xfer

Release 1.0.0

Feb 01, 2019

Tutorials

Release: 1.0.0
Date: Feb 01, 2019

Xfer is a Transfer Learning framework written in Python.

Xfer features Repurposers that can be used to take an MXNet model and train a meta-model or modify the model for a new target dataset. To get started with Xfer checkout our introductory tutorial *here*.

The code can be found on our Github project page. It is open source and provided using the Apache 2.0 license.

CHAPTER 1

Deep Transfer Learning with Xfer

1.1 Transfer learning in 3 lines of code:

```
repurposer = xfer.LrRepurposer(source_model, feature_layer_names=['fc7'])
repurposer.repurpose(train_iterator)
predictions = repurposer.predict_label(test_iterator)
```

Keep reading below to see Xfer in action!

1.2 Overview

1.2.1 What is Xfer?

Xfer is a library that allows quick and easy transfer of knowledge stored in deep neural networks. It can be used for the classification of data of arbitrary numeric format, and can be applied to the common cases of image or text data.

Xfer can be used as a pipeline that spans from extracting features to training a repurposer. The repurposer is then an object that performs classification in the target task.

You can also use individual components of Xfer as part of your own pipeline. For example, you can leverage the feature extractor to extract features from deep neural networks or ModelHandler, which allows for quick building of neural networks, even if you are not an MXNet expert.

1.2.2 How can Xfer help me?

- *Resource efficiency*: you don't have to train big neural networks from scratch.
- *Data efficiency*: by transferring knowledge, you can classify complex data even if you have very few labels.
- *Easy access to neural networks*: you don't need to be an ML ninja in order to leverage the power of neural networks. With Xfer you can easily re-use them or even modify existing architectures and create your own solution.

- *Uncertainty modeling*: With the Bayesian neural network (BNN) or the Gaussian process (GP) repurposers, you can obtain uncertainty in the predictions of the repurposer.
- *Utilities for feature extraction from neural networks*.
- *Rapid prototyping*.

1.2.3 This Demo

In this notebook we demonstrate Xfer in an image classification task. A pre-trained neural network is selected, from which we transfer knowledge for the classification task in the *target* domain. The *target* task is a much smaller set of images that come from a different domain (hand-drawn sketches), therefore the classifier from the *source* task cannot be used as is, without repurposing. Therefore, the aim is to train a new classifier and it is vital to transfer knowledge from the *source* task, due to the extremely scarce *target* dataset. The new classifier for the *target* task is either a meta-model or a modified and fine-tuned clone of the *source* task's neural network.

1.2.4 Components

Xfer is comprised of 2 components:

- `ModelHandler` - Extracts features from pretrained model and performs model manipulation
- `Repurposer` - Repurposes model for target task

1.2.5 Transfer Learing Pipeline

In the following, we demonstrate the Xfer workflow:

1. A data iterator creation
2. A pre-trained model selection (i.e. picking a *source task*)
3. Feature extraction with the `ModelHandler`
4. `Repurposer` used to perform transfer learning from the source task to the target task

First we import or define all relevant modules and utilities.

```
In [1]: import numpy as np
        import os
        import json
        import random
        import logging
        import glob

        import mxnet as mx
        from sklearn.model_selection import StratifiedShuffleSplit
        from sklearn.metrics import classification_report
        from matplotlib import pylab as plt
        %matplotlib inline

        import xfer

        seed=2
        random.seed(seed)
        np.random.seed(seed)
        mx.random.seed(seed)
```

```
        logger = logging.getLogger()
        logger.setLevel(logging.INFO)

        # Change the default option below to test Xfer on other datasets (or use your own!).
        TEST_IMAGES = 'test_sketches/' # Options: 'test_images' or 'test_sketches' or 'test_images_sk
```

```
In [2]: def get_iterators_from_folder(data_dir, train_size=0.6, batchsize=10, label_name='softmax_lab
            """
            Method to create iterators from data stored in a folder with the following structure:
            /data_dir
                /class1
                    class1_img1
                    class1_img2
                    ...
                    class1_imgN
                /class2
                    class2_img1
                    class2_img2
                    ...
                    class2_imgN
                ...
                /classN
            """
            # assert dir exists
            if not os.path.isdir(data_dir):
                raise ValueError('Directory not found: {}'.format(data_dir))
            # get class names
            classes = [x.split('/')[-1] for x in glob.glob(data_dir+'/*')]
            classes.sort()
            fnames = []
            labels = []
            for c in classes:
                    # get all the image filenames and labels
                    images = glob.glob(data_dir+'/'+c+'/*')
                    images.sort()
                    fnames += images
                    labels += [c]*len(images)
            # create label2id mapping
            id2label = dict(enumerate(set(labels)))
            label2id = dict((v,k) for k, v in id2label.items())

            # get indices of train and test
            sss = StratifiedShuffleSplit(n_splits=2, test_size=None, train_size=train_size, random_st
            train_indices, test_indices = next(sss.split(labels, labels))

            train_img_list = []
            test_img_list = []
            train_labels = []
            test_labels = []
            # create imglist for training and test
            for idx in train_indices:
                train_img_list.append([label2id[labels[idx]], fnames[idx]])
                train_labels.append(label2id[labels[idx]])
            for idx in test_indices:
                test_img_list.append([label2id[labels[idx]], fnames[idx]])
                test_labels.append(label2id[labels[idx]])

            # make iterators
            train_iterator = mx.image.ImageIter(batchsize, (3,224,224), imglist=train_img_list, label
```

```
                                        path_root='')
    test_iterator = mx.image.ImageIter(batchsize, (3,224,224), imglist=test_img_list, label_
                                       path_root='')

    return train_iterator, test_iterator, train_labels, test_labels, id2label, label2id


def get_images(iterator):
    """
    Returns list of image arrays from iterator
    """
    iterator.reset()
    images = []
    while True:
        try:
            batch = iterator.next().data[0]
            for n in range(batch.shape[0]):
                images.append(batch[n])
        except StopIteration:
            break
    return images


def show_predictions(predictions, images, id2label, uncertainty=None, figsize=(9,1.2), fontsi
    """
    Plots images with predictions as labels. If uncertainty is given then this is plotted bel
    series of horizontalbar charts.
    """
    num_rows = 1 if uncertainty is None else 2

    plt.figure(figsize=figsize)
    for cc in range(n):
        plt.subplot(num_rows,n,1+cc)
        plt.tick_params(
                        axis='both',          # changes apply to the x-axis
                        which='both',      # both major and minor ticks are affected
                        bottom=False,      # ticks along the bottom edge are off
                        top=False,         # ticks along the top edge are off
                        left=False,
                        labelleft=False,
                        labelbottom=False) # labels along the bottom edge are off
        plt.imshow(np.uint8(images[cc].asnumpy().transpose((1,2,0))))
        plt.title(id2label[predictions[cc]].split(',')[0], fontsize=fontsize)
        plt.axis

    if uncertainty is not None:
        pos = range(len(id2label.values()))
        for cc in range(n):
            plt.subplot(num_rows,n,n+1+cc)
            # Normalize the bars to be 0-1 for better readability.
            xx = uncertainty[cc]
            xx = (xx-min(xx))/(max(xx)-min(xx))
            plt.barh(pos, xx, align='center', height=0.3)
            if cc == 0:
                plt.yticks(pos, id2label.values())
            else:
                plt.gca().set_yticklabels([])
            plt.gca().set_xticklabels([])
            plt.grid(True)
```

1.3 Data Handling

In order for Xfer to process data, it must be given as an MXNet data iterator (`mxnet.io.DataIter`). MXNet expects labels to be sequential integers starting at zero so we have mapped all our string labels to integers to avoid any unexpected behaviours.

The data handling portion of the workflow is made up of the following steps:

- Get iterators
- Get labels
- Get label to idx mapping dictionary

```
In [3]: # We have chosen to split the data into train and test at a 60:40 ratio and use a batchsize o
        train_iterator, test_iterator, train_labels, test_labels, id2label, label2id = get_iterators_
```

```
INFO:root:Using 1 threads for decoding...
INFO:root:Set enviroment variable MXNET_CPU_WORKER_NTHREADS to a larger number to use more threads.
INFO:root:ImageIter: loading image list...
INFO:root:Using 1 threads for decoding...
INFO:root:Set enviroment variable MXNET_CPU_WORKER_NTHREADS to a larger number to use more threads.
INFO:root:ImageIter: loading image list...
```

1.4 Source Model

ModelHandler is an Xfer module which handles everything related to the source pre-trained neural network. It can extract features given a target dataset and source model, and it can also manipulate the pre-trained network by adding/removing/freezing layers (we'll see this functionality in the next section). For now, we simply:

- Load MXNet Module from file
- Instantiate `ModelHandler` object with VGG-19 model as source model

The VGG-19 model is a convolutional neural network trained on ImageNet and is good at image classification. Other models trained on ImageNet are likely to be good source models for this task.

```
In [4]: # Download model
        path = 'http://data.mxnet.io/models/imagenet/'
        [mx.test_utils.download(path+'vgg/vgg19-0000.params'),
        mx.test_utils.download(path+'vgg/vgg19-symbol.json')]
```

```
INFO:root:vgg19-0000.params exists, skipping download
INFO:root:vgg19-symbol.json exists, skipping download
```

```
Out[4]: ['vgg19-0000.params', 'vgg19-symbol.json']
```

```
In [5]: source_model = mx.module.Module.load('vgg19', 0, label_names=['prob_label'])
        mh = xfer.model_handler.ModelHandler(source_model)
```

