Scaling ML using Cloud ML Engine

Contents

1. Data Science Workflow

- Goal is to standardise the development of models
    - Checklist of necessary technical steps

Vision: Achieve an first end-to-end model in production within a productincrement of 10 weeks

Scale out: Scale without having to rewrite your model

Data Science Process - Proposal

Step 1: Preparation	**Step 2: Data exploration and model building**	**Step 3: Model deployment**
1.1 Project setup	2.1 One click to start the Data Scientist Exploration Environment	3.1 Model serving
1.2 Quick data exploration	2.2 Setup for Data exploration and Machine Learning	3.2 Model deployment (load balancing ...)
1.3 Data visualization	2.3 Deep dive in data exploration	3.3 Model versioning
-	2.4 Data visualization and profiling	3.4 Model monitoring
-	2.5 Feature engineering	-
-	2.6 Model building	-
-	2.7 Model training	-
-	2.8 Model testing	-
-	2.9 Hyparameters tuning	-
-	2.10 Model visualisation	-

steps 1 and 2 can be done only locally

We will look today at

```
- 2.7 How to train a model?
- 2.8 How to evaluate a model?
- 3.1 How to make predictions?
- 3.2 How to deploy a model?
```

Should help to answer:

- Where do we need to improve?
- Where to go next?

Process description will be refined.

2. Where to develop?

Locally using

- Google SDK on your laptop (CLI)
- your IDE (e.g. PyCharme)
- Juypter Notebook
- `gcloud ml-engine local`

Simple Cloud setup using

- [Google Console (https://console.cloud.google.com/)](https://console.cloud.google.com/) Compute Engine with 5 GB storage
- Cloud Editor
- datalab
- `gcloud ml-engine(local)`

Proposal

- when to migrate to GCP:
  - distribute learning on several machines
  - serve model 24/7

develop locally

3. Hands-ON Tutorial: Running MNIST on ML-Engine

- deep dive into step 2 and 3 of Data Science process
- data exploration is omitted as a curated dataset is used

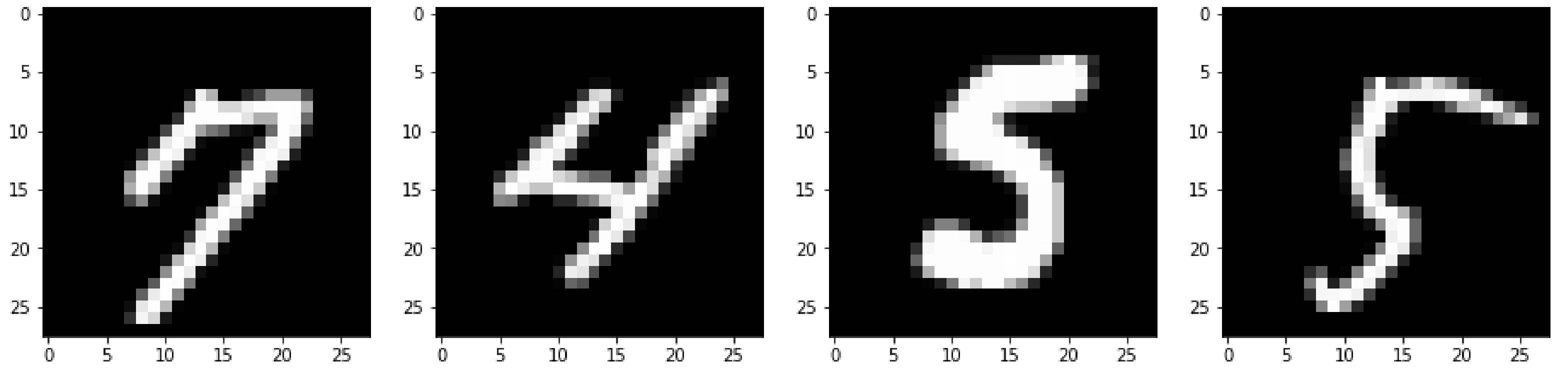
Adapted from Notebook (https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/machine_learning/cloudmle/cloudmle.ipynb) of Google Coursera Course Serverless Machine Learning with Tensorflow on Google Cloud Platform (https://www.coursera.org/learn/serverless-machine-learning-gcp/)

