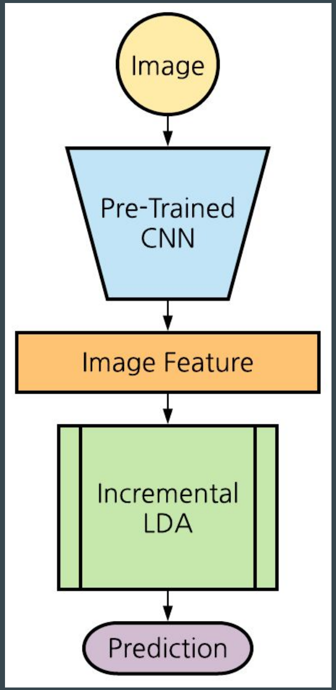
Module 3: PyTorch Demo for Online Streaming Image Classification

Tyler Hayes
PhD Candidate, Rochester Institute of Technology
https://tyler-hayes.github.io/

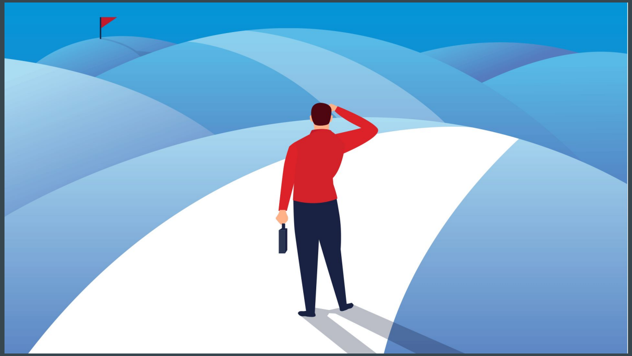
Recap of Deep Streaming Linear Discriminant Analysis (SLDA)

1. Extract image feature from **pre-trained deep CNN**
2. Update **class-specific running mean vector** and running **shared covariance matrix** among classes
3. During inference, a prediction is made by assigning the label of the **closest Gaussian in feature space** defined by the class mean vectors and covariance matrix



Goals of Tutorial

1. Load the CORe50 training/testing datasets and scenarios using Avalanche
2. Define the Deep SLDA model and ResNet-18 backbone network
3. Train and evaluate the network on each incremental batch
4. Save results out
5. Plot incremental accuracies



Follow along! https://github.com/tyler-hayes/SLDA-Tutorial

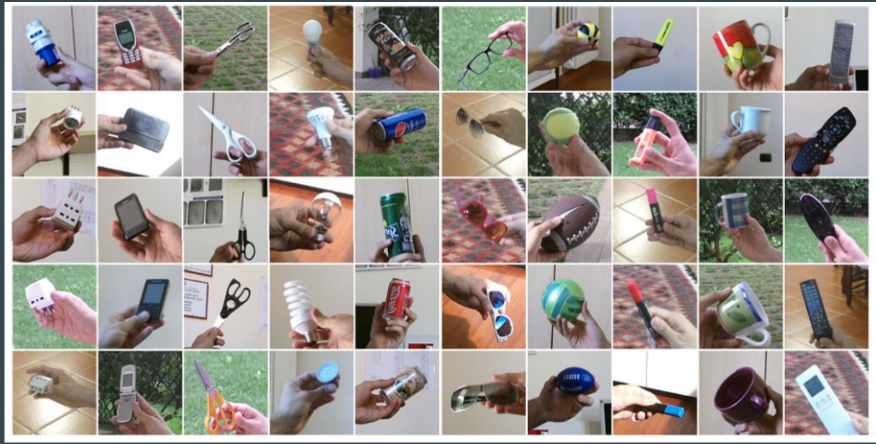
What is Avalanche?

- Avalanche is an **End-to-End Continual Learning Library** based on **PyTorch**
  - https://avalanche.continualai.org/
- Used for fast prototyping, training, and reproducible evaluation of continual learning algorithms
- Contains popular continual learning benchmarks
  - We are going to use it here to gather the **CORe50** training scenarios

Avalanche *powered by* ContinualAI

CORe50 Scenarios

- **New Instances (NI):** new training patterns of the same classes become available in subsequent batches with new poses and conditions (illumination, background, occlusion, ...)
- **New Classes (NC):** new training patterns belonging to different classes become available in subsequent batches
- **New Instances and Classes (NIC):** new training patterns belonging both to known and new classes become available in subsequent training batches



Lomonaco, Vincenzo, and Davide Maltoni. "Core50: a new dataset and benchmark for continuous object recognition." CoRL, 2017.

