Universidad de Granada

Inteligencia Computacional

Máster Profesional en Ingeniería Informática

Reconocimiento óptico de caracteres MNIST

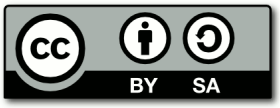
Autor
Ernesto Serrano Collado

Profesor
Fernando Berzal Galiano

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 20 de noviembre de 2017



Reconocimiento óptico de caracteres MNIST

Ernesto Serrano Collado

Resumen

Palabras clave: *software libre, inteligencia artificial, redes neuronales*

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales.

Se deberá evaluar el uso de varios tipos de redes neuronales para resolver un problema de reconocimiento óptico de caracteres: el reconocimiento de dígitos manuscritos de la base de datos MNIST.

Yo, **Ernesto Serrano Collado**, alumno de la titulación **Máster Profesional en Ingeniería Informática** de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, autorizo la ubicación de la siguiente copia de mi Trabajo (*Reconocimiento óptico de caracteres MNIST*) en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Además, este mismo trabajo está publicado bajo la licencia **Creative Commons Attribution-ShareAlike 4.0** [**?**], dando permiso para copiarlo y redistribuirlo en cualquier medio o formato, también de adaptarlo de la forma que se quiera, pero todo esto siempre y cuando se reconozca la autoría y se distribuya con la misma licencia que el trabajo original. El documento en formato `LaTeX` se puede encontrar en el siguiente repositorio de `GitHub`: `https://github.com/erseco/ugr_inteligencia_computacional`.

Fdo: Ernesto Serrano Collado

Granada, a 20 de noviembre de 2017

Índice general

Capítulo 1

Introducción

Las redes neuronales (también conocidas como sistemas conexionistas) son un modelo computacional basado en un gran conjunto de unidades neuronales simples (neuronas artificiales), de forma aproximadamente análoga al comportamiento observado en los axones de las neuronas en los cerebros biológicos.

El objetivo de una red neuronal es resolver los problemas de la misma manera que el cerebro humano, aunque las redes neuronales son más abstractas. Los proyectos de redes neuronales modernas suelen trabajar con unos pocos miles a unos pocos millones de unidades neuronales y millones de conexiones, que sigue siendo varios órdenes de magnitud menos complejo que el cerebro humano y más cercano a la potencia de cálculo de un gusano.

Hoy día las redes neuronales son ampliamente utilizadas para el reconocimiento de patrones. Uno de los más típicos ejemplos de red neuronal se basa en la base de datos MNIST[1] que es considerado el "Hola mundo" de la Inteligencia Artificial.

El conjunto de entrenamiento de dicha base de datos contiene 60.000 ejemplos etiquetados. Los ejemplos de entrenamiento son imágenes normalizadas de 28x28 píxeles y se encuentran en el fichero "train-images-idx3-ubyte", mientras que las etiquetas correspondientes a los ejemplos se pueden encontrar en el fichero "train-labels-idx1-ubyte".

El conjunto de prueba, almacenado en el mismo formato que el conjunto de entrenamiento, puede encontrarlo en los ficheros "t10k-images-idx3-ubyte" (imágenes) y "t10k-labels-idx1-ubyte" (etiquetas).

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales donde se deberá evaluar el

[1] `http://yann.lecun.com/exdb/mnist/`

uso de varios tipos de redes neuronales para resolver un problema de OCR[2].

En esta práctica podíamos optar por una implementación desde 0 o bien usar una librería ya creada. Antes de tomar una decisión estuve docmentandome sobre el "estado del arte" de las librerías de redes neuronales.

1.0.1. TensorFlow

TensorFlow es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. Está desarrollada en Python y C++.

1.0.2. Theano

Theano es una biblioteca de computación numérica para Python. En Theano, los cálculos se expresan usando una sintaxis NumPy y se compilan para funcionar eficientemente en arquitecturas de CPU o GPU.

1.0.3. Cognitive Toolkit

Microsoft Cognitive Toolkit, anteriormente conocido como CNTK y en ocasiones denominado "The Microsoft Cognitive Toolkit", es un marco de aprendizaje profundo desarrollado por Microsoft Research. Microsoft Cognitive Toolkit describe las redes neuronales como una serie de pasos computacionales a través de un gráfico dirigido. Está desarrollada en C++.

