

Neutrino interaction classification with ResNet18

Andy Chappell

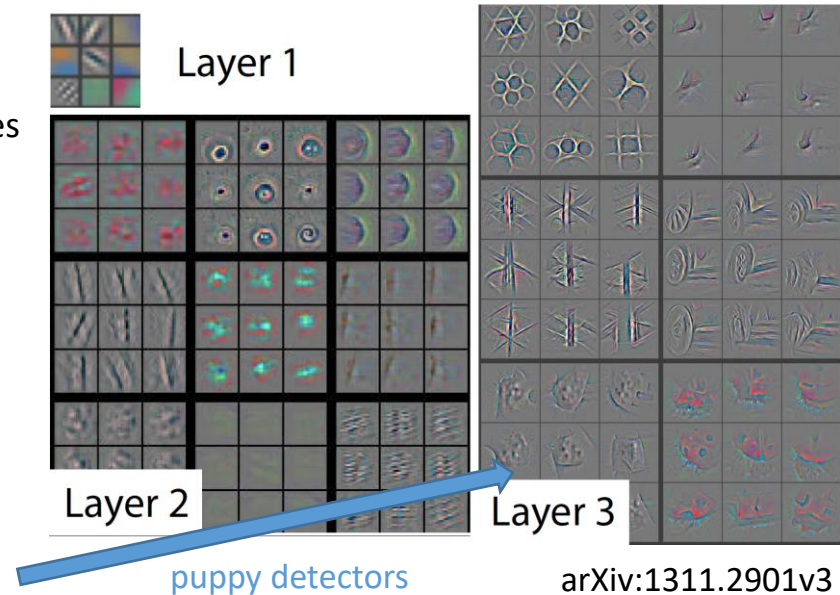
08/11/2022

LArSoft Workshop

- An overview of transfer learning
 - We'll be working with a pretrained ResNet18
 - How was the ResNet trained previously?
 - What happens when you introduce never-seen-before classes?
- Task 1: Introduce quasi elastic and resonant neutral current interactions to a network that has never seen them before
- Task 2: Investigate the network performance
- Task 3: Fine-tune the network to improve performance
- Task 4: Re-investigate the network performance

- In this tutorial we'll be introducing you to the practical use of transfer learning
- This is in part to encourage you to make use of transfer learning, because we don't do it enough in neutrino physics – GPUs are expensive and resource hungry, and it's no fun to wait 3 days for your results
- When available, transfer learning can save you a tremendous amount of time in achieving high performance
- Some training tasks won't even be tractable without it (e.g. limited training sample)
- However, today we need to come up with a task that can be undertaken in an afternoon using resources available on Google Colab
 - This necessarily requires that the task is quite simple
 - As such, transfer learning here is not likely to be obviously superior to training from scratch
 - However, the mechanics of the process are 'real-world' and so will transfer to your real-world use cases
 - It also means we won't have a test set, only training and validation (why is this potentially problematic?)

- ResNet18 is one of many pretrained networks available through PyTorch torchvision
 - Pretrained on [ImageNet](#) – a database of millions of photographs, comprising 1000 classes
 - 11,178,051 trainable parameters (this is the smallest ResNet)
- I don't recall ever seeing a puppy in a LArTPC, how is this relevant to neutrinos?
 - Networks learn features at different scales
 - Shallow layers learn the most primitive structures
 - Deep layers learn the most abstract features
 - Those early layers are still relevant in neutrino interactions



- The standard pretrained ResNet18 targeted the 1000 classes from ImageNet
 - That's not what we want for classifying neutrino interactions
 - The network that you'll use as the starting point for the tasks was adapted to target two classes, quasi elastic and resonant charged current events
- What does this look like?

```
1 import torchvision as tv
2 model = tv.models.resnet18(pretrained=True)
3 model
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  ...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

We want to change the number of output features, but we need to match the number of features output by the previous layer

- ResNet is structured such that the simplest change we can make to adapt to our use case is to replace the f_c (fully connected) layer

- The standard pretrained ResNet18 targeted the 1000 classes from ImageNet
 - That's not what we want for classifying neutrino interactions
 - The network that you'll use as the starting point for the tasks was adapted to target two classes, quasi elastic and resonant charged current events
- How do we modify the network?

```
1 import torch
2 in_features = model.fc.in_features
3 out_features = 2
4 model.fc = torch.nn.Linear(in_features, out_features)
5 model
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  ...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=2, bias=True)
)
```

Same number of input features,
but 2 output features matching
our two interaction classes

- We now have a network where almost all of the weights are those determined from training on ImageNet, but we now have an **untrained** `fc` layer

- In the previous slides we took advantage of torchvision features to quickly and easily define our model architecture (ResNet18) and download and apply weights learned by training on ImageNet
- What if we have custom weights?
- We've already transfer learned the 1000-class ResNet to the 2-class network for you and stored the weights in a file called `model_baseline.pt`

- We can load custom weights via

```
1 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
2 model.load_state_dict(torch.load(filename, map_location=device))
```

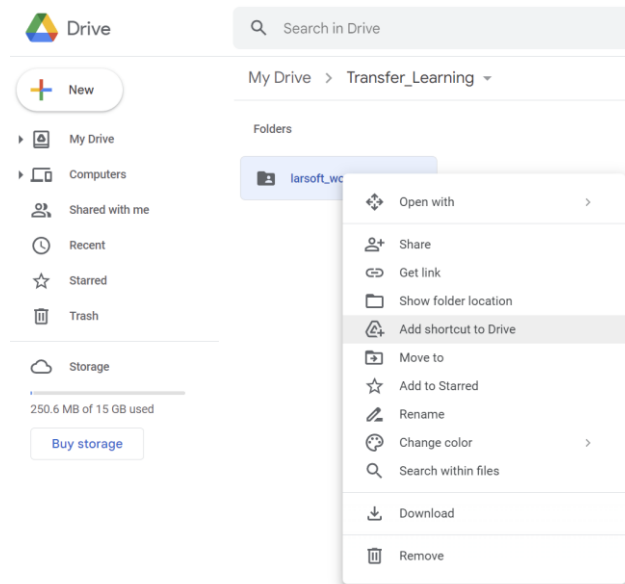
- So, we now have a ResNet18 network initialized with weights that can classify CCQE and CCRES interactions

Getting key files into Google Colab

- There are a few Python scripts, a baseline set of model weights and many training images that you need access to for this tutorial
- You can find all of these files in this location

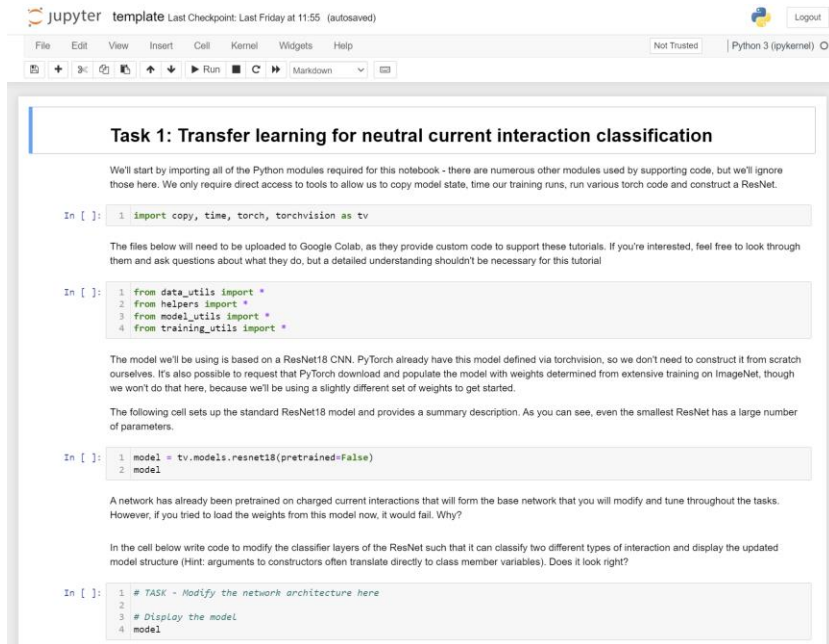
<https://drive.google.com/drive/folders/1M2KU1i8BtTeDAQdVYqyWBOJMOzhSyNIO?usp=sharing>

- Log into your Google account so you can create a short-cut to this location within your own Google Drive
- You'll need to move to the containing `Transfer_Learning` folder so you can see the `larsoft_workshop_files` folder
- Adding a shortcut will greatly ease the process of making these files accessible to Google Colab
- You should now see a folder in your personal Google Drive called `larsoft_workshop_files`, the notebook will handle the rest, you'll just need to give Colab permission to access your Google Drive when it asks



Working with the notebook

- You can find a template notebook at
 - https://github.com/AndyChappell/larsoft_workshop_2022
- You can load this following the same approach from the previous tutorial
- The notebook is self-describing and includes both pre-existing support code, but also a number of # TASK comments indicating where you are expected to write new code



The screenshot shows a Jupyter Notebook interface with the following content:

Task 1: Transfer learning for neutral current interaction classification

We'll start by importing all of the Python modules required for this notebook - there are numerous other modules used by supporting code, but we'll ignore those here. We only require direct access to tools to allow us to copy model state, time our training runs, run various torch code and construct a ResNet.

```
In [ ]: 1 import copy, time, torch, torchvision as tv
```

The files below will need to be uploaded to Google Colab, as they provide custom code to support these tutorials. If you're interested, feel free to look through them and ask questions about what they do, but a detailed understanding shouldn't be necessary for this tutorial

```
In [ ]: 1 from data_utils import *
2 from helpers import *
3 from model_utils import *
4 from training_utils import *
```

The model we'll be using is based on a ResNet18 CNN. PyTorch already have this model defined via torchvision, so we don't need to construct it from scratch ourselves. It's also possible to request that PyTorch download and populate the model with weights determined from extensive training on ImageNet, though we won't do that here, because we'll be using a slightly different set of weights to get started.

The following cell sets up the standard ResNet18 model and provides a summary description. As you can see, even the smallest ResNet has a large number of parameters.

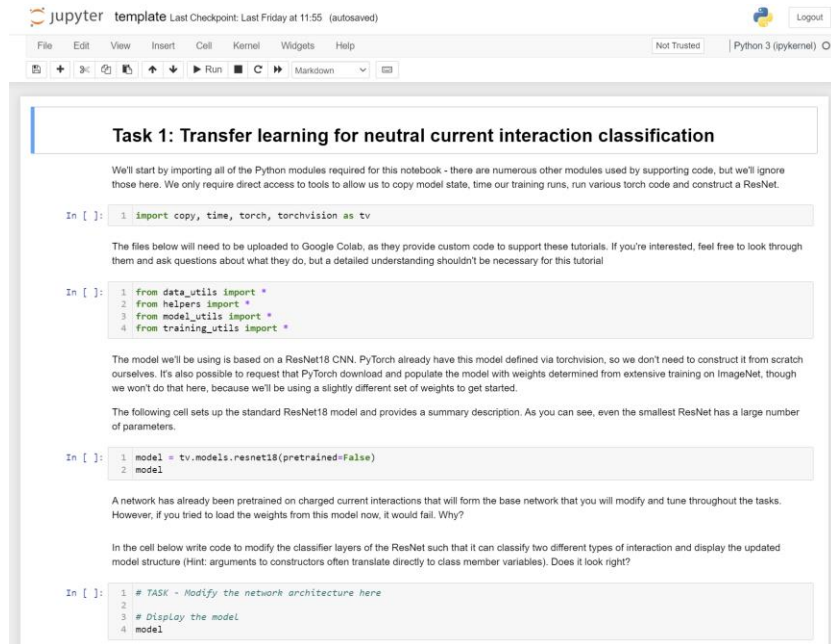
```
In [ ]: 1 model = tv.models.resnet18(pretrained=False)
2 model
```

A network has already been pretrained on charged current interactions that will form the base network that you will modify and tune throughout the tasks. However, if you tried to load the weights from this model now, it would fail. Why?

In the cell below write code to modify the classifier layers of the ResNet such that it can classify two different types of interaction and display the updated model structure (Hint: arguments to constructors often translate directly to class member variables). Does it look right?

```
In [ ]: 1 # TASK - Modify the network architecture here
2
3 # Display the model
4 model
```

- The notebook references various helper functions defined in four Python scripts included at the start of the notebook
 - If you start working with neural networks more in the future, you'll quickly discover that a lot of work goes into defining truth information and managing datasets (torchvision provides a lot of help here too, but sometimes you need custom datasets and more rarely custom dataloaders – as in the vertex finding network)
 - We've eliminated much of this overhead in this tutorial in the interest of time, but feel free to look through these scripts and ask questions if you're interested



The screenshot shows a Jupyter Notebook interface with the following content:

Task 1: Transfer learning for neutral current interaction classification

We'll start by importing all of the Python modules required for this notebook - there are numerous other modules used by supporting code, but we'll ignore those here. We only require direct access to tools to allow us to copy model state, time our training runs, run various torch code and construct a ResNet.

```
In [ ]: 1 import copy, time, torch, torchvision as tv
```

The files below will need to be uploaded to Google Colab, as they provide custom code to support these tutorials. If you're interested, feel free to look through them and ask questions about what they do, but a detailed understanding shouldn't be necessary for this tutorial

```
In [ ]: 1 from data_utils import *
2 from helpers import *
3 from model_utils import *
4 from training_utils import *
```

The model we'll be using is based on a ResNet18 CNN. PyTorch already have this model defined via torchvision, so we don't need to construct it from scratch ourselves. It's also possible to request that PyTorch download and populate the model with weights determined from extensive training on ImageNet, though we won't do that here, because we'll be using a slightly different set of weights to get started.

The following cell sets up the standard ResNet18 model and provides a summary description. As you can see, even the smallest ResNet has a large number of parameters.

```
In [ ]: 1 model = tv.models.resnet18(pretrained=False)
2 model
```

A network has already been pretrained on charged current interactions that will form the base network that you will modify and tune throughout the tasks. However, if you tried to load the weights from this model now, it would fail. Why?

In the cell below write code to modify the classifier layers of the ResNet such that it can classify two different types of interaction and display the updated model structure (Hint: arguments to constructors often translate directly to class member variables). Does it look right?

```
In [ ]: 1 # TASK - Modify the network architecture here
2
3 # Display the model
4 model
```

Task 1: Classifying neutral current interactions

- The network defined in `model_baseline.pt` has never seen a neutral current interaction
- We've provided you with a set of images containing, in addition to CC interactions, quasi elastic and resonant NC interactions
- Your task is to adapt the network so that, in addition to CC interactions, it can also classify these two new neutral current classes, but without modifying the weights in the feature extractor layers (that is, everything until the final, fully connected layer)
- To do this, you'll need to freeze the existing network weights and modify the fully connected layer

Task 2: Investigating network performance

- Hopefully you have a trained network
- Now it's time to assess its performance
- What is the classification accuracy?
 - Are you impressed? Underwhelmed?
- What, if any, classes are being mixed up?
 - Produce a confusion matrix from the validation set
 - Do the errors make sense?

- The network you trained in task 1 only permitted changes to the classifier layer
- This is a particularly good way to train if you have some new examples from within existing classes that the network is familiar with
 - Maybe all of your CCRES interactions had zero or one proton and now you have some two proton examples
 - Primitive features of the network are unlikely to change, so you really just want it to learn the final classification step
 - Freezing the early layers and training only the fc layer permits rapid tuning of a network
- Introducing NC interactions however will yield some more fundamental differences in topologies and so we probably want to be able to finetune the earlier layers too
- Your task is to unlock all of the network parameters and continue training
 - What should the learning rate be?
 - For how long should you train?

Task 4: Re-investigating network performance

- Same as task 2, but with your fully fine-tuned network