Loading CORe50 Incremental Batches Based on Scenario

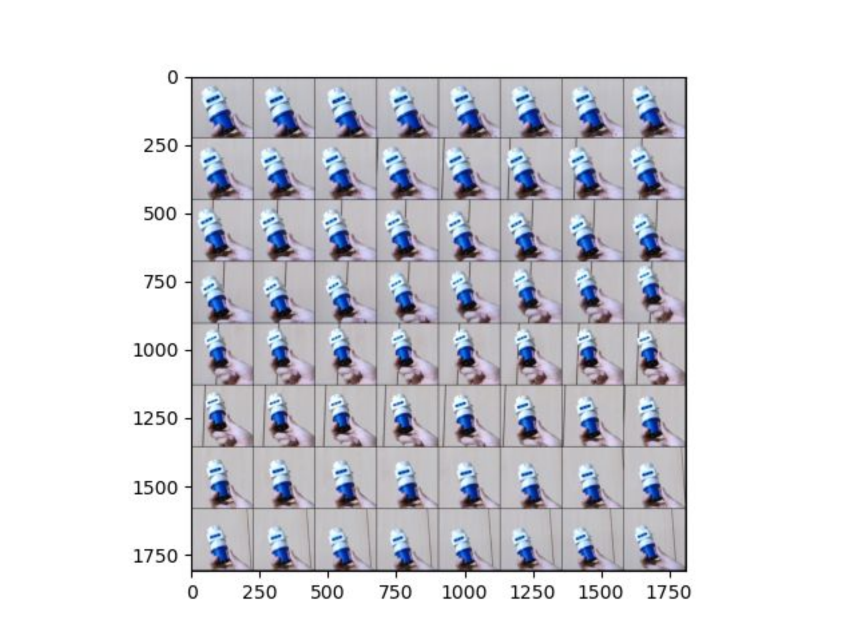
- Given one of the CORe50 training scenarios (NI, NC, NIC), we can define a scenario object using Avalanche
- The scenario object will return a new PyTorch dataset for each incremental batch
- Remember to normalize the data using ImageNet mean and standard deviation statistics

```
def get_data(args):
    # Create the dataset scenario object
    _mu = [0.485, 0.456, 0.406]  # imagenet normalization
    _std = [0.229, 0.224, 0.225]
    t = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=_mu,
                             std=_std)
    ])
    scenario = CORe50(root=args.dataset_dir, scenario=args.scenario,
                      train_transform=t, eval_transform=t)
    return scenario
```

Loading and Viewing CORe50 Data

```
def show_sample_images(dataset_dir, scenario):
    t = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor()
    ])
    scenario = CORe50(root=dataset_dir, scenario=scenario, train_transform=t,
                      eval_transform=t)
    loader = DataLoader(scenario.train_dataset, batch_size=64, shuffle=False,
                        num_workers=8)

    def imshow(img):
        npimg = img.numpy()
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()

    # get some random training images
    dataiter = iter(loader)
    images, labels, _ = dataiter.next()

    # show images
    imshow(torchvision.utils.make_grid(images))
```

Output:



Define the Pre-Trained ResNet-18 Backbone as a Feature Extractor

- We define a ResNet-18 model in PyTorch and load in ImageNet pre-trained weights
- We can then register hooks to extract features from a particular layer of the network
- For this model, we would like to extract features from the penultimate layer

```
def get_feature_extraction_model(arch, imagenet_pretrained, feature_size,
                                 num_classes):
    feature_extraction_model = models.__dict__[arch](
        pretrained=imagenet_pretrained)
    feature_extraction_model.fc = nn.Linear(feature_size, num_classes)
    return feature_extraction_model
```

```
feature_extraction_wrapper = retrieve_any_layer.ModelWrapper(
    feature_extraction_model.eval().cuda(),
    ['layer4.1'], return_single=True).eval()
```

