My solution is written in python. I am not sure if my interpretation of the desired neural network structure was correct since I have little experience with branching DNNs. My understanding of what you meant by the inference server was that you want me to create an API that can be called via a POST request from the internet or local network.

Note that I coded everything on macOS. This will definitely run on Linux but I had no opportunity to try it on Windows.

# The project consists of the following files:

**learner.py**

* contains the code for downloading the MINST dataset and training of the DNN classifier

**client\_example.py**

* contains an example of how to call the inference server using PyCurl

**inference\_server.py**

* contains the code running the server itself

**my\_model.h5**

* my resulting model

**image.raw**

* an example of raw image file compatible with the inference server API

# Usage:

The DNN can be trained by running the command

python learner.py

After the network is trained launch the server by entering the command

python inference\_server.py

The API is accessible via http://127.0.0.1:5000/upload. It takes an uncompressed file containing 784 bytes where every byte represents one pixel of the 28x28 image.

The API returns a json file formatted as {‘0’:XX, ‘1’:XX, ‘2’:XX, ‘3’:XX, ‘4’:XX, ‘5’:XX, ‘6’:XX, ‘7’:XX, ‘8’:XX, ‘9’:XX } where XX are the scores associated with the given class multiplied by 100 and truncated to integer.

Here is an example of how one may use the command curl to access the API:

curl -X POST -H 'enctype:ltipart/form-data ; Content-Type:multipart/form-data' -H "Accept: application/json" -F file=@image.raw -s "http://127.0.0.1:5000/upload"

To see an example of the API being accessed from python please run the command

python client\_example.py

# Training and performance:

The structure of the neural network that I have created is the following:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

==================================================================================================

input\_1 (InputLayer) (None, 1, 28, 28) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_1 (Conv2D) (None, 1, 28, 32) 702496 input\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2D) (None, 1, 14, 16) 0 conv2d\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 1, 14, 64) 200768 max\_pooling2d\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 1, 14, 64) 200768 max\_pooling2d\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2D) (None, 1, 7, 32) 0 conv2d\_2[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_3 (MaxPooling2D) (None, 1, 7, 32) 0 conv2d\_3[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 1, 7, 512) 803328 max\_pooling2d\_2[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_5 (Conv2D) (None, 1, 7, 512) 803328 max\_pooling2d\_3[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_4 (MaxPooling2D) (None, 1, 3, 256) 0 conv2d\_4[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_5 (MaxPooling2D) (None, 1, 3, 256) 0 conv2d\_5[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

concatenate\_1 (Concatenate) (None, 1, 3, 512) 0 max\_pooling2d\_4[0][0]

max\_pooling2d\_5[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 1536) 0 concatenate\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 1000) 1537000 flatten\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 500) 500500 dense\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 10) 5010 dense\_2[0][0]

==================================================================================================

Total params: 4,753,198

Trainable params: 4,753,198

Non-trainable params: 0

The resulting accuracy was 0.977.

Note that since I am not routinely training DNNs on my laptop I was unable to make use of my GPU. The structure of this neural network is also very different from the neural networks I have used before – I would normally use very small 3x3 or 4x4 feature extractors as my first layer.

The resulting classifier compares poorly to other classifiers from similar category. According to <http://yann.lecun.com/exdb/mnist/> . The convolutional neural network LeNet-1 for example is a lot smaller and has a test accuracy of 98.3 .

Note that since the training with 5 epochs was taking over 2 hours on my computer I decided not to experiment and to hand over the results that I got.

Bellow I am attaching the confusion matrix on the test data:

**0 1 2 3 4 5 6 7 8 9**

**0** [[ 978 0 0 0 0 0 0 1 1 0]

**1** [ 2 1120 1 1 2 2 1 5 1 0]

**2** [ 3 4 1012 8 1 0 0 3 1 0]

**3**  [ 0 0 3 1001 0 3 0 1 0 2]

**4** [ 0 0 2 0 971 0 3 2 0 4]

**5** [ 2 0 0 27 0 862 1 0 0 0]

**6** [ 9 2 2 1 7 7 930 0 0 0]

**7** [ 1 2 18 7 0 1 0 998 0 1]

**8** [ 8 0 10 35 4 21 2 3 890 1]

**9** [ 7 1 4 34 34 5 0 49 2 873]]

One can see that the most common mistake is classifying 7 and 9.

If I was to improve the classifier further I would generate new artificial data by applying different affine transforms to the training data.

# Inference server:

I decided to make use of the library called flask. The API itself works via a POST request from a client to the server containing the file with a JSON containing the scores being returned.

The API could be able to handle larger loads since flask is multithreaded.