1.5 How well the pre-trained network alone is doing (without repurposing)?

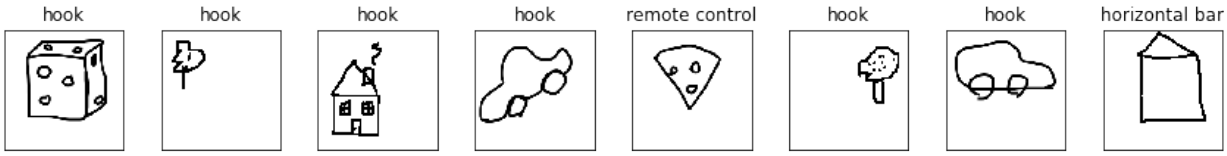
This section will show how well the pre-trained *source* model performs before any repurposing is applied.

```
In [6]: # Get pre-trained model without modifications
        model = mh.get_module(iterator=test_iterator)
```

```
        # Predict on our test data
        predictions = np.argmax(model.predict(test_iterator), axis=1).asnumpy().astype(int)
```

```
In [7]: # This utility just allows us to translate image-id's of the imagenet dataset to human-readab
        with open('imagenet1000-class-to-human.json', 'r') as fp:
            imagenet_class_to_human = json.load(fp)

        imagenet_class_to_human = {int(k): v for k, v in imagenet_class_to_human.items()}
```

```
In [8]: # Plot all test images along with the predicted labels
        images = get_images(test_iterator)

        show_predictions(predictions, images, imagenet_class_to_human, None, (15, 1.5))
```



The model is performing badly on our sketch images - it thinks most of our drawings are hooks! The reason for this is that the label and image distribution in the target task are different (having come from a different dataset) i.e the model has been trained on photographs of objects and so cannot sensibly classify these sketches. The results would get worse if the source/target dataset mismatch was larger. A **repurposing** step is required to better align the pre-trained model with the target data.

1.6 Repurposing

1.6.1 (a) Repurposing with meta-models

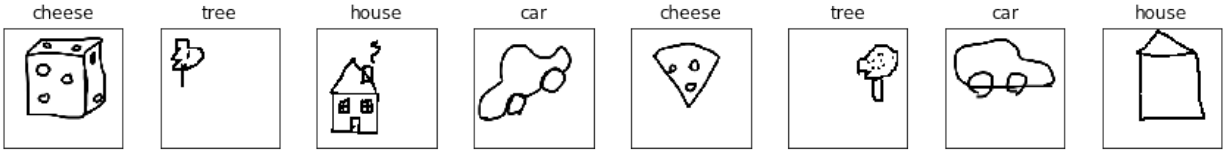
By repurposing with meta models, we use the neural network as a feature extractor and fit a different model on these features.

```
In [9]: # Instantiate a Logistic Regression repurposer (other options: SVM; GP; NN, BNN repurposers)
        logging.info("Logistic Regression (LR) Repuroser")
        repLR = xfer.LrRepurposer(source_model=source_model, feature_layer_names=['fc7'])
        repLR.repurpose(train_iterator)
        predictionsLR = repLR.predict_label(test_iterator)

        logging.info("LR Repurposer - Classification Results")
        print(classification_report(test_labels, predictionsLR, target_names=list(id2label.values()),
```

```
INFO:root:Logistic Regression (LR) Repuroser
INFO:root:Extracting features from layers: fc7
INFO:root:Processed batch 1
INFO:root:Processed batch 2
INFO:root:Processed batch 3
INFO:root:Processed batch 4
INFO:root:Processed batch 5
INFO:root:Processed batch 6
 /anaconda/envs/matplotlib-backend/lib/python3.5/site-packages/sklearn/linear_model/sag.py:326: Conve
INFO:root:Extracting features from layers: fc7
INFO:root:Processed batch 1
INFO:root:Processed batch 2
INFO:root:Processed batch 3
INFO:root:Processed batch 4
INFO:root:LR Repurposer - Classification Results
```

```
             precision    recall  f1-score   support

       tree      1.000     0.750     0.857         4
        car      1.000     1.000     1.000         4
     cheese      1.000     1.000     1.000         4
      house      0.800     1.000     0.889         4

avg / total      0.950     0.938     0.937        16
```

```
In [10]: show_predictions(predictionsLR, images, id2label, None, (15,1.5))
```



1.6.2 (b) Fine-tuning Neural Network repurposer

Neural network repurposers will:

- Modify the pretrained neural network architecture by adding and removing layers
- Retrain the network with certain layers held fixed or randomised

```
In [11]: # Choose which layers of the model to fix during training - more fixed layers lead to faster
         fixed_layers = ['conv1_1','conv1_2','conv2_1','conv2_2','conv3_1','conv3_2','conv3_3','conv3
                         'conv4_1','conv4_2', 'conv4_3','conv4_4','conv5_1','conv5_2','conv5_3', 'con
         # Choose which layers of the model to randomise before training - we may want to forget some
         # this model knows
         random_layers = []

         repNN = xfer.NeuralNetworkRandomFreezeRepurposer(source_model, target_class_count=4, fixed_l
         repNN.repurpose(train_iterator)
         predictionsNN = repNN.predict_label(test_iterator)
         logging.info("NN Repurposer - Classification Results")
         print(classification_report(test_labels, predictionsNN, target_names=list(id2label.values())
```

```
INFO:root:fc8, prob deleted from model top
INFO:root:Added new_fully_connected_layer, prob to model top
WARNING:root:Already bound, ignoring bind()
 /anaconda/envs/matplotlib-backend/lib/python3.5/site-packages/mxnet/module/base_module.py:488: UserW
INFO:root:Epoch[0] Train-accuracy=0.541667
INFO:root:Epoch[0] Time cost=33.846
INFO:root:Epoch[1] Train-accuracy=1.000000
INFO:root:Epoch[1] Time cost=22.451
INFO:root:Epoch[2] Train-accuracy=1.000000
INFO:root:Epoch[2] Time cost=24.884
INFO:root:Epoch[3] Train-accuracy=1.000000
INFO:root:Epoch[3] Time cost=23.454
INFO:root:Epoch[4] Train-accuracy=1.000000
INFO:root:Epoch[4] Time cost=24.850
INFO:root:NN Repurposer - Classification Results
             precision    recall  f1-score   support

       tree      1.000     0.750     0.857         4
        car      1.000     1.000     1.000         4
```

```
     cheese      1.000     1.000     1.000         4
      house      0.800     1.000     0.889         4

avg / total      0.950     0.938     0.937        16
```

The neural network repurposer will likely not be great if the target dataset is extremely small.

1.6.3 (c) Repurposing with probability and uncertainty

Two repurposers offer well-calibrated probability for predictions: `GPRepurposer` and `BNNRepurposer`. Here we explore the former (the latter can be used for non-tiny datasets).

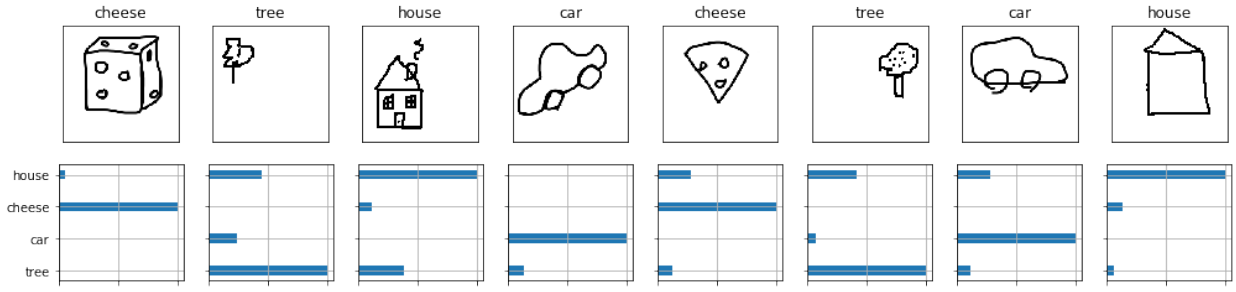
```
In [12]: # Instantiate a GP repurposer
         repGP = xfer.GpRepurposer(source_model, feature_layer_names=['fc6'], apply_l2_norm=True)
         repGP.repurpose(train_iterator)

         logging.info("GP Repurposer - Classification Results")
         uncertaintyGP = repGP.predict_probability(test_iterator)
         predictionsGP = np.argmax(uncertaintyGP, axis=1)

         print(classification_report(test_labels, predictionsGP,
             target_names=list(id2label.values()), digits=3))
```

```
INFO:root:Extracting features from layers: fc6
INFO:root:Processed batch 1
INFO:root:Processed batch 2
INFO:root:Processed batch 3
INFO:root:Processed batch 4
INFO:root:Processed batch 5
INFO:root:Processed batch 6
INFO:GP:initializing Y
INFO:GP:initializing inference method
INFO:GP:adding kernel and likelihood as parameters
INFO:GP:initializing Y
INFO:GP:initializing inference method
INFO:GP:adding kernel and likelihood as parameters
INFO:GP:initializing Y
INFO:GP:initializing inference method
INFO:GP:adding kernel and likelihood as parameters
INFO:GP:initializing Y
INFO:GP:initializing inference method
INFO:GP:adding kernel and likelihood as parameters
INFO:root:GP Repurposer - Classification Results
INFO:root:Extracting features from layers: fc6
INFO:root:Processed batch 1
INFO:root:Processed batch 2
INFO:root:Processed batch 3
INFO:root:Processed batch 4
```

```
             precision    recall  f1-score   support

       tree      1.000     0.750     0.857         4
        car      1.000     1.000     1.000         4
     cheese      1.000     1.000     1.000         4
      house      0.800     1.000     0.889         4

avg / total      0.950     0.938     0.937        16
```

The code below will plot the predictions and the probability for each class.

```
In [13]: show_predictions(predictionsGP, images, id2label, uncertaintyGP, (17,3.8))
```



We not only have predictions from this model but we can also see the uncertainty in the model for any given predition which allows us to make better decisions on our data.