Define the Deep SLDA Model for Online Classification

- We define Deep SLDA as a PyTorch Module
- The required parameters are: the shape of the input vectors, the total number of classes, a test batch size, a shrinkage parameter, and a boolean indicating if the covariance matrix should be plastic
- We initialize the class means to zeros and the covariance matrix to ones

```
class StreamingLDA(nn.Module):
    """
    This is an implementation of the Deep Streaming Linear Discriminant
    Analysis algorithm for streaming learning.
    """

    def __init__(self, input_shape, num_classes, test_batch_size=1024,
                 shrinkage_param=1e-4, streaming_update_sigma=True):
        """
        Init function for the SLDA model.
        :param input_shape: feature dimension
        :param num_classes: number of total classes in stream
        :param test_batch_size: batch size for inference
        :param shrinkage_param: value of the shrinkage parameter
        :param streaming_update_sigma: True if sigma is plastic else False
        """

        super(StreamingLDA, self).__init__()

        # SLDA parameters
        self.device = 'cuda'
        self.input_shape = input_shape
        self.num_classes = num_classes
        self.test_batch_size = test_batch_size
        self.shrinkage_param = shrinkage_param
        self.streaming_update_sigma = streaming_update_sigma

        # setup weights for SLDA
        self.muK = torch.zeros((num_classes, input_shape)).to(self.device)
        self.cK = torch.zeros(num_classes).to(self.device)
        self.Sigma = torch.ones((input_shape, input_shape)).to(self.device)
        self.num_updates = 0
        self.Lambda = torch.zeros_like(self.Sigma).to(self.device)
        self.prev_num_updates = -1
```

Define the Deep SLDA Model for Online Classification

- To fit the model on a single new sample, we use a running mean and covariance update

$$\Delta_t = \frac{t\left(\mathbf{z}_t - \boldsymbol{\mu}_{(k=y,t)}\right)\left(\mathbf{z}_t - \boldsymbol{\mu}_{(k=y,t)}\right)^T}{t+1}$$

Covariance Update

$$\boldsymbol{\Sigma}_{t+1} = \frac{t\boldsymbol{\Sigma}_t + \Delta_t}{t+1}$$

Mean Update

$$\boldsymbol{\mu}_{(k=y,t+1)} \leftarrow \frac{c_{(k=y,t)}\boldsymbol{\mu}_{(k=y,t)} + \mathbf{z}_t}{c_{(k=y,t)} + 1}$$

```python
def fit(self, x, y):
    """
    Fit the SLDA model to a new sample (x,y).
    :param x: a torch tensor of the input data (must be a vector)
    :param y: a torch tensor of the input label
    :return: None
    """
    x = x.to(self.device)
    y = y.long().to(self.device)

    # make sure things are the right shape
    if len(x.shape) < 2:
        x = x.unsqueeze(0)
    if len(y.shape) == 0:
        y = y.unsqueeze(0)

    with torch.no_grad():

        # covariance updates
        if self.streaming_update_sigma:
            x_minus_mu = (x - self.muK[y])
            mult = torch.matmul(x_minus_mu.transpose(1, 0), x_minus_mu)
            delta = mult * self.num_updates / (self.num_updates + 1)
            self.Sigma = (self.num_updates * self.Sigma + delta) / (
                        self.num_updates + 1)

        # update class means
        self.muK[y, :] += (x - self.muK[y, :]) / (self.cK[y] + 1).unsqueeze(
            1)
        self.cK[y] += 1
        self.num_updates += 1
```

Define the Deep SLDA Model for Online Classification

- To make predictions, we assign the label of the closest Gaussian in feature space
- First, compute precision matrix Λ
- Then compute weight and bias term for each class
- Prediction = Wz+b

$$\mathbf{\Lambda} = [(1-\varepsilon)\,\mathbf{\Sigma} + \varepsilon \mathbf{I}]^{-1}$$

$$\mathbf{w}_k = \mathbf{\Lambda}\boldsymbol{\mu}_k$$

$$b_k = -\frac{1}{2}\left(\boldsymbol{\mu}_k \cdot \mathbf{\Lambda}\boldsymbol{\mu}_k\right)$$

$$\mathbf{W}\mathbf{z}_t + \mathbf{b}$$

```python
def predict(self, X, return_probas=False):
    """
    Make predictions on test data X.
    :param X: a torch tensor that contains N data samples (N x d)
    :param return_probas: True if the user would like probabilities instead
    of predictions returned
    :return: the test predictions or probabilities
    """
    X = X.to(self.device)

    with torch.no_grad():
        # initialize parameters for testing
        num_samples = X.shape[0]
        scores = torch.empty((num_samples, self.num_classes))
        mb = min(self.test_batch_size, num_samples)

        # compute/load Lambda matrix
        if self.prev_num_updates != self.num_updates:
            # there have been updates to the model, compute Lambda
            Lambda = torch.pinverse(
                (
                        1 - self.shrinkage_param) * self.Sigma +
                self.shrinkage_param * torch.eye(
                    self.input_shape).to(
                    self.device))
            self.Lambda = Lambda
            self.prev_num_updates = self.num_updates
        else:
            Lambda = self.Lambda

        # parameters for predictions
        M = self.muK.transpose(1, 0)
        W = torch.matmul(Lambda, M)
        c = 0.5 * torch.sum(M * W, dim=0)

        # loop in mini-batches over test samples
        for i in range(0, num_samples, mb):
            start = min(i, num_samples - mb)
            end = i + mb
            x = X[start:end]
            scores[start:end, :] = torch.matmul(x, W) - c

        # return predictions or probabilities
        if not return_probas:
            return scores.cpu()
        else:
            return torch.softmax(scores, dim=1).cpu()
```

