Approach pre-trained deep learning models with caution

Pre-trained models are easy to use, but are you glossing over details that could impact your model performance?

How many times have you run the following snippets:

```python
import torchvision.models as models
inception = models.inception_v3(pretrained=True)
```

or

```python
from keras.applications.inception_v3 import InceptionV3
base_model = InceptionV3(weights='imagenet',
include_top=False)
```

It seems like using these pre-trained models have become a new standard for industry best practices. After all, why wouldn’t you take advantage of a model that’s been trained on more data and compute than you could ever muster by yourself?

See the discussion on Reddit and HackerNews

Long live pre-trained models!

There are several substantial benefits to leveraging pre-trained models:

- super simple to incorporate
• achieve solid (same or even better) model performance quickly

• there’s not as much labeled data required

• versatile uses cases from transfer learning, prediction, and feature extraction

Advances within the NLP space have also encouraged the use of pretrained language models like GPT and GPT-2, AllenNLP’s ELMo, Google’s BERT, and Sebastian Ruder and Jeremy Howard’s ULMFit (for an excellent overview of these models, see this TOPBOTs post).

One common technique for leveraging pretrained models is feature extraction, where you’re retrieving intermediate representations produced by the pretrained model and using those representations as inputs for a new model. These final fully-connected layers are generally assumed to capture information that is relevant for solving a new task.

**Everyone’s in on the game**

Every major framework like Tensorflow, Keras, PyTorch, MXNet, etc… offers pre-trained models like Inception V3, ResNet, AlexNet with weights:

• Keras Applications

• PyTorch torchvision.models

• Tensorflow Official Models (and now TensorFlow Hubs)

• MXNet Model Zoo

• Fast.ai Applications
But are these benchmarks reproducible?

The article that inspired this post came from [Curtis Northcutt](https://www.cs.ucsb.edu/~northcutt/), a computer science PhD candidate at MIT. His article ‘Towards Reproducibility: Benchmarking Keras and PyTorch’ made several interesting claims —

1. **resnet** architectures perform better in PyTorch and **inception** architectures perform better in Keras

2. The published benchmarks on Keras Applications cannot be reproduced, even when exactly copying the example code. In fact, their reported accuracies (as of Feb. 2019) are usually higher than the actual accuracies (citing 1 and 2)

3. Some pre-trained Keras models yield inconsistent or lower accuracies when deployed on a server (3) or run in sequence with other Keras models (4)

4. Keras models using batch normalization can be unreliable. For some models, forward-pass evaluations (with gradients supposedly off) still result in weights changing at inference time. (See 5)

You might be wondering: **How is that possible? Aren’t these the same model and shouldn’t they have the same performance if**
trained with the same conditions?

Well, you’re not alone. Curtis’ article also sparked some reactions on Twitter:

![Twitter](image)

“[to mitigate reproducibility issues] In Keras I
1) avoid batches during inference.”
!!!

wtf?

@yoavgo

Delip Rao @deliprao

#deeplearning reproducibility: “Resnet architectures
perform better in PyTorch and inception architectures
perform better in Keras” 😲😭

Jeremy Howard @jeremyphoward

Benchmarking Keras and PyTorch Pre-Trained Models

and some interesting insights into the reason for these differences:

Andrei Bursuc @abursuc

Replying to @jeremyphoward
In PyTorch all models in the zoo are trained by the dev
team in similar conditions. In Keras most of the pre-
trained models are imported from authors’
implementation (e.g. ResNet from initial caffe

Knowing (and trusting) these benchmarks are important because they allow you to make informed decisions around which framework to use
and are often used as baselines for research and implementation.

So what are some things to look out for when you're leveraging these pre-trained models?

**Considerations for using pre-trained models**

1. **How similar is your task? How similar is your data?**

   Are you expecting that cited 0.945% validation accuracy for the Keras Xception model you're using with your new dataset of x-rays? First, you need to check how similar your data is to the original dataset that the model was trained on (in this case: ImageNet). You also need to be aware of where the features have been transferred from (the bottom, middle, or top of the network) because that will impact model performance depending on task similarity.

   Read [CS231n—Transfer Learning](https://cs231n.github.io/transfer-learning/) and ‘How transferable are features in deep neural networks?’

2. **How did you preprocess the data?**

   Your model's pre-processing should be the same as the original model's training. With almost all torchvision models, they use the same pre-processing values. For Keras models, you should always use the `preprocess_input` function for the corresponding model-level module. For example:

   ```python
   # VGG16
   keras.applications.vgg16.preprocess_input
   # InceptionV3
   keras.applications.inception_v3.preprocess_input
   # ResNet50
   keras.applications.resnet50.preprocess_input
   ```

3. **What’s your backend?**
There were some rumblings on HackerNews that changing the Keras’ backend from TensorFlow to CNTK (Microsoft Cognitive toolkit) improved the performance. Since Keras is a model-level library, it does not handle lower-level operations such as tensor products, convolutions, etc… so it relies on other tensor manipulation frameworks like the TensorFlow backend and the Theano backend.

Max Woolf provided an excellent benchmarking project that found that while accuracy was the same between CNTK and TensorFlow, CNTK was faster at LSTMs and Multilayer Perceptions (MLPs) while TensorFlow was faster at CNNs and embeddings.

Woolf’s post is from 2017, so It’d be interesting to get an updated comparison that also includes Theano and MXNet as a backend (although Theano is now deprecated).

There are also some claims that there are certain versions of Theano that may ignore your seed (for a relevant post form Keras, see this).

4. What’s your hardware?

Are you using an Amazon EC2 NVIDIA Tesla K80 or a Google Compute NVIDIA Tesla P100? Maybe even a TPU? Check out these useful benchmark resources for run times for these different pretrained models.

- Apache MXNet’s GluonNLP 0.6: Closing the Gap in Reproducible Research with BERT
- Caleb Robinson’s ‘How to reproduce ImageNet validation results’ (and of course, again, Curtis’ benchmarking post)
- DL Bench
- Stanford DAWNBench
- TensorFlow’s performance benchmarks

5. What’s your learning rate?

In practice, you should either keep the pre-trained parameters fixed (ie. use the pre-trained models as feature extractors) as or tune them with a
fairly small learning in order to not unlearn everything in the original model.

6. Is there a difference in how you use optimizations like batch normalization or dropout, especially between training mode and inference mode?

As Curtis' post claims:

Keras models using batch normalization can be unreliable. For some models, forward-pass evaluations (with gradients supposedly off) still result in weights changing at inference time. (See 5)

But why is this the case?

According to Vasilis Vryniotis, Principal Data Scientist at Expedia, who first identified the issue with the frozen batch normalization layer in Keras (see Vasilis' PR here and detailed blog post here):

The problem with the current implementation of Keras is that when a batch normalization (BN) layer is frozen, it continues to use the mini-batch statistics during training. I believe a better approach when the BN is frozen is to use the moving mean and variance that it learned during training. Why? For the same reasons why the mini-batch statistics should not be updated when the layer is frozen: it can lead to poor results because the next layers are not trained properly.

Vasilis also cited instances where this discrepancy led to significant drops in model performance (“from 100% down to 50% accuracy) when the Keras model is switched from train mode to test mode.

Use these questions to guide how you interact with pre-trained models for your next project. Have comments, questions, or additions? Comment below!

.fs-text-boldWant more practical machine learning content? Follow us on Medium!