1.6.4 Other repurposers

We have seen the use of LrRepurposer, NeuralNetworkRandomFreezeRepurposer and GpRepurposer. Other repurposers offered are: SvmRepurposer, BnnRepurposer, NeuralNetworkFineTuneRepurposer.

You can also *write your own repurposer*

1.7 Using Xfer on your own data

All you need to do is generate your own data iterator and use it instead of the iterators used above.

For more details see the API documentation

```
In [ ]:
```

CHAPTER 2

Model Handler

ModelHandler is a utility class for manipulating and inspecting MXNet models. It can be used to:

- Add and remove layers from an existing model or "freeze" selected layers
- Discover information such as layer names and types
- Extract features from pretrained models

In this tutorial, we will demonstrate some of the key capabilities of ModelHandler.

2.1 Initialisation

```
In [1]: import mxnet as mx
        import logging

        import xfer

        logger = logging.getLogger()
        logger.setLevel(logging.INFO)
```

```
In [2]: # Download vgg19 (trained on imagenet)
        path = 'http://data.mxnet.io/models/imagenet/'
        [mx.test_utils.download(path+'vgg/vgg19-0000.params'),
        mx.test_utils.download(path+'vgg/vgg19-symbol.json')]
```

```
INFO:root:vgg19-0000.params exists, skipping download
INFO:root:vgg19-symbol.json exists, skipping download
```

```
Out[2]: ['vgg19-0000.params', 'vgg19-symbol.json']
```

```
In [3]: source_model = mx.module.Module.load('vgg19', 0, label_names=['prob_label'])

        # The ModelHandler constructor takes an MXNet Module as input
        mh = xfer.model_handler.ModelHandler(source_model)
```

2.2 Model Inspection

2.2.1 Layer names

```
In [4]: print(mh.layer_names)
```

```
['conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1', 'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'po
```

2.2.2 Layer Types

Given the name of a layer, this function returns the type.

```
In [5]: print(mh.get_layer_type('relu5_2'))
        print(mh.get_layer_type('flatten_0'))
        print(mh.get_layer_type('fc7'))
        print(mh.get_layer_type('conv5_3'))
        print(mh.get_layer_type('prob'))
```

```
Activation
Flatten
FullyConnected
Convolution
SoftmaxOutput
```

ModelHandler can be used to get a list of layers that are of a specific type.

```
In [6]: import xfer.model_handler.consts as consts

        print(mh.get_layer_names_matching_type('Convolution'))
        print(mh.get_layer_names_matching_type('Pooling'))
        print(mh.get_layer_names_matching_type('Activation'))
        print(mh.get_layer_names_matching_type('BatchNorm'))
```

```
['conv1_1', 'conv1_2', 'conv2_1', 'conv2_2', 'conv3_1', 'conv3_2', 'conv3_3', 'conv3_4', 'conv4_1',
['pool1', 'pool2', 'pool3', 'pool4', 'pool5']
['relu1_1', 'relu1_2', 'relu2_1', 'relu2_2', 'relu3_1', 'relu3_2', 'relu3_3', 'relu3_4', 'relu4_1',
[]
```

2.2.3 Architecture Visualization

```
In [7]: mh.visualize_net()
```

2.3 Feature Extraction

ModelHandler makes it easy to extract features from a dataset using a pretrained model.

By passing an MXNet DataIterator and a list of the layers to extract features from the `get_layer_output()` method will return a feature dictionary and an ordered list of labels.

```
In [8]: imglist = [[0, 'test_images/accordion/accordion_1.jpg'], [0, 'test_images/accordion/accordion
                   [0, 'test_images/accordion/accordion_4.jpg'], [0, 'test_images/accordion/accordion
                   [1, 'test_images/ant/ant_2.jpg'], [1, 'test_images/ant/ant_3.jpg'], [1, 'test_imag
                   [2, 'test_images/anchor/anchor_1.jpg'], [2, 'test_images/anchor/anchor_2.jpg'], [2
                   [2, 'test_images/anchor/anchor_4.jpg'], [2, 'test_images/anchor/anchor_5.jpg'], [3
```

```
                  [3, 'test_images/airplanes/airplanes_2.jpg'], [3, 'test_images/airplanes/airplanes
                  [3, 'test_images/airplanes/airplanes_5.jpg']]
        iterator = mx.img.ImageIter(imglist=imglist, batch_size=4, path_root='', data_shape=(3, 224,

INFO:root:Using 1 threads for decoding...
INFO:root:Set enviroment variable MXNET_CPU_WORKER_NTHREADS to a larger number to use more threads.
INFO:root:ImageIter: loading image list...
```

```
In [9]: features, labels = mh.get_layer_output(data_iterator=iterator, layer_names=['fc6', 'fc8'])

        print('Shape of output from fc6:', features['fc6'].shape)
        print('Shape of output from fc8:', features['fc8'].shape)
        print('Labels:', labels)

        print('Subset of example feature:', features['fc8'][0,:100])
```

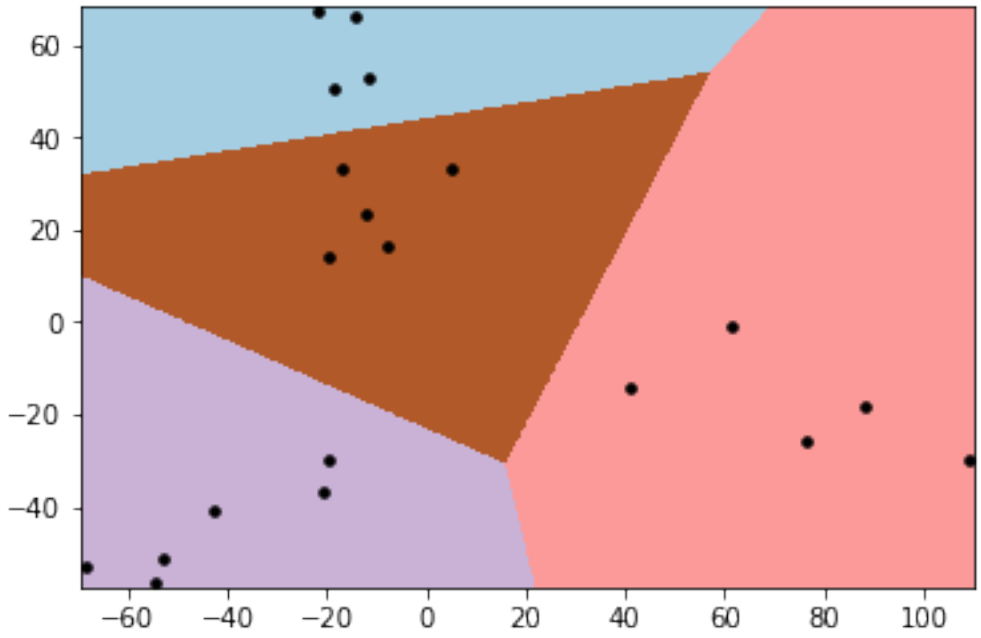
```
INFO:root:Extracting features from layers: fc6 fc8
INFO:root:Processed batch 1
INFO:root:Processed batch 2
INFO:root:Processed batch 3
INFO:root:Processed batch 4
INFO:root:Processed batch 5

Shape of output from fc6: (20, 4096)
Shape of output from fc8: (20, 1000)
Labels: [0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3]
Subset of example feature: [-2.5173218  -3.0606608  -0.18566455 -1.195584   -1.6561157  -0.3466432
 -0.27523482 -3.0833778  -3.987122   -4.711858   -2.5934136  -1.1821984
  0.06423883 -3.5403426   0.36486915 -2.0515091  -3.3651357   0.47566187
 -0.93592185 -0.53005326 -2.7707744  -1.2674817  -2.3202353  -0.33125317
 -1.5847255  -4.2490544  -4.2170153  -5.6999183  -2.6653297  -3.4800928
 -4.693992   -3.4104934  -3.673527   -4.2224913  -0.29074478 -6.513745
 -4.4927287  -4.5361094  -2.549627    2.1703975  -1.3125131  -2.1347325
 -3.761081   -2.3712082  -3.8052034  -1.6259451  -1.68117    -1.481512
 -2.2081814  -1.5731778  -1.287838    1.2327844  -3.9466934  -3.9385183
 -0.87836707 -2.9489741  -3.4411037  -4.030957   -1.4967936  -3.7117271
 -2.2397022  -3.325867   -2.8145652  -0.63274264 -3.2671835  -1.8046627
 -3.0445974  -1.5151932  -2.7235372   6.5550556  -1.62281    -1.5104069
  1.3592944   0.8891826  -0.14048216 -1.0063077  -1.5578198  -0.45763612
 -2.0689113   3.2839453  -2.0749338  -4.179339   -0.49392343 -0.5244163
 -1.9723302  -0.07367857 -2.2878125  -0.96980214 -2.8648748  -2.6847577
 -3.5610118  -3.7286394  -4.5710897  -4.949738   -0.80546796 -5.8007493
 -3.260846   -6.434879   -4.7502995  -4.953493  ]
```