Training SLDA on a New “Batch” of Data

- For each data batch defined by the CORe50 training scenarios:
  - Extract the ResNet-18 pre-trained features
  - Then train SLDA one sample at a time
- We can track extra statistics about the model like how much RAM and external memory it uses in MB

```
def train(model, feature_extraction_wrapper, train_loader):
    print('\nTraining on %d images.' % len(train_loader.dataset))

    stats = {"ram": [], "disk": []}
    stats['disk'].append(check_ext_mem("cl_ext_mem"))
    stats['ram'].append(check_ram_usage())

    for train_x, train_y, _ in tqdm(train_loader, total=len(train_loader)):
        batch_x_feat = feature_extraction_wrapper(train_x.cuda())
        batch_x_feat = pool_feat(batch_x_feat)

        # train one sample at a time
        for x_pt, y_pt in zip(batch_x_feat, train_y):
            model.fit(x_pt.cpu(), y_pt.view(1, ))

    return stats
```

Testing SLDA After Each New "Batch" of Data

- For batches of test data:
  - Extract the ResNet-18 pre-trained features
  - Compute predictions using SLDA model
- We can track the model's accuracy on the entire test set over time

```
def evaluate(model, feature_extraction_wrapper, test_loader):
    print('\nEvaluating on %d images.' % len(test_loader.dataset))

    preds = []
    correct = 0

    for it, (test_x, test_y, _) in tqdm(enumerate(test_loader),
                                         total=len(test_loader)):
        batch_x_feat = feature_extraction_wrapper(test_x.cuda())
        batch_x_feat = pool_feat(batch_x_feat)

        logits = model.predict(batch_x_feat, return_probas=True)

        _, pred_label = torch.max(logits, 1)
        correct += (pred_label == test_y).sum()
        preds += list(pred_label.numpy())

    acc = correct.item() / len(test_loader.dataset)
    return acc, preds
```

Training/Evaluation Setup

- Start timing the experiment
- Define scenario object and test data loader
- Define feature extraction model and Deep SLDA model

```
def main(args):
    # start timing experiment
    start = time.time()

    scenario = get_data(args)
    test_loader = DataLoader(scenario.test_dataset, batch_size=args.batch_size,
                             shuffle=False, num_workers=8)

    # create SLDA model
    model = StreamingLDA(args.feature_size, args.n_classes,
                         test_batch_size=args.batch_size,
                         shrinkage_param=args.shrinkage,
                         streaming_update_sigma=args.plastic_cov)

    # create feature extraction model pre-trained on imagenet
    feature_extraction_model = get_feature_extraction_model(arch=args.arch,
                                                            imagenet_pretrained=True,
                                                            feature_size=args.feature_size,
                                                            num_classes=args.n_classes)
    # layer 4.1 is the final layer in resnet18 (need to change this code
    # for other architectures)
    feature_extraction_wrapper = retrieve_any_layer.ModelWrapper(
        feature_extraction_model.eval().cuda(),
        ['layer4.1'], return_single=True).eval()
```

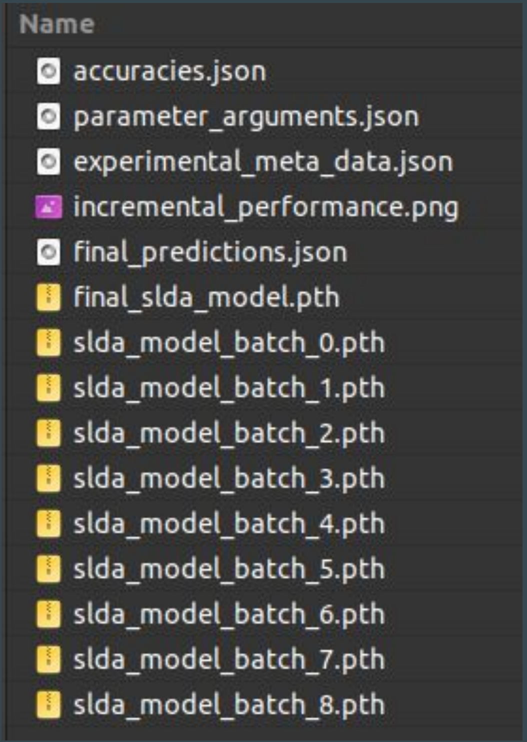
Training/Evaluation Loop

- Define variables to track model statistics over time
- Loop over the training scenario batches and fit SLDA model one sample at a time
- After training on an entire batch, evaluate the model on all test data
- Update model statistics and save them out to files

