UDACITY

< Return to Classroom

Landmark Classification & Tagging for Social Media 2.0

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Congratulations on finishing this project successfully 🎉 Good luck with the next project in your nanodegree. Check out the below review for more feedback :)

Some excellent work you have in here. The project is now complete and meets all the requirements.
I liked the way how the project is structured and is a wonderful read for anyone attempting to do similar work.

I hope you had a great learning time doing this project and will keep the same spirit going forward when you tackle a real-world problem

All the best and keep learning

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

Great start

The project has all the required files.

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

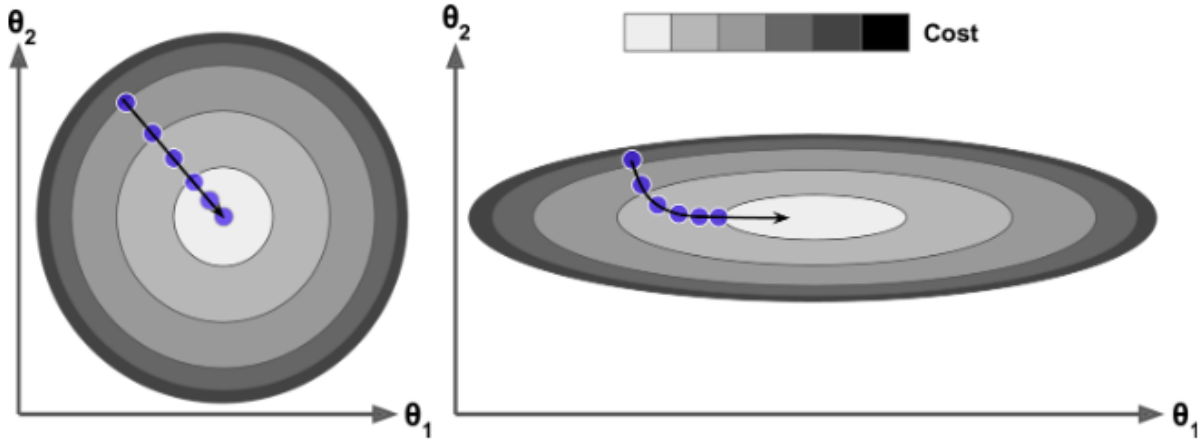
- In the file 'src/data.py', all the YOUR CODE HERE sections have been replaced with code
- The data_transforms dictionary contains train, valid and test keys. The values are instances of transforms.Compose. At the minimum, the 3 set of transforms contains a Resize(256) step, a crop step (RandomCrop 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

- You have used RandomCrop instead of CenterCrop in the training transforms. RandomCrop takes a random crop of the image, which provides more variability during training time and results in a more robust model.
- In the test data loader, you have added shuffle=False to prevent shuffling. This is a best practice to avoid variability when running the test, especially across models.

Check out this Udacity video to see why the splitting of the data is needed to be done; It tells you the difference between the validation set and test set

Nice work in normalizing the input features in the data. It is a necessary step. The cost function can have the shape of an elongated bowl if the features have very different scales. The figure below shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set

where feature 1 has much smaller values than feature 2 (on the right).



As you can see, on the left the Gradient Descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

Check out this Udacity video

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

Data Augmentation steps help to create more data which can be used to train the model. Bigger the size of the data, better the model will be trained. Data augmentation also helps to generalize the model as the data augmentation can create input distribution which may not be given in the original training set's distribution. But you also need to make sure that the data augmentation processing creates the input which will reflect the real world conditions for the model to learn 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**

Nice work in using data_loaders as a dictionary. The dictionary is a hash table that has constant search time (i.e. O(1)) while looking up for the element. Check out this course which is about data structures and algorithms

Check out this Udacity video that highlights the benefit of batching of the data

There are also computational benefits of using the batches to train the machine learning model. If you load the entire dataset (which can very large) at once to train the machine learning, this can result in memory resource issues. Instead, we can iterate over the data as batches to be loaded on to the memory. Python iterator is used for iterating over the data that brings in memory performance. Python iterator gives out the

batch for the training and then in the next iteration gives out another batch of the data after freeing up the previous batch from the memory. In this way, the iterator brings memory efficiency.

- 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

Good job

- Model.py is correctly setup, as a suggestion, you can remove the `YOUR CODE HERE` when you update with the code
- My model class is set up right, both the init and forward method is correct
- Num classes is not hardcoded in the CNN arch and is taken as a parameter.
- Use of a drop-out layer is also seen in your cnn architecture

All the test functions are passing
A good read on Drop out

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.

Usually (outside of this project) your choice of the hyper-parameter should not only improve the accuracy levels of the ML applications but also improve the latency, computational demands, and various other metrics like biasness etc.

For example, here is an interesting paper in which the CNNs were used to improve the resolutions of the images; https://arxiv.org/abs/1501.00092

Then in later years, the researchers tried to accelerate the CNN processing by tweaking the hyper-parameter without compromising on the image resolution quality. The main idea was to lower the number of trainable parameters by tweaking the hyper-parameters. Here's the paper https://arxiv.org/abs/1608.00367

- 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

- optimization.py is correctly setup, as a suggestion, you can remove the YOUR CODE HERE when you update with the code
  get loss function is setup right
- Test cases are passed in the notebook
- Both SGD and Adam are optimized

- 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.

- Train.py is correctly set up, as a suggestion, you can remove the YOUR CODE HERE when you update with the code
  train_one_epoch is set up right and test cases are passing too.
- The scheduler is initialized and used, more on it here
- Nice work, All the test cases are also passing from the training sections

- 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)

Good job

- The hyperparameter tuning is a difficult art, your parameters are in good range and work well for the project. They can tuned further but it is okay to pass
- Losses are decreasing for each epoch

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

Test accuracy level meets the requirements of this section :)

- **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**

- Predictor.py file is wonderfully done, all test cases are passing.
- As a suggestion, remove the instruction from the file, it should not like you are following instructions
- Checkpoint is created and used, this is handy when we need to stop training and train at later date

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**

- The layers of the pre-trained models are frozen and a linear layer is added at the end. Try adding drop out to the linear layer
- Test cases are passing from all the transfer learning cases

- 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)

Since you have used ResNet in the transfer learning section, you should read some ResNet papers to understand how hyper-parameters influences the performances of the ResNet model.

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

Residual networks (ResNets) add the skip connections to form residual blocks in which the layer's activation function is added element-wise with the previous layer's input identity function. There's a trend in the area of neural networks which is "the Deeper the Better". A very deep network can use a large receptive field that takes a large image context into account. So may be use the larger ResNet model with more number of layers.

- Test accuracy is at least 60%

Test accuracy level meets the requirements of this section :)

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

Nice work in saving the pytorch model into the disk. The saved model can be used for the inference and after certain steps can be deployed to be used by other people.

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.

I would encourage you to go further steps and deploy the machine learning model on to the cloud. You can use FastAPI/Flask as a Web API that serves the client's requests and provide the predictions as a response.

You can use any cloud platforms and services like AWS lambda or Heroku to deploy your machine learning model.

DOWNLOAD PROJECT

RETURN TO PATH