ResNet CNN Pipeline Documentation

- This is documentation/writeup for the `ResNet.py`, `ActiveTrainer.py`, `ActiveTester.py`, `Data.py`, `DataHelper.py` files.
- The pipeline(**is still being tweaked**) consists of the `ActiveTrainer.py` training a ResNet CNN model(running multiple tests for differing experiments) and then using the code snippet in `ActiveTester.py` to run files
- I sort of glossed over much of the pipeline that has to do with saving results in between training(I used `tag` and `experiment` variables to make tests easier) **since it will most likely change when added to the transformer pipeline.**

> **Disclaimer**
> I think this ResNet code has been worked on for the past 3 years(Im not certain of the exact origin of it). I didnt write the code in the `ResNet.py` file, but I wrote most of the code in the `ActiveTrainer.py` file(while using some snippets online for multiprocessing)
>
> Sorry if this is overexplained or too in-depth, its because im using this as a chance to recap/collect my thoughts :)

- Important Sections:
  - Explaining ResNet.py Class (you can skip the breakdown)
  - Explaining Training Setup
  Table of Contents`

ResNet Theory

- The core motivation for ResNets were simple: as a model gets larger, it looses out on the input signal(since the input values get transformed so much).
  - This causes the error of networks to stay stagnant or increase as epochs increase
- ResNet models has the layers learning **residual functions** between the output and input values
- Residual functions are the "difference" between input and output.
- Simply put:
  - ResNets are split into residual blocks, where instead of having a feed forward network, ResNets have the output of each layer added to the input of the layer steps ahead(residual connections)
- The Model can learn the residual mapping, which means that it "carries" the context/information of inputs and adds them to outputs.

File: ResNet.py

- Starting from the first used class, our ActiveTrainer file calls the ResnetConfig class.

Class: ResnetConfig (L:146)

> **TLDR**
>
> Config class for resnet model, only important attributes are input_dim(1) and output_dim (number of classes),
> Object is later passed into ResNet Model.

Within our `ActiveTrainer.py` file, we initlize our models using the code below.

```
def init_models(gpu, num_classes):
    config = ResnetConfig(
        input_dim = 1,
        output_dim = num_classes,
        res_dims=[32, 64, 64, 64],
        res_kernel=[5, 7, 17, 13],
        res_stride=[4, 4, 5, 3],
        num_blocks=[2, 2, 2, 2],
        first_kernel_size = 13,
```

```
        first_stride = 1,
        first_pool_kernel_size = 7,
        first_pool_stride = 7,

    )
    learning_rate = 0.5e-3
    model = ResnetClassifier(config)
    opt_SD=torch.optim.Adam(model.parameters(), lr=learning_rate, eps=1e-5)
    return model, opt_SD, learning_rate
```

- The two most important parameters are the input_dim and output_dim. The input_dim should be one since we are just throwing in our areas
- The **ResNet config class acts as a object that just stores all of the configuration parameters** for the later classes, if you want you can play around with the resent parameters.

Class: ResnetClassifier (L:236)

TLDR

ResnetClassifer is our model file.
Constructor takes in previous config object and number of classes
Forward method takes in inputs, with the option of taking in labels.
(if needed in the future, you can also give it its own loss function and turn it into an unsupervised model)
Forward method returns `ModelOutput` object which computes top predictions and accuracy.

- Constructor Class(L:237)
  - The default of the construer method just setups the classifier with default parameters, we instead throw our own classifier in there
  - Default objets includes:
    - `self.resnet` object: 1D
    - `self.classifer` object: Linear layer for classification, which takes the output of resnet model (which whatever size the final layer is set to) and outputs sizes of our output dimension.
    - `self.adv` object: Linear layer for unsupervised learning. takes output of resnet model and squeezes it to 1D array.

- Forward Method(L:251)
  - Takes inputs into ResNet model, applies a average pool to the output of the resnet, and then throws the output into the classifier object.
  - From here, the supervised logits(s_logits) and unsupervised logits(u_logits) are computed
  - Since we are using supervised learning, we use a cross-entropy loss function(`loss_fct`) to compute our loss
  - The forward method(most of the time) returns a `ModelOutput` object which is detailed below

Class: ModelOutput(L8)

- Constructor takes in logits and loss, computes the predictions(by taking the index of the highest value)
- You interact wi
- Method: `accuracy()` : takes label tensor and computes the accuracy of the model
- Method `top_k_preds()` : Takes the top `k` predictions(top `k` indices with highest probabilities)
- Method `top_k_acc()` : Computes top `k` accuracy

Breakdown of Actual ResNet Architecture

- Class **CNN1dLayer**: Normal CNN Layer with dropout and GELU activation
  - 1D Convolutional Layer
  - Batch Normalization
  - GELU
  - Dropout
- Class **CNN1dlayerNoAct**: Normal CNN Layer with dropout with no activation
  - 1D Convolutional Layer
  - Batch Normalization
  - Dropout
- **self.downsample**: Down sampling layer
  - If input and output dimensions doesn't match, a 1x1 convolution helps match shape of residual with output from CNN blocks
  - Allows for the residual + hidden output
- Class Resnet1DBlock:
  - Residual Path: If downsampling is needed, sends data through self.downsample, else is a straight path to output
  - Hidden Path: CNN1dLayer -> CNN1dlayerNoAct

    - Adds residual + hidden
    - Applies GELU activation
- Class: Resnet
    - Constructor sets up each layer
    - Method: `_make_resnet_layer` :
        - creates a resnet layer with `num_blocks` residual blocks
        - first residual block uses downsampling to go from `prev_dim` to `dim`
        - rest of the residual blocks have the same input and output dimension
    - Forward Pass:
        - Initial Convolutional Layer
        - Maxpool
        - 4 Residual Layers
            - `layer1` takes input channels = `config.res_dims[0]` and outputs `config.res_dims[0]`
            - `layer2` takes input channels = `config.res_dims[0]` and outputs `config.res_dims[1]`
            - `layer3` takes input channels = `config.res_dims[1]` and outputs `config.res_dims[2]`
            - `layer4` takes input channels = `config.res_dims[2]` and outputs `config.res_dims[3]`
-

File: ActiveTrainer.py

- lll quickly summarize the more less important parts of this file below, the only methods of interest are most likely `init_process()` , `init_models()` , and `train()`
- Main
    - my pipeline splits training into tags and experiments for large scale testing, its not too important but it effects the save paths
    - Spawns processes amongst 5 gpus, using a certain amount of epochs, batch_size, background, etc

Method: train

- Parameters:
    - gpu(mp purposes)
    - epochs
    - world_size(mp purposes
    - batch_size

    - background
    - tensors: array of testing and validation tensors
    - exper (pipeline purposes)
    - tag (pipeline purposes)
    - num_classes
- **Since its using multiprocessing, you will see a lot of `if gpu =0`, all evaluation is done on gpu `0`**
  Rundown:
- Calls `init_process()` and `init_models()` methods to setup model and optimizer files
- Sets up distributed data parallel processing (ik fancy words)
- Loads data using `DiffractionDataset` class(described under `Data.py`) file and sets up data loader files
- Training:
    - Uses CrossEntropy Loss Function
    - Initializes arrays that store weighted average of train accuracy, top 1 test accuracy, and top 3 test accuracy
    - During each epoch
        - Computes loss and updates gradients for each batch
        - Computes train accuracy(`t1_train_acc`) for each batch
        - Saves weighted average for that epoch (`t1_train_acc /= train_examples`)
        - Computes top1 test accuracy (`top1_test_acc`) and top 3 test accuracy (`top3_test_acc`) for each batch
        - Saves weighted average for that epoch (`top1_test_acc/=total_samples`)
        - Logs files in console: `L.log("[Epoch %d/%d] [Train Acc: %d%%] [T1 Test Acc: %d%%] [Top 3 acc: %d%%]\n"% (epoch+1, epochs, (100 * t1_train_acc), (100*top1_test_acc), (100*top3_test_acc)))`
        - Saves current state of model and optimizer in `model_path` file
    - **Saves an tensor with the train accuracy, top 1 test accuracy, and top 3 test accuracy for each epoch(for post-training analysis) under `train-t1_test-t3-test.pt`**

Method: init_process

- Used for MP purposes

Method: init_models

- intializes model(REGARDLESS OF WHICH MP GPU IS BEING USED) using bellow model architecture and hyperparameters:

```
config = ResnetConfig(
    input_dim = 1,
    output_dim = num_classes,
    res_dims=[32, 64, 64, 64],
    res_kernel=[5, 7, 17, 13],
    res_stride=[4, 4, 5, 3],
    num_blocks=[2, 2, 2, 2],
    first_kernel_size = 13,
    first_stride = 1,
    first_pool_kernel_size = 7,
    first_pool_stride = 7,
)
learning_rate = 0.5e-3
model = ResnetClassifier(config)
opt_SD=torch.optim.Adam(model.parameters(), lr=learning_rate, eps=1e-5)
```

File: ActiveTester.py

- Method: return_analystics
  - input params:
    - model_path: path of dictionary with model that was saved during training
    - training_analytics_path: path of tensor that was saved during training
    - num_classes: number of classses
  - Output:
    - epochs,
    - train_accuracy_array
    - top1_validation_acc_array
    - top3_validation_acc_array
    - top1_test_accuracy
    - top3_test_accuracy
    - top1_predictions
    - top3_predictions

File: Data.py

- Data includes a child class of the pytorch dataset class
- The `DiffrationDataset` class constructor takes:
  - num_classes: number of classes
  - background: uniform background added to each pattern
  - data: our data tensor

- labels: our label tensor
- cat: Either "Bravais Lattice" or "Space Group", which is later used for a mapping

File: DataHelper.py

- Includes helper function `createTrainAndValidation` which intakes
  - path: path to training tensor
  - validationSize: % amount of training tensor that will be used for validation
  - savePath: path to directory where data tensors are saved
  - tag: used for saving purposes
- Only bit of important code imo:

```
dTrainPath = os.path.join(savePath, tag + '_Train.pt')
dValdPath = os.path.join(savePath, tag + '_Validation.pt')
torch.save({'X': data_train_tensor, 'Y': labels_train_tensor}, dTrainPath)
torch.save({'X': data_test_tensor, 'Y': labels_test_tensor}, dValdPath)
```