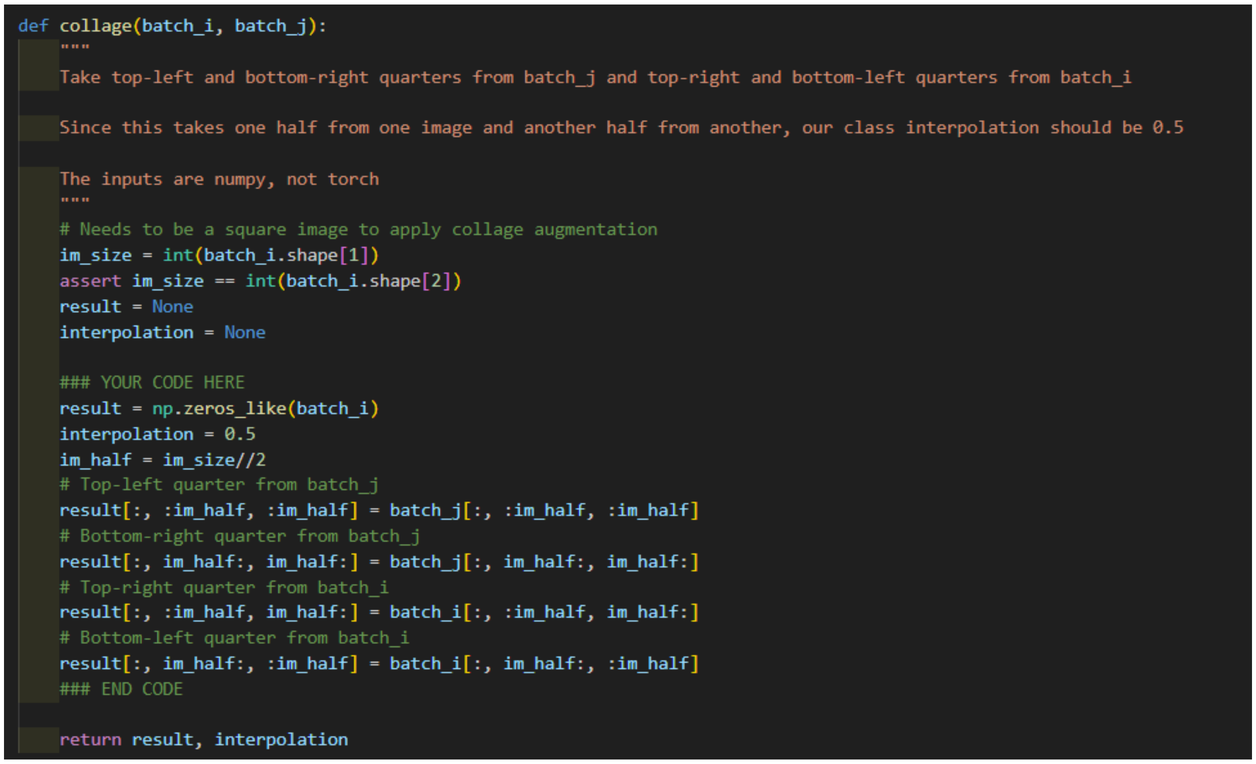
Machine Learning Hand In 3

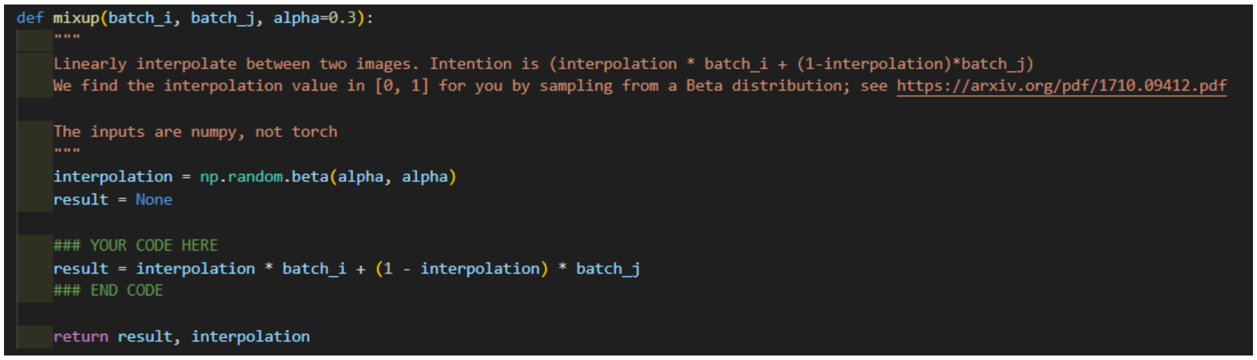
Group 41
Tan Wei Yuan (Student ID: 202300860)
Yu Juan Ang (Student ID: 202300862)
Alton ZhiXian Cheah (Student ID: 202300865)