1.0.4. Keras

Keras es una biblioteca de red neuronal de código abierto escrita en Python. Es capaz de ejecutarse sobre MXNet, Deeplearning4j, Tensorflow, CNTK o Theano. Está diseñada para permitir una rápida experimentación con redes neuronales profundas, se enfoca en ser mínimo, modular y extensible. Fue desarrollado como parte del esfuerzo de investigación del proyecto ONEIROS (Sistema Operativo de Robot Inteligente Neuroelectrónico de código abierto), y su principal autor y mantenedor es François Chollet, un ingeniero de Google. Está desarrollada en Python.

1.0.5. DeepCL

Biblioteca OpenCL para entrenar redes neuronales convolucionales profundas. Está desarrollada en Python.

[2]Optical Character Recognition

1.0.6. mnist-1lnn

Implementación en C de una red neuronal sencilla para resolver el problema de la base de datos MINST. Al estar desarrollada en C es muy veloz en su ejecución a costa de lidiar con una programación a bajo nivel.

1.0.7. Deeplearning4j

Deeplearning4j es una biblioteca de programación de aprendizaje profundo escrita en Java que provee marco con amplio soporte para algoritmos de aprendizaje profundo.

Capítulo 2

Implementación

En un primer momento instalé en mi máquina Keras usando TensorFlow como backend, y ejecuté varios ejemplos en los que noté que dicha libreria es muy exigente si no se tiene una GPU y un procesador potente. Como no dispongo de una máquina con dichas características pensé en utilizar una AMI en Amazon EC2 especializada en 'deep learning' con 2 GPU CUDA.

Con dicha máquina los ejemplos se ejecutaban a muchísima velocidad y los tutoriales de la base de datos MNIST me daban resultados de 2,8 %. Dejé esa opción para más adelante por si pensaba competir con el resto de compañeros en cuando a rendimiento y me propuse utilizar algo que requiriera algo de programación por mi parte.

Probé DeepCL ya que mi máquina tiene una tarjeta Intel HD 5000 que soporta OpenCL, pero el rendimiento con los ejemplos no era demasiado bueno. Además según leí en varios artículos dicha librería tenía errores que hacía que no fuera todo lo óptima que debería ser.

Mas tarde probé la implementación en C "minst-1lnn" que resolvía el problema con una red neuronal muy simple y muy rápida. Pero los resultados no eran demasiado buenos ya que daba entorno al 5 % de error. Porté el código C a Python ya que es una buena forma de ver como funciona el algoritmo utilizado, pero aun dando los mismos resultados la ejecución del mismo era demasiado lenta como para ser usable.

También, tras leer algunos capítulos del libro "Neural Networks and Deep Learning" de Michael Nielsen, decidí probar algunos de los ejemplos ya que da el código base para una red neuronal multicapa bastante sencilla en Python y que se ejecutaba con bastante velocidad al utilizar la librería "numpy".

Estuve utilizando como base los ejemplos de Michael Nielsen adaptándolos para la práctica y realizado diversas mejoras pero no conseguía bajar del

resultado obtenido en la máquina AWS.

Así que tras muchas pruebas finalmente decidí volver a Keras en Amazon AWS probando muchas de las opciones que brinda la librería para conseguir un mejor resultado que el que dan los ejemplos proporcionados por la misma.

Los resultados obtenidos tras 12 épocas con los ejemplos de Keras era de un 99.25 % en el conjunto de prueba, con una tasa de error de 2.8 %. Para mejorar dichos resultados me propuse usar una configuración de capas más agresiva. En los ejemplos utilizaba una capa oculta de tipo convolucional con activación 'relu' de 32 neuronas. Posteriormente fui agregando capas convolucionales duplicando las neuronas de cada una de ellas. Este tipo de configuración requería mas tiempo de proceso pero al final la red sobreaprendía demasiado y aunque daba un error del 0.0 % en el conjunto de entrenamiento daba unos resultados parecidos a los de los ejemplos en el conjunto de prueba.

