the notes for those who want to play around here

Here are some tips on what you could do:

- **Network size**
- More neurons,
- More layers, (docs)
- Other nonlinearities in the hidden layers
  - tanh, relu, leaky relu, etc
- Larger networks may take more epochs to train, so don't discard your net just because it could didn't beat the baseline in 5 epochs.
- **Early Stopping**
- Training for 100 epochs regardless of anything is probably a bad idea.
- Some networks converge over 5 epochs, others - over 500.
- Way to go: stop when validation score is 10 iterations past maximum
- **Faster optimization**
- rmsprop, nesterov_momentum, adam, adagrad and so on.
  - Converge faster and sometimes reach better optima

- It might make sense to tweak learning rate/momentum, other learning parameters, batch size and number of epochs

- **Regularize** to prevent overfitting
- Add some L2 weight norm to the loss function, theano will do the rest
    - Can be done manually or via - https://keras.io/regularizers/
- **Data augmemntation** - getting 5x as large dataset for free is a great deal
- https://keras.io/preprocessing/image/
- Zoom-in+slice = move
- Rotate+zoom(to remove black stripes)
- any other perturbations
- Simple way to do that (if you have PIL/Image):
    - `from scipy.misc import imrotate,imresize`
    - and a few slicing
- Stay realistic. There's usually no point in flipping dogs upside down as that is not the way you usually see them.

```
In [ ]:
```