Code

augmentations.py

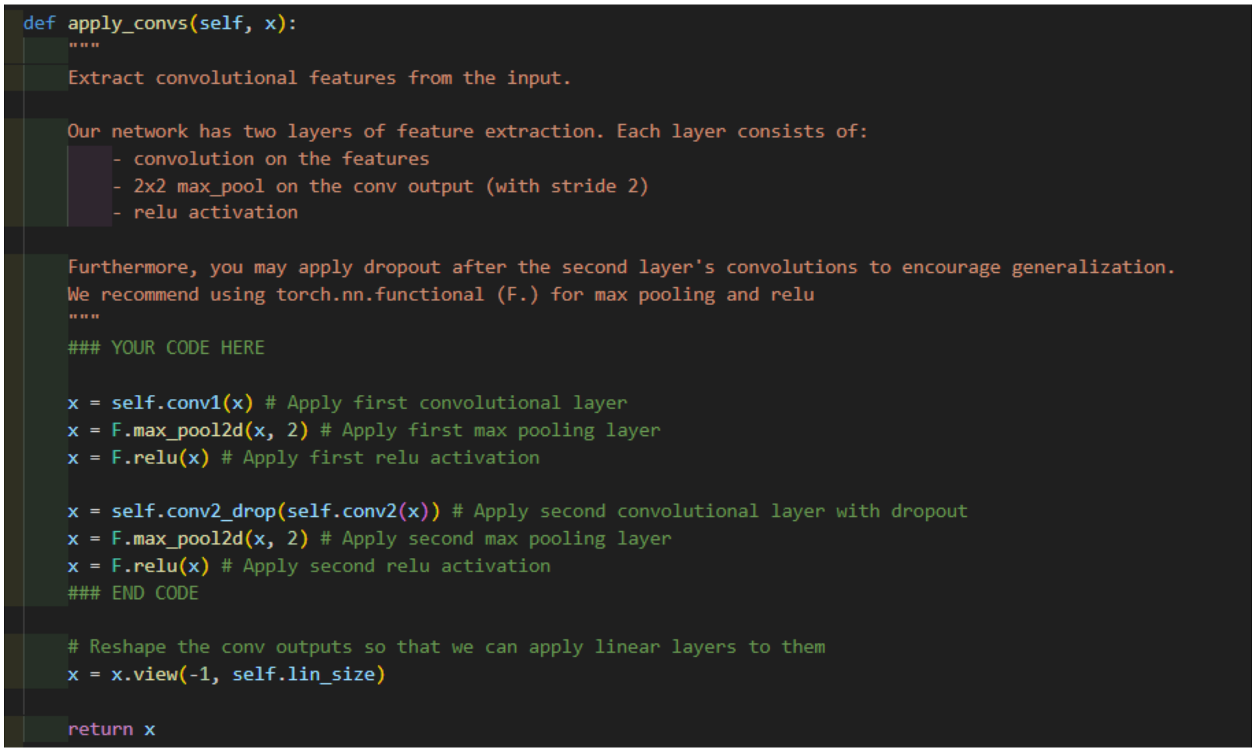


```python
def collage(batch_i, batch_j):
    """
    Take top-left and bottom-right quarters from batch_j and top-right and bottom-left quarters from batch_i

    Since this takes one half from one image and another half from another, our class interpolation should be 0.5

    The inputs are numpy, not torch
    """
    # Needs to be a square image to apply collage augmentation
    im_size = int(batch_i.shape[1])
    assert im_size == int(batch_i.shape[2])
    result = None
    interpolation = None

    ### YOUR CODE HERE
    result = np.zeros_like(batch_i)
    interpolation = 0.5
    im_half = im_size//2
    # Top-left quarter from batch_j
    result[:, :im_half, :im_half] = batch_j[:, :im_half, :im_half]
    # Bottom-right quarter from batch_j
    result[:, im_half:, im_half:] = batch_j[:, im_half:, im_half:]
    # Top-right quarter from batch_i
    result[:, :im_half, im_half:] = batch_i[:, :im_half, im_half:]
    # Bottom-left quarter from batch_i
    result[:, im_half:, :im_half] = batch_i[:, im_half:, :im_half]
    ### END CODE

    return result, interpolation
```

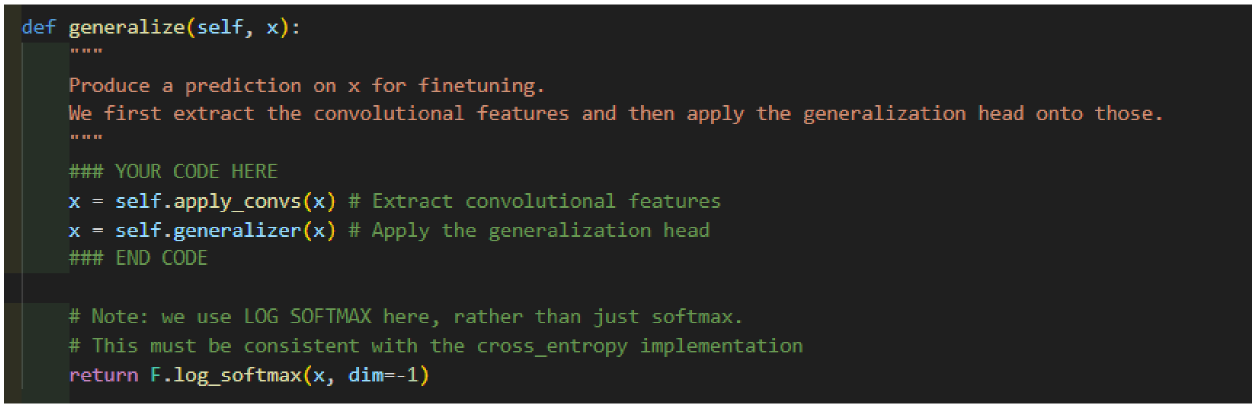


```python
def mixup(batch_i, batch_j, alpha=0.3):
    """
    Linearly interpolate between two images. Intention is (interpolation * batch_i + (1-interpolation)*batch_j)
    We find the interpolation value in [0, 1] for you by sampling from a Beta distribution; see https://arxiv.org/pdf/1710.09412.pdf

    The inputs are numpy, not torch
    """
    interpolation = np.random.beta(alpha, alpha)
    result = None

    ### YOUR CODE HERE
    result = interpolation * batch_i + (1 - interpolation) * batch_j
    ### END CODE

    return result, interpolation
```

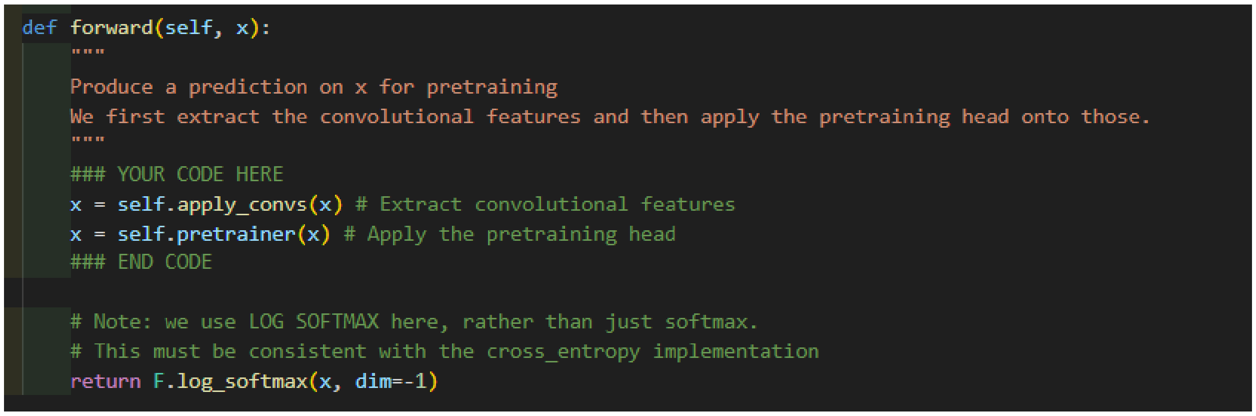
network.py



```python
def apply_convs(self, x):
    """
    Extract convolutional features from the input.

    Our network has two layers of feature extraction. Each layer consists of:
        - convolution on the features
        - 2x2 max_pool on the conv output (with stride 2)
        - relu activation

    Furthermore, you may apply dropout after the second layer's convolutions to encourage generalization.
    We recommend using torch.nn.functional (F.) for max pooling and relu
    """
    ### YOUR CODE HERE

    x = self.conv1(x) # Apply first convolutional layer
    x = F.max_pool2d(x, 2) # Apply first max pooling layer
    x = F.relu(x) # Apply first relu activation

    x = self.conv2_drop(self.conv2(x)) # Apply second convolutional layer with dropout
    x = F.max_pool2d(x, 2) # Apply second max pooling layer
    x = F.relu(x) # Apply second relu activation
    ### END CODE

    # Reshape the conv outputs so that we can apply linear layers to them
    x = x.view(-1, self.lin_size)

    return x
```

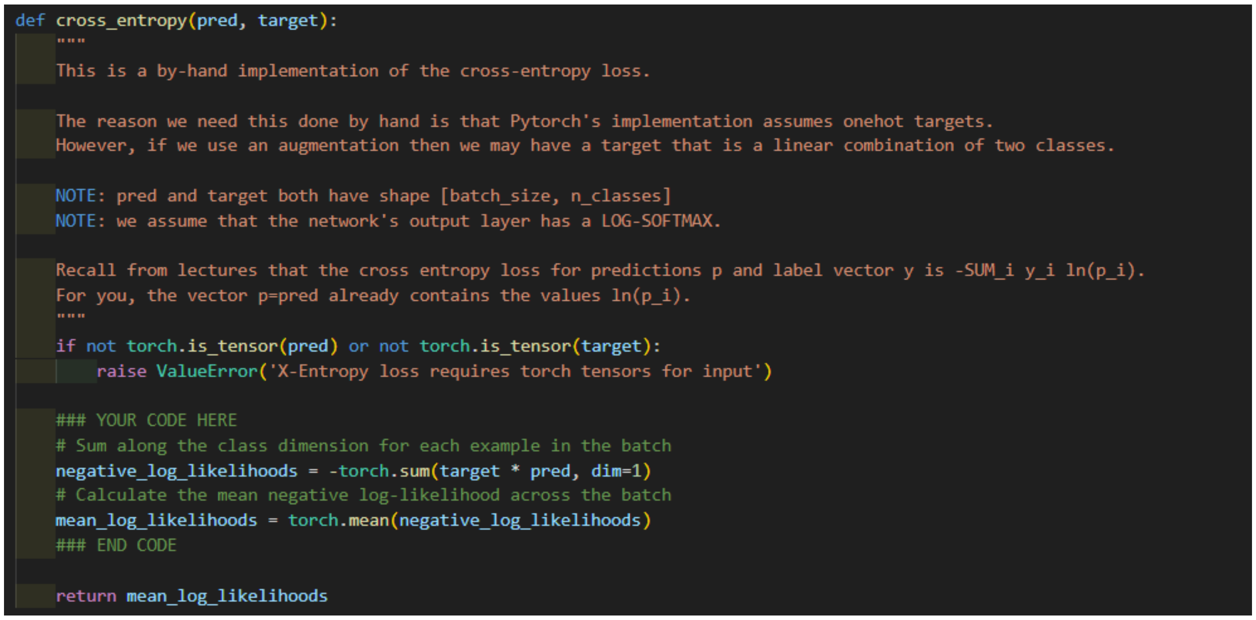


```python
def generalize(self, x):
    """
    Produce a prediction on x for finetuning.
    We first extract the convolutional features and then apply the generalization head onto those.
    """
    ### YOUR CODE HERE
    x = self.apply_convs(x) # Extract convolutional features
    x = self.generalizer(x) # Apply the generalization head
    ### END CODE

    # Note: we use LOG SOFTMAX here, rather than just softmax.
    # This must be consistent with the cross_entropy implementation
    return F.log_softmax(x, dim=-1)
```



```python
def forward(self, x):
    """
    Produce a prediction on x for pretraining
    We first extract the convolutional features and then apply the pretraining head onto those.
    """
    ### YOUR CODE HERE
    x = self.apply_convs(x) # Extract convolutional features
    x = self.pretrainer(x) # Apply the pretraining head
    ### END CODE

    # Note: we use LOG SOFTMAX here, rather than just softmax.
    # This must be consistent with the cross_entropy implementation
    return F.log_softmax(x, dim=-1)
```

utils.py

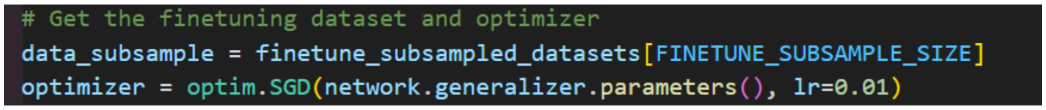


```python
def cross_entropy(pred, target):
    """
    This is a by-hand implementation of the cross-entropy loss.

    The reason we need this done by hand is that Pytorch's implementation assumes onehot targets.
    However, if we use an augmentation then we may have a target that is a linear combination of two classes.

    NOTE: pred and target both have shape [batch_size, n_classes]
    NOTE: we assume that the network's output layer has a LOG-SOFTMAX.

    Recall from lectures that the cross entropy loss for predictions p and label vector y is -SUM_i y_i ln(p_i).
    For you, the vector p=pred already contains the values ln(p_i).
    """
    if not torch.is_tensor(pred) or not torch.is_tensor(target):
        raise ValueError('X-Entropy loss requires torch tensors for input')

    ### YOUR CODE HERE
    # Sum along the class dimension for each example in the batch
    negative_log_likelihoods = -torch.sum(target * pred, dim=1)
    # Calculate the mean negative log-likelihood across the batch
    mean_log_likelihoods = torch.mean(negative_log_likelihoods)
    ### END CODE

    return mean_log_likelihoods
```

Tasks

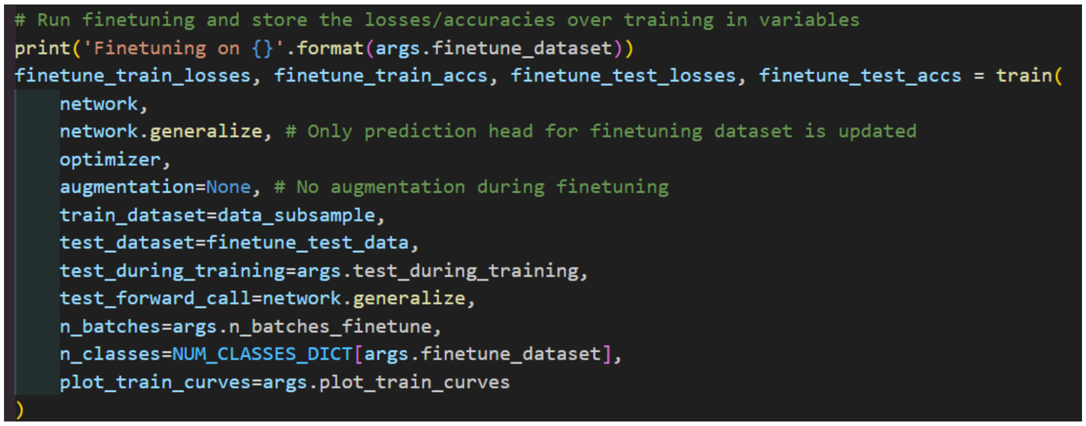
Code explanations

Where are we ensuring that the finetuning does not affect the feature extractor?



```
# Get the finetuning dataset and optimizer
data_subsample = finetune_subsampled_datasets[FINETUNE_SUBSAMPLE_SIZE]
optimizer = optim.SGD(network.generalizer.parameters(), lr=0.01)
```

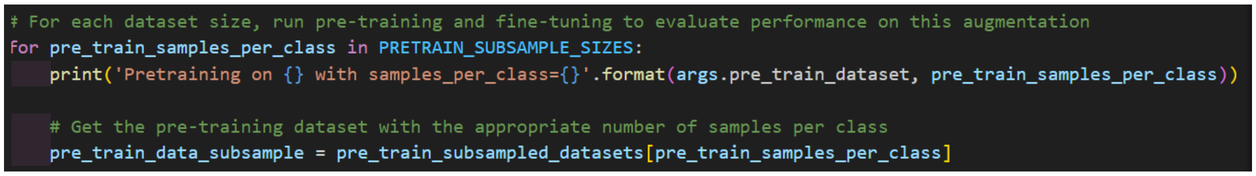
When finetuning, the optimizer is applied specifically to the parameters of the generalizer prediction head responsible for classification on the finetuning dataset D_F. The parameters of the feature extractor (eg. the convolutional layers) are not included in this optimization step, hence only the generalization head is updated during finetuning while the feature extractor remains fixed.



```
# Run finetuning and store the losses/accuracies over training in variables
print('Finetuning on {}'.format(args.finetune_dataset))
finetune_train_losses, finetune_train_accs, finetune_test_losses, finetune_test_accs = train(
    network,
    network.generalize, # Only prediction head for finetuning dataset is updated
    optimizer,
    augmentation=None, # No augmentation during finetuning
    train_dataset=data_subsample,
    test_dataset=finetune_test_data,
    test_during_training=args.test_during_training,
    test_forward_call=network.generalize,
    n_batches=args.n_batches_finetune,
    n_classes=NUM_CLASSES_DICT[args.finetune_dataset],
    plot_train_curves=args.plot_train_curves
)
```

During finetuning, *network.generalize* is passed as the *forward_call* argument, which means that during training, only the parameters of the *generalize* method and hence the prediction head for D_F will have their gradients updated.

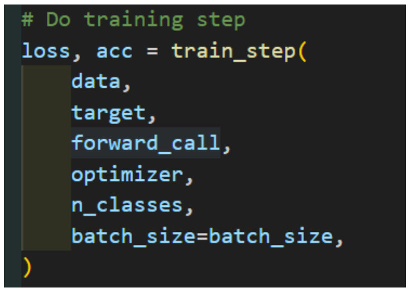
How does the code work that gets *s* samples per class for the pre-training dataset?



```
# For each dataset size, run pre-training and fine-tuning to evaluate performance on this augmentation
for pre_train_samples_per_class in PRETRAIN_SUBSAMPLE_SIZES:
    print('Pretraining on {} with samples_per_class={}'.format(args.pre_train_dataset, pre_train_samples_per_class))

    # Get the pre-training dataset with the appropriate number of samples per class
    pre_train_data_subsample = pre_train_subsampled_datasets[pre_train_samples_per_class]
```

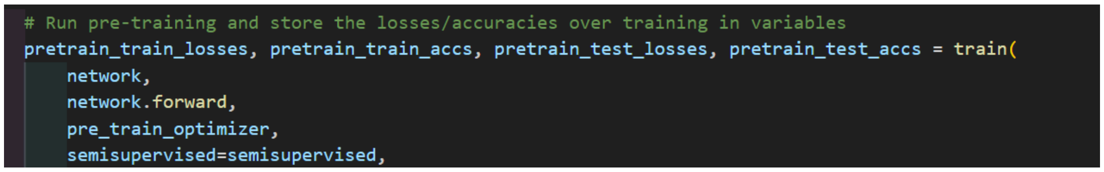
The network is pre-trained on different subsets of the pre-training dataset, each subset having a different number of samples *s* per class. This is useful for studying the impact of varying amounts of data during the pre-training phase.
The subsampled datasets for pre-training are obtained from the *get_subsampled_datasets()* method used in *get_data* which randomly generates *s* samples for each class from the original dataset for pre-training.

What is the *forward_call* parameter responsible for in the *train()* method (located in *network_training.py*)?



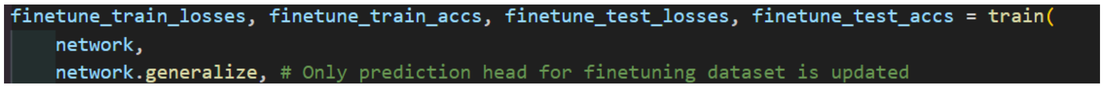
```
# Do training step
loss, acc = train_step(
    data,
    target,
    forward_call,
    optimizer,
    n_classes,
    batch_size=batch_size,
)
```

The *forward_call* parameter in the *train()* method is responsible for specifying the forward pass of the neural network on the input data, which is necessary for computing the loss. It is used to control which parts of the network are used for training, allowing for different parts of the network to be trained during the pre-training and finetuning stages.



```
# Run pre-training and store the losses/accuracies over training in variables
pretrain_train_losses, pretrain_train_accs, pretrain_test_losses, pretrain_test_accs = train(
    network,
    network.forward,
    pre_train_optimizer,
    semisupervised=semisupervised,
```

During pre-training, the *forward_call* is set to *network.forward*, which means that the entire network (including the feature extractor and the pre-training prediction head) is used for training:



```
finetune_train_losses, finetune_train_accs, finetune_test_losses, finetune_test_accs = train(
    network,
    network.generalize, # Only prediction head for finetuning dataset is updated
```

During finetuning, the *forward_call* is set to *network.generalize*, which means that only the finetuning prediction head is used for training, while the feature extractor's parameters are fixed.

Describe how the augment() method works (located in augmentations.py).

The *augment()* method applies the specified augmentation (eg. no augmentation, collage, or mixup) to the input batch. For each image in the batch, it randomly selects another image from the batch, while ensuring that an image is not augmented with itself (since we do not allow merge_indices[i] = i). The augmentation is applied to the current image and the randomly selected image from the batch, then the targets are computed by taking a linear combination of the original labels and the labels of the images that the current images are augmented with.

If we pre-train and finetune on the same dataset, is there any reason to do the finetuning step?

The effects of finetuning might be less significant when pre-training and finetuning on the same dataset compared to when using different but related datasets as in transfer learning. Finetuning on the same dataset may increase the risk of overfitting, especially if the dataset is small. If the improvement in performance from additional training is limited, the computational cost of finetuning may outweigh the benefits.

However, the finetuning step may also allow the network to specialize and adapt to the specific classification task. The initial pre-training on D_P may have provided generic feature extraction capabilities, and finetuning refines these features for the specific classes in D_F.

Whether to do the finetuning steps depends on the given context such as the task domain, size of the dataset, model performance, and computational resources available.

Predictions

Will the collage and mixup data augmentations help achieve higher finetune accuracies? Which do you expect will be more effective?
The collage and mixup data augmentations will help achieve higher finetune accuracies with limited training data as they effectively increase the size of the training dataset. The augmentations introduce variations in the training data, which may make the model more robust to variations in input (eg. different perspectives, orientations, deformations).

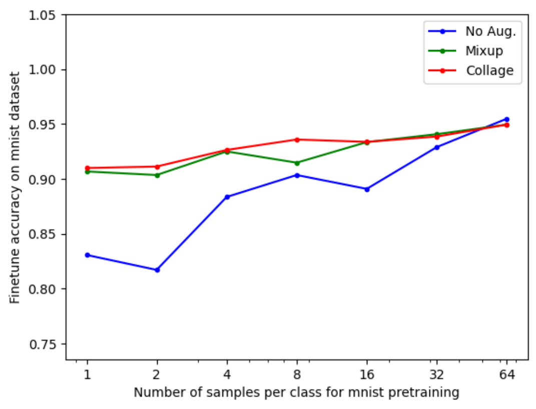
We expect mixup to be more effective than collage as the randomness in sampling the interpolation values from the continuous distribution introduces more diversity in the training data and may allow the network to learn and generalize better.

What relationship do you expect between the number of samples in the pre-training dataset and the finetuning accuracy? Does this change with data augmentations?
We expect increasing the number of samples in the pre-training dataset to have better finetuning accuracy, as it may improve feature extraction and generalization.
Data augmentations may reduce the dependence on an excessively large pre-training dataset. The finetuning process may be more effective as the network's ability to generalize from a smaller set of samples may be enhanced.

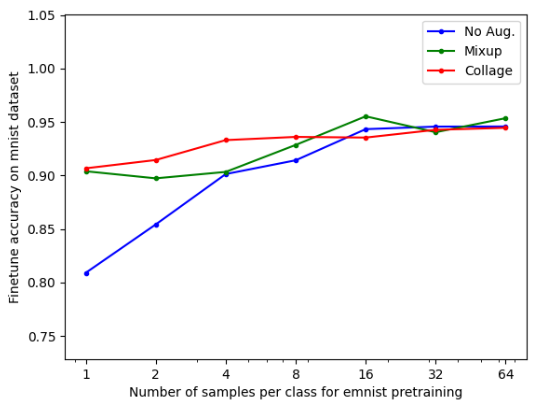
Report

Training Plots

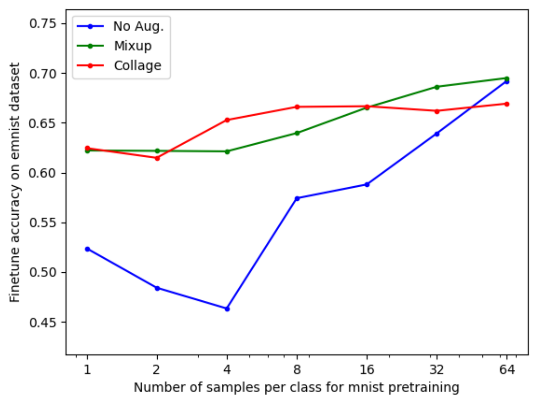
D_P=mnist, D_F=mnist



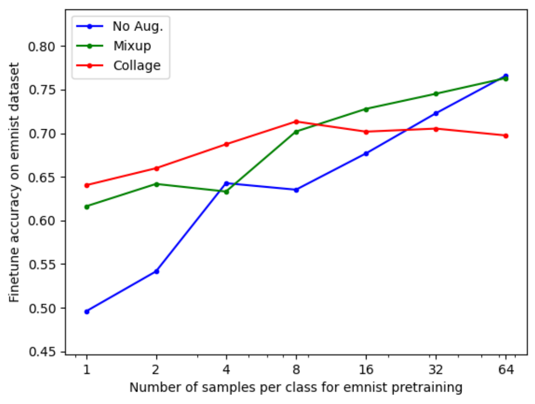
D_P=emnist, D_F=mnist



D_P=mnist, D_F=emnist



D_P=emnist, D_F=emnist



How does the number of samples per class affect training performance? Does this get affected by the augmentations?

The finetune accuracy generally increases as the number of samples per class *s* increases, both with and without augmentations. Having more samples per class provides more instances for the network to adjust its parameters to better fit the underlying patterns in the input data.

Augmentation increases the training performance up until a large enough *s* (eg. 32), as they artificially increase the effective size of the training dataset. Hence the impact of augmentations is most significant when the dataset is small, and it diminishes as the dataset size increases.

Which augmentation performs better? Why?

Collage augmentation generally performs better for smaller *s* (eg. 1-16), but mixup augmentation performs better for larger *s* (eg. 32-64).

With fewer samples per class, collage augmentation is able to create diverse images that preserve spatial relationships within images (especially for handwritten characters) for more effective exploration of the feature space.

With a larger number of samples per class, mixup introduces a smooth interpolation between samples that helps prevent overfitting and improves generalization of the network.

Does finetuning and pre-training on the same dataset obtain better performance than pre-training on one dataset and finetuning on another? Why?
Finetuning and pre-training on the same dataset obtains higher finetune accuracy.

Since we only apply the gradients to the prediction head responsible for D_F, the feature extraction pipeline remains fixed. If we obtain high accuracy scores on D_F after finetuning, then the two datasets D_P and D_F must share a lot of similar information. Finetuning and pre-training on the same dataset would hence mean that D_P and D_F share the same or most information. As the network has already learned to extract useful features from the pre-training dataset, it can predict the target variable in the fine-tuning dataset more accurately.