Caffe to TensorFlow Conversion and Benchmarking Models

Chao Hsiang Chung
University of Illinois, Urbana-Champaign
chchung2@illinois.edu

Abstract
The CarML (Cognitive ARtifacts for Machine Learning) is a platform allowing people to easily deploy and experiment ML (Machine Learning)/DL (Deep Learning) frameworks and models. It allows ML/DL software developers to deploy their software packages, ML model developers to publish and evaluate their models, users to experiment with different models and frameworks, all through a web user interface or a REST api, and system architects to capture system resource usage to inform future system and hardware configuration. In order to compare the accuracy that a model predicts among frameworks, instead of using a existing model from different framework, it is better to use a model in one framework to convert to other frameworks. There are many existing converters for framework conversion; however, none of them is able to convert all kinds of models. Therefore, I researched on how to solve the issues happened when converting from Caffe to TensorFlow, and composed a script to automatically collect and analyze results by inquiring CarML by images.

Introduction
With the rise of popularity of ML/DL, there are numerous converters available online. I adopted two converters MMdnn by Microsoft, and caffe-tensorflow by ethereo. Unfortunately, Caffe is far different from TensorFlow. For instance, in the graph of Caffe, each node represents a layer. In TensorFlow, however, each node is a tensor operation (e.g. matrix add/multiply, convolution, etc.). Since the similarity between Caffe and TensorFlow is not as high as Caffe and Caffe2, it needs an indirect state to be converted successfully. The conversion procedure is as Figure1:

Besides, although there are many resources (converters) available, most of these them are not perfect. They are either rife with bugs or not supporting Caffe models. The details will be discussed later.

Conversion
Common Layers in Caffe Graphs
The following layers are used mostly in every models I converted:
1. Convolution: By weighted moving average, find out edges, corners, or other features.
2. ReLU (Rectified Linear Units): Introduce non-linearity to a system that basically has just been computing linear operations during the conv layers, it is faster than Tanh or Sigmoid functions. Also alleviate the vanishing gradient problem.
3. Pooling: Maxpooling is the most popular. It takes a filter (normally of size 2x2) and a stride of the same length to down-sample by outputs the maximum number in every subregion that the filter convolves around.
4. Dropout: This layer drops out a random set of activations in that layer by setting them to zero, in order to help alleviate the over-fitting problem.

Figure 1: Conversion from Caffe to TensorFlow with an indirect state Numpy

Figure 2: Mechanism of Dropout
Structure, Features of Models

AlexNet  AlexNet is one of the first models which proved the effectiveness of application of CNN (Convolutional Neural Network) on complex model (with error rate 15.3% in ImageNet LSVRC-2012). It consists of 5 convolutional layers followed by 3 fully connected layers as Figure 3. It adopts ReLU, Dropout, LRN 1, and data augmentation such as image translations, horizontal reflections, and patch extractions, which were innovative then, so that it could increase the accuracy significantly compared to the past.

Figure 3: AlexNet Structure.

Inception  From Figure 4 we can observe the structures of Inception. The bottom green box is input and the top one is output. The naive version of Inception allows to perform all of the convolution or the pooling operations in parallel. Nonetheless, this would lead to too many outputs. The solution is to add 1x1 convolution operations before the 3x3 and 5x5 layers since 1x1 convolutions 2 provide a way of a method of dimensionality reduction.

Figure 4: Inception Structures.

The evolution of Inception:

- V1 to V2: A variant of Inception-v2 which adds BN-auxiliary, which is the version in which the fully connected layer of the auxiliary classifier is also-normalized, not just convolutions. Besides, it tears down 3x3 convolution in to 3x1 and 1x3 to reduce parameters. (As Figure 6)

- V3 to V4: A streamlined version of v3 with a more uniform architecture and better recognition performance.

Figure 5: By using smaller kernel, the calculation will be more efficient.

Figure 6: By using 1x3 and 3x1, it can reduce parameter usage.

GoogLeNet  The appearance of GoogLeNet proved that with more convolutions, deeper layer, it is more likely to obtain a better result (accuracy). It won ILSVRC 2014 with a top 5 error rate of 6.7%, and it was one of the first CNN architectures that really strayed from the general approach of simply stacking convolution and pooling layers on top of each other in a sequential structure. GoogLeNet has 22 layers, which means almost 12x less parameters, so it is faster and less than AlexNet and much more accurate.