These features can be used for training a meta-model or for clustering as shown in this example.

```
In [10]: from sklearn.decomposition import PCA
         from sklearn.cluster import KMeans
         import matplotlib.pyplot as plt
         import numpy as np
         %matplotlib inline

         reduced_data = PCA(n_components=2).fit_transform(features['fc8'])

         kmeans = KMeans(init='k-means++', n_clusters=4, n_init=10)
         kmeans.fit(reduced_data)

         h=0.1

         x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
         y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
```

```
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

        Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)

        plt.imshow(Z, interpolation='nearest',
                   extent=(xx.min(), xx.max(), yy.min(), yy.max()),
                   cmap=plt.cm.Paired,
                   aspect='auto', origin='lower')
        plt.plot(reduced_data[:, 0], reduced_data[:, 1], 'k.', markersize=7)
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x11a199128>]
```



2.4 Model Manipulation

Modifying models in MXNet can be problematic because symbols are held as graphs. This means that modifiying the input of the model requires the graph to be reconstructed above any changes made. ModelHandler takes care of this for you which means that adding and removing layers from either end of a neural network can be done with 1-2 lines of code.

2.4.1 Remove layers

```
In [11]: # Dropping 4 layers from the top of the layer hierarchy (where top = output)
         mh.drop_layer_top(4)
         print(mh.layer_names)
```

```
INFO:root:relu7, drop7, fc8, prob deleted from model top
```

```
['conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1', 'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'po
```

```
In [12]: # Dropping a layer from the bottom of the layer hierarchy (where bottom = input)
         mh.drop_layer_bottom(1)
         print(mh.layer_names)
```

```
INFO:root:conv1_1 deleted from model bottom
```

```
['relu1_1', 'conv1_2', 'relu1_2', 'pool1', 'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2', 'conv
```

2.4.2 Add layers

Layers can be added to models by first defining the layer with an `mxnet.symbol` object and using `add_layer_top()` or `add_layer_bottom()` to add the layer to the model.

```
In [13]: # define layer symbols
         fc = mx.sym.FullyConnected(name='fullyconntected1', num_hidden=4)
         softmax = mx.sym.SoftmaxOutput(name='softmax')
         conv1 = mx.sym.Convolution(name='convolution1', kernel=(20,20), num_filter=64)

         # Add layer to the bottom of the layer hierarchy (where bottom = input)
         mh.add_layer_bottom([conv1])
         # Add layer to the top of the layer hierarchy (where top = output)
         mh.add_layer_top([fc, softmax])

         print(mh.layer_names)
```

```
INFO:root:Added convolution1 to model bottom
INFO:root:Added fullyconntected1, softmax to model top
```

```
['convolution1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1', 'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2
```

Once a model has been modified, ModelHandler can be used to return an MXNet Module which can then be used for training.

There is an option to specify parameters which should stay fixed during training or should be randomised before training to allow different modes of transfer learning.

```
In [14]: # In this case, the conv1_1 layer will stay fixed during training and the layers fc6 and fc7
         mod = mh.get_module(iterator,
                             fixed_layer_parameters=mh.get_layer_parameters(['conv1_1']),
                             random_layer_parameters=mh.get_layer_parameters(['fc6', 'fc7']))
```

```
In [15]: iterator.reset()
         mod.fit(iterator, num_epoch=5)
```

```
WARNING:root:Already bound, ignoring bind()
/anaconda/envs/xfer_env/lib/python3.5/site-packages/mxnet/module/base_module.py:488: UserWarning: Par
  allow_missing=allow_missing, force_init=force_init)
INFO:root:Epoch[0] Train-accuracy=0.100000
INFO:root:Epoch[0] Time cost=58.709
INFO:root:Epoch[1] Train-accuracy=0.250000
INFO:root:Epoch[1] Time cost=56.932
INFO:root:Epoch[2] Train-accuracy=0.250000
INFO:root:Epoch[2] Time cost=52.735
INFO:root:Epoch[3] Train-accuracy=0.250000
INFO:root:Epoch[3] Time cost=61.472
INFO:root:Epoch[4] Train-accuracy=0.250000
INFO:root:Epoch[4] Time cost=58.052
```

We now have a trained model ready to be used for prediction. This new model isn't very useful but demonstrates the concept - to train a better model, use more data and experiment with combinations of fixed and random layers.

Now you have seen what ModelHandler can do, you should try it out for yourself!

For more details see the API docs

```
In [ ]:
```

CHAPTER 3

Transfer learning for text categorization

In this notebook we showcase how to use Xfer to tackle a simple task of transfer learning for text categorization. To that end, we use the 20 newsgroups text dataset (http://qwone.com/~jason/20Newsgroups/) which is comprised of a collection of 20K newsgroups posts.

We use a Convolutional Neural Network (CNN) pre-trained on a subset of 13 classes (~12K instances). For the target task, we assume that we have access to a much smaller dataset with 100 posts from the remaining 7 categories. This is a common situation in many real world applications where the number of categories grows with time as we collect new data. In this case, we will always start with a low number of labelled instances for the new categories (this is the cold-start problem).

In this scenario, training a NN from scratch on the target task is not feasible: due to the scarcity of labeled instances (100 in this case) and the large number of parameters, the model will be prone to overfitting. Instead, we will use Xfer to transfer the knowledge from the source model and propose a data efficient classifier.

```
In [1]: import warnings
        warnings.filterwarnings('ignore')

        import re
        import numpy as np
        import pickle
        from collections import Counter
        import itertools
        import numpy as np
        import mxnet as mx

        import xfer

        mx.random.seed(1)
        np.random.seed(1)

        %matplotlib inline
        import matplotlib.pyplot as plt
```

```
In [2]: config = {
            "source_vocab": "NewsGroupsSourceVocabulary.pickle",
            "model_prefix_source":'NewsGroupsSourceModel',
```

```
        "num_epoch_source": 100,
        "batch_size": 100,
        "num_train": 100,
        "context": mx.cpu(),
    }
```

3.1 A) Load the pre-trained model

Before building the target dataset, we load the pre-trained model into a mxnet `Module`. In this case, the pre-trained model is a CNN that was trained with instances beloging to the following 13 categories that are not used in the target task:

- comp.graphics
- comp.os.ms-windows.misc
- comp.sys.ibm.pc.hardware
- comp.windows.x
- misc.forsale
- rec.motorcycles
- rec.sport.baseball
- sci.crypt
- sci.med
- sci.space
- talk.politics.mideast
- talk.politics.misc
- talk.religion.misc

```
In [3]: import zipfile
        with zipfile.ZipFile("{}-{:04}.params.zip".format(config["model_prefix_source"], config["num_
            zip_ref.extractall()
```

```
In [4]: sym, arg_params, aux_params = mx.model.load_checkpoint(config["model_prefix_source"], config
        mx.viz.plot_network(sym)
```

3.2 B) Load the target dataset into an iterator

We define a helper function to download the dataset. The 7 classes used for the target task are the following:

- alt.atheism
- comp.sys.mac.hardware
- rec.autos
- rec.sport.hockey
- sci.electronics
- soc.religion.christian

- talk.politics.guns

```
In [5]: def download_dataset():
            from sklearn.datasets import fetch_20newsgroups
            categories=['alt.atheism',
                        'comp.sys.mac.hardware',
                        'rec.autos',
                        'rec.sport.hockey',
                        'sci.electronics',
                        'soc.religion.christian',
                        'talk.politics.guns'
                       ]

            newsgroups_train = fetch_20newsgroups(subset='train',categories=categories)
            newsgroups_test = fetch_20newsgroups(subset='test',categories=categories)

            x_text = np.concatenate((newsgroups_train.data, newsgroups_test.data), axis=0)
            labels = np.concatenate((newsgroups_train.target, newsgroups_test.target))

            return x_text, labels, categories
```