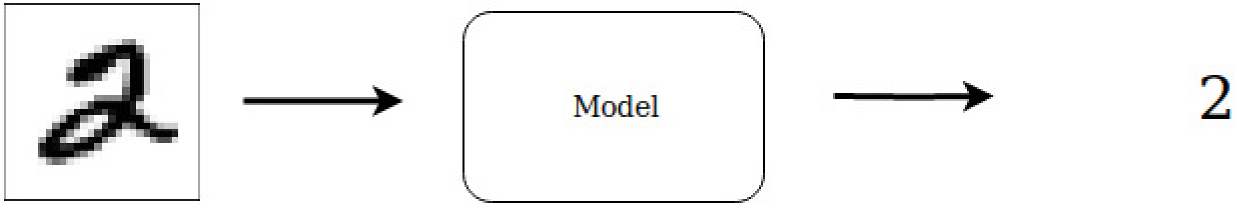
- In order to import from `src` functionality later in this notebook, it is necessary to change to the root directory of the notebooks directory

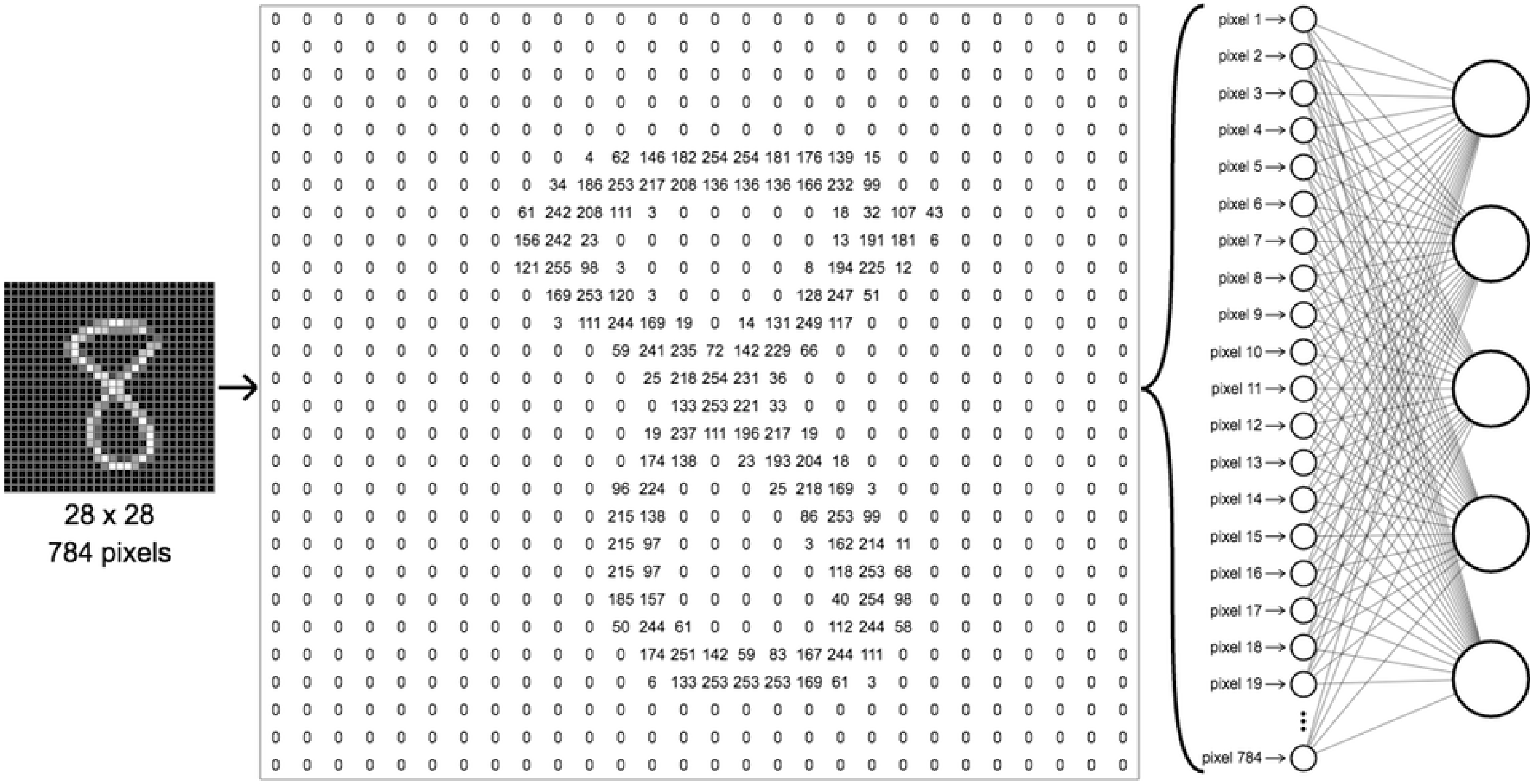
MNIST use-case

- recognise hand-written digits (e.g. on a postal card)

In [20]:
```
from src.utils.mnist_utils import plot_mnist_testdata
plot_mnist_testdata()
```







Why MNIST?

