UDACITY

‹ Return to Classroom

Landmark Classification & Tagging for Social Media 2.0

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Hello!

Congratulations for passing the project on your first submission 🎉 this was very strong and almost perfect!

A few tips and tricks, as well as elaborations are added in the project review please see rubrics below for this 👌

For your next CNN models, another baseline approach / cheatsheet including links to case studies is this article.

You have put the work in and you have made it, feel free to revisit the project at any given time!

Keep up the good work!

File Requirements

The submission includes at least the following files:
cnn_from_scratch.ipynb
transfer_learning.ipynb
app.ipynb
src/train.py

src/model.py
src/helpers.py
src/predictor.py
src/transfer.py
src/optimization.py
src/data.py

All required files included in this submission, appreciated! 👌🏼

Thank you for successfully submitting the worked through notebooks, and reports in form of a .html versions, way to go!

See here for further elaborations on why this is important for cross-platform specific readability/functionality of the project.

Create a CNN to Classify Landmarks from Scratch (cnn_from_scratch.ipynb notebook)

- In the file 'src/data.py', all the YOUR CODE HERE sections have been replaced with code
- The data_transforms dictionary contains train, valid and test keys. The values are instances of transforms.Compose. At the minimum, the 3 set of transforms contains a Resize(256) step, a crop step (RandomCrop for train and CenterCrop for valid and test), a ToTensor step and finally a Normalize step (which uses the mean and std of the dataset). The train transforms should also contain, in-between the crop and the ToTensor, one or more data augmentation transforms.
- The ImageFolder instances for train, valid and test use the appropriate transform from the data_transforms dictionary (using the “transform” keyword of ImageFolder)
- The data loaders for train, valid and test use the right ImageFolder instance and use the batch_size, sampler, and num_workers that are given in input to the function
- In the notebook, the tests for this function were run and they are all PASSED

Pytorch ImageFolder functionality applied to load images generically ✅

In cnn_from_scratch.py for src/data.py function test are passing ✅

- Answer describes each step of the image preprocessing and augmentation. Augmentation (cropping, rotating, etc.) is not a requirement, but highly recommended

Excellent 👏🏼, you have precisely described the steps you have taken to load the data, including resizing the images to speed up training, and adding augmentations on the training set to enlarge the data, to further explore this venue you can have a look at this repository for what is further possible in regards to augmentations, and see here torchvision native augmentations which can freely be chosen from!

- The code gets an iterator from the train data loader and then uses it to obtain a batch of images and labels
- The code gets the class names from the train data loader
- In the notebook, the 'get_data_loaders' function is used to get the 'data_loaders' dictionary
- Then the function 'visualize_one_batch' is called and 5 images from the train data loader are shown with their labels

This DataLoader lies at the heart of pytorch torch.utils.data utility, you have correctly returned the batch_data utilising this functionality ✨

Well done, you have displayed a reasonable number of images like a good citizen, meeting requirements here 💯

You can further costumize this by changing e.g. the figure size if the image labels overlap, great that you left the class-number in the label, for orientation!

- Within 'src/model.py', all the YOUR CODE HERE sections have been replaced with code
- Both the **init** and the forward method of the class MyModel have been filled
- The class MyModel implements a CNN architecture
- The output layer of the CNN architecture has num_classes outputs (i.e., the number of outputs should not be hardcoded, but should instead use the num_classes parameter passed to the constructor)
- If the CNN architecture uses DropOut, then the amount of dropout should be controlled by the "dropout" parameter of the **init** method
- The .forward method should *NOT* include the application of Softmax
- In the notebook, the tests for this function were run and they are all PASSED

The CNN architecture is built neatly with a good number convolutional layers, including dropout and the needed flattening step in the forward pass.

As a sidenote, newer papers have shown to achieve better results and more effective training with smaller kernels (kernel_size=3), as you did, and using a smaller kernel may allow better feature detection 🔎

Further the relatively large images are better accomodated with 4-6 convolutional layers, increasing the size of the network like this will significantly increase the accuracy 👌

Answer describes the reasoning behind the selection of architecture type, layer types and so on. The students should reuse some of the concepts learned during the class.

