NVIDIA DIGITS with TensorFlow

Getting Started Guide

DU-09197-001 _v1.0 | September 2021

Table of Contents

Chapter 1. Overview

DIGITS (the Deep Learning GPU Training System) is a webapp for training deep learning models. TensorFlow is the currently supported framework. DIGITS puts the power of deep learning into the hands of engineers and data scientists.

DIGITS is not a framework. DIGITS is a wrapper for TensorFlow; which provides a graphical web interface to those frameworks rather than dealing with them directly on the command-line.

DIGITS can be used to rapidly train highly accurate deep neural network (DNNs) for image classification, segmentation, object detection tasks, and more. DIGITS simplifies common deep learning tasks such as managing data, designing and training neural networks on multi-GPU systems, monitoring performance in real time with advanced visualizations, and selecting the best performing model from the results browser for deployment. DIGITS is completely interactive so that data scientists can focus on designing and training networks rather than programming and debugging.

DIGITS is available through multiple channels such as:

- GitHub download
- NVIDIA's Docker repository, `nvcr.io`

DIGITS also includes the NVIDIA Caffe and TensorFlow deep learning frameworks.

1.1. Contents Of The DIGITS Application

The container image available in the NVIDIA® GPU Cloud™ (NGC) registry and NVIDIA® DGX™ container registry, `nvcr.io`, is pre-built and installed into the `/usr/local/python/` directory.

DIGITS also includes the TensorFlow deep learning framework.

Chapter 2. Enabling support for TensorFlow in DIGITS

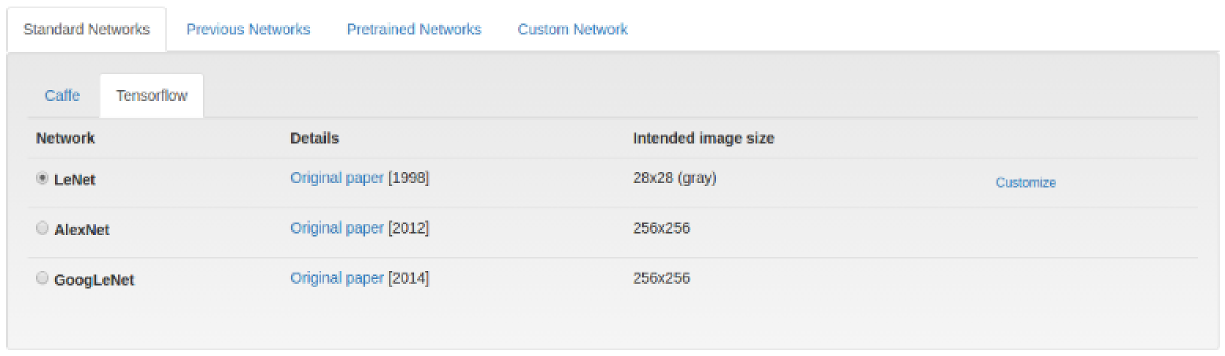
DIGITS will automatically enable support for TensorFlow if it detects that TensorFlow-gpu is installed in the system. This is done by a line of python code that attempts to import tensorflow to see if it actually imports.

If DIGITS cannot enable TensorFlow, a message will be printed in the console saying: TensorFlow support is disabled.

Chapter 3. Selecting TensorFlow when creating a model in DIGITS

About this task

Click the **TensorFlow** tab on the Model Creation page:



Network	Details	Intended image size
● LeNet	Original paper [1998]	28x28 (gray)
○ AlexNet	Original paper [2012]	256x256
○ GoogLeNet	Original paper [2014]	256x256

Note:

By default, Torch7 initializes the weights of linear and convolutional layers according to the method introduced in LeCun, Yann A., et al. "Efficient backprop." Neural networks: Tricks of the trade. Springer Berlin Heidelberg, 2012. 9-48.. Although this weight initialization scheme performs reasonably well under many diverse circumstances, this is rarely optimal and you might notice that Caffe is sometimes able to learn more quickly when using e.g. Xavier initialization. See these examples for more information.

3.1. Defining a TensorFlow model in DIGITS

To define a TensorFlow model in DIGITS , you need to write a python class that follows this basic template:

```
class UserModel(Tower):
    @model_propertyOther TensorFlow Tools in DIGITS
    def inference(self):
        # Your code here
        return model

    @model_property#with tf.variable_scope(digits.GraphKeys.MODEL, reuse=None):
    def loss(self):
        # Your code here
        return loss
```

For example, this is what it looks like for LeNet-5, a model that was created for the classification of hand written digits by Yann Lecun:

```
class UserModel(Tower):

    @model_property
    def inference(self):
        x = tf.reshape(self.x, shape=[-1, self.input_shape[0], self.input_shape[1],
 self.input_shape[2]])
        # scale (divide by MNIST std)
        x = x * 0.0125
        with slim.arg_scope([slim.conv2d, slim.fully_connected],

 weights_initializer=tf.contrib.layers.xavier_initializer(),
                            weights_regularizer=slim.l2_regularizer(0.0005) ):
            model = slim.conv2d(x, 20, [5, 5], padding='VALID', scope='conv1')
            model = slim.max_pool2d(model, [2, 2], padding='VALID', scope='pool1')
            model = slim.conv2d(model, 50, [5, 5], padding='VALID', scope='conv2')
            model = slim.max_pool2d(model, [2, 2], padding='VALID', scope='pool2')
            model = slim.flatten(model)
            model = slim.fully_connected(model, 500, scope='fc1')
            model = slim.dropout(model, 0.5, is_training=self.is_training,
 scope='do1')
            model = slim.fully_connected(model, self.nclasses, activation_fn=None,
 scope='fc2')
            return model

    @model_property
    def loss(self):
        loss = digits.classification_loss(self.inference, self.y)
        accuracy = digits.classification_accuracy(self.inference, self.y)
        self.summaries.append(tf.summary.scalar(accuracy.op.name, accuracy))
        return loss
```

The properties inference and loss must be defined and the class must be called UserModel and it must inherit Tower. This is how DIGITS will interact with the python code.

3.1.1. Provided properties

Properties that are accessible through `self`:

Property name	Type	Description
nclasses	number	Number of classes (for classification datasets). For

Property name	Type	Description
		other type of datasets, this is undefined.
input_shape	Tensor	Shape (1D Tensor) of the first input Tensor. For image data, this is set to height, width, and channels accessible by [0], [1], and [2] respectively.
is_training	boolean	Whether or not this is a training graph.
is_inference	boolean	Whether or not this is a graph is created for inference/ testing.
x	Tensor	The input node, with the shape of [N, H, W, C].
y	Tensor	The label, [N] for scalar labels, [N, H, W, C] otherwise. Defined only if self.is_training is True.
fineTuneHook	function	A function(net) that returns the model to be used for fine-tuning. The untuned model is passed as a function parameter.
disableAutoDataParallelism	boolean	By default models are encapsulated in a nn.DataParallelTable container to enable multi-GPU training when more than 1 GPUs are selected. Setting this flag to true disables this mechanism.

3.1.2. Internal properties

These properties are in the UserModel class written by the user:

Property name	Type	Description
__init()__	None	The constructor for the UserModel class.
inference()	Tensor	Called during training and inference.

Property name	Type	Description
loss()	Tensor	Called during training to determine the loss and variables to train.

3.1.3. Tensors

Networks receive TensorFlow Tensor objects as input in the NxCxHxW format (index in batch x channels x height x width). If a GPU is available, Tensors are provided as Cuda tensors and the model and criterion are moved to GPUs through a call to their cuda() method. In the absence of GPUs, Tensors are provided as Float tensors.

3.2. Other TensorFlow Tools in DIGITS

DIGITS provides a few useful tools to help with your development with TensorFlow.

3.2.1. Provided Helpful Functions

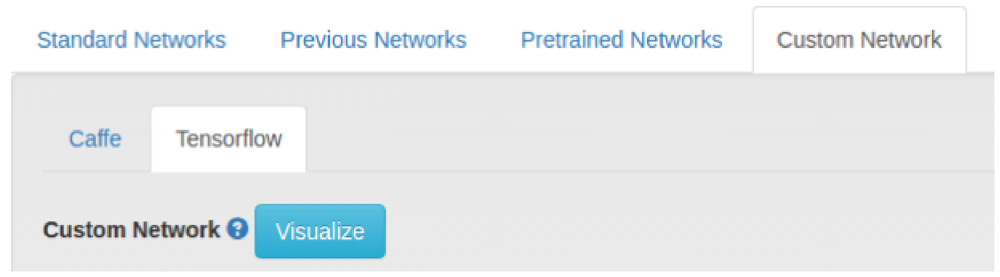
DIGITS provides a few helpful functions to help you with creating the model. Here are the functions we provide inside the digits class:

Function Name	Parameters	Description
classification_loss	pred - the images to be classified y - the labels	Used for classification training to calculate the loss of image classification.
mse_loss	lhs - left hand tensor rhs - right hand tensor	Used for calculating the mean square loss between 2 tensors.
constrastive_loss	lhs - left hand tensor rhs - right hand tensor y - the labels	Calculates the contrastive loss with respect to the Caffe definition.
classification_accuracy	pred - the image to be classified y - the labels	Used to measure the accuracy of the classification task.
nhwc_to_nchw	x - the tensor to transpose	Transpose the tensor that was originally NHWC format to NCHW. The tensor must be a degree of 4.
nchw_to_nhwc	x - the tensor to transpose	Transpose the tensor that was originally NCHW format