- images need pre-processing
- black and white images are numeric vectors

3.A Setup

1. ML Engine Runtimes
2. Repository Structure
3. Configuration Variables
    - Environment variables to set
    - How to add them to your runtime
4. Setup `gcloud` runtime

Create conda environement

- `conda env create -f environment.yml -n env_gcp_dl`

1 ML Engine Runtimes

Default ML-Engine Runtimes depend on the Tensorflow Version

- [list of runtimes (https://cloud.google.com/ml-engine/docs/tensorflow/runtime-version-list)](https://cloud.google.com/ml-engine/docs/tensorflow/runtime-version-list)
- Current Version: `1.12`

In [ ]:
```
#!conda install tensorflow=1.12
```

In [21]:
```
import tensorflow as tf
tf.__version__
```

Out[21]:
```
'1.12.0'
```

2. Repository structure

In [ ]:
```
ls | grep "/\|yaml"
```

Key Directories containing information

```
.
+-- data
+-- src
|   +-- models
|   +-- packages
config.yaml
```

In the next step the contents of `config.yaml` (config.yaml) will be important

3. GCP Environment Variables

- `PROJECT_ID`: unique ID that identifies your project, e.g. **ml-productive-pipeline-12345**
- `BUCKET`: BLOB-store ID. Each project has per default an bucket named by the `PROJECT_ID`
- `REGION`: Which data center to use

Additional Environment Variables needed for ML-Engine

- `PKG_NAME`: Package Name which will contain your model
- `TF_VERSION`: Tensorflow Version

In [ ]:

```
import yaml
from pprint import pprint
with open("config.yaml", "r", encoding = "utf8") as f:
    config = yaml.load(f)
pprint(config)
```

Adding Environment Variables to your runtime

- add variables **persistently** to the runtime of your kernel from jupyter (or datalab)
- use `os.environ` dictionary

In [ ]:
```
import os
PROJECT = config['project-id']
REGION = config['region'] # Choose an available region for Cloud MLE from http
s://cloud.google.com/ml-engine/docs/regions.
BUCKET = config['bucket'] # REPLACE WITH YOUR BUCKET NAME. Use a regional bucke
t in the region you selected.
PKG_NAME = config['pkg-name']

os.environ['PROJECT'] = PROJECT
os.environ['BUCKET'] = BUCKET
os.environ['REGION'] = REGION
os.environ['TFVERSION'] = str(config['tf-version'])  # Tensorflow version 1.4 b
efore
os.environ['PKG_NAME'] = config['pkg-name']
```

Now, you can access the environement variable in the terminal where your jupyter, datalab or iphyton.

In [ ]:
```
!echo "Using Tensorflow Version: $TFVERSION"
```

4. Setup gcloud runtime

In [ ]:

```
%%bash
gcloud config set project $PROJECT
gcloud config set compute/region $REGION
```

Access Control

- not necessary if you use
  - datalab
  - local sdk
- Service Accounts (Creating and Managing Service Accounts (https://cloud.google.com/iam/docs/creating-managing-service-accounts))
  - need be assigned read/write permission to BUCKET

Beyond Scripting: Packaging up the code

Take your code and put into a standard Python package structure, see [pkg_mnist_fnn/model.py (../src/pkg_mnist_fnn/model.py)](../src/pkg_mnist_fnn/model.py)

Key-Idea:

- define entry point which can be called
- write all tasks as a function (callable)

Why a package?

- can be called from other scripts `import model`

`model.py`

load most recent version, if needed:

In [ ]:
```
%load src/pkg_mnist_fnn/model.py
```

```python
import tensorflow as tf
import numpy as np

from .utils import load_data
##########################################################################
#Factor into config:
N_PIXEL = 784
OUTDIR = 'trained'
USE_TPU = False
EPOCHS = 5

IMAGE_SIZE = 28 * 28
NUM_LABELS = 10
BATCH_SIZE = 128
##########################################################################
def parse_images(x):
    return x.reshape(len(x), -1).astype('float32')


def parse_labels(y):
    return y.astype('int32')
```

```python
def numpy_input_fn(images: np.ndarray,
                   labels: np.ndarray,
                   mode=tf.estimator.ModeKeys.EVAL):
    """
    Return depending on the `mode`-key an Interator which can be use to feed i
nto
    the Estimator-Model.

    Alternative if a `tf.data.Dataset` named `dataset` would be created:
    `dataset.make_one_shot_iterator().get_next()`
    """
    if mode == tf.estimator.ModeKeys.TRAIN:
        _epochs = EPOCHS
        _shuffle = True
        _num_threads = 2
    else:
        _epochs = 1
        _shuffle = False
        _num_threads = 1

    return tf.estimator.inputs.numpy_input_fn(
        {'x': images},
        y=labels,
        batch_size=BATCH_SIZE,
        num_epochs=_epochs, # Boolean, if True shuffles the queue.
                            # Avoid shuffle at prediction time.
        # Boolean, if True shuffles the queue. Avoid shuffle at prediction
        shuffle=_shuffle,
        queue_capacity=1000, # Integer, number of threads used for reading
        # and enqueueing. To have predicted order of reading and enqueueing,
        # such as in prediction and evaluation mode, num_threads should be 1.
        num_threads=_num_threads
    )
```

```python
def serving_input_fn():
    feature_placeholders = {
        'x': tf.placeholder(tf.float32, shape=[None, N_PIXEL])
    }
    features = feature_placeholders
    return tf.estimator.export.ServingInputReceiver(
         features=features,
         receiver_tensors=feature_placeholders,
         receiver_tensors_alternatives=None
         )
```

```python
def train_and_evaluate(args):
    """
    Utility function for distributed training on ML-Engine
    www.tensorflow.org/api_docs/python/tf/estimator/train_and_evaluate
    """
    # Load Data in Memoery
    (x_train, y_train), (x_test, y_test) = load_data(
        rel_path=args['data_path'])

    x_train = parse_images(x_train)
    x_test = parse_images(x_test)

    y_train = parse_labels(y_train)
    y_test = parse_labels(y_test)

    model = tf.estimator.DNNClassifier(
        hidden_units=[256, 128, 64],
        feature_columns=[tf.feature_column.numeric_column(
            'x', shape=[N_PIXEL, ])],
        model_dir=args['output_dir'],
        n_classes=10,
        optimizer=tf.train.AdamOptimizer,
        # activation_fn=,
        dropout=0.2,
        batch_norm=False,
        loss_reduction='weighted_sum',
        warm_start_from=None,
        config = None
    )
    train_spec = tf.estimator.TrainSpec(
    # see next slide
```

```python
def train_and_evaluate(args):
    """
    Utility function for distributed training on ML-Engine
    www.tensorflow.org/api_docs/python/tf/estimator/train_and_evaluate
    """
    # see previous slide

    model = tf.estimator.DNNClassifier(
    # see previous slide
    )
    train_spec = tf.estimator.TrainSpec(
        input_fn=numpy_input_fn(
            x_train, y_train, mode=tf.estimator.ModeKeys.TRAIN),
        max_steps=args['train_steps'],
        hooks = None
    )
    exporter = tf.estimator.LatestExporter('exporter', serving_input_fn)
    eval_spec = tf.estimator.EvalSpec(
        input_fn=numpy_input_fn(
            x_test, y_test, mode=tf.estimator.ModeKeys.EVAL),
        steps=None,
        start_delay_secs=args['eval_delay_secs'],
        throttle_secs=args['min_eval_frequency'],
        exporters=exporter
    )
    tf.estimator.train_and_evaluate(
        estimator=model, train_spec=train_spec, eval_spec=eval_spec)
```

`task.py`

load most recent file using:

In [ ]:
```
%load src/pkg_mnist_fnn/task.py
```

```python
"""
Parse arguments and call main function
"""
import os
import argparse
import shutil

from .model import train_and_evaluate

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--data_path',
        help='GCS or local path to training data',
        required=True
    )
    parser.add_argument(
        '--output_dir',
        help='GCS location to write checkpoints and export models',
        required=True
    )
    parser.add_argument(
        '--train_batch_size',
        help='Batch size for training steps',
        type=int,
        default='128'
    )
    parser.add_argument(
        '--train_steps',
        help='Steps to run the training job for',
        type=int,
        default='200'
    )
```

```python
    parser.add_argument(
            '--hidden_units',
            help='List of hidden layer sizes to use for DNN feature columns',
            nargs='+',
            type=int,
            default=[128, 64, 32]
        )
        parser.add_argument(
            '--job_dir',
            help='this model ignores this field, but it is required by gcloud',
            default='junk'
        )
        # Eval arguments
        parser.add_argument(
            '--eval_delay_secs',
            help='How long to wait before running first evaluation',
            default=1,
            type=int
        )
        parser.add_argument(
            '--min_eval_frequency',
            help='Seconds between evaluations',
            default=10,
            type=int
        )

        args = parser.parse_args().__dict__

        OUTDIR = args['output_dir']
        # ########################################
        # # Train and Evaluate (use TensorBoard to visualize)
        train_and_evaluate(args)
```