When observing the structure of GoogLeNet (as Figure 7), it is obvious that not everything is happening sequentially. Some of pieces of this network which are happening in parallel, which is Inception module actually. GoogLeNet has 9 Inception modules in the entire structure, and with over 100 layers in total. In addition, it did not use fully connected layers, which could save a huge number of parameters. It is important that it combined with Inception module so

1LRN layers are used in Alexnet to normalize pixels across channels in the first convolutional layers, in order to to ensure that very bright or dark pixels do not dominate their neighbors.

21x1 convolutions (Network in Network layer) span a certain depth, so we can think of it as a 1 x 1 x N convolution where N is the number of filters applied in the layer. Effectively, this layer is performing a N-D element-wise multiplication where N is the depth of the input volume into the layer.
that it can have amazing performance and computationally efficiency.

**VGG (Visual Geometry Group)**  The characteristics of VGG are simplicity and depth. Though it did not win the championship of ILSVRC 2014 (it was GoogLeNet), it still performed marvelously with only 7.3% error rate. By utilizing small filters (3x3), it effectively reduces the need of the amount of parameters, and thereby it is able to use more ReLU layers and go deeper. The structure is as Figure 8. The usage of smaller filters influenced some renowned models such as ResNet and GoogLeNet. However, it is time-consuming to train VGG and its weights are large (VGG16 is over 533MB and VGG19 is over 574MB).

**ResNet**  ResNet is known for its extremely huge number of layers (can be as large as 269), which helped it won ILSVRC 2015 with an incredible error rate of 3.6%. The reason that it can achieve numerous layers is that it utilize residual block. The idea of residual block is that when the input x was originally going to go through conv-relu-conv series (as Figure 9), instead of going through them, it just bypasses them and go to the output, then add the F(x) to the output to achieve similar result of going through the series. By doing so, a tremendous number of layers is achievable. Also, its fantastic performance inspired people to focus on residual learning.

The evolution of ResNet is from small number of layers to huge number of layers, so does the performance (accuracy rate). However, it should be noted that ResNet team did try a 1202 layers network, but ended up having lower performance owing to over-fitting.

**SqueezeNet**  Unlike other models which pursue higher accuracy, SqueezeNet was invented to reduce the complexity of network. Although deep residual learning models (e.g., ResNet) have implemented small kernel (3x3), SqueezeNet substituted some of 3x3 kernel to 1x1. This can effectively reduce 9 times smaller parameters. The reason why SqueezeNet did not replace them all is not to cause negative effect on accuracy.

SqueezeNet also reduce the number of inputs for the remaining 3x3 filters. This strategy reduces the number of parameters by basically just using fewer filters. The mechanism behind this is by feeding *squeeze* layers into what they term *expand* layers (as Figure 10). By reducing the number of filters in the squeeze layer feeding into the expand layer, they are reducing the number of...
connections entering these 3x3 filters thus reducing the total number of parameters. (This is also called FireModule)

Finally, it down-samples late in the network so that convolution layers have large activation maps. The principle is to decrease the stride with later convolution layers and thus creating a larger activation/feature map later in the network, classification accuracy actually increases.

SqueezeNet takes advantages of fire module and chains many of these modules together to arrive at a smaller model. *Figure 11* shows some variants of this chaining process.

**Figure 11**: Left: SqueezeNet. Middle: SqueezeNet with simple bypass. Right: SqueezeNet with complex bypass.

Note that SqueezeNet architecture was able to achieve a 50X reduction in model size compared to AlexNet while meeting or exceeding the top-1 and top-5 accuracy of AlexNet. Also, its size is 510x smaller than AlexNet. These results are really encouraging because SqueezeNet shows the potential for combining different approaches for compression.

**NIN (Network-in-network)** NIN is proposed to enhance model discriminability for local patches within the receptive field. Rather than using the conventional convolutional layer, whose linear filters followed by a nonlinear activation function to scan the input, NIN builds micro neural networks with more complex structures to abstract the data within the receptive field. (As *Figure 12*).

*Figure 12*: The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (multilayer perceptron)

The feature maps are obtained by sliding the a multilayer perceptron (MLP) over the input in a similar manner as CNN and are then fed into the next layer. Therefore, the overall structure of the NIN is the stacking of multiple mlpconv layers. (As *Figure 13*).

Note that the idea of using MLP is adopted by ResNet and Inception.