Function Name	Parameters	Description
		to NHWC. The tensor must be a degree of 4.
hwc_to_chw	x - the tensor to transpose	Transpose the tensor that was originally HWC format to CHW. The tensor must be a degree of 3.
chw_to_hwc	x - the tensor to transpose	Transpose the tensor that was originally CHW format to HWC. The tensor must be a degree of 3.
bgr_to_rgb	x - the tensor to transform	Transform the tensor that was originally in BGR channels to RGB.
rgb_to_bgr	x - the tensor to transform	Transform the tensor that was originally in RGB channels to BGR.

3.2.2. Visualization With TensorBoard

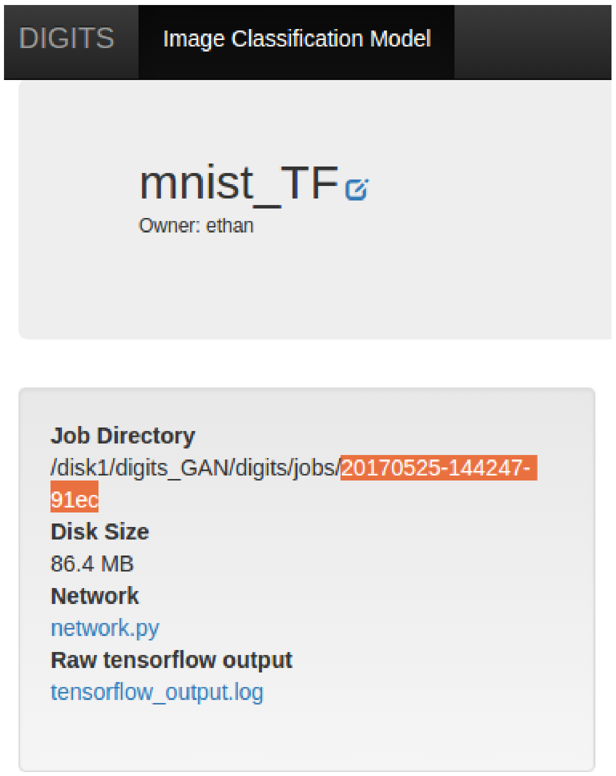
TensorBoard is a visualization tool provided by TensorFlow to see a graph of your neural network. DIGITS provides easy access to TensorBoard network visualization for your network while creating it. This can be accessed by clicking on the Visualize button under Custom Network as seen in the image below.



If there is something wrong with the network model, DIGITS will automatically provide you with the stacktrace and the error message to help you locate the problem. You can also spin up the full TensorBoard server while your model is training using this command

```
$ tensorboard --logdir <job_dir>/tb/
```

where `<job_dir>` is the directory where them model is being trained at, which can be found here:

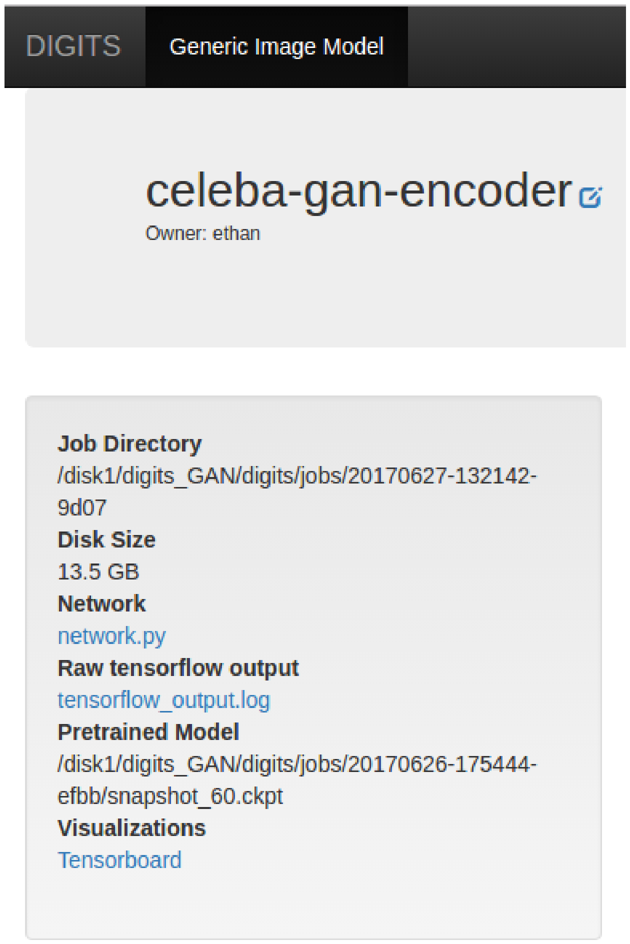


mnist_TF

Owner: ethan

Job Directory
/disk1/digits_GAN/digits/jobs/20170525-144247-91ec
Disk Size
86.4 MB
Network
network.py
Raw tensorflow output
tensorflow_output.log

Afterwards, go to `http://localhost:6006` to open the TensorBoard page or click **TensorBoard** under Visualization.



celeba-gan-encoder

Owner: ethan

Job Directory
/disk1/digits_GAN/digits/jobs/20170627-132142-9d07
Disk Size
13.5 GB
Network
network.py
Raw tensorflow output
tensorflow_output.log
Pretrained Model
/disk1/digits_GAN/digits/jobs/20170626-175444-efbb/snapshot_60.ckpt
Visualizations
Tensorboard

For more information on using TensorBoard see: https://www.tensorflow.org/guide/summaries_and_tensorboard.

3.3. Examples

3.3.1. Simple Auto-Encoder Network

The following network is a simple auto encoder to demostate the structure of how to use TensorFlow in DIGITS. An auto encoder is a 2 part network that basically acts as a compression mechanism. The first part will try to compress an image to a size smaller than original while the second part will try to decompress the compressed representation created by the compression network.

```
class UserModel(Tower):

    @model_property
    def inference(self):

        # the order for input shape is [0] -> H, [1] -> W, [2] -> C
        # this is because tensorflow's default order is NHWC
        model = tf.reshape(self.x, shape=[-1, self.input_shape[0],
self.input_shape[1], self.input_shape[2]])
        image_dim = self.input_shape[0] * self.input_shape[1]

        with slim.arg_scope([slim.fully_connected],
                        weights_initializer=tf.contrib.layers.xavier_initializer(),
                        weights_regularizer=slim.l2_regularizer(0.0005)):

            # first we reshape the images to something
            model = tf.reshape(_x, shape=[-1, image_dim])

            # encode the image
            model = slim.fully_connected(model, 300, scope='fc1')
            model = slim.fully_connected(model, 50, scope='fc2')

            # decode the image
            model = slim.fully_connected(model, 300, scope='fc3')
            model = slim.fully_connected(model, image_dim, activation_fn=None,
scope='fc4')

            # form it back to the original
            model = tf.reshape(model, shape=[-1, self.input_shape[0],
self.input_shape[1], self.input_shape[2]])

            return model

    @model_property
    def loss(self):

        # In an autoencoder, we compare the encoded and then decoded image with the
original
        original = tf.reshape(self.x, shape=[-1, self.input_shape[0],
self.input_shape[1], self.input_shape[2]])

        # self.inference is called to get the processed image
        model = self.inference
        loss = digits.mse_loss(original, model)

        return loss
```

3.3.2. Freezing Variables in Pre-Trained Models by Renaming

The following is a demonstration of how to specify which weights you would like to use for training. This works best if you are using a pre-trained model. This is applicable for fine tuning a model.

When you first train a model, TensorFlow will save the variables with their specified names. When you reload the model to retrain it, tensorflow will simutainously reload all those variables and mark them available to retrain if they are specified in the model definition. When you change the name of the variables in the model, TensorFlow will then know to not train that variable and thus "freezes" it.

```
class UserModel(Tower):

    @model_property
    def inference(self):

        model = construct_model()
        """code to construct the network omitted"""

        # assuming the original model have weight2 and bias2 variables
        # in here, we renamed them by adding the suffix _not_in_use
        # this tells TensorFlow that these variables in the pre-trained model should
        # not be retrained and it should be frozen
        # If we would like to freeze a weight, all we have to do is just rename it
        self.weights = {
            'weight1': tf.get_variable('weight1', [5, 5, self.input_shape[2], 20],
initializer=tf.contrib.layers.xavier_initializer()),
            'weight2': tf.get_variable('weight2_not_in_use', [5, 5, 20, 50],
initializer=tf.contrib.layers.xavier_initializer())
        }

        self.biases = {
            'bias1': tf.get_variable('bias1', [20],
initializer=tf.constant_initializer(0.0)),
            'bias2': tf.get_variable('bias2_not_in_use', [50],
initializer=tf.constant_initializer(0.0))
        }

        return model

    @model_property
    def loss(self):
        loss = calculate_loss()
        """code to calculate loss omitted"""
        return loss
```

Chapter 4. Troubleshooting

For troubleshooting tips see the Nvidia DIGITS Troubleshooting and Support Guide.

4.1. Support

For the latest Release Notes, see the DIGITS Release Notes Documentation website.

For more information about DIGITS, see:

- DIGITS website
- DIGITS 6.0 project
- GitHub documentation

Note: There may be slight variations between the nvidia-docker images and this image.