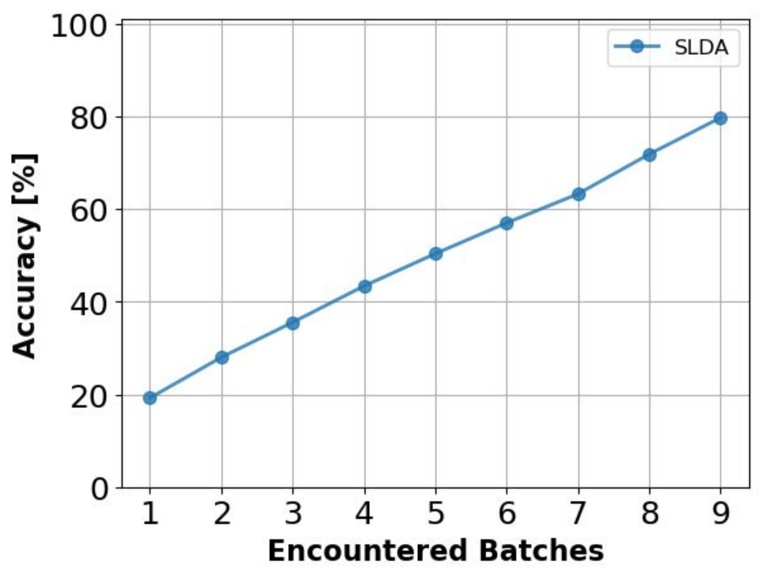
```python
# variables to update over time
test_acc = []
ext_mem_sz = []
ram_usage = []

# loop over the training incremental batches
for i, batch in enumerate(scenario.train_stream):
    print("\n---------- Batch {0}/{1} -------------".format(i + 1, len(
        scenario.train_stream)))
    train_loader = DataLoader(batch.dataset, batch_size=args.batch_size,
                              shuffle=False, num_workers=8)

    # fit SLDA model to batch (one sample at a time)
    stats = train(model, feature_extraction_wrapper, train_loader)

    # save SLDA model weights after current batch has been fit
    model.save_model(args.save_dir, 'slda_model_batch_%d' % i)

    # evaluate model on test data
    acc, preds = evaluate(model, feature_extraction_wrapper, test_loader)

    print("------------------------------------------")
    print("Test Accuracy: %0.3f" % acc)
    print("------------------------------------------")

    # update stats
    test_acc += [acc]
    ext_mem_sz += stats['disk']
    ram_usage += stats['ram']

    # update test accuracy list
    save_accuracies(test_acc, args.save_dir)

# total elapsed time
elapsed = (time.time() - start) / 60
print("Total Experiment Time: %0.2f minutes" % elapsed)

# save experimental results
save_experimental_results(args.save_dir, model, test_acc, elapsed,
                          ram_usage, ext_mem_sz, preds)

# plot final results
plot_results(test_acc, 'incremental_performance', args.save_dir)
```

Saved Files From Experiment

- JSON file containing model accuracies on the entire test set after every training increment
- JSON file containing parameter settings for experiment
- JSON file containing additional experiment statistics such as RAM and external memory usage
- Plot of incremental performance (can be made from accuracies.json)
- JSON file containing final model predictions
- Checkpoints for the SLDA model after training on each scenario batch
- Checkpoint for the final SLDA model



Live Code Demonstration (Requires ~6 Minutes for Entire Dataset)



SLDA performance on CORe50 test set (44,972 images) after each training session

Wrap-Up

- We demonstrated how to use **Avalanche** to load CORe50 training scenarios, setup a pre-trained feature extraction neural network, and setup the Deep SLDA model
- We showed how all of these parts of code can be put together to perform **online continual learning for image classification**
- Deep SLDA achieves high performance, while running much faster and using significantly less memory than competing models
  - **100x faster and 1000x less memory** than previous models on ImageNet
- Our code is available on GitHub and can be easily extended to other datasets using Avalanche
  - https://github.com/tyler-hayes/SLDA-Tutorial

Thank You!

Questions?

Tyler Hayes
tlh6792@rit.edu
https://tyler-hayes.github.io

- We demonstrated how SLDA can be implemented in PyTorch on CORe50 for streaming image classification
- SLDA is a simple baseline that requires little compute and memory
- Our code can be easily extended to additional datasets and feature extraction models