**Figure 13**: NIN Structure

**Xception** The Xception model is an extension of Inception. It takes advantage of depth-wise separable convolutions and the residual block from ResNet. By using depth-wise separable convolutions and the residual block, Xception can reduce a huge amount of computational resources, and thus having around 91 MB for its weights. The structure of Xception can refer to *Figure 14*.

**WRN (Wide Residual Networks)** To WRN, residual blocks of ResNet tends to provide little amount of information since ResNet is too deep; namely, WRN considers that the success of ResNet was credited to residual block instead of deeper layers. Also, WRN thinks that each fraction of a percent of improved accuracy costs nearly doubling the number of layers, and so training very deep residual...
networks has a problem of diminishing feature reuse, which makes these networks very slow to train.

As a result, WRN decreases depth and increases width of residual networks. To achieve it, WRN utilizes the following three way: More convolution layers; More feature planes; Increment filter size of convolution layers. Besides, with the rise of width, parameters will also be increased by the power of two. To reduce the number of parameters, WRN adopts dropout.

The performance and the number of parameters of WRN40-4 is similar to ResNet1001; however, WRN’s training time is less 8 times than ResNet’s.

**DPN (Dual Path Networks)** ResNet focuses on the reuse of the characteristics, but giving up exploring new characteristics. The DenseNet is the opposite. DPN takes advantages of ResNet and DenseNet. As Figure 16 (e), DPN uses ResNet as main body, aided by DenseNet.

Note that DPN won the championship in Object Localization Task in ILSVRC 2017, with all competition tasks within Top 3.
FCN (Fully Convolutional Networks) FCN adapts contemporary classification networks (AlexNet, VGG, and GoogLeNet) into fully convolutional networks and transfer their learned representations by fine-tuning to the segmentation task. Then, it defines a novel architecture that combines semantic information from a deep, coarse layer with appearance information from a shallow, fine layer to produce accurate and detailed segmentations. In FCN, the features are merged from different stages in the encoder which vary in coarseness of semantic information. Also, the up-sampling of learned low resolution semantic feature maps is done using de-convolutions which are initialized with bilinear interpolation filters. For traditional up-sampling, due to the fixed stride, it does not take care of margin, and also limits the scale of detail of the up-sampling output. FCN use skip layer, which will decrease the size of stride around margin area, and incorporate the fine layer and the coarse layer, then do the up-sampling. This method takes care of local and global information. Hence it enhances the performance.

Conversion Results
In 31 models, I successfully converted 27 Caffe models into TensorFlow. The 4 failed ones are DPN68, DPN92, voc-fcn16s, and voc-fcn32s. During the conversion, there were many bugs found and fixed. The detail about bug fixing will be elaborated in next section.

From Caffe to Numpy
1. The first issue I encountered was Unable to determine kernel parameter. I tried to set lots of break points to find out the cause. Finally, I found that this happened because the converter did not know the input shape when trying to get the stride kernel output shape. The solution is to set the input shape in the...
file layers.py and shapes.py before using layers to get the result.

2. Another issue was Unknown layer type encountered. I tried to traced the cause but everything looked well. Then I tried to find resource from the internet, and I noticed that this converter needs a great amount of memory space. I expanded the memory size (from 2GB to 4GB) that the converter can use and solved this issue.

3. Some of the models were out of date so that the converter could not use those models as inputs. The solution is simply to update Caffe models by using the tool in the Caffe Library.

4. One more issue was that parsing slice layer is not implemented in the converter. This one led to DPN and fcn models conversion. I have posted an issue on GitHub and the author replied that he will get it fixed soon.

**From Numpy to TensorFlow**

1. There was one error message saying that TypeError: Input split_dim of Split Op has type float32 that does not match expected type of int32. I tried to change the parameter type before passing to the function; however, this caused other problems that the function wants the original type. Then I set break points trying to figure out what happened. I found out this was nothing to do with type error; instead, when the converter trying to call a function split in python library, it passed the parameters with the wrong order. It happened since the documentation of that function was wrong:

   On the documentation, it shows that the order of the parameter of the function is `tf.split(split_dim, num_splits, value, name = 'split')`. But actually it is defined as `split(value, num_or_size_splits, axis = 0, num = None, name = "split")`.

   Since the converter used the function based on the documentation, I changed the order and it worked.

2. Another issue was assertion error. In Caffe, it accepts more types of padding, but TensorFlow only accepts either SAME or VALID. Therefore, when I converted models from Caffe, some of them may have padding type None (i.e., without any padding). After looking for information about SAME and VALID, I realized that VALID itself is without padding. So I change None to VALID and resolved the issue.

3. The last one was caused by the dimension of the layers of models. Some of them exceeds the dimension defined in functions of the converter so that there will be negative dimension when doing convolution. This issue happens in the converter caffe-tensorflow by ethereo. The solution is to observe what is the maximum number of dimension needed by the input, then modified it in the functions of the converter. However, since the converter MMdnn by Microsoft has dealt with this situation, we can use it instead, and the problem resolved.

**Conclusion** Before fixing the bugs, I could only converted 18 models from Caffe to Numpy, and only 13 of them could be converted from Numpy to TensorFlow. Nevertheless, after bugs fixed, I am certainly sure that every Numpy can be converted to TensorFlow, and after MMdnn updates, every caffe should be able to be converted from Caffe to Numpy.

**Benchmark**

**Data Collection Implementation**

My script for inference is adapted from Abdul Dakkak’s script. In his script, when a website of an URL is corrupted, the entire results will be gone. Thus, I tried two methods to improve Abdul Dakkak’s script for inference:

1. First, I tried to examine the HTTP response code to check if the website is still accessible. By this method I can eliminate many corrupted website. However, I found out even some of the responses are 404, the images on those website are still available. As a result, after checking the HTTP response code, I also try to load the image to validate if the image is available. This will slow down the speed of inference, but this helps a lot since there are many resources showing 404 while the image exists.

   Unfortunately, there are some URLs with HTTP response code 200 (i.e., the request has succeeded.) will also lead to the script crashed and lose the result. What’s worse, when trying to check those URLs by inputting them in browser, they just show up normally. Therefore, if we want to use this method, we have to manually take out these URLs.

2. Since the first method was not ideal, I try to utilize another strategy: do inference one URL by one URL. Instead of checking the validation of an URL, I just input it and try to analyze the result on CarML. If this URL is corrupted, pass it and try the next one; if it works, then write the result into output file.

   This method effectively eliminate any possibility of program crash, as well as the lose of results. What’s more, without trying to check validation of images by loading the image, this method is far faster than method...
1. Therefore, I adopt the second method to do inference and result collection.

**Detailed Methodology Used for Benchmarking**

**Data** To analyze the performance of models, first thing is to have sufficient data to input. I used URLs from *ImageNet*. Since sometimes an output from model may have names in the format of scientific name or some unusual name, it is not easy to analyze the statistical result. Thus, I chose images of Bear for the most of the outputs of bear include 'Bear'.

**Result Collection** By using the script, For each output of an input, I will write three results with Top 3 probability to the output file. For example, when there is an output as following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cheeseburger</td>
<td>0.9993</td>
</tr>
<tr>
<td>Meat loaf</td>
<td>0.0003</td>
</tr>
<tr>
<td>Ice cream</td>
<td>0.0002</td>
</tr>
<tr>
<td>Hotdog</td>
<td>0.0001</td>
</tr>
<tr>
<td>Guacamole</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

I will only output Cheeseburger, Meat loaf, and Ice cream with their probabilities individually.

**Analysis** After having sufficient data of results (i.e., around 6000.), I can start analyze. First, I check the name each data of results whether has Bear'.

- If it does not match, then add 0 to the variable sumProbability.
- If it matches, choose the highest probability among three results, and also add it into the variable sumProbability. Meanwhile, if the result with the highest probability matches, increment sumAccuracy by 1.

After checking 6000 results, we will derive the overall average of the probability and the overall accuracy by the following two equations:

\[
\text{OverallAccuracy} = \frac{\text{sumAccuracy}}{6000}
\]

\[
\text{OverallProbability} = \frac{\text{sumProbability}}{6000}
\]

By having the overall average of the performance (accuracy) and probability for each model, we will be able to see the difference between models (i.e., which one has higher performance, etc.). The next step is to analyze the same model perform on different frameworks (which framework has higher performance, etc.).

<table>
<thead>
<tr>
<th>Frameworks</th>
<th>Caffe</th>
<th>TensorFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models</td>
<td>Accuracy</td>
<td>Probability</td>
</tr>
<tr>
<td>BVLC-AlexNet</td>
<td>0.820</td>
<td>0.677</td>
</tr>
<tr>
<td>BVLC-GoogLeNet</td>
<td>0.877</td>
<td>0.783</td>
</tr>
<tr>
<td>BVLC-Reference-CaffeNet</td>
<td>0.817</td>
<td>0.681</td>
</tr>
<tr>
<td>DPN68</td>
<td>0.945</td>
<td>0.818</td>
</tr>
<tr>
<td>DPN92</td>
<td>0.952</td>
<td>0.844</td>
</tr>
<tr>
<td>Inception-ResNet_v2</td>
<td>0.949</td>
<td>0.926</td>
</tr>
<tr>
<td>Inception-v3</td>
<td>0.943</td>
<td>0.809</td>
</tr>
<tr>
<td>Inception-v4</td>
<td>0.950</td>
<td>0.822</td>
</tr>
<tr>
<td>ResNet101</td>
<td>0.872</td>
<td>0.827</td>
</tr>
<tr>
<td>ResNet101-v2</td>
<td>0.934</td>
<td>0.881</td>
</tr>
<tr>
<td>ResNet152</td>
<td>0.877</td>
<td>0.832</td>
</tr>
</tbody>
</table>

Table 1: Results of Each Model in Different Frameworks. Caffe has 6000s inputs. TensorFlow has 1000s inputs.

**Result**

**Result Analysis** From Table 1 and Figure 19, it can be observed that

1. Most of the results are similar to the expected ones. However, some of ResNets have incredible accuracy (i.e., 95%). I believe this is caused by over-fitting since ResNet typically has a huge number of layers, but it can only pass a small amount of information to the next layer. If using more kinds of input (i.e., using various kinds of images) and calculate the average accuracy, it will reduce the problem with such a high accuracy due to over-fitting.

2. With the improvement of the model (e.g., Inception-v3 to Inception-v4, etc.), the overall accuracy is enhanced.
3. Since AlexNet is the earliest model here and almost every other model is based on AlexNet, its accuracy is relatively low than others.

4. In this result table, we cannot analyze the difference between frameworks since Tensorflow only has \( \frac{1}{6} \) inputs compared to Caffe. Nonetheless, it is still somewhat observable that the overall accuracy on TensorFlow is close to one on Caffe.

**Difference between Caffe and TensorFlow**

- **TensorFlow**
  - Pros:
    * Python + Numpy
    * Computational graph abstraction
    * TensorFlow for visualization
    * Data and model parallelism
  - Cons:
    * Slower than other frameworks
    * Not many pretrained models
    * Computational graph is pure Python, therefore slow
    * Drops out to Python to load each new training batch
    * Not very toolable

- **Caffe**
  - Pros:
    * Good for feedforward networks and image processing
    * Train models without writing any code
  - Cons:
    * Need to write C++ / CUDA for new GPU layers
    * Not good for recurrent networks
    * Cumbersome for big networks (GoogLeNet, ResNet)
    * Not extensible, bit of a hairball

**Explanation of Insufficient Data**

The reason that there is only few result is that

1. I used a huge number of inputs for each model. When I tried to run 6000s inputs, it took around 2 hours.

2. The CarML website is not stable. It is easy to crash when receiving a certain number of inferences.

I was told that 1000 inputs is enough for analyzing during the last meeting with **AbdulDakkak**. This is also why there are only 1000 inputs for TensorFlow.

I am confident that my script is perfect, so if I can have enough time (including restarting CarML server), I can get all of results from models on frameworks.

**Conclusion/Prospect**

Basically, almost every model can be converted from Caffe to TensorFlow, and results of each model can be derived given time.

By using models from one specific framework to convert to other frameworks, it can used to analyze the difference between frameworks since these model is from the same resource instead of re-training on different frameworks. Also, with more and more models available on frameworks on CarML, CarML can be a public platform for analysis of Machine Learning/Deep Learning models!

**References**

Francois Chollet. *Deep Learning with Depthwise Separable Convolutions*.

Min Lin, Qiang Chen, Shuicheng Yan. *Network In Networks*.

Sergey Zagoruyko, Nikos Komodakis. *Wide Residual Networks*.

Sergey Zagoruyko, Nikos Komodakis. *Wide Residual Networks*.

Yunpeng Chen, Jianan Li, Huaxin Xiao, Xiaojie Jin, Shuicheng Yan, Jiashi Feng. *Dual Path Networks*.


Karen Simonyan, Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*.

Microsoft. [https://github.com/Microsoft/MMdnn](https://github.com/Microsoft/MMdnn)

ethereon. [https://github.com/ethereon/caffe-tensorflow](https://github.com/ethereon/caffe-tensorflow)