Take Home Assessment – RetiSpec

Objective: This assignment aims to create a custom data loader in PyTorch to load a custom image dataset. The images are in the .tif format; each is a 4-channel image. A convolutional neural network (CNN) model is also designed to classify these images into Forest' 0' and River' 1'. Moreover, the TensorBoard provides the measurements and visualizations needed during the CNN model learning process. The TensorBoard enables tracking the CNN model metrics such as loss and accuracy, visualizing the model graph and weight histograms. The TensorBoard can be used to judge model performance and compare different models. Finally, the trained model is used to score a validation dataset that was never seen during the training.

Steps – Solution:

1- Custom Data Loader

A class "ForestRiverDataset(Dataset)" is created and extends the methods in the PyTorch Dataset class. The class's constructor has three parameters (csv_file, root_dir, transform) that will be used to locate and load the data to the PyTorch device (CUDA or CPU). The csv_file is the path to the CSV file. The CSV file contains the list of file names and the class of each file for the training and validation data. The CSV files are created by executing cells 1 to 7 of the ipynb found in this git repo (LINK). The root_dir points to the image data folder and transform is a composite transform method that allows applying a different number of transformations on the loaded image datasets. The ForestRiverDataset class has a get_item method to read and adequately normalize the tif images. The ForestRiverDataset class has a method len to return the length of the loaded image dataset. Finally, the DataLoader method from the Dataset class is used to load the custom data.

2- Baseline Convolutional Neural Network

The class ForestRiverModel(nn.Module) is created and extends the PyTorch nn.Module class to define the CNN model's layers. Figure 1 shows a graphical representation of a ForestRiverModel CNN model instance. The CNN input layer shape is (32,4,64,64) represents a PyTorch tensor of 32 images; each image is 4 channels with 64x64 resolution. The CNN model is composed of three repeated blocks. Each block is three consecutive layers (Conv2d, BatchNormalization, and Relu activation function). The output of the first block is max-pooled to reduce the resolution of the input images. Each block output channel is 2x of the previous block output channels. The output is a linear, fully connected layer that flattens all the output channels of the last block. During training, the softmax function will be used to convert the fully connected layer output into class probability. Please refer to the ipynb cell 18 to get more information about each layer's input/output shape.

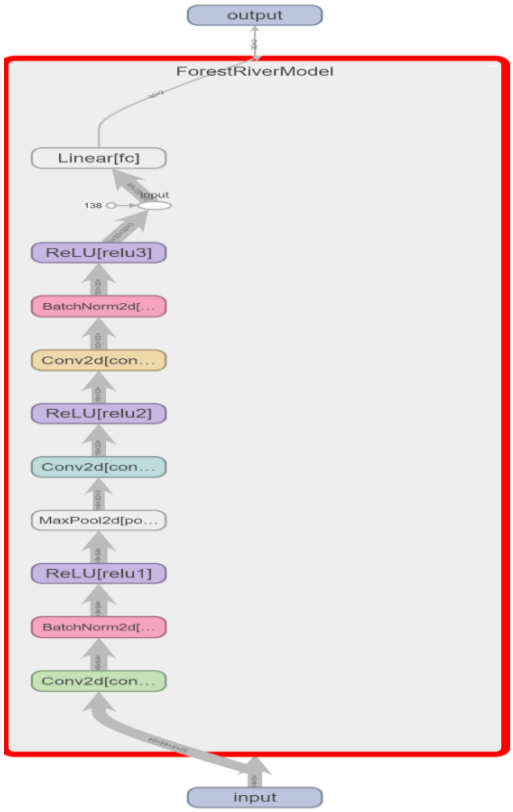

Fig.1. A graphical representation of a ForestRiverModel CNN model instance

3- Training and validation

The training loop hyperparameters are defined in cell 17. Instantiate the CNN model in cell 19 and define the loss function to be minimized with the desired optimizer in cell 20. The actual training cycle starts in cell 27. For every batch during the training phase, we want to set the gradients to zero before starting backpropagation (i.e., updating the weights and biases of every layer) because PyTorch sums up the gradients on subsequent backward passes. This accumulating behaviour is convenient while computing the loss gradient summed over multiple batches. Thus the training loop accumulates the gradients on every loss.backward() call. With every batch, the training loss, train and test accuracy are recorded, and the CNN model with the highest accuracy is saved in best_ForestRiver_Model.model. The saved model can be found on the git repo (see link above). The best model accuracy is 0.5520, slightly better than a random chance.

The validation is conducted by scoring the trained model (i.e., model trained at epoch =50). The validation code is shown in cell 28. It is worth noting that the model never saw the validation data before the validation scoring. The validation accuracy is 0.4501.

4- TensorBoard

The TensorBoard logs information about the model, model parameters and training performance.

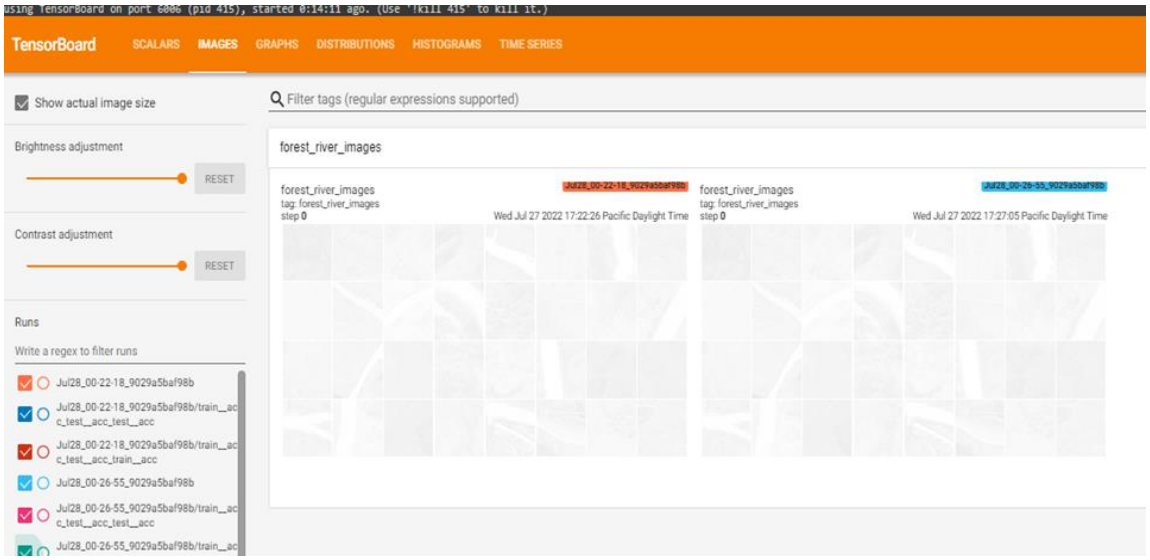

Fig.2. Two batch of random images collected from the train data loader

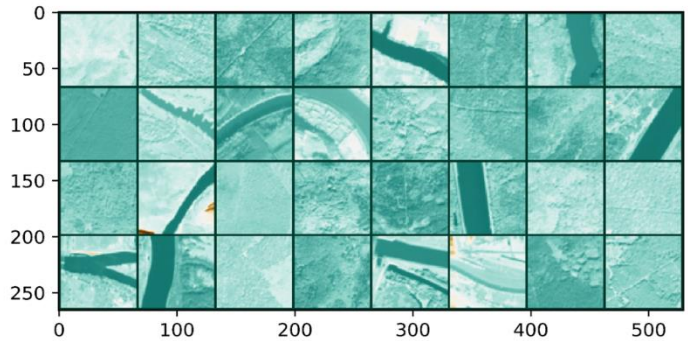

Fig.3. One batch of random images collected from the train data loader with adjusted color map for visualization

Figure 2 shows two random batched collected from the train data loader. Using the SummaryWriter class, we can log these batches by multiple executions of writer.add_image as illustrated in cell 24. Figure 3 shows one set of batched collected from the train data loader with proper colour mapping for visualization purposes.