In addition, we use two helper classes to create the corpus: * **Vocabulary**: It creates the lexicon for a given corpus. In addition, it provides a basic string cleaning function based on regular expressions. * **Corpus**: Given a corpus (text and labels) and a Vocabulary object, it converts the text instances into a numerical format. In particular, it uses the provided vocabulary object to tokenize and clean the text instances. Then, it pads the sentences using max_length/fix_length and the padding symbol defined in the vocabulary object. Finally, each token is encoded into a one-hot vector using the vocabulary. In addition, it provides a helper function to build the training and test sets.

```
In [6]: class Vocabulary(object):
            def __init__(self, sentences, padding_word="</s>", unknown_word="</ukw>"):
                self.padding_word = padding_word
                self.unknown_word = unknown_word
                sentences = [self.clean_str(sent).split(" ") for sent in sentences]
                self.max_length = max(len(x) for x in sentences)
                self.word_counts = Counter(itertools.chain(*sentences))
                self.id2word = [x[0] for x in self.word_counts.most_common()]
                self.id2word.append(self.padding_word)
                self.id2word.append(self.unknown_word)
                self.word2id = {x: i for i, x in enumerate(self.id2word)}

                print('Vocabulary size', len(self.id2word))

            def clean_str(self, string):
                string = re.sub(r"[^A-Za-z0-9(),;!?\']", " ", string)
                contractions = ["\'t", "\'ve", "\'d", "\'s", "\'ll", "\'m", "\'er"]
                punctuations =  [",", ";", "!", "\?", "\)", "\("]
                for ee in contractions + punctuations:
                    string = re.sub(r"{}".format(ee), " {} ".format(ee), string)
                return string.strip().lower()
```

```
In [7]: class Corpus(object):
            def __init__(self, sentences, labels, vocabulary, max_length=None, fix_length=None):
                self.vocabulary = vocabulary
                self.max_length = max_length
                self.fix_length = fix_length
                sentences = [self.vocabulary.clean_str(sent).split(" ") for sent in sentences]
                sentences_padded = self._pad_sentences(sentences, self.vocabulary.padding_word, self
                x = []
                for sentence in sentences_padded:
                    x.append([self.vocabulary.word2id.get(word, self.vocabulary.word2id[self.vocabula
```

```
                self.x = np.array(x)
                self.y = np.array(labels)

                print('Data shape:', self.x.shape)
                print('Vocabulary size', len(vocabulary.id2word))
                print('Maximum number words per sentence', self.x.shape[1])
                print('Number of labels', len(np.unique(self.y)))

            def _pad_sentences(self, sentences, padding_word="</s>", max_length=None, fix_length=None
                sequence_length = max(len(x) for x in sentences)
                if max_length is not None:
                    sequence_length = min(sequence_length, max_length)
                if fix_length is not None:
                    sequence_length = fix_length
                padded_sentences = []
                for i in range(len(sentences)):
                    sentence = sentences[i]
                    if len(sentence) > sequence_length:
                        sentence = sentence[0:sequence_length]
                    num_padding = sequence_length - len(sentence)
                    new_sentence = sentence + [padding_word] * num_padding
                    padded_sentences.append(new_sentence)

                return padded_sentences

            def split_train_test(self, num_train):
                shuffle_indices = np.random.permutation(np.arange(len(self.y)))
                x_shuffled = self.x[shuffle_indices]
                y_shuffled = self.y[shuffle_indices]
                x_train, x_dev = x_shuffled[:num_train], x_shuffled[num_train:]
                y_train, y_dev = y_shuffled[:num_train], y_shuffled[num_train:]

                print('Train/Test split: %d/%d' % (len(y_train), len(y_dev)))
                print('Train shape:', x_train.shape)
                print('Test shape:', x_dev.shape)

                return x_train, x_dev, y_train, y_dev
```

We import the vocabulary object of the source model

```
In [8]: with open(config["source_vocab"], 'rb') as handle:
            vocab_source = pickle.load(handle)
```

We build the dataset for the target task and split it in training/test sets. Notice that we use only 100 instances for the training set since we care about scenarios in which the target dataset is small compared to the source dataset.

```
In [9]: x_text, labels, categories = download_dataset()
        corpus = Corpus(x_text, labels, vocab_source, fix_length=arg_params["vocab_embed_weight"].sha
        x_train, x_dev, y_train, y_dev = corpus.split_train_test(config["num_train"])
```

```
Data shape: (6642, 300)
Vocabulary size 149371
Maximum number words per sentence 300
Number of labels 7
Train/Test split: 100/6542
Train shape: (100, 300)
Test shape: (6542, 300)
```

We finally load the training/test datasets into mxnet iterators:

```
In [10]: train_iter = mx.io.NDArrayIter(x_train, y_train, config['batch_size'], shuffle=True)
         val_iter = mx.io.NDArrayIter(x_dev, y_dev, config['batch_size'], shuffle=False)
```

3.3 C) How well we predict without repurposing?

We create a new mxnet `Module`, bind the training/test iterators and load the parameters of the pre-trained model.

```
In [11]: mod = mx.mod.Module(symbol=sym,
                             context=config["context"],
                             data_names=['data'],
                             label_names=['softmax_label'])
         mod.bind(data_shapes=train_iter.provide_data, label_shapes=train_iter.provide_label)
         mod.set_params(arg_params, aux_params)

In [12]: mod.save_checkpoint("NewsGroupsSourceModelCompressed",1)
```

If we use the pre-trained network directly, we obtain a poor performance since it was trained for a different task (different set of classes).

```
In [13]: y_predicted = mod.predict(val_iter).asnumpy().argmax(axis=1)
         print("Accuracy score before repurposing: {}".format(np.mean((y_predicted == y_dev))))

Accuracy score before repurposing: 0.07169061449098135
```

3.4 D) Repurposing

For this demo, we use one class of repurposers called `MetaModelRepurposer`. These are repurposers that use the pre-trained model as a feature extractor and fit a different model using the extracted features. In particular, we use an `LrRepurposer` which fits a logistic regression using the extracted features.

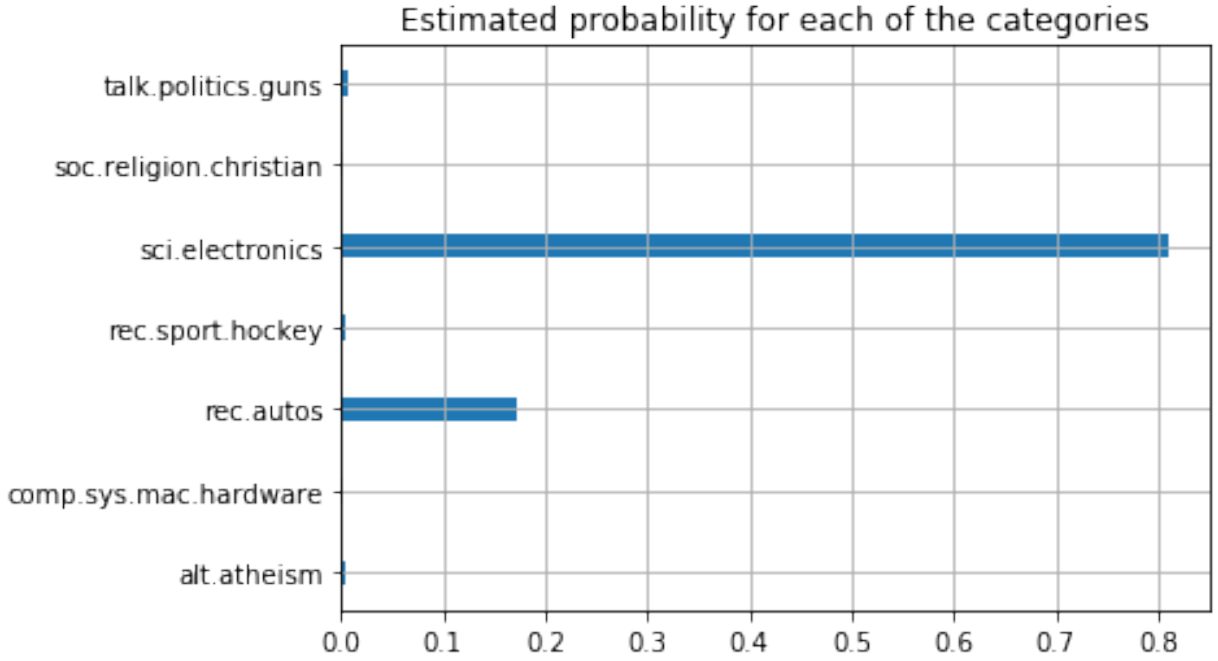
```
In [14]: lr_repurposer = xfer.LrRepurposer(mod, ["dropout0"])
         lr_repurposer.repurpose(train_iter)
         y_predicted = lr_repurposer.predict_label(val_iter)

In [15]: print("Accuracy score after repurposing: {}".format(np.mean((y_predicted == y_dev))))

Accuracy score after repurposing: 0.5889636196881688
```

Training the model from scratch results in an accuracy of approximately 0.5 and it is much slower to train having to resort to GPUs to speed up the process.

Finally, we show the full pipeline to categorize a new test example. In this case, we have used the first paragraphs of the wikipedia article for **"Automotive Electronics"**. (Feel free to replace it with your own example!!!)

```
In [16]: test_example = ["""
         Automotive electronics are electronic systems used in vehicles, including engine management,

         Electronic systems have become an increasingly large component of the cost of an automobile,

         The earliest electronics systems available as factory installations were vacuum tube car rad
         """]

In [17]: corpus = Corpus(test_example, [1], vocab_source, fix_length=arg_params["vocab_embed_weight"]

Data shape: (1, 300)
Vocabulary size 149371
Maximum number words per sentence 300
Number of labels 1

In [18]: test_example_iter = mx.io.NDArrayIter(corpus.x, corpus.y, len(corpus.y), shuffle=False)

In [19]: y_predicted_prob = lr_repurposer.predict_probability(test_example_iter)

In [20]: plt.barh(list(range(y_predicted_prob.shape[1])), y_predicted_prob[0,:], align='center', heig
         plt.yticks(list(range(y_predicted_prob.shape[1])), categories)
```

```
plt.title("Estimated probability for each of the categories")
plt.grid(True)
```



We can see that the main two identified topics are electronics and autos, which is what we would expect given the title of the article.

In [ ]:

CHAPTER 4

Xfer with HyperParameter Optimization

When training neural networks, hyperparameters may have to be tuned to improve accuracy metrics. The purpose of this notebook is to demonstrate how to do *HyperParameter Optimization* (HPO) when repurposing neural networks in Xfer. Here, we use GPyOpt to do HPO through Bayesian Optimization.