La estructura de datos utilizada por Keras está organizada por capas, permitiendo de forma fácil ir agregando capas de diferentes tipos. Además tiene una cantidad enorme de tipos de capas predefinidos. Tenemos capas de activación, capas softmax, capas lineales y muchísimos tipos mas que podemos ir agregando de forma sencilla.

La primera capa que agreguemos será la que se defina como capa de entrada y deberá tener el mismo tamaño que los datos de entrada. La segunda capa toma la salida de la primera y establece las dimensiones de salida. Esto significa que la segunda capa tiene tantos nodos como tiene su salida. Esta configuración se irá encadenando hasta la capa de salida que en nuestro caso es de 10 (los números del 0 al 9 que vamos a reconocer).

Una vez que tengamos nuestro modelo debemos compilarlo. La compilación utiliza las librerías del backend utilizado (en nuestro caso TensorFlow). El backend selecciona automáticamente la mejor forma de representar la red para el entrenamiento, y hará que las predicciones se ejecuten en el hardware disponible de la forma mas óptima utilizando la GPU si está disponible.

Al la hora de compilar debemos indicar algunas propiedades adicionales para entrenar la red, como la función de pérdida (loss) el número de épocas así como optimizador utilizado para buscar en diferentes pesos de la red o cualquier métrica opcional que deseemos recopilar durante el entrenamiento. En nuestras pruebas hemos probado diferentes funciones loss así como optimizadores. Para esta práctica como los datos están categorizados hemos utilizado la función categorical_crossentropy pues es la mas adecuada en base a nuestros datos pues están categorizados. De igual forma podemos elegir

el algoritmo optimizador, hemos probado 'sgd' (stocastic gradient descent), 'adam' y 'adadelta' siendo este último el que nos ha dado mejores resultados.

La fase de entrenamiento se ejecutará en un número fijo de iteraciones a través del conjunto de datos llamado 'epochs' (épocas). Definiremos tamaño del lote o numero de imágenes que se evaluarán antes de que se realice una actualización de pesos en la red con el argumento batch_size.

Una vez entrenada nuestra red neuronal ya podemos evaluar su eficiencia utilizando el conjunto de datos de prueba.

2.1. Configuracion final

Tras varias pruebas, mi implementación final fue la siguiente:

- **Número de capas**: Ocho capas.
- **Tamaño de capa de entrada**: 28x28. Tantas neuronas como píxeles en la imagen
- **Tamaño de capa de salida**: 10. Tantas neuronas como posibles soluciones.
- **Función de activación**: relu
- **Tipo de aprendizaje**: Estocástico. Reajuste de pesos tras cada imagen

2.2. Definición de capas

- La primera capa oculta es una capa convolucional llamada Convolution2D. La capa tiene 32 mapas característicos, que con el tamaño de 5×5 y una función de activación del rectificador. Esta es la capa de entrada.
- A continuación definimos una capa de 'pooling' que toma el máximo llamado MaxPooling2D. Está configurado con un tamaño de pool de 2×2.
- Capa convolucional con 15 mapas de características de tamaño 3×3.
- La siguiente capa es una capa de regularización que utiliza una deserción llamada "Dropout". Está configurado para excluir aleatoriamente el 20 % de las neuronas en la capa para reducir el exceso de sobreaprendizaje.

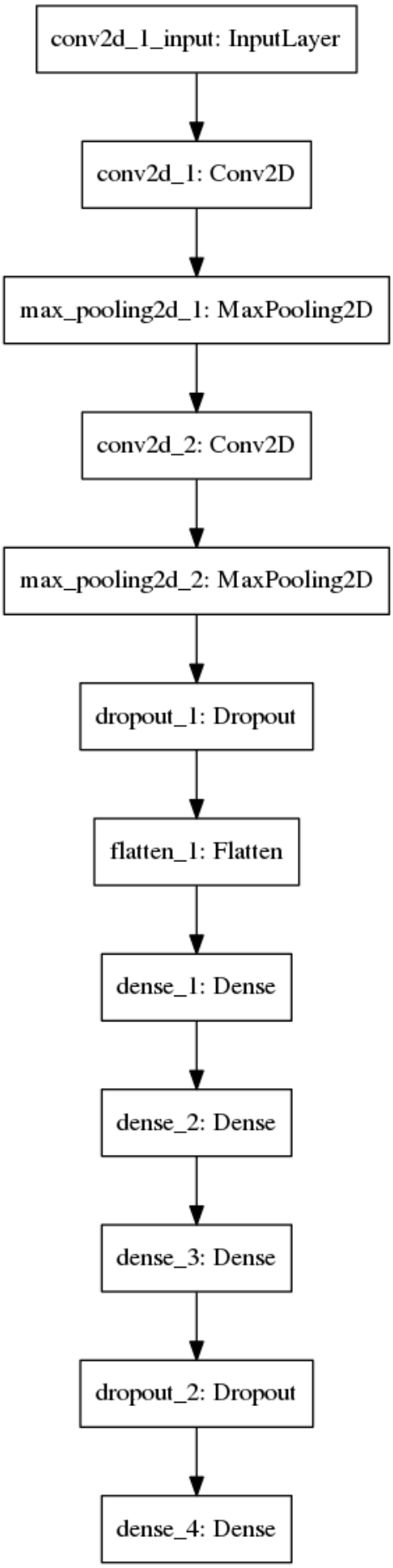
- A continuación se convierte la capa que convierte los datos de la matriz 2D en un vector llamado "Flatten". Permite que la salida sea procesada por capas estándar completamente conectadas.

- A continuación, una capa completamente conectada con 128 neuronas y función de activación del rectificador.

- Capa totalmente conectada con 50 neuronas y activación del rectificador.

- Por último, la capa de salida tiene 10 neuronas para las 10 clases y una función de activación softmax para emitir predicciones probabilisticas para cada clase.

En la figura 2.1 podemos ver como están definidas las capas.


conv2d_1_input: InputLayer
conv2d_1: Conv2D
max_pooling2d_1: MaxPooling2D
conv2d_2: Conv2D
max_pooling2d_2: MaxPooling2D
dropout_1: Dropout
flatten_1: Flatten
dense_1: Dense
dense_2: Dense
dense_3: Dense
dropout_2: Dropout
dense_4: Dense

Capítulo 3

Resultados

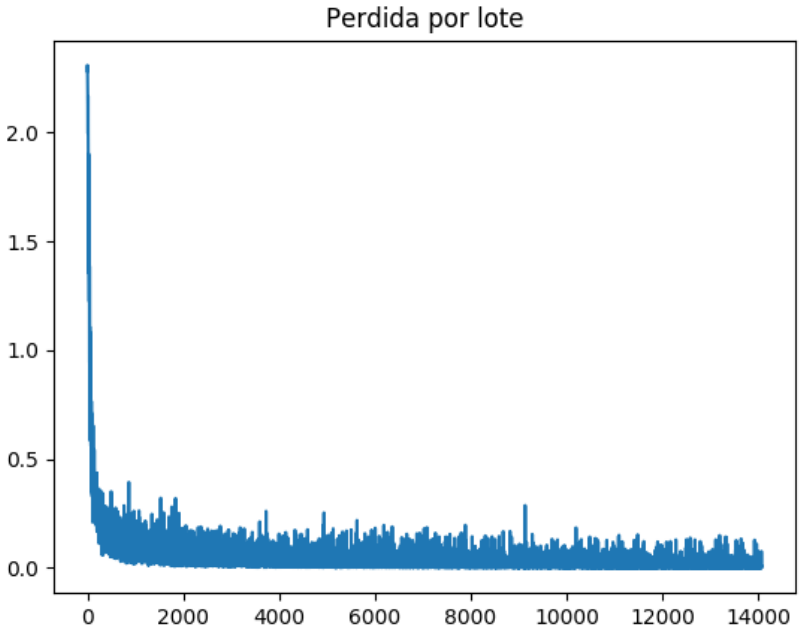
Tras probar diferentes ajustes he detectado que había 'sobreaprendizaje', esto es que aunque se mejora mucho el resultado para el conjunto de entrenamiento, a la hora de aplicar lo aprendido sobre el conjunto de prueba diera malos resultados. Tras ir ajustando los parámetros de las diferentes capas añadidas así como el número de neuronas se ha alcanzado una solución que daba unos resultados mas similares entre ambos conjuntos, siendo por tanto esta solución mucho mejor que un resultado dispar.

Los resultados obtenidos tras diversos ajustes en el algoritmo son bastante buenos, pasamos a detallarlos:

Tiempo de entrenamiento (en segundos)	187
Tasa de error sobre el conjunto de entrenamiento (%)	0.05
Precisión sobre el conjunto de entrenamiento (%)	99.95
Tasa de error sobre el conjunto de prueba (%)	0.64
Precisión sobre el conjunto de prueba (%)	99.36

Tabla 3.1: Tabla de resultados

En la figura 3 podemos ver como ha ido bajando el porcentaje de error en cada lote.


Perdida por lote
2.0
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000
12000
14000

Capítulo 4

Conclusiones

La realización de esta práctica me ha servido como introducción al basto mundo de las redes neuronales. Como nos comentó el profesor, necesitaríamos mucho mas tiempo que un simple cuatrimestre para profundizar mas, pues es un campo muy amplio. Aun así ha sido muy gratificante poner a prueba los conocimientos adquiridos en la clase de teoría y viendo como los algoritmos van aprendiendo a resolver el problema de forma automática.

Aunque en un principio barajé la posibilidad de realizar una implementación por mi mismo, la falta de tiempo por motivos tanto laborales como académicos me lo ha impedido. De ahí a utilizar la librería Keras con TensorFlow como *backend*. El poder utilizar una máquina en Amazon AWS con varias GPU me ha facilitado el poder probar en horas diferentes configuraciones que en mi máquina me hubiera llevado semanas a costa del desembolso económico.

Capítulo 5

Anexos

5.1. Código fuente de la red neuronal implementada usadon la librería Keras

```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Activation
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

import keras.callbacks as cb

import numpy as np

from keras.utils import plot_model

import time


from matplotlib import pyplot as plt
plt.switch_backend('agg')

class LossHistory(cb.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        batch_loss = logs.get('loss')
        self.losses.append(batch_loss)


batch_size = 128
num_classes = 10
epochs = 30
# epochs = 1

# input image dimensions
img_rows, img_cols = 28, 28

```

```
# the data, shuffled and 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)

model = Sequential()

# model.add(Conv2D(30, kernel_size=(5, 5), padding='valid',
    input_shape=(1, 28, 28), activation='relu'))
model.add(Conv2D(32, kernel_size=(5, 5),
                 activation='relu',
                 input_shape=input_shape))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.summary()

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

history = LossHistory()

start = time.time()

model.fit(x_train, y_train,
          batch_size=batch_size,
          callbacks=[history],
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))


end = time.time()
print("training time:")
print(end - start)
```

```
99
100 score = model.evaluate(x_train, y_train, verbose=0)
101 print('Train loss:', score[0]*100, '%')
102 print('Train accuracy:', score[1]*100, '%')
103
104 score = model.evaluate(x_test, y_test, verbose=0)
105 print('Test loss:', score[0]*100, '%')
106 print('Test accuracy:', score[1]*100, '%')
107
108 result = model.predict(x_test)
109
110 class_result=np.argmax(result,axis=-1)
111
112 print(class_result)
113
114 np.savetxt('test.txt', class_result.astype('int'), delimiter="")
115 plot_model(model, to_file='model.png')
116
117
118 plt.switch_backend('agg')
119 plt.ioff()
120 fig = plt.figure()
121
122 ax = fig.add_subplot(111)
123 ax.plot(history.losses)
124 ax.set_title('Perdida por lote')
125
126 fig.savefig('plot.png')
```

5.2. Salida del script

```
1  Using TensorFlow backend.
2  x_train shape: (60000, 28, 28, 1)
3  60000 train samples
4  10000 test samples
5  _________________________________________________________________
6  Layer (type)                 Output Shape              Param #
7  =================================================================
8  conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
9  _________________________________________________________________
10 max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
11 _________________________________________________________________
12 conv2d_2 (Conv2D)            (None, 10, 10, 64)        18496
13 _________________________________________________________________
14 max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
15 _________________________________________________________________
16 dropout_1 (Dropout)          (None, 5, 5, 64)          0
17 _________________________________________________________________
18 flatten_1 (Flatten)          (None, 1600)              0
19 _________________________________________________________________
20 dense_1 (Dense)              (None, 512)               819712
21 _________________________________________________________________
22 dense_2 (Dense)              (None, 128)               65664
23 _________________________________________________________________
24 dense_3 (Dense)              (None, 50)                6450
25 _________________________________________________________________
26 dropout_2 (Dropout)          (None, 50)                0
27 _________________________________________________________________
```

```
28 dense_4 (Dense)              (None, 10)                510
29 =================================================================
30 Total params: 911,664
31 Trainable params: 911,664
32 Non-trainable params: 0
33 _________________________________________________________________
34 Train on 60000 samples, validate on 10000 samples
35 Epoch 1/30
36 2017-11-18 12:50:11.715760: W tensorflow/core/platform/
       cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
        use SSE4.1 instructions, but these are available on your machine
       and could speed up CPU computations.
37 2017-11-18 12:50:11.715800: W tensorflow/core/platform/
       cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
        use SSE4.2 instructions, but these are available on your machine
       and could speed up CPU computations.
38 2017-11-18 12:50:11.715810: W tensorflow/core/platform/
       cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
        use AVX instructions, but these are available on your machine and
        could speed up CPU computations.
39 2017-11-18 12:50:11.715816: W tensorflow/core/platform/
       cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
        use AVX2 instructions, but these are available on your machine
       and could speed up CPU computations.
40 2017-11-18 12:50:11.715825: W tensorflow/core/platform/
       cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
        use FMA instructions, but these are available on your machine and
        could speed up CPU computations.
41 2017-11-18 12:50:11.906171: I tensorflow/stream_executor/cuda/
       cuda_gpu_executor.cc:893] successful NUMA node read from SysFS had
        negative value (-1), but there must be at least one NUMA node, so
        returning NUMA node zero
42 2017-11-18 12:50:11.906675: I tensorflow/core/common_runtime/gpu/
       gpu_device.cc:955] Found device 0 with properties:
43 name: Tesla K80
44 major: 3 minor: 7 memoryClockRate (GHz) 0.8235
45 pciBusID 0000:00:1e.0
46 Total memory: 11.17GiB
47 Free memory: 11.11GiB
48 2017-11-18 12:50:11.906703: I tensorflow/core/common_runtime/gpu/
       gpu_device.cc:976] DMA: 0
49 2017-11-18 12:50:11.906711: I tensorflow/core/common_runtime/gpu/
       gpu_device.cc:986] 0:   Y
50 2017-11-18 12:50:11.906730: I tensorflow/core/common_runtime/gpu/
       gpu_device.cc:1045] Creating TensorFlow device (/gpu:0) -> (device
       : 0, name: Tesla K80, pci bus id: 0000:00:1e.0)
51 60000/60000 [==============================] - 9s - loss: 0.4390 - acc
       : 0.8630 - val_loss: 0.0676 - val_acc: 0.9786
52 Epoch 2/30
53 60000/60000 [==============================] - 6s - loss: 0.1182 - acc
       : 0.9691 - val_loss: 0.0462 - val_acc: 0.9854
54 Epoch 3/30
55 60000/60000 [==============================] - 6s - loss: 0.0871 - acc
       : 0.9784 - val_loss: 0.0334 - val_acc: 0.9896
56 Epoch 4/30
57 60000/60000 [==============================] - 6s - loss: 0.0721 - acc
       : 0.9819 - val_loss: 0.0324 - val_acc: 0.9899
58 Epoch 5/30
59 60000/60000 [==============================] - 6s - loss: 0.0598 - acc
       : 0.9850 - val_loss: 0.0290 - val_acc: 0.9916
60 Epoch 6/30
```

```
60000/60000 [==============================] - 6s - loss: 0.0516 - acc
    : 0.9868 - val_loss: 0.0281 - val_acc: 0.9914
Epoch 7/30
60000/60000 [==============================] - 6s - loss: 0.0474 - acc
    : 0.9878 - val_loss: 0.0267 - val_acc: 0.9922
Epoch 8/30
60000/60000 [==============================] - 6s - loss: 0.0417 - acc
    : 0.9896 - val_loss: 0.0283 - val_acc: 0.9926
Epoch 9/30
60000/60000 [==============================] - 6s - loss: 0.0354 - acc
    : 0.9902 - val_loss: 0.0263 - val_acc: 0.9927
Epoch 10/30
60000/60000 [==============================] - 6s - loss: 0.0342 - acc
    : 0.9910 - val_loss: 0.0273 - val_acc: 0.9929
Epoch 11/30
60000/60000 [==============================] - 6s - loss: 0.0317 - acc
    : 0.9920 - val_loss: 0.0259 - val_acc: 0.9931
Epoch 12/30
60000/60000 [==============================] - 6s - loss: 0.0285 - acc
    : 0.9924 - val_loss: 0.0269 - val_acc: 0.9931
Epoch 13/30
60000/60000 [==============================] - 6s - loss: 0.0276 - acc
    : 0.9932 - val_loss: 0.0286 - val_acc: 0.9930
Epoch 14/30
60000/60000 [==============================] - 6s - loss: 0.0257 - acc
    : 0.9934 - val_loss: 0.0259 - val_acc: 0.9938
Epoch 15/30
60000/60000 [==============================] - 6s - loss: 0.0231 - acc
    : 0.9943 - val_loss: 0.0273 - val_acc: 0.9933
Epoch 16/30
60000/60000 [==============================] - 6s - loss: 0.0216 - acc
    : 0.9944 - val_loss: 0.0310 - val_acc: 0.9931
Epoch 17/30
60000/60000 [==============================] - 6s - loss: 0.0194 - acc
    : 0.9950 - val_loss: 0.0302 - val_acc: 0.9934
Epoch 18/30
60000/60000 [==============================] - 6s - loss: 0.0190 - acc
    : 0.9949 - val_loss: 0.0282 - val_acc: 0.9931
Epoch 19/30
60000/60000 [==============================] - 6s - loss: 0.0178 - acc
    : 0.9951 - val_loss: 0.0277 - val_acc: 0.9937
Epoch 20/30
60000/60000 [==============================] - 6s - loss: 0.0170 - acc
    : 0.9958 - val_loss: 0.0318 - val_acc: 0.9933
Epoch 21/30
60000/60000 [==============================] - 6s - loss: 0.0163 - acc
    : 0.9958 - val_loss: 0.0307 - val_acc: 0.9934
Epoch 22/30
60000/60000 [==============================] - 6s - loss: 0.0147 - acc
    : 0.9963 - val_loss: 0.0296 - val_acc: 0.9939
Epoch 23/30
60000/60000 [==============================] - 6s - loss: 0.0140 - acc
    : 0.9962 - val_loss: 0.0372 - val_acc: 0.9929
Epoch 24/30
60000/60000 [==============================] - 6s - loss: 0.0136 - acc
    : 0.9964 - val_loss: 0.0340 - val_acc: 0.9933
Epoch 25/30
60000/60000 [==============================] - 6s - loss: 0.0131 - acc
    : 0.9968 - val_loss: 0.0292 - val_acc: 0.9936
Epoch 26/30
60000/60000 [==============================] - 6s - loss: 0.0130 - acc
    : 0.9965 - val_loss: 0.0308 - val_acc: 0.9935
```

```
Epoch 27/30
60000/60000 [==============================] - 6s - loss: 0.0119 - acc
    : 0.9966 - val_loss: 0.0284 - val_acc: 0.9940
Epoch 28/30
60000/60000 [==============================] - 6s - loss: 0.0115 - acc
    : 0.9968 - val_loss: 0.0340 - val_acc: 0.9932
Epoch 29/30
60000/60000 [==============================] - 6s - loss: 0.0114 - acc
    : 0.9970 - val_loss: 0.0351 - val_acc: 0.9935
Epoch 30/30
60000/60000 [==============================] - 6s - loss: 0.0115 - acc
    : 0.9970 - val_loss: 0.0357 - val_acc: 0.9936
training time: 187.433431149 seconds
Train loss: 0.205168554096
Train accuracy: 99.9566666667 %
Test loss: 3.57385053659
Test accuracy: 99.36 %
```