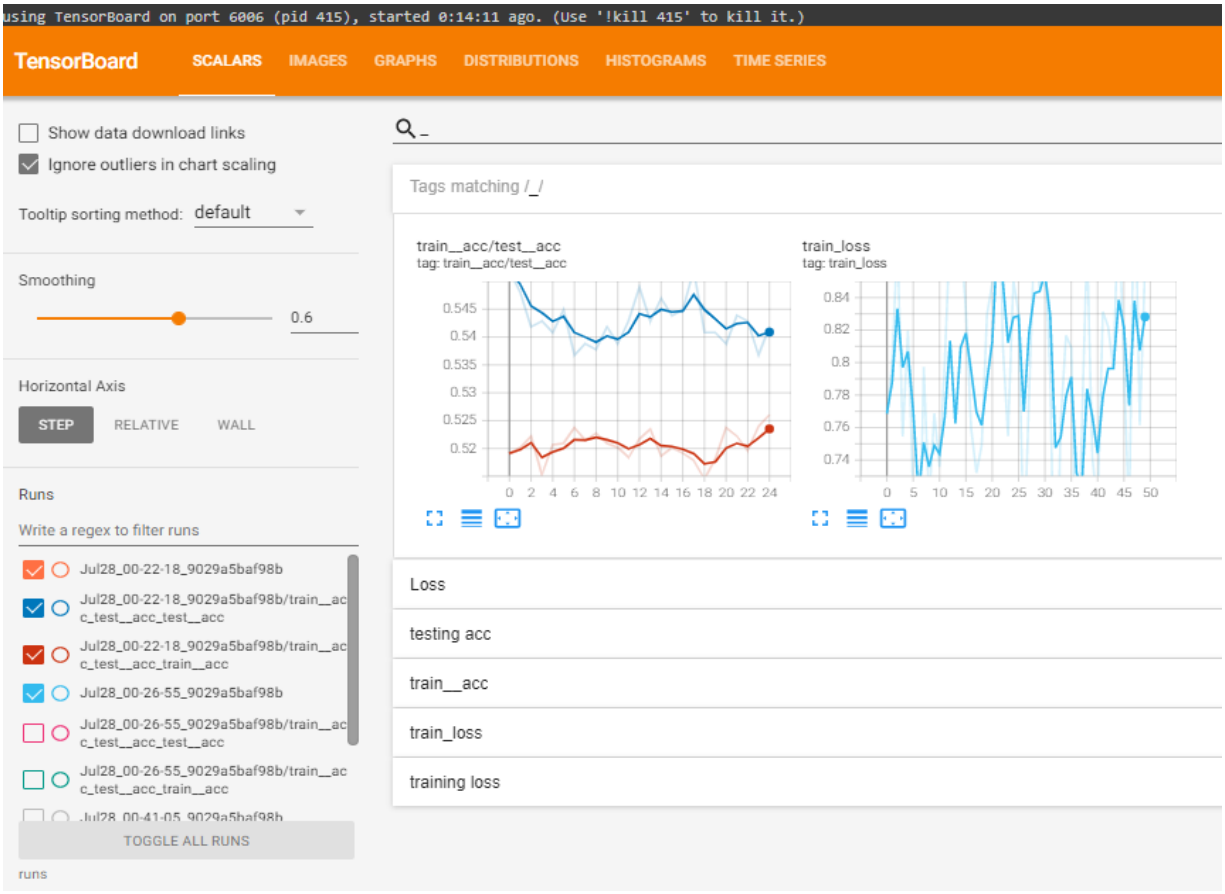

Fig.4. Visualization of the train loss, train and testing accuracy

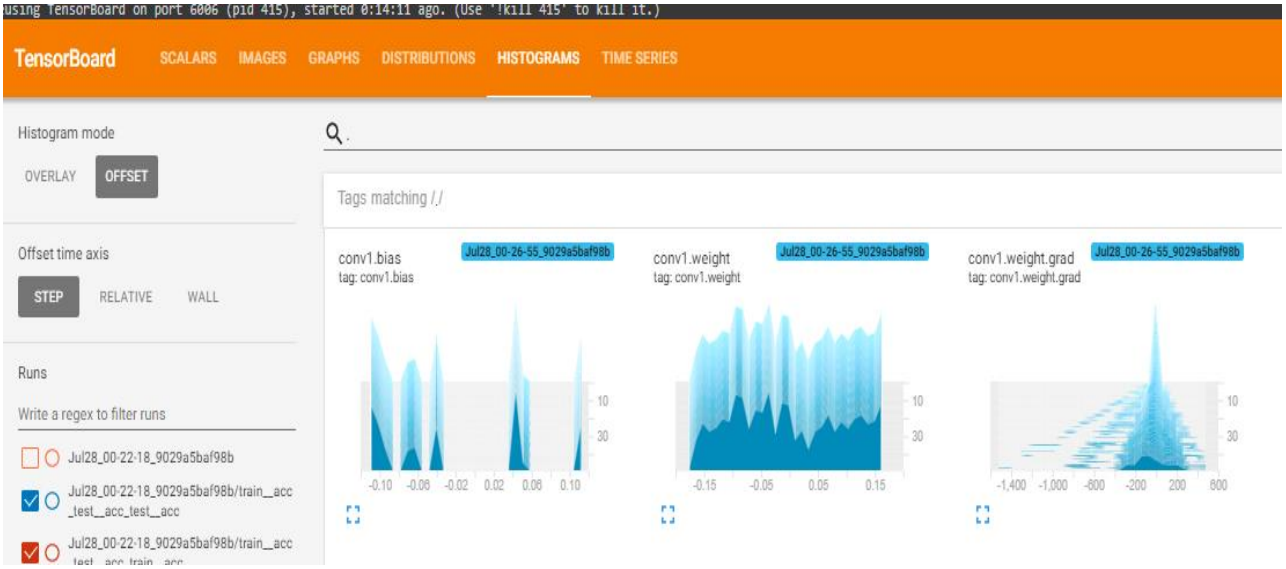

Fig.5. Visualization of the train loss, train and testing accuracy

Figure 4 shows different training information, including one two-trace plot of the training and testing accuracy and the training loss. Figure 5 shows the distribution (normalized histogram) of the first layer (i.e., Conv1 bias, weight, and weight grade per epoch) of the CNN model as described by the code snippet.

```
# visualize the weight distribution in some layers closer to the input #images.
# It helps in understanding the effecting weight/layer initialization method

    writer.add_scalar('training loss',train_loss,epoch)# * len(train_loader) + i)
    writer.add_histogram('conv1.bias', model.conv1.bias, epoch)
    writer.add_histogram('conv1.weight', model.conv1.weight, epoch)
    writer.add_histogram('conv1.weight.grad',model.conv1.weight.grad,epoch)
```

5- Results, conclusions, and future works

The results show that the supplied notebook (see link above) has achieved the assignment requirements by creating a custom data loader, training, validating a CNN model, saving the best accuracy model and using the TensorBoard to examine the CNN model performance. However, the designed model is a base model that was not tuned for better performance, and thus it has a slightly better than random chance accuracy during training (0.5520).

Future work will address the limitation of this training to obtain better classification accuracy. Better model accuracy can be applying several techniques, including:

1- Data augmentation: is the method used to increase the volume of data by adding slightly modified copies of already existing data. It regularizes and helps reduce overfitting when training a CNN model.
2- Hyperparameters optimization: finding the best combination of hyperparameter values to achieve maximum performance on the data in a reasonable amount of time using grid or random search methods.
3- Use the learning rate scheduler: adjust the learning rate during training by reducing the learning rate according to a pre-defined schedule. The learning rate affects how quickly our model can converge to local minima (i.e., best accuracy).
4- Early stopping: Early stopping is a method that allows specifying an arbitrarily large number of training epochs and stops training once the model performance stops improving on the validation dataset.
5- Weight decay: Weight decay adds a penalty term to the loss function of a CNN model, which shrinks the weights during backpropagation. This prevents the CNN model overfitting the training data and the exploding gradient problem.
6- Use pre-trained models (transfer learning): previously trained CNN models on a large dataset, typically on a large-scale image-classification task. We can either use the pre-trained model or transfer learning (unfreez and tweek the last few layers) to customize this model to a given task.
7- Ensemble learning: combines the predictions from multiple CNN models to reduce the variance of predictions and reduce generalization error. Techniques for ensemble learning can be grouped by the varied element, such as training data, the model, and how predictions are combined.