Note that depending on number of epochs, the target data set and transferability between source and target tasks, the default hyperparameter settings in Xfer could give desired results and HPO may not be required. If someone wants to try HPO, this notebook shows how to do it using GPyOpt.

In [1]:
```
import warnings
warnings.filterwarnings("ignore")

import logging
logging.disable(logging.WARNING)

import gc
import glob
import os
import random
import time

import GPyOpt
import mxnet as mx
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit
import xfer

from matplotlib import pylab as plt
%matplotlib inline
```

```
/anaconda/envs/xfer_env/lib/python3.5/site-packages/urllib3/contrib/pyopenssl.py:46: DeprecationWarn
```

4.1 Utility methods

```
In [2]: def set_random_seeds():
            seed = 1234
            np.random.seed(seed)
            mx.random.seed(seed)
            random.seed(seed)
            GPyOpt.util.general.np.random.seed(seed)
            GPyOpt.optimization.acquisition_optimizer.np.random.seed(seed)

        def get_iterators(data_dir, train_size=0.3, validation_size=0.3, test_size=0.4, batch_size=1,
                          label_name='softmax_label', data_name='data', random_state=1):
            """
            Method to create iterators from data stored in a folder with the following structure:
            /data_dir
                /class1
                    class1_img1 ... class1_imgN
                /class2
                    class2_img1 ... class2_imgN
                ...
                /classN
            """
            set_random_seeds()
            # Assert data_dir exists
            if not os.path.isdir(data_dir):
                raise ValueError('Directory not found: {}'.format(data_dir))
            # Get class names
            classes = [x.split('/')[-1] for x in glob.glob(data_dir+'/*')]
            classes.sort()
            fnames = []
            labels = []
            for c in classes:
                    # Get all the image filenames and labels
                    images = glob.glob(data_dir+'/'+c+'/*')
                    images.sort()
                    fnames += images
                    labels += [c]*len(images)
            # Create label2id mapping
            id2label = dict(enumerate(set(labels)))
            label2id = dict((v,k) for k, v in id2label.items())

            # Split training(train+validation) and test data
            sss = StratifiedShuffleSplit(n_splits=2, test_size=None, train_size=train_size+validatio
            train_indices, test_indices = next(sss.split(labels, labels))

            # Training data (train+validation)
            train_validation_images = []
            train_validation_labels = []
            for idx in train_indices:
                train_validation_images.append([label2id[labels[idx]], fnames[idx]])
                train_validation_labels.append(label2id[labels[idx]])

            # Test data
            test_images = []
            test_labels = []
            for idx in test_indices:
                test_images.append([label2id[labels[idx]], fnames[idx]])
                test_labels.append(label2id[labels[idx]])
```

```
    # Separate validation set and train set
    train_percent = train_size / (train_size+validation_size)
    sss_1 = StratifiedShuffleSplit(n_splits=2, test_size=None, train_size=train_percent, rand
    train_indices, validation_indices = next(sss_1.split(train_validation_labels, train_valid
    train_images = []
    train_labels = []
    for idx in train_indices:
        train_images.append(train_validation_images[idx])
        train_labels.append(train_validation_labels[idx])
    validation_images = []
    validation_labels = []
    for idx in validation_indices:
        validation_images.append(train_validation_images[idx])
        validation_labels.append(train_validation_labels[idx])

    # Create iterators
    train_iterator = mx.image.ImageIter(batch_size, (3,224,224), imglist=train_images, label_
                                        data_name=data_name, path_root='')
    validation_iterator = mx.image.ImageIter(batch_size, (3,224,224), imglist=validation_imag
                                             data_name=data_name, path_root='')
    train_validation_iterator = mx.image.ImageIter(batch_size, (3,224,224), imglist=train_val
                                                   label_name=label_name, data_name=data_name
    test_iterator = mx.image.ImageIter(batch_size, (3,224,224), imglist=test_images, label_na
                                       data_name=data_name, path_root='')

    return train_iterator, validation_iterator, train_validation_iterator, test_iterator, id2

def get_labels(iterator):
    """ Return labels from data iterator """
    iterator.reset()
    labels = []
    while True:
        try:
            labels = labels + iterator.next().label[0].asnumpy().astype(int).tolist()
        except StopIteration:
            break
    return labels

def get_images(iterator):
    """ Return list of image arrays from iterator """
    iterator.reset()
    images = []
    while True:
        try:
            batch = iterator.next().data[0]
            for n in range(batch.shape[0]):
                images.append(batch[n])
        except StopIteration:
            break
    return images

def show_predictions(predictions, images, id2label, figsize=(15,1.5), fontsize=12, n=None):
    """ Display images along with predicted labels """
    n = len(images) if n is None else n
    num_rows = 1
    plt.figure(figsize=figsize)
    for cc in range(n):
        plt.subplot(num_rows,n,1+cc)
```

```
                plt.tick_params(
                                axis='both',          # changes apply to the x-axis
                                which='both',      # both major and minor ticks are affected
                                bottom=False,      # ticks along the bottom edge are off
                                top=False,         # ticks along the top edge are off
                                left=False,
                                labelleft=False,
                                labelbottom=False) # labels along the bottom edge are off
                plt.imshow(np.uint8(images[cc].asnumpy().transpose((1,2,0))))
                plt.title(id2label[predictions[cc]].split(',')[0], fontsize=fontsize)
                plt.axis
```

4.1.1 A) Source model

In Transfer Learning, the model from which knowledge is transferred is called the source model. Here, we use vgg19 model from MXNet Model Zoo as the source model. vgg19 is a convolutional neural network trained on the ImageNet dataset which contains 1 million natural images categorized into 1000 classes.

```
In [3]: # Download source model
        path = 'http://data.mxnet.io/models/imagenet/'
        [mx.test_utils.download(path+'vgg/vgg19-0000.params'), mx.test_utils.download(path+'vgg/vgg19

        # Load source model from file
        source_model = mx.module.Module.load('vgg19', 0, label_names=['prob_label'])
```

4.1.2 B) Target data

Target data is a much smaller dataset with 40 images categorized into 4 classes from a different domain (hand-drawn sketches). We'll demonstrate how to use Xfer along with HPO to learn classifying this target data by transferring knowledge from vgg19 model.

```
In [4]: TARGET_DATA_DIR = 'test_sketches'
        set_random_seeds()
        train_iterator, validation_iterator, train_validation_iterator, test_iterator, id2label = get
        train_labels = get_labels(train_iterator)
        validation_labels = get_labels(validation_iterator)
        test_labels = get_labels(test_iterator)
        train_validation_labels = get_labels(train_validation_iterator)

        print('Number of train images: {}'.format(len(train_labels)))
        print('Number of validation images: {}'.format(len(validation_labels)))
        print('Number of test images: {}'.format(len(test_labels)))
```

```
Number of train images: 12
Number of validation images: 12
Number of test images: 16
```

4.2 How are these data sets used?

During HPO, we train the model using the training data and evaluate the hyperparameters with the validation data. Once we find an optimized learning rate, we do a final train of the model using both the training data and the validation data, and report precision on our withheld testing data.

4.2.1 C) Repurpose without HPO

This section demonstrates how to repurpose the source model to target data with default hyperparameters.

```
In [5]: # Default optimizer, learning rate and number of epochs used in Xfer to train neural network
        DEFAULT_OPTIMIZER = 'sgd'
        DEFAULT_OPTIMIZER_PARAMS = {'learning_rate': 0.001}
        DEFAULT_NUM_EPOCHS = 4

        TARGET_CLASS_COUNT = 4  # 4 classes of sketch images are used for this demo (car, cheese, hou
        CONTEXT_FUNCTION = mx.cpu # 'mx.gpu' or 'mx.cpu' (MXNet context function to train neural netw

        # Layers to freeze or randomly initialize during neural network training.
        # For this demo, we freeze the first 16 convolutional layers transferred from 'vgg19' model.
        FIXED_LAYERS = ['conv1_1','conv1_2','conv2_1','conv2_2','conv3_1','conv3_2','conv3_3','conv3_
                        'conv4_1','conv4_2','conv4_3','conv4_4','conv5_1','conv5_2','conv5_3','conv5_
        RANDOM_LAYERS = []

        # Method to repurpose neural network with given hyperparameters
        def train_and_predict(source_model, train_data_iterator, test_data_iterator, optimizer, optir
            set_random_seeds()
            repurposer = xfer.NeuralNetworkRandomFreezeRepurposer(source_model,
                                                                  optimizer=optimizer,
                                                                  optimizer_params=optimizer_params,
                                                                  target_class_count=TARGET_CLASS_COU
                                                                  fixed_layers=FIXED_LAYERS,
                                                                  random_layers=RANDOM_LAYERS,
                                                                  context_function=CONTEXT_FUNCTION,
                                                                  num_epochs=DEFAULT_NUM_EPOCHS)
            repurposer.repurpose(train_data_iterator)
            predictions = repurposer.predict_label(test_data_iterator)
            return predictions

        # Train neural network with default hyperparameters
        predictions = train_and_predict(source_model, train_validation_iterator, test_iterator,
                                        DEFAULT_OPTIMIZER, DEFAULT_OPTIMIZER_PARAMS)
        precision_default = np.mean(predictions == test_labels)
        print('Trained neural network with default hyperparameters. Precision: {}'.format(precision_d

        # Display test images and predictions
        test_images = get_images(test_iterator)
        show_predictions(predictions, test_images, id2label)
```

```
Trained neural network with default hyperparameters. Precision: 0.4375
```

Note that the quality of the results may vary due to randomness e.g. in the initialization of the neural network weights. However, the idea is that, since the precision varies with different choices of the learning rate and in certain cases the default learning rate might not be the best. We therefore wish to find a good learning rate automatically rather than with trial-and-error. HPO helps us achieve this, as demonstrated below.