Well done, you have precisely described the steps you have taken to built the model, choosing relu() activation as the default, in all except the last layer, which is activated by LogSoftmax()(inside CrossEntropy()), since we are performing a **multiclass classification**. ✨

- In the file 'src/optimization.py', all the YOUR CODE HERE sections have been replaced with code

- **The get_loss function returns the appropriate loss for a multiclass classification (CrossEntropy loss)**
- **The relative test for the 'get_loss function' is run in the notebook and is PASSED**
- **In the 'get_optimizer' function, both the SGD and the Adam optimizer are initialized with the provided input model, as well as with the learning_rate, momentum (for SGD), and weight_decay provided in input**
- **The relative test for the 'get_optimizer' function is run in the notebook and is PASSED**

For the CNN network you have perfectly letting the user choose between SGD and Adam as the optimiser, SGD is expected to decrease the losses a bit better when it comes to a larger number of epochs, again it should be experimented with the learning rate here, to see how training progresses!

Great work choosing CrossEntropyLoss() as the loss function, this is 2 steps in 1: NLLLoss() + LogSoftmax() 👏

- **In 'src/train.py', all the YOUR CODE HERE sections have been replaced with code**
- **In the function 'train_one_epoch', the model is set to training mode. Then a proper training loop is completed: the gradient is cleared, a forward pass is completed, the value for the loss is computed, and a backward pass is completed. Finally, the parameters are updated by completing an optimizer step.**
- **The tests relative to the function 'train_one_epoch' are run (from the notebook) and are all PASSED**
- **In the function 'valid_one_epoch', the model is set to evaluation mode (so that no gradients are computed), then within the loop a forward pass is completed, and the validation loss is calculated. There should be no backward pass here (this is different than the training loop).**
- **The tests relative to the function 'valid_one_epoch' are run (from the notebook) and are all PASSED**
- **In the 'optimize' function, the learning rate scheduler that reduces the learning rate on plateau is initialized. Then within the loop, the weights are saved if the validation loss decreases by more than 1% from the previous minimum validation loss. Then the learning rate scheduler is triggered by making a step.**
- **The tests relative to the function 'optimize' are run (from the notebook) and are all PASSED**
- **In the function 'one_epoch_test', the model is set to evaluation mode, a forward pass is completed within the loop, and the loss value is computed. Finally, the prediction is computed by taking the argmax of the logits.**
- **The tests relative to the function 'one_epoch_test' are run (from the notebook) and are all PASSED.**

Great job, training algorithm correctly uses train and validation data, and the epochs which improved the validation loss are saved as the model, kudos to you! 👌

Further you have correctly implemented the Learning Rate Scheduler, to come closer to local minima at potential plateaus during last strides of training process.

Also great work with this function and formatting, you can use the new improved f-string formatting functionalities in python!

- Sensible hyperparameters are used (the default or not)
- The solution gets the dataloaders, the model, the optimizer and the loss using the functions completed in the previous steps
- The model is trained successfully (the train and validation loss decrease with the epochs)

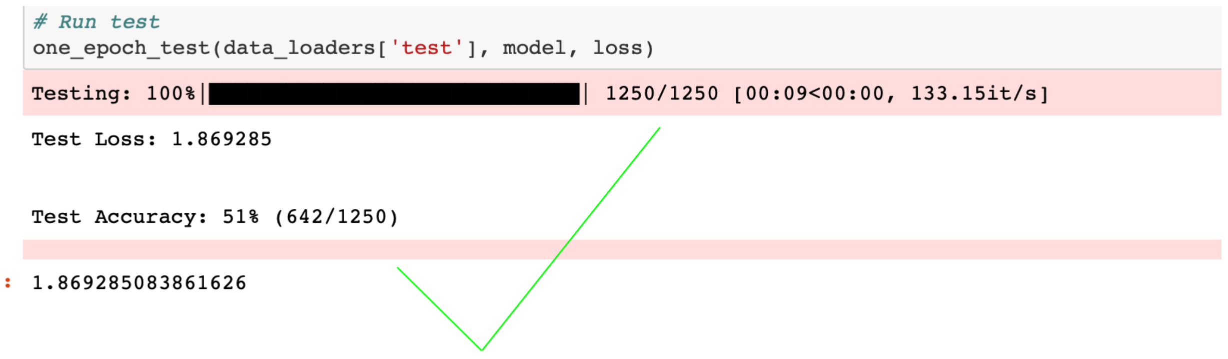
Sensible hyperparameters are added, resulting in an overall good training outcome ✅

- The student runs the testing code in the notebook and obtains a test accuracy of at least 50%

Great job achieving 51% accuracy on the testset! ⭐

More is possible by tweaking the parameters, even if not as much as the transfer learning model coming up, still, you can experiment with:

- optimizer
- number and sizes of convolutional layers
- kernel_size
- dropout rate
- learning rate
- epochs
- weight_decay



```
# Run test
one_epoch_test(data_loaders['test'], model, loss)
```

```
Testing: 100%|██████████████████████████████| 1250/1250 [00:09<00:00, 133.15it/s]
Test Loss: 1.869285


Test Accuracy: 51% (642/1250)

: 1.869285083861626
```

- In 'src/predictor.py', the .forward method is completed so that it applies the transforms defined in **init** (using self.transforms), uses the model to get the logits, applies the 'softmax' function across dim=1, and returns the result
- In the notebook, the tests are run and they are all PASSED
- In the notebook, all the YOUR CODE HERE sections have been replaced with code. In particular, the best weights from the training run are loaded, and torch.jit.script is used to generate the TorchScript serialization from the model
- In the next cell, the saved checkpoints/original_exported.pt file is loaded back using torch.jit.load, then all the remaining cells are run to get the confusion matrix of the exported model
- The diagonal of the confusion matrix should have a lighter color, signifying that most test examples are correctly classified

The predictor is correctly built and functional, by applying the forward method, as transforms are used correctly, and output softmax activated. 💯

Use Transfer Learning (transfer_learning.ipynb notebook)

- **In 'src/transfer.py', all the YOUR CODE HERE sections have been replaced with code**
- **All parameters of the loaded architecture are frozen, and a linear layer at the end has been added using the appropriate input features (as returned by the backbone), and the appropriate output features, as specified by the n_classes parameter**
- **The tests in Step 1 in the notebook are run and they are all PASSED**

Transfer model implemented correctly, resnet18 was chosen, and all tests pass requirements ✅

- **The hyperparameters in the notebook are reasonable**
- **The function 'get_model_transfer_learning' is used to get a model**
- **The model is trained successfully (the train and validation loss decrease with the epoch)**

Sensible hyperparameters are added, resulting in an overall good training outcome ✅

- **The submission provides an appropriate explanation why the chosen architecture is suitable for this classification task.**

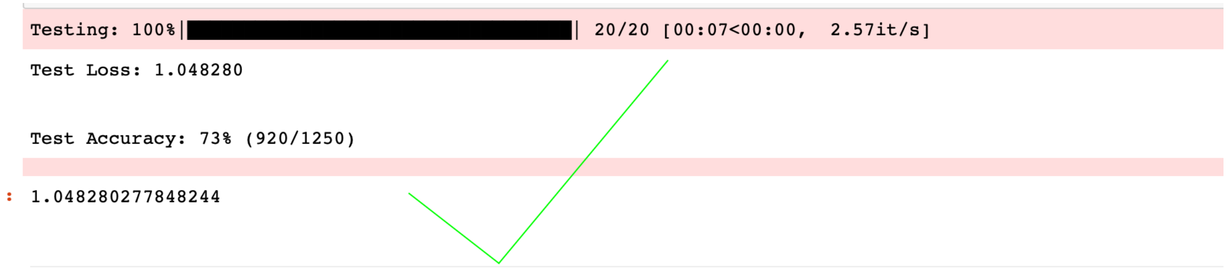
You are meeting requirements here ✔ your reason for why resnet was suitable as being consolidated as a standard as on of the winners of the ImageNet competition is on point.

- **Test accuracy is at least 60%**

With the transfer learning model you have achieved 73% accuracy on the testset, which is a great score and makes the model production ready immediately ⭐

Again, when considering using the model refining can be done by experimenting with the hyperparameters, like normal:

- optimizer
- weight decay
- learning rate
- epochs



```
Testing: 100%|████████████████████████████| 20/20 [00:07<00:00,  2.57it/s]
Test Loss: 1.048280


Test Accuracy: 73% (920/1250)

: 1.048280277848244
```

- Appropriate cells have been run to save the transfer learning model into “checkpoints/transfer_exported.pt”

Model saved to checkpoint ✔

Write Your App (app.ipynb notebook)

- All the YOUR CODE HERE sections in the notebook have been replaced with code
- One of the two TorchScript exports (either the cnn from scratch or the transfer learning model) are loaded using torch.jist.load
- The app is run. In the saved notebook, an image is shown that is not part of the training nor the test set, along with the predictions of the model.

❕ **[Note]** There is an issue with the installation of the environment for running the app, you will first need to install:

```
!pip install ipywidgets
```

including extensions:

```
!jupyter nbextension enable --py widgetsnbextension
```

Whereafter the notebook must be restarted, like so the app is functional.

I leave this for your information.

DOWNLOAD PROJECT

RETURN TO PATH