Solving CNN using TensorFlow

Ying-Ting Wang

Abstract –

CIFAR-10 classification is a common benchmark problem in machine learning. The problem is to classify RGB 32x32 pixel images across 10 categories:

Airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

.lki

Goals –

The goal of this tutorial is to build a relatively small convolutional neural network (CNN) for recognizing images. In the process, this tutorial:

1. Highlights a canonical organization for network architecture, training and evaluation.
2. Provides a template for constructing larger and more sophisticated models.

The reason CIFAR-10 was selected was that it is complex enough to exercise much of TensorFlow’s ability to scale to large models. And the model is small enough to train fast, which is ideal for trying out new ideas and experimenting with new techniques.

Highlights of the Tutorial –

The tutorial demonstrates several important constructs for designing larger and more sophisticated models in TensorFlow:

* Core mathematical components including convolution, rectified linear activations, max pooling and local response normalization.
* Visualization of network activities during training, including input images, losses and distributions of activations and gradients.
* Routines for calculating the moving average of learned parameters and using these averages during evaluation to boost predictive performance.
* Implementation of a learning rate schedule that systematically decrements over time.
* Prefetching queues for input data to isolate the model from disk latency and expensive image pre-processing.

We also provide a multi-GPU version of the model which demonstrates:

* Configuring a model to train across multiple GPU cards in parallel.
* Sharing and updating variables among multiple GPUs.

Model Architecture –

The model in this CIFAR-10 tutorial is a multi-layer architecture consisting of alternating convolutions and nonlinearities. These layers are followed by fully connected layers leading into a softmax classifier. The model follows the architecture described by Alex Krizhevsky, with a few differences in the top few layers.

This model achieves a peak performance of about 86% accuracy within a few hours of training time on a GPU. It consists of 1,068,298 learnable parameters and requires about 19.5M multiply-add operations to compute inference on a single image.

Code Organization –

The code for this tutorial resides in <https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/>

|  |  |
| --- | --- |
| File | Purpose |
| Cifar10\_input.py | Reads the native CIFAR-10 binary file format. |
| Cifar10.py | Builds the CIFAR-10 model. |
| Cifar10\_train.py | Trains a CIFAR-10 model on a CPU or GPU. |
| Cifar10\_multi\_gpu\_train.py | Trains a CIFAR-10 model on multiple GPUs. |
| Cifar10\_eval.py | Evaluates the predictive performance of a CIFAR-10 model. |

Fig 1. File structure in this tutorial

CIFAR-10 model –

The CIFAR-10 network is largely contained in cifar10.py. The complete training graph contains roughly 765 operations. The code can be most reusable by constructing the graph with the following modules:

1. Model inputs: inputs() and distorted\_inputs() add operations that read and preprocess CIFAR images for evaluation and training, respectively.
2. Model prediction: inference() adds operations that perform inference, i.e. classification, on supplied images.
3. Model training: loss() and train() add operations that compute the loss, gradients, variable updates and visualization summaries.
4. Model Inputs:

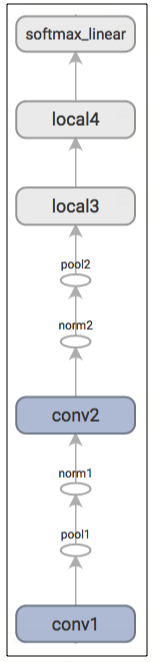
The input part of the model is built by the functions inputs() and distorted\_inputs() which read images from the CIFAR-10 binary data files. These files contain fixed byte length records, so we use tf.FixedLengthRecordReader.

The images are processed as follows:

* They are cropped to 24 x 24 pixels, centrally for evaluation or randomly for training.
* They are approximately whitened to make the model insensitive to dynamic range.

For training, we additionally apply a series of random distortions to artificially increase the data set size:

* Random flip the image from left to right
* Randomly distort the image brightness.
* Randomly distort the image contrast.

1. Model Prediction

The prediction part of the model is constructed by the inference() function which adds operations to compute the *logits* of the predictions. That part of the model is organized as follows:

|  |  |
| --- | --- |
| Layer Name | Description |
| Conv1 | Convolution and rectified linear activation. |
| Pool1 | Max pooling. |
| Norm1 | Local response normalization. |
| Conv2 | Convolution and rectified linear activation. |
| Norm2 | Local response normalization. |
| Pool2 | Max pooling. |
| Local3 | Fully connected layer with rectified linear activation. |
| Local4 | Fully connected layer with rectified linear activation. |
| Softmax\_linear | Linear transformation to produce logits. |

Fig. 2. Left table is inference operation steps; Right graph is generated from TensorBoard describing the inference operation

**Notice: The output of inference are un-normalized logits. Editing the network architecture by using**

**tf.nn.softmax can return normalized predictions**

The inputs() and inference() functions provide all the components necessary to perform evaluation on a model.

**Notice: The model architecture in inference() differs slightly from the CIFAR-10 model specified in cuda-convnet. In particular, the top layers of Alex’s original model are locally connected and not fully connected. Editing the architecture can exactly reproduce the locally connected architecture in the top layer.**

1. Model Training

The usual method for training a network to perform N-way classification is multinomial logistic regression, aka. *Softmax regression*. Softmax regression applies a softmax nonlinearity to the output of the network and calculates the cross-entropy between the normalized predictions and a 1-hot encoding of the label. For regularization, we also apply the usual weight decay losses to all learned variables. The objective function for the model is the sum of the cross entropy loss and all these weight decay terms, as returned by the loss() function.

Visualize it in TensorBoard with a tf.summary.scalar:

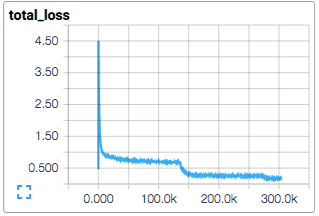


Fig 3. Sum of the cross entropy loss.

Train the model using standard gradient descent algorithm with a learning rate that exponentially decays over time.

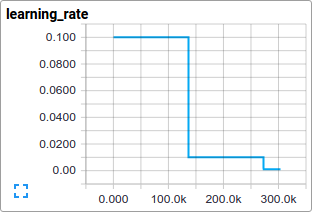


Fig 4. Learning rate that exponentially decays over time.

The train() function adds the operations needed to minimize the objective by calculating the gradient and updating the learned variables. It returns an operation that executes all the calculations needed to train and update the model for one batch of images.

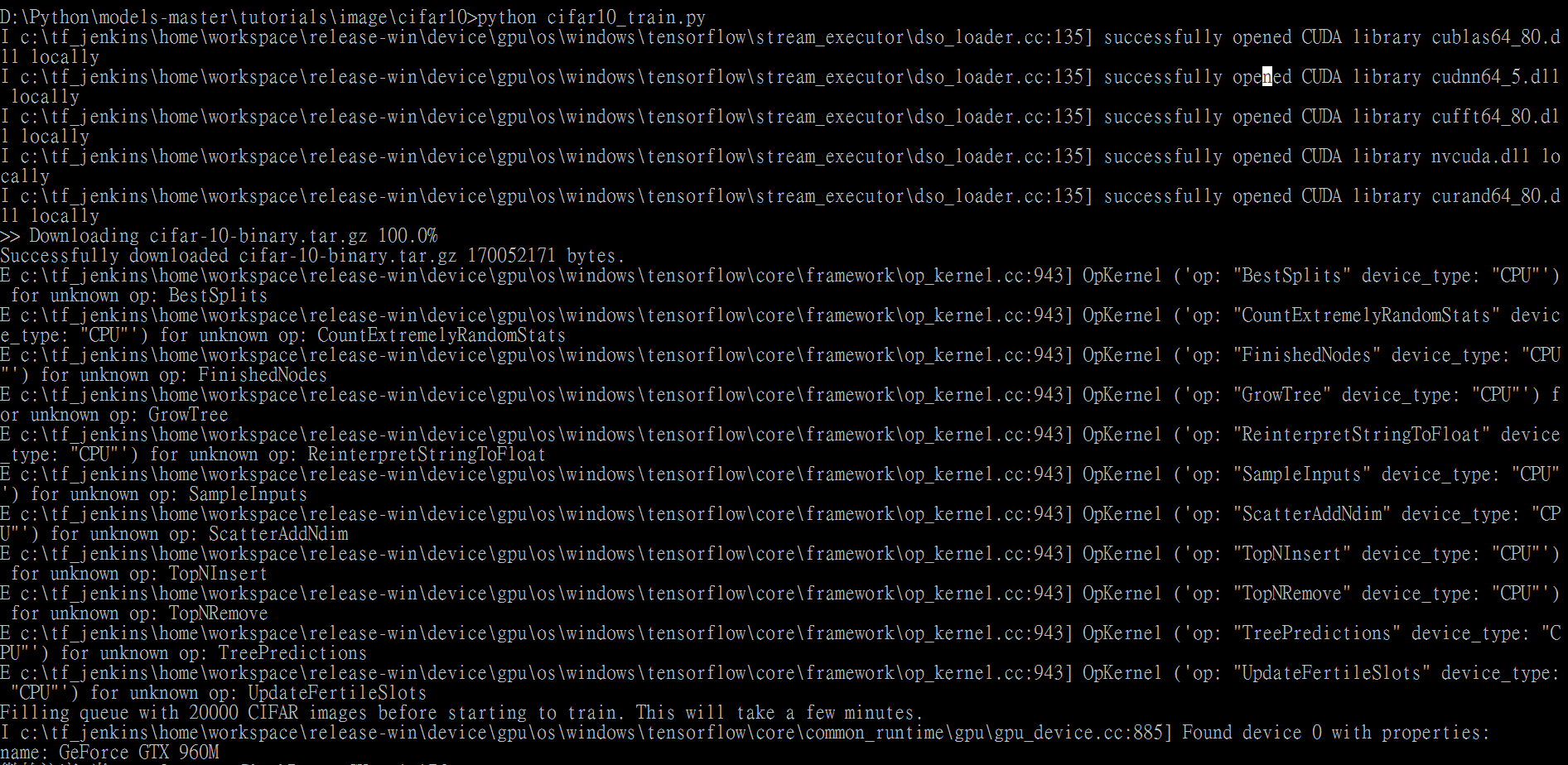
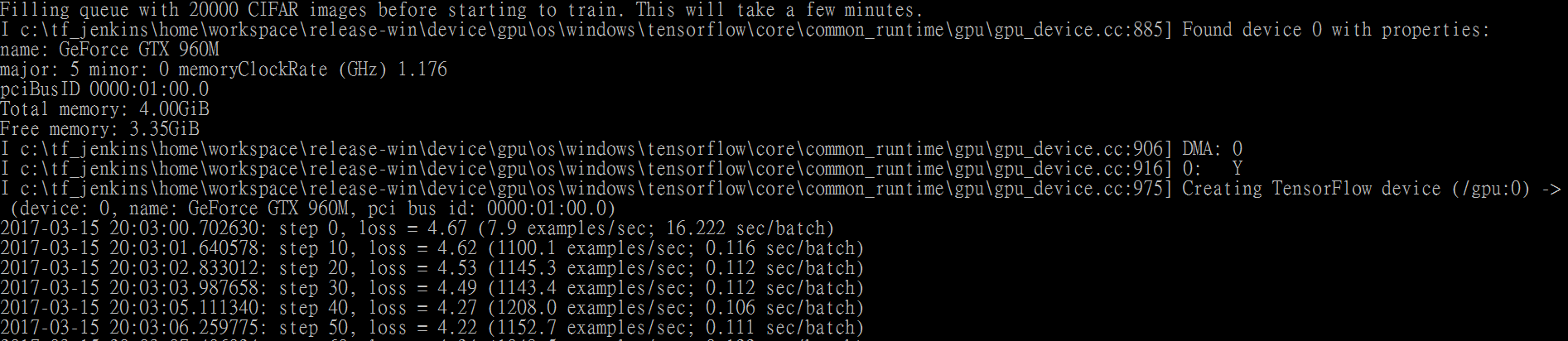
Launching and Training the Model –

Launch it and run the training operation with the script cifar10\_train.py

*>>> Python cifar10\_train.py*

**NOTE: The first time you run any target in the CIFAR-10 tutorial, the CIFAR-10 dataset is automatically downloaded. The data set is about 160MB.**

We should see the output:



Python command to execute cifar10\_train.py file

If install tensorflow GPU version,

it will open CUDA library

Start training process

Automatically download CIFAR-10 image set

Fig 5. The executing process after excute cifar10\_train.py

Fig 5. Shows that after executing cifar10\_train.py, the program will open relative libraries, show your machine configuration, and start training process.

Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.  
2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/batch)  
2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/batch)  
2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/batch)  
2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/batch)  
2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/batch)  
2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)

…

Fig 6. (Reference from tensorflow official website)The output after we execute cifar10\_train.py

The script reports the total loss every 10 steps as well as the speed at which the last batch of data was processed. A few comments:

* The first batch of data can be very slow (e.g. several minutes) as the preprocessing threads fill up the shuffling queue with 20,000 processed CIFAR images.
* The reported loss is the average loss of the most recent batch. Remember that this loss is the sum of the cross entropy and all weight decay terms.
* Keep an eye on the processing speed of a batch. The numbers shown above were obtained on a Tesla K40c. If running on a CPU, expect slower performance.

**NOTE: If the first training step take so long, try decreasing the number of images that initially fill up the queue. Search for *min\_fraction\_of\_exapmles\_in\_queue* in *cifar10\_input.py***

Cifar10\_train.py periodically saves all model parameters in checkpoint files but it does not evaluate the model. The checkpoint file will be used by cifar10\_eval.py to measure the predictive performance.

The terminal text returned from cifar10\_train.py provides minimal insight into how the model is training. We want more insight into the model during training:

* Is the loss really decreasing or is that just noise?
* Is the model being provided appropriate images?
* Are the gradient, activations and weight reasonable?
* What is the learning rate currently at?

TensorBoard provides this functionality, displaying data exported periodically from cifar10\_train.py via a tf.summary.FileWriter.

For instance, we can watch how the distribution of activations and degree of sparsity in local3 features evolve during training:

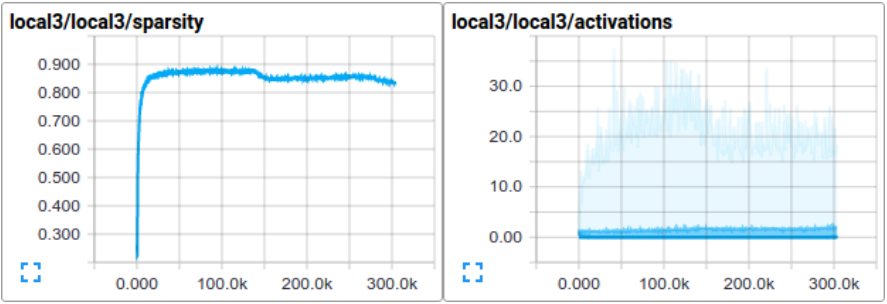


Fig 6. Degree of sparsity & distribution of activations.

Individual loss functions, as well as the total loss, are particularly interesting to track over time. However, the loss exhibits a considerable amount of noise due to the small batch size employed by training. In practice we find it extremely useful to visualize their moving averages in addition to their raw values. The script use “tf.train.ExponentialMovingAverage” for this purpose.

Evaluating a Model –

The model is evaluated by the script cifar10\_eval.py. It constructs the model with the inference() function and uses all 10,000 images in the evaluation set of CIFAR-10. It calculates the precision @1: how often the top prediction matches the true label of the image.

To monitor how the model improves during training, the evaluation script runs periodically on the lastest checkpoint files created by the cifar10\_train.py.

>>> python cifar10\_eval.py

Be careful not to run the evaluation and training binary on the same GPU or else you might run out of memory. Consider running the evaluation on a separate GPU if available or suspending the training binary while running the evaluation on the same GPU.

We should see the output:

2015-11-06 08:30:44.391206: precision @ 1 = 0.860  
...

The script merely returns the precision @1 periodically – in this case it returned 86% accuracy. Cifar10\_eval.py also exports summaries that may be visualized in TensorBoard. These summaries provide additional insight into the model during evaluation.

The training script calculated the moving average version of all learned variables.

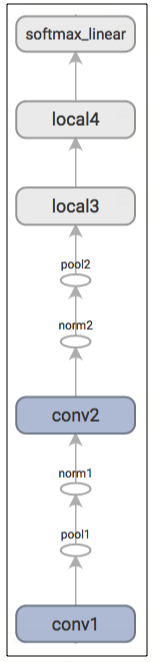
The evaluation script substitutes all learned model parameters with the moving average version. This substitution boosts model performance at evaluation time.

The flowchart of CNN using Tensorflow –

Model inputs

CIFAR

images



Model predictioin

Inference( )

Distorted\_inputs( )

1. 32x32 -> 24x24
2. whitened

Model train

Train( )

1. Calculatin gradient
2. Update learn variables

Loss( )

Inputs( )

Read

images

Preprocess

Compute

logits

Softmax

regression

Minimize

objective