4.2.2 D) Repurpose with HPO: Optimizing learning rate

4.3 i) Declare the hyperparameter to optimize and its domain

```
In [6]: # Learning rate is the hyperparameter we will optimize here
        # We allow GPyOpt to operate in a normalized domain [0,1] and map the value to a desired log
        # This helps GPyOpt to learn a smooth underlying function in fewer iterations
        gpyopt_domain = [{'name': 'learning_rate', 'type': 'continuous', 'domain': (0,1)}]

        # Method to map a value given by GPyOpt in [0, 1] to a desired range of learning rate
        def map_learning_rate(source_value):
            if(source_value < 0 or source_value > 1):
                raise ValueError('source_value must be in the range [0,1]')

            # We explore learning rate in the range [1e-6 , 1e-1]. You can choose a different range t
            # Log scale is used here because it is an intuitive way to explore learning rates
            # For example, if 1e-2 doesn't work, we tend to explore 1e-3 or 1e-1 which is a jump in l
            log_learning_rate_start = -6  # 1e-6 in linear scale
            log_learning_rate_end = -1  # 1e-1 in linear scale

            log_span = abs(log_learning_rate_end - log_learning_rate_start)
            log_mapped_value = log_learning_rate_start + (source_value * log_span)
            mapped_value = 10 ** log_mapped_value  # Convert from log scale to linear scale
            return mapped_value
```

4.4 ii) Define an objective function to optimize the hyperparameter

```
In [7]: def get_hyperparameters_from_config(config):
            """
            Extract hyperparameters from input configuration provided by GPyOpt.
            Refer the caller 'hpo_objective_function' for more details.
            """
            learning_rate = map_learning_rate(config[0]) # Map learning_rate value given by GPyOpt to
            optimizer = DEFAULT_OPTIMIZER  # Using default optimizer here i.e. 'sgd'
            return optimizer, learning_rate

        def hpo_objective_function(config_matrix):
            """
            Objective function to optimize the hyperparameters for
            This method is called by GPyOpt internally to get outputs of objective function for diffe

            We train a neural network with given hyperparameters and return (1-precision) on validati
            You can choose to optimize for a different measure and create the objective function acco
            Here, we consider one hyperparameter (learning_rate) to optimize precision

            Note: config_matrix has m rows and n columns
            m denotes the number of experiments to run i.e. each row would contain input configuratio
            n denotes the number of hyperparameters (e.g. 2 columns for learning_rate and batch_size)
            """
            # Output of objective function for each input configuration
            function_output = np.zeros((config_matrix.shape[0], 1))

            # For each input configuration, train a nerual network and calculate accuracy on validati
            for idx, config in enumerate(config_matrix):
                optimizer, learning_rate = get_hyperparameters_from_config(config)
```

```
            # Train neural network with the mapped learning rate and get predictions on validatio
            predictions = train_and_predict(source_model=source_model,
                                            train_data_iterator=train_iterator,
                                            test_data_iterator=validation_iterator,
                                            optimizer = optimizer,
                                            optimizer_params = {'learning_rate': learning_rate

            # Calculate precision on validation set and update function_output with (1-precision)
            precision = np.mean(predictions == validation_labels)
            function_output[idx][0] = (1.0 - precision)  # (1-precision) to keep a minimization o
            print('learning_rate: {}. optimizer: {}. precision: {}'.format(learning_rate, optimiz
            gc.collect()

        return function_output
```

4.5 iii) Initialize a Bayesian optimizer using GPyOpt

```
In [8]: set_random_seeds()
        NUM_INITIAL_POINTS = 2
        hyperparameter_optimizer = GPyOpt.methods.BayesianOptimization(f=hpo_objective_function,
                                                                       domain=gpyopt_domain,
                                                                       initial_design_numdata=NUM_INI
```

```
learning_rate: 9.069790423538591e-06. optimizer: sgd. precision: 0.75
learning_rate: 0.0012898638021921914. optimizer: sgd. precision: 0.25
```

4.6 iv) Run more iterations to identify a better learning rate for our objective function

```
In [9]: set_random_seeds()
        NUM_ITERATIONS_TO_RUN = 5
        hyperparameter_optimizer.run_optimization(max_iter = NUM_ITERATIONS_TO_RUN)
```

```
 /anaconda/envs/xfer_env/lib/python3.5/site-packages/paramz/parameterized.py:266: DeprecationWarning:
 /anaconda/envs/xfer_env/lib/python3.5/site-packages/paramz/parameterized.py:267: DeprecationWarning:
 /anaconda/envs/xfer_env/lib/python3.5/site-packages/paramz/core/parameter_core.py:290: DeprecationWa
 /anaconda/envs/xfer_env/lib/python3.5/site-packages/paramz/core/parameter_core.py:291: DeprecationWa
```

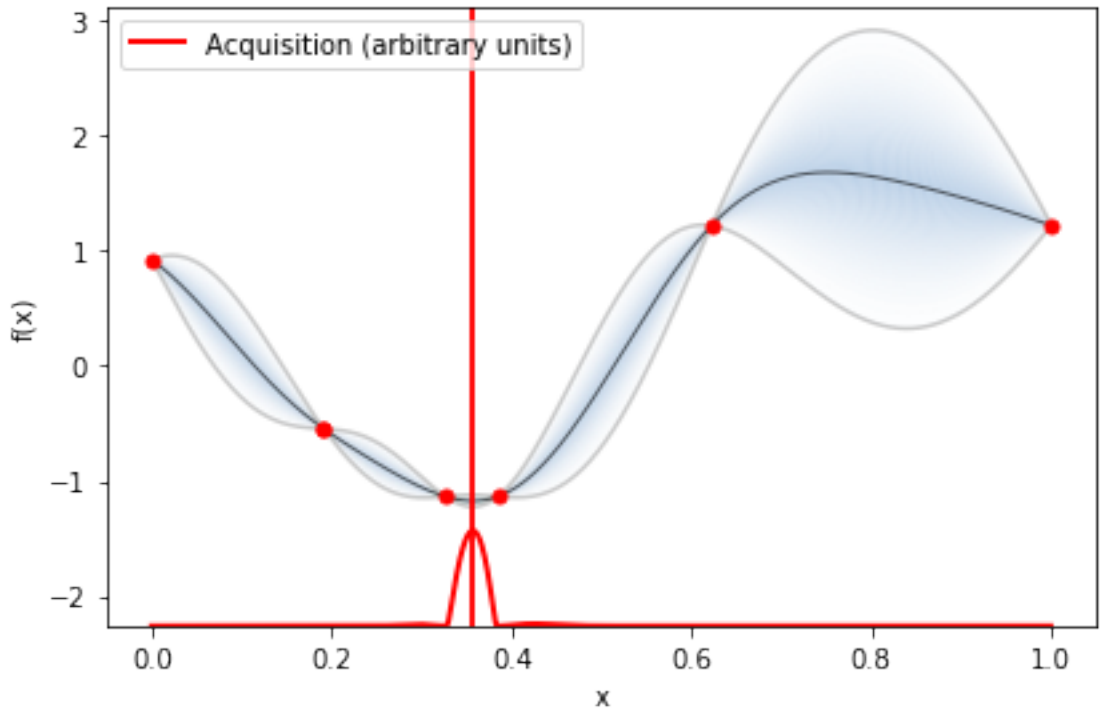
```
learning_rate: 9.013709494113845e-06. optimizer: sgd. precision: 0.75
learning_rate: 1e-06. optimizer: sgd. precision: 0.3333333333333333
learning_rate: 4.282331345714957e-05. optimizer: sgd. precision: 0.9166666666666666
learning_rate: 8.400871344074512e-05. optimizer: sgd. precision: 0.9166666666666666
learning_rate: 0.1. optimizer: sgd. precision: 0.25
```

4.7 v) Results

4.8 Learning rates evaluated

The plot below shows the different learning rates explored by GPyOpt in the normalized domain [0, 1].

```
In [10]: hyperparameter_optimizer.plot_acquisition()
```



4.9 Optimized learning rate

```
In [11]: # Take hyperparameters that minimized the objective function output
         x_best = hyperparameter_optimizer.X[np.argmin(hyperparameter_optimizer.Y)]
         optimized_learning_rate = map_learning_rate(x_best[0])
         precision = 1.0 - min(hyperparameter_optimizer.Y)[0]  # Objective was to minimize 1-precisio
         print('Optimized learning rate: {}. Precision on validation data: {}'.format(optimized_learn
```

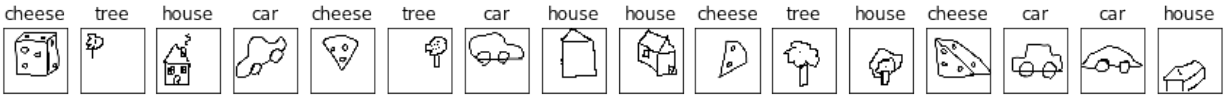
```
Optimized learning rate: 4.282331345714957e-05. Precision on validation data: 0.9166666666666666
```

Note that the optimized learning rate found is from the iterations run so far. Based on time available, one can run more iterations which may help in obtaining a more optimized learning rate.

