# Transfer Learning tutorial

# Install Anaconda(<https://www.anaconda.com/download/>)

# Anaconda is a free and open source distribution of the Python and R programming languages for data science and machine learning related applications (large-scale data processing, predictive analytics, scientific computing), that aims to simplify package management and deployment.

# 

# Install Pytorch(http://pytorch.org)

# Tensors and Dynamic neural networks in Python with strong GPU acceleration.

# 

# Install torchvision

# Run this command in Anaconda Prompt:

pip install torchvision

1. The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.

In this tutorial, you will learn how to train your network using transfer learning.

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

These two major transfer learning scenarios look as follows:

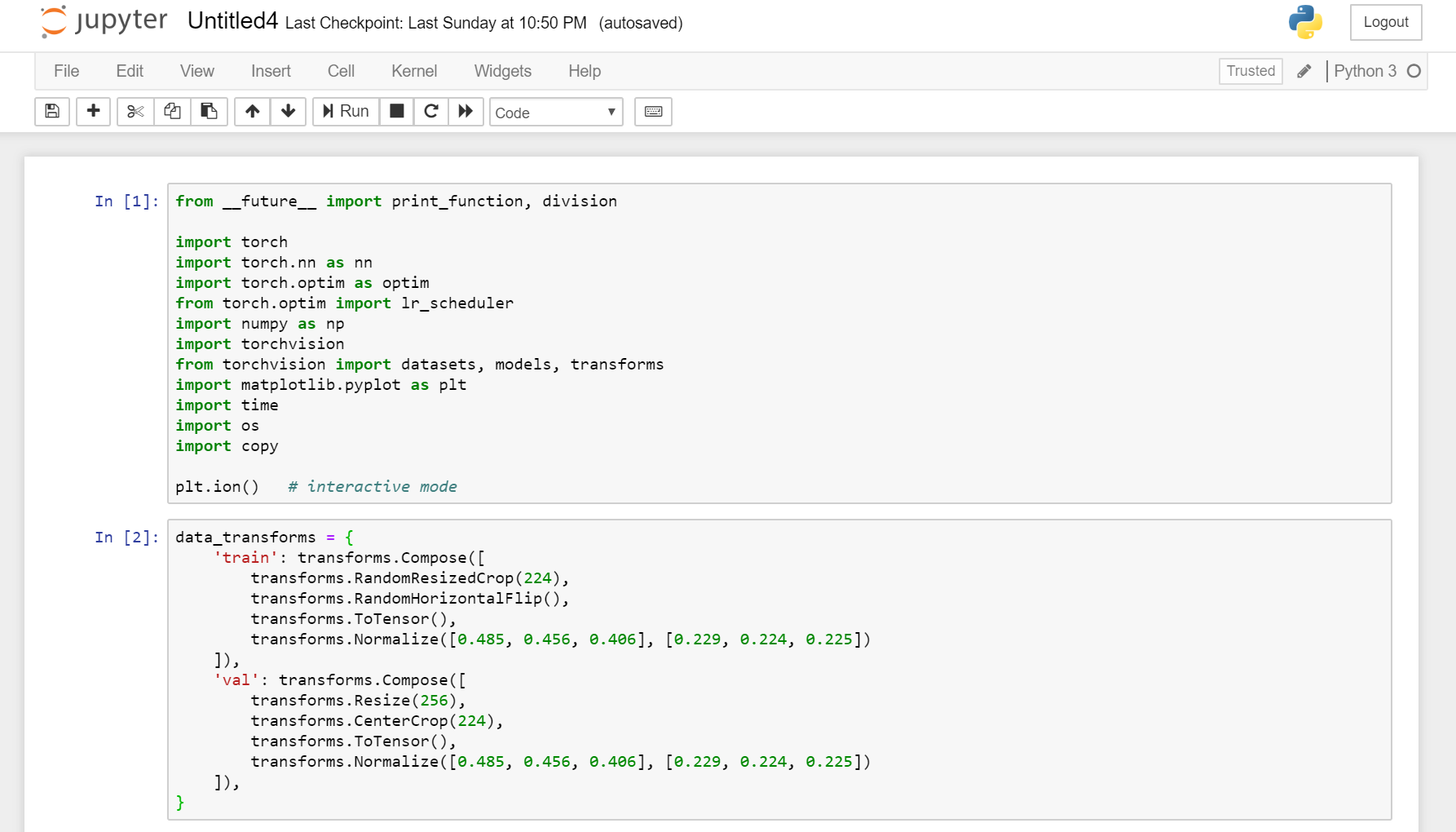
**Finetuning the convnet**: Instead of random initializaion, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.

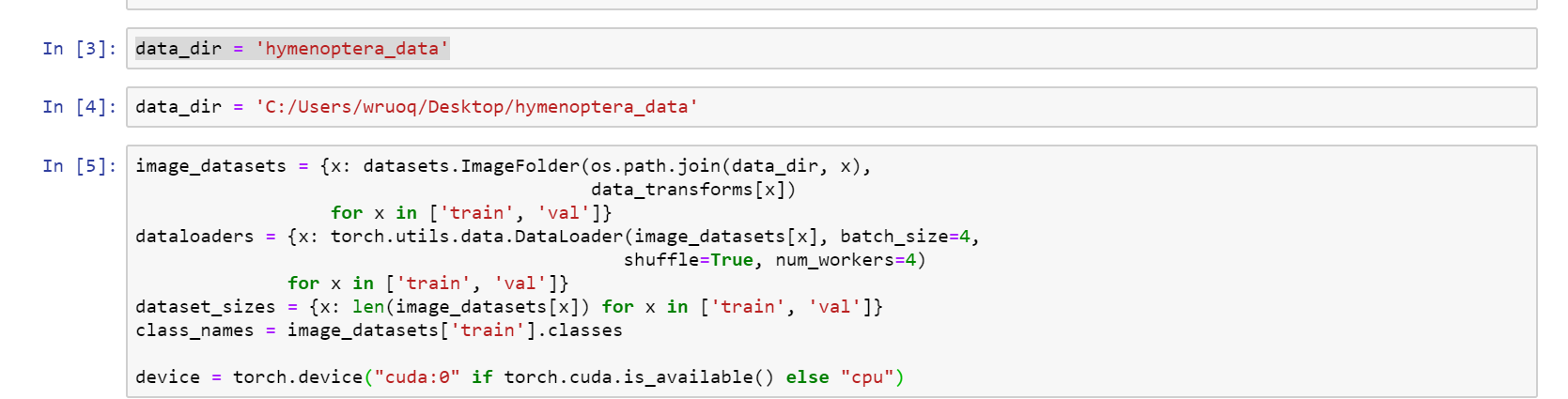
**ConvNet as fixed feature extracto**r: Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

* 1. **Load Data**

We will use torchvision and torch.utils.data packages for loading the data.

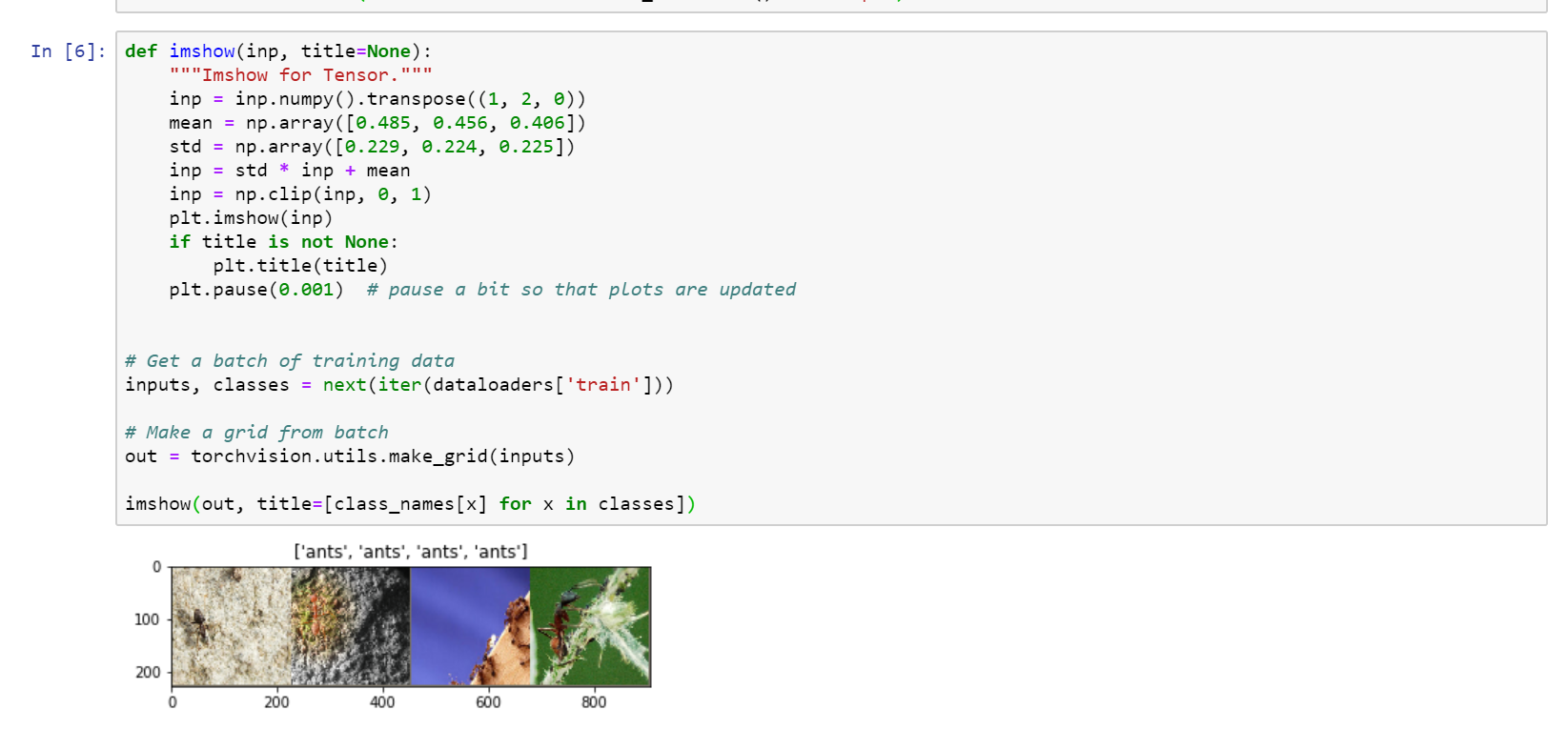
The problem we’re going to solve today is to train a model to classify **ants** and **bees**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually, this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well.This dataset is a very small subset of imagenet.Note : Download the data from here <https://download.pytorch.org/tutorial/hymenoptera\_data.zip>and extract it to the current directory.





**4.1.1 Visualize a few images**

Let’s visualize a few training images so as to understand the data augmentations.

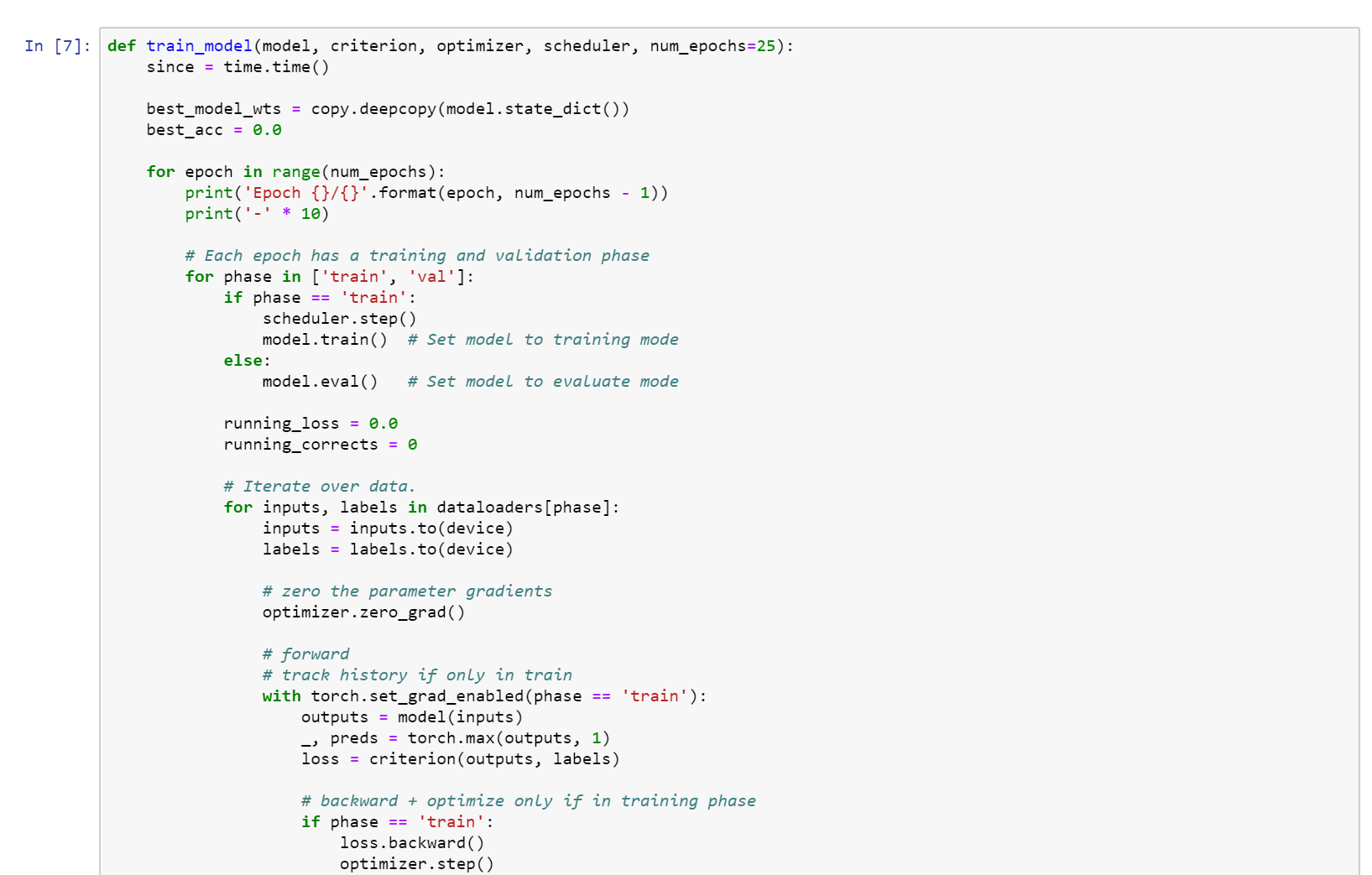


**4.2 Training the model**

Now, let’s write a general function to train a model. Here, we will illustrate:

* Scheduling the learning rate
* Saving the best model

In the following, parameter scheduler is an LR scheduler object fromtorch.optim.lr\_scheduler.





### 4.2.1 Visualizing the model predictions

### Generic function to display predictions for a few images

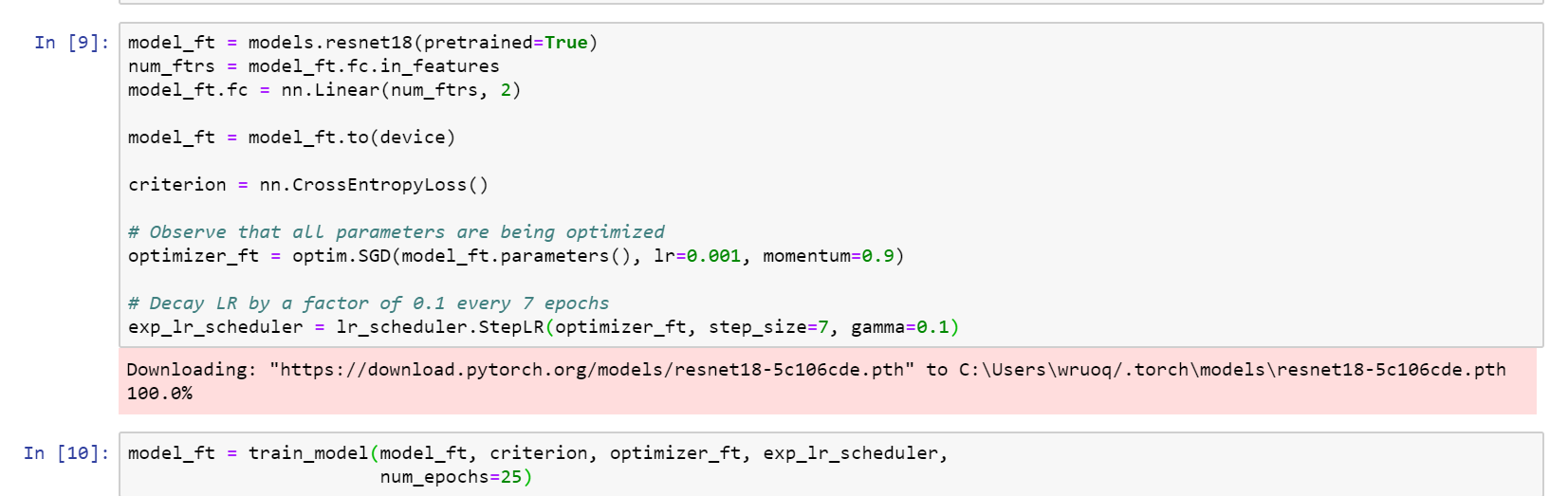


## 4.3 Finetuning the convnet

Load a pretrained model and reset final fully connected layer.

### Train and evaluate

It should take around 15-25 min on CPU. On GPU though, it takes less than a minute.



OUT:

Epoch 0/24

----------

train Loss: 0.5596 Acc: 0.7172

val Loss: 0.2106 Acc: 0.9216

Epoch 1/24

----------

train Loss: 0.4152 Acc: 0.8115

val Loss: 0.6844 Acc: 0.7712

Epoch 2/24

----------

train Loss: 0.3846 Acc: 0.8607

val Loss: 0.2728 Acc: 0.9085

Epoch 3/24

----------

train Loss: 0.6967 Acc: 0.7664

val Loss: 0.4466 Acc: 0.8627

Epoch 4/24

----------

train Loss: 0.5282 Acc: 0.8156

val Loss: 0.5506 Acc: 0.7843

Epoch 5/24

----------

train Loss: 0.5530 Acc: 0.7705

val Loss: 0.3770 Acc: 0.9020

Epoch 6/24

----------

train Loss: 0.4715 Acc: 0.7910

val Loss: 0.2892 Acc: 0.8889

Epoch 7/24

----------

train Loss: 0.3106 Acc: 0.8443

val Loss: 0.2509 Acc: 0.9346

Epoch 8/24

----------

train Loss: 0.2589 Acc: 0.8975

val Loss: 0.2698 Acc: 0.8954

Epoch 9/24

----------

train Loss: 0.3522 Acc: 0.8525

val Loss: 0.2441 Acc: 0.9150

Epoch 10/24

----------

train Loss: 0.2450 Acc: 0.8852

val Loss: 0.2306 Acc: 0.9216

Epoch 11/24

----------

train Loss: 0.2908 Acc: 0.8852

val Loss: 0.2233 Acc: 0.9216

Epoch 12/24

----------

train Loss: 0.2713 Acc: 0.9016

val Loss: 0.2212 Acc: 0.9216

Epoch 13/24

----------

train Loss: 0.2378 Acc: 0.8893

val Loss: 0.2169 Acc: 0.9216

Epoch 14/24

----------

train Loss: 0.4035 Acc: 0.8402

val Loss: 0.2231 Acc: 0.9216

Epoch 15/24

----------

train Loss: 0.3664 Acc: 0.8320

val Loss: 0.2242 Acc: 0.9281

Epoch 16/24

----------

train Loss: 0.2601 Acc: 0.8811

val Loss: 0.2157 Acc: 0.9346

Epoch 17/24

----------

train Loss: 0.2566 Acc: 0.8893

val Loss: 0.2307 Acc: 0.9216

Epoch 18/24

----------

train Loss: 0.3377 Acc: 0.8689

val Loss: 0.2235 Acc: 0.9150

Epoch 19/24

----------

train Loss: 0.3003 Acc: 0.8607

val Loss: 0.2328 Acc: 0.9216

Epoch 20/24

----------

train Loss: 0.2363 Acc: 0.8975

val Loss: 0.2621 Acc: 0.9020

Epoch 21/24

----------

train Loss: 0.2169 Acc: 0.9016

val Loss: 0.2336 Acc: 0.9085

Epoch 22/24

----------

train Loss: 0.2599 Acc: 0.8852

val Loss: 0.2334 Acc: 0.9085

Epoch 23/24

----------

train Loss: 0.2586 Acc: 0.8648

val Loss: 0.2174 Acc: 0.9412

Epoch 24/24

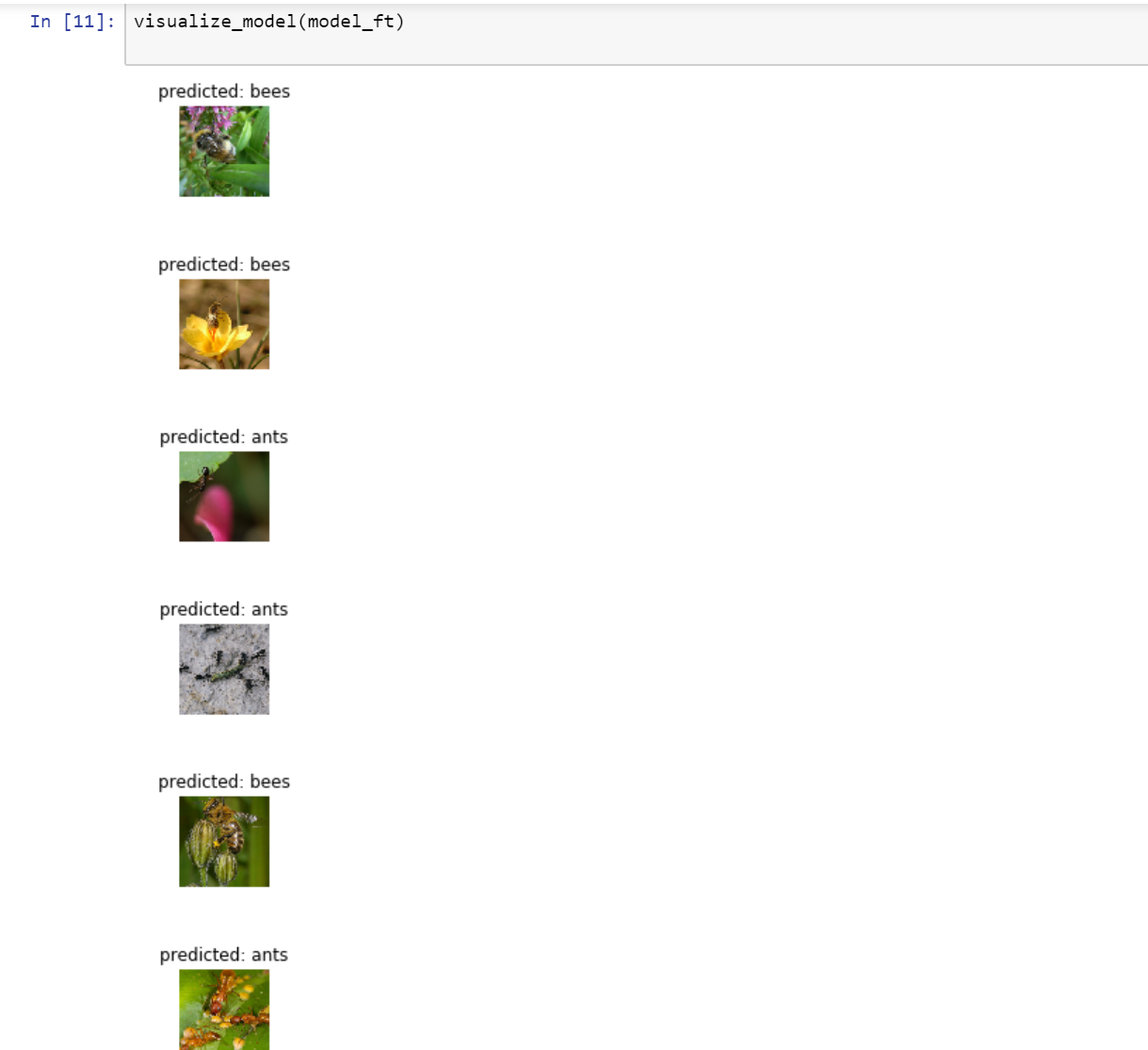
----------

train Loss: 0.2213 Acc: 0.9057

val Loss: 0.2324 Acc: 0.9150

Training complete in 42m 44s

Best val Acc: 0.941176

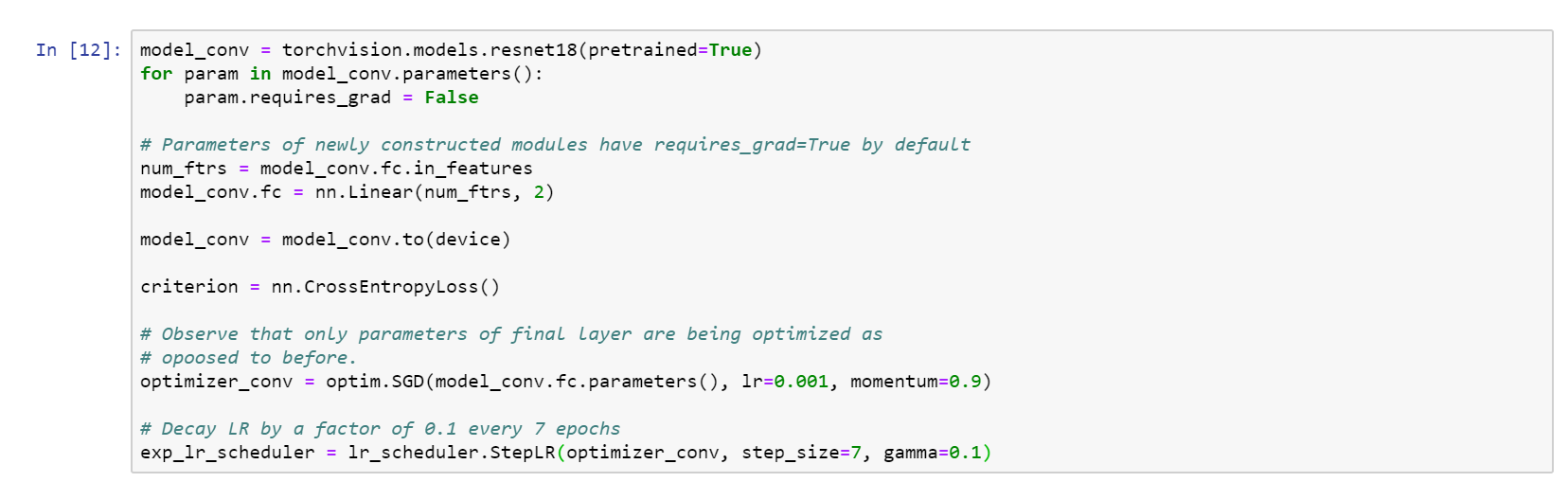


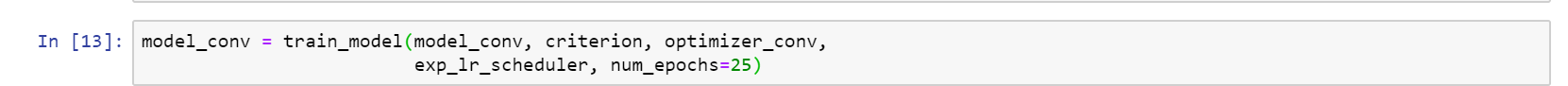
**4.4 ConvNet as fixed feature extractor**

Here, we need to freeze all the network except the final layer. We need to set requires\_grad == False to freeze the parameters so that the gradients are not computed in backward().

**Train and evaluate**

On CPU this will take about half the time compared to previous scenario. This is expected as gradients don’t need to be computed for most of the network. However, forward does need to be computed.





Epoch 0/24

----------

train Loss: 0.6271 Acc: 0.6885

val Loss: 0.5396 Acc: 0.7516

Epoch 1/24

----------

train Loss: 0.5643 Acc: 0.7377

val Loss: 0.1949 Acc: 0.9346

Epoch 2/24

----------

train Loss: 0.5512 Acc: 0.7541

val Loss: 0.1840 Acc: 0.9346

Epoch 3/24

----------

train Loss: 0.5408 Acc: 0.7664

val Loss: 0.1624 Acc: 0.9542

Epoch 4/24

----------

train Loss: 0.4277 Acc: 0.8115

val Loss: 0.1756 Acc: 0.9412

Epoch 5/24

----------

train Loss: 0.4372 Acc: 0.8197

val Loss: 0.1795 Acc: 0.9412

Epoch 6/24

----------

train Loss: 0.4244 Acc: 0.8279

val Loss: 0.1810 Acc: 0.9412

Epoch 7/24

----------

train Loss: 0.3780 Acc: 0.8279

val Loss: 0.1840 Acc: 0.9412

Epoch 8/24

----------

train Loss: 0.3627 Acc: 0.8197

val Loss: 0.1860 Acc: 0.9346

Epoch 9/24

----------

train Loss: 0.3690 Acc: 0.8484

val Loss: 0.1994 Acc: 0.9281

Epoch 10/24

----------

train Loss: 0.4171 Acc: 0.8197

val Loss: 0.1662 Acc: 0.9477

Epoch 11/24

----------

train Loss: 0.3367 Acc: 0.8320

val Loss: 0.2038 Acc: 0.9216

Epoch 12/24

----------

train Loss: 0.3618 Acc: 0.8443

val Loss: 0.1772 Acc: 0.9477

Epoch 13/24

----------

train Loss: 0.3677 Acc: 0.8484

val Loss: 0.1971 Acc: 0.9281

Epoch 14/24

----------

train Loss: 0.2710 Acc: 0.8975

val Loss: 0.1773 Acc: 0.9412

Epoch 15/24

----------

train Loss: 0.3547 Acc: 0.8525

val Loss: 0.1679 Acc: 0.9477

Epoch 16/24

----------

train Loss: 0.3423 Acc: 0.8402

val Loss: 0.1848 Acc: 0.9346

Epoch 17/24

----------

train Loss: 0.3559 Acc: 0.8525

val Loss: 0.1724 Acc: 0.9477

Epoch 18/24

----------

train Loss: 0.3623 Acc: 0.8484

val Loss: 0.1775 Acc: 0.9542

Epoch 19/24

----------

train Loss: 0.3157 Acc: 0.8566

val Loss: 0.1667 Acc: 0.9542

Epoch 20/24

----------

train Loss: 0.3553 Acc: 0.8525

val Loss: 0.1822 Acc: 0.9412

Epoch 21/24

----------

train Loss: 0.3598 Acc: 0.8320

val Loss: 0.1733 Acc: 0.9412

Epoch 22/24

----------

train Loss: 0.3624 Acc: 0.8566

val Loss: 0.2122 Acc: 0.9281

Epoch 23/24

----------

train Loss: 0.3753 Acc: 0.8361

val Loss: 0.1961 Acc: 0.9346

Epoch 24/24

----------

train Loss: 0.2795 Acc: 0.8811

val Loss: 0.1724 Acc: 0.9412

Training complete in 24m 19s

Best val Acc: 0.954248