3.B Train using ML-Engine on

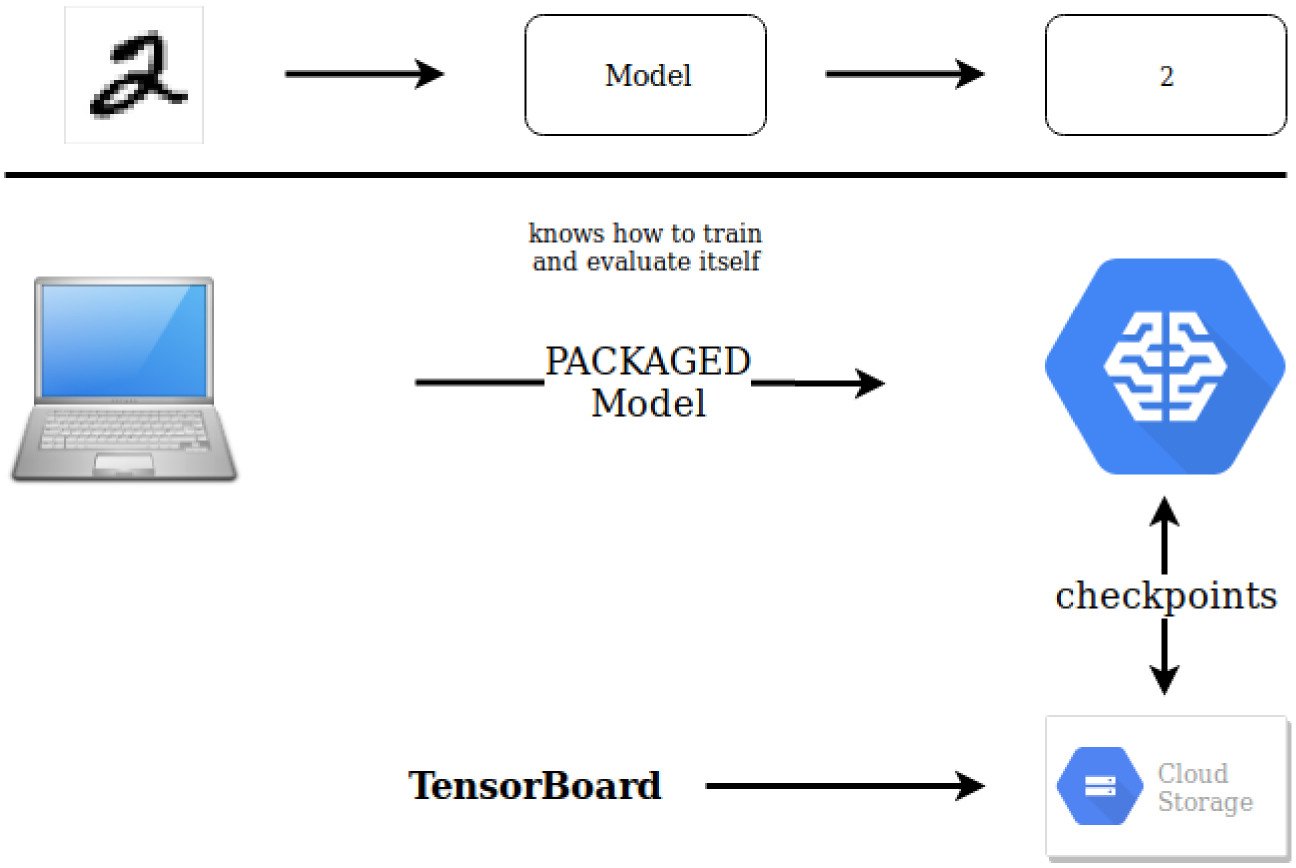
your local machine

1. Call your python script (module)
2. Use `gcloud ml-engine local train`

a cluster of machines using ML-Engine service

1. Use `gcloud ml-engine train`

Modeling and ML-Engine



- Environment Variables with absolut paths to relevant folders:
  - `PWD`: where your project folder lies
  - `PKG_NAME`: Self-Contained Package to be exported into `site-packages` in `venv`
  - `trained`: Where to store checkpoints (logs, weights, graph)

In [ ]:

```
%%bash
echo "Working Directory:     $PWD"
echo "Local data Directory:  $PWD/data"
echo "Package Directory:     $PWD/src/$PKG_NAME"
echo "Saved Model Directory: $PWD/src/$PKG_NAME/trained/"
rm -rf $PWD/src/$PKG_NAME/trained/ # start fresh
echo "Erased previously saved models"
```

1. Running the Python `module` without gcp ml-engine

- Entry point is defined in `task.py`
    - parses command line arguments
- conda env has to be active

In [ ]:
```
%%bash
source activate gcp_dl
rm -rf $PKG_NAME.tar.gz ${PWD}/$PKG_NAME/trained
export PYTHONPATH=${PYTHONPATH}:${PWD}/../
echo "Python Package Path: src.${PKG_NAME}.task"
```

In [ ]:
```
%%bash
source activate gcp_dl
python -m src.${PKG_NAME}.task \
   --data_path="${PWD}/data" \
   --output_dir=${PWD}/src/${PKG_NAME}/trained \
   --train_steps=1000 \
   --job_dir=tmp
echo "Saved model: ${PWD}/src/${PKG_NAME}/trained/export/exporter/ "
```

Saved Model

In [ ]:
```
%%bash
DATE=$(ls $PWD/src/$PKG_NAME/trained/export/exporter/ |tail -1)
echo "Date as integer: $DATE"
echo
date -d @${DATE}
```

And we would be ready to deploy

... but of course not without looking at performance metrics or predictions!

2. Training using `gcloud ml-engine local train`

- continue training using `ml-engine local`

In [ ]:

```
%%bash
source activate gcp_dl
gcloud ml-engine local train \
   --module-name=${PKG_NAME}.task \
   --package-path=${PWD}/src/${PKG_NAME} \
   -- \
   --data_path="${PWD}/data" \
   --output_dir=${PWD}/src/${PKG_NAME}/trained \
   --train_steps=3000 \
   --job_dir=./tmp
```

In [ ]:

```
!gcloud ml-engine local train  --help
```

3. Training Cloud using `gcloud ml-engine train`

- a copy of the data is in Google Storage (buckets)
- `gcloud ml-engine` output is saved to `OUTDIR`in Google Storage
    - checkpoints (logs)
    - model graph and weights
- data is copied to Google Storage

*NOTE: No with-spaces behind line break symbol *

In [ ]:
```
OUTDIR = '/'.join(['gs:/', BUCKET, PKG_NAME, 'trained'])
os.environ['OUTDIR'] = OUTDIR
```

In [ ]:
```
!gsutil -m cp ${PWD}/data/mnist/raw/mnist.npz gs://${BUCKET}/$PKG_NAME/data/mni
st.npz
```

Start Job

In [ ]:

```
%%bash
OUTDIR=gs://${BUCKET}/$PKG_NAME/trained
JOBNAME=mnist_$(date -u +%y%m%d_%H%M%S)
echo $OUTDIR $REGION $JOBNAME
gsutil -m rm -rf $OUTDIR
gcloud ml-engine jobs submit training $JOBNAME \
   --region=$REGION \
   --module-name=$PKG_NAME.task \
   --package-path=${PWD}/src/$PKG_NAME \
   --staging-bucket=gs://$BUCKET \
   --scale-tier=BASIC \
   --python-version 3.5 \
   --runtime-version=$TFVERSION \
   -- \
   --data_path="gs://${BUCKET}/$PKG_NAME/data/" \
   --output_dir=$OUTDIR \
   --train_steps=5000 \
   --job_dir=$OUTDIR/jobs
```

Fetch logs from ml-engine job

- replace *mnist_190226_135612* with your `JOBNAME`

In [ ]:
```
!gcloud ml-engine jobs describe    mnist_190226_135612
```

In [ ]:
```
!gcloud ml-engine jobs stream-logs mnist_190226_135612
```

Check Results in TensorBoard

- metrics and variables are inspected from the logs, called checkpoints (`ckpt`)
- Dashboard on localhost: `TensorBoard`

Inspect Models trained on your machine:

- `tensorboard --logdir src/pkg_mnist_fnn/trained`

In [ ]:

```
%%bash
source activate gcp_dl
tensorboard --logdir $PWD/src/$PKG_NAME/trained
```

Or trained on GCP, where results are store in Google Cloud Storage

Deploy model - from any previous step

- `tf.estimator.LatestExporter` is used to store a model for deployment in the cloud
- See also: `tf.estimator.export`, `tf.saved_model`

[Link to Console (https://console.cloud.google.com/)](https://console.cloud.google.com/)

Check that a model has been saved on your Bucket:

In [ ]:
```
%%bash
gsutil ls gs://${BUCKET}/${PKG_NAME}/trained/export/exporter
```

Deploy

Identifier for deployed model:

- `MODEL_NAME`
- `MODEL_VERSION`

In [ ]:

```
%%bash
MODEL_NAME="MNIST_MLENGINE"
MODEL_VERSION="v2"
MODEL_LOCATION=$(gsutil ls gs://${BUCKET}/$PKG_NAME/trained/export/exporter | t
ail -1)
echo "Run these commands one-by-one (the very first time, you'll create a model
and then create a version)"
#gcloud ml-engine versions delete ${MODEL_VERSION} --model ${MODEL_NAME}
#gcloud ml-engine models delete ${MODEL_NAME}
gcloud ml-engine models   create ${MODEL_NAME} --regions $REGION
gcloud ml-engine versions create ${MODEL_VERSION} --model ${MODEL_NAME} --origi
n ${MODEL_LOCATION} --runtime-version $TFVERSION
```

Predictions

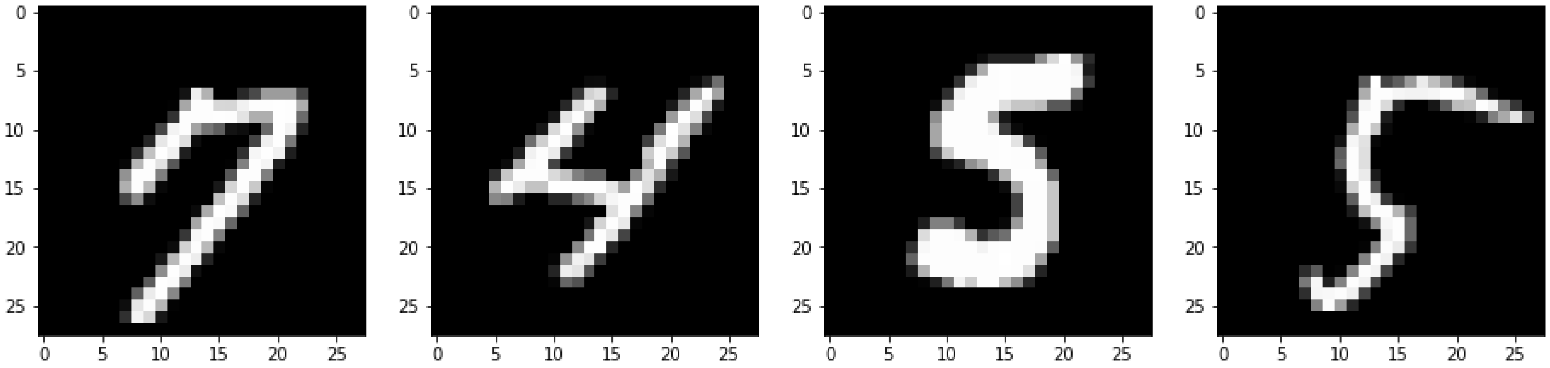
1. Using the Model saved by Python Module
2. Using Model saved by `ml-engine local`
3. Using Model trained online

Tools get predictions:

- Command Line Interfaces
    - `gcloud ml-engine local predict`
    - `gcloud ml-engine predict`
- Python Client

Let's look again at our four examples

In [22]:
```
from src.utils.mnist_utils import plot_mnist_testdata
plot_mnist_testdata()
```



ML-Engine: `ml-engine local predict`

- Using Model saved
  - Python module
  - `ml-engine local`

In [ ]:
```
%%bash
source activate gcp_dl
model_dir=$(ls $PWD/src/$PKG_NAME/trained/export/exporter/ | tail -1)
echo "Selected Model:  $model_dir"
gcloud ml-engine local predict \
    --model-dir=${PWD}/src/${PKG_NAME}/trained/export/exporter/${model_dir} \
    --json-instances=./data/test.json \
    --verbosity debug > data/test_predictions
cat data/test_predictions
```

In [ ]:

```
%%bash
gcloud ml-engine local predict --help
```

Online Prediction - Command Line

- same output format as before

In [ ]:

```
%%bash
gcloud ml-engine predict --model=MNIST_MLENGINE --version=v1 --json-instances=data/test.json
```

Check Console

Online Prediction - Python Client

- Get predictions using the [Python-Client-Library, see Tutorial (https://cloud.google.com/ml-engine/docs/tensorflow/python-client-library)](https://cloud.google.com/ml-engine/docs/tensorflow/python-client-library).
- [API-Reference (https://cloud.google.com/ml-engine/reference/rest/)](https://cloud.google.com/ml-engine/reference/rest/)
- service account authentification: [link (https://cloud.google.com/iam/docs/creating-managing-service-accounts)](https://cloud.google.com/iam/docs/creating-managing-service-accounts)

In [ ]:
```
from oauth2client.client import GoogleCredentials
from googleapiclient import
from googleapiclient import errors
import json
```

In [ ]:
```
api = discovery.build(serviceName='ml', version='v1',
                      http=None,
                      discoveryServiceUrl='https://www.googleapis.com/discover
y/v1/apis/{api}/{apiVersion}/rest',
                      developerKey=None,
                      model=None,
                      #requestBuilder= googleapiclient.http.HttpRequest,
                      credentials=None,
                      cache_discovery=True,
                      cache=None)
```

```
In [ ]: MODEL_NAME = 'MNIST_MLENGINE'
        VERSION = 'v1'
```

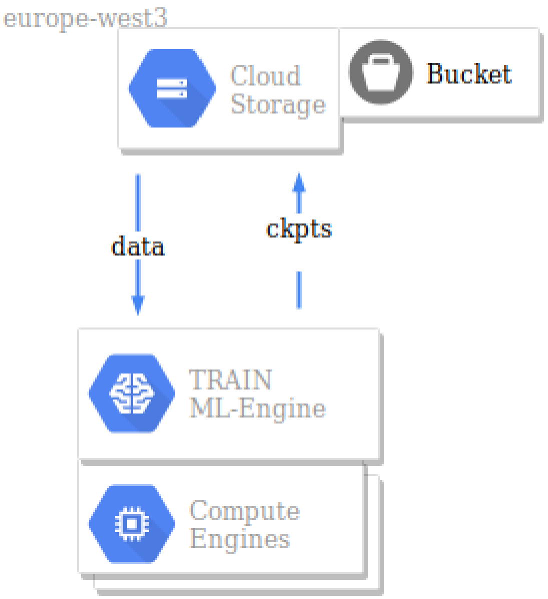
```
In [ ]: # Load data
        from src.pkg_mnist_fnn.utils import load_data
        from src.pkg_mnist_fnn.model import parse_images
        (_,_), (x_test, y_test) = load_data(rel_path='data')
        N=4
        test_indices = np.random.randint(low=0, high=len(y_test), size=N)
        x_test, y_test = x_test[test_indices], y_test[test_indices]
        x_test = parse_images(x_test).tolist()

        eol = "\r\n"
        n_lines = len(y_test)
        instances = []
        with open("data/test.json", "r") as f:
            for image, label in zip(x_test, y_test):
                instances.append({"x": image}) #, "y": int(label)}
```

```
In [ ]: project_id = 'projects/{}/models/{}/versions/{}'.format(PROJECT, MODEL_NAME, VE
        RSION)
        request_data = {"instances":
            instances
        }
        request = api.projects().predict(body=request_data, name=project_id).execute()
        print(request)
```

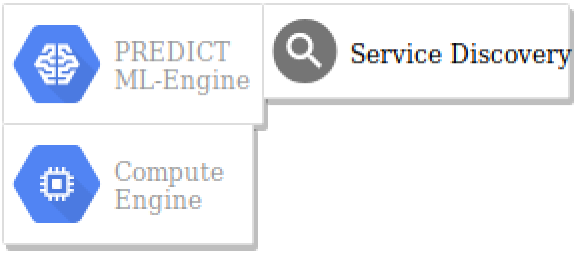
```
In [ ]: for i, pred in enumerate(request['predictions']):
            print("Predicted class: {}, True Class:\t{}".format(pred['classes'][0], y_t
        est[i]))
```

Recap

TRAIN



PREDICT



Outlook

- Add different models types
    - different layers of abstraction in tensorflow
    - sklearn
- Show how to use `ml-engine` in SQL in BigQuery

Appendix

Notes on Jupyter Slides

- Activate: View -> Cell Toolbar -> Slideshow
- [nbextensions (https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/install.html)](https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/install.html)
  - [split cells vertically (https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/splitcell/readme.htm](https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/splitcell/readme.htm)
  - install into base conda environment
- [RISE (https://damianavila.github.io/RISE/installation.html)](https://damianavila.github.io/RISE/installation.html) for interactive presentations
  - using conda: `conda install -c damianavila82 rise`
  - activte scrolling in Notebook-Metadata, see [link (https://damianavila.github.io/RISE/customize.html#config-right-scroll)](https://damianavila.github.io/RISE/customize.html#config-right-scroll)
  - adapt width and height of your slides to your machine and needs. [link (https://damianavila.github.io/RISE/customize.html#change-the-width-and-height-of-slides)](https://damianavila.github.io/RISE/customize.html#change-the-width-and-height-of-slides)