4.10 Precision on test data with optimal learning rate

```
In [12]: # Train neural network with optimal learning rate and get predictions on test data
         # Along with training data, validation data is also used to train the final model
         predictions = train_and_predict(source_model=source_model,
                                         train_data_iterator=train_validation_iterator,  # train d
                                         test_data_iterator=test_iterator,  # predict on test data
                                         optimizer = DEFAULT_OPTIMIZER,
                                         optimizer_params = {'learning_rate': optimized_learning_r
         precision_optimized = np.mean(predictions == test_labels)
         print('Optimized learning rate: {}. Precision on test data: {}'.format(optimized_learning_ra
         show_predictions(predictions, test_images, id2label)
```

```
Optimized learning rate: 4.282331345714957e-05. Precision on test data: 0.9375
```



4.10.1 E) Repurpose with HPO: Optimizing multiple hyperparameters

The following section can be used for reference when someone wants to optimize multiple hyperparameters to repurpose models using Xfer. Here, the hyperparameters chosen are: 1. Optimizer for neural network (sgd or adam). 2. Learning rate.

Note that running more iterations could be useful here because there are more combination of values to explore.

```
# Choose the hyperparameters and specify the domain
optimizer_id_to_name = {1: 'sgd', 2:'adam'}
domain_with_2_hyperparams = [{'name': 'learning_rate', 'type': 'continuous', 'domain
↪': (0,1)},
                             {'name': 'optimizer', 'type': 'discrete', 'domain': (1,
↪2)}]

# Override this method to extract the optimizer in addition to learning_rate from␣
↪GPyOpt config
def get_hyperparameters_from_config(config):
    """
    Extract hyperparameters from input configuration provided by GPyOpt.
    Refer the caller 'hpo_objective_function' for more details.
    """
    learning_rate = map_learning_rate(config[0]) # Map learning_rate value given by␣
↪GPyOpt to the desire range
    optimizer = optimizer_id_to_name[config[1]]  # Using optimizer given by GPyOpt
    return optimizer, learning_rate

# Initialize GPyOpt with new domain and run optimization
set_random_seeds()
hyperparameter_optimizer2 = GPyOpt.methods.BayesianOptimization(f=hpo_objective_
↪function,
                                                                domain=domain_with_2_
↪hyperparams,
                                                                initial_design_
↪numdata=5)
hyperparameter_optimizer2.run_optimization(max_iter=5)

# Take hyperparameters that minimized the objective function output
x_best2 = hyperparameter_optimizer2.X[np.argmin(hyperparameter_optimizer2.Y)]
optimized_learning_rate2 = map_learning_rate(x_best2[0])
precision2 = 1.0 - min(hyperparameter_optimizer2.Y)[0]  # Objective was to minimize 1-
↪precision
print('Optimized learning rate: {}. Optimizer: {}. Precision on validation data: {}'
      .format(optimized_learning_rate2, optimizer_id_to_name[x_best2[1]], precision2))

# Train neural network with optimal (learning rate, optimizer) and get predictions on␣
↪test data
# Along with training data, validation data is also used to train the final model
predictions2 = train_and_predict(source_model=source_model,
                                 train_data_iterator=train_validation_iterator,  #␣
↪train on (train + validation) set
                                 test_data_iterator=test_iterator,  # predict on␣
↪test data
```

(continues on next page)

(continued from previous page)

```
                                    optimizer = DEFAULT_OPTIMIZER,
                                    optimizer_params = {'learning_rate': optimized_
↪learning_rate2})
precision_optimized2 = np.mean(predictions2 == test_labels)
print('Optimized learning rate: {}. Optimizer: {}. Precision on test data: {}'
      .format(optimized_learning_rate2, optimizer_id_to_name[x_best2[1]], precision_
↪optimized2))
show_predictions(predictions2, test_images, id2label)
```

In [ ]:

CHAPTER 5

Repurposing

Base Classes:

xfer.Repurposer	Base Class for repurposers that train models using Transfer Learning (source_model -> target_model).
xfer.MetaModelRepurposer	Base class for repurposers that extract features from layers in source neural network (Transfer) and train a meta-model using the extracted features (Learn).
xfer.NeuralNetworkRepurposer	Base class for repurposers that create a target neural network from a source neural network through Transfer Learning.

5.1 xfer.Repurposer

class xfer.**Repurposer**(*source_model: mxnet.module.module.Module*, *context_function=<function cpu>*, *num_devices=1*)

Bases: object

Base Class for repurposers that train models using Transfer Learning (source_model -> target_model).

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer learning from.
- **context_function** (function(int)->:class:*mx.context.Context*) – MXNet context function that provides device type context.
- **num_devices** (*int*) – Number of devices to use with context_function.

Methods

`__init__`	Initialize self.
deserialize	Abstract method.
get_params	Get parameters of repurposer that are in the constructor's argument list.
predict_label	Abstract method.
predict_probability	Abstract method.
repurpose	Abstract method.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Abstract method.

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Abstract method.

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Abstract method.

predict_label(*test_iterator: mxnet.io.io.DataIter*)
Abstract method.

get_params()
Get parameters of repurposer that are in the constructor's argument list.

Return type dict

serialize(*file_prefix*)
Abstract method.

deserialize(*input_dict*)
Abstract method.

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

5.2 xfer.MetaModelRepurposer

class xfer.**MetaModelRepurposer**(*source_model: mxnet.module.module.Module*, *feature_layer_names*, *context_function=<function cpu>*, *num_devices=1*)
Bases: xfer.repurposer.Repurposer

Base class for repurposers that extract features from layers in source neural network (Transfer) and train a meta-model using the extracted features (Learn).

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer learning from.
- **feature_layer_names** (*list[str]*) – Name of layer(s) in source_model from which features should be transferred.
- **context_function** (function(int)->:class:*mx.context.Context*) – MXNet context function that provides device type context.
- **num_devices** (*int*) – Number of devices to use to extract features from source_model.

Methods

__init__	Initialize self.
deserialize	Abstract method.
get_features_from_source_model	Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.
get_params	Get parameters of repurposer that are in the constructor's argument list.
predict_label	Predict class labels on test data using the target_model (repurposed meta-model).
predict_probability	Predict class probabilities on test data using the target_model (repurposed meta-model).
repurpose	Train a meta-model using features extracted from training data through the source neural network.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Abstract method.

Attributes

feature_layer_names	Names of the layers to extract features from.
source_model	Model to extract features from.

get_params()
Get parameters of repurposer that are in the constructor's argument list.

Return type dict

feature_layer_names
Names of the layers to extract features from.

source_model
Model to extract features from.

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Train a meta-model using features extracted from training data through the source neural network.

Set self.target_model to the trained meta-model.

Parameters **train_iterator** – Training data iterator to use to extract features from source_model.

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Predict class probabilities on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted probabilities.

Return type numpy.ndarray

predict_label(*test_iterator: mxnet.io.io.DataIter*)
Predict class labels on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted labels.

Return type numpy.ndarray

get_features_from_source_model(*data_iterator: mxnet.io.io.DataIter*)
Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.

In addition, return mapping of feature_layer_name to indices in feature array.

Parameters **data_iterator** (mxnet.io.DataIter) – Iterator for data to be passed through the source network and extract features.

Returns features, feature_indices_per_layer and labels.

Return type MetaModelData

deserialize(*input_dict*)
Abstract method.

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

serialize(*file_prefix*)
Abstract method.

5.3 xfer.NeuralNetworkRepurposer

class xfer.**NeuralNetworkRepurposer**(*source_model: mxnet.module.module.Module*, *context_function=<function cpu>*, *num_devices=1*, *batch_size=64*, *num_epochs=5*, *optimizer='sgd'*, *optimizer_params=None*)
Bases: xfer.repurposer.Repurposer

Base class for repurposers that create a target neural network from a source neural network through Transfer Learning.

- Transfer layer architecture and weights from a source neural network (Transfer) and
- Train a target neural network adapting the transferred network to new data set (Learn)

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer leaning from
- **context_function** (*function*) – MXNet context function that provides device type context
- **num_devices** (*int*) – Number of devices to use to train target neural network
- **batch_size** (*int*) – Size of data batches to be used for training the target neural network
- **num_epochs** (*int*) – Number of epochs to be used for training the target neural network
- **optimizer** (*str*) – Optimizer required by MXNet to train target neural network. Default: 'sgd'
- **optimizer_params** (*dict(str, float)*) – Optimizer params required by MXNet to train target neural network. Default: {'learning_rate': 1e-3}

Methods

__init__	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.
get_params	Get parameters of repurposer that are in the constructor
predict_label	Perform predictions on test data using the target_model (repurposed neural network).
predict_probability	Perform predictions on test data using the target_model (repurposed neural network).
repurpose	Train a neural network by transferring layers/weights from source_model.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Serialize repurposer to dictionary.

get_params()
Get parameters of repurposer that are in the constructor

Return type dict

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Train a neural network by transferring layers/weights from source_model. Set self.target_model to the repurposed neural network.

Parameters **train_iterator** (mxnet.io.DataIter) – Training data iterator to use to extract features from source_model

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Perform predictions on test data using the target_model (repurposed neural network).

Parameters **test_iterator** (mxnet.io.DataIter) – Test data iterator to return predictions for

Returns Predicted probabilities

Return type numpy.ndarray

predict_label (*test_iterator: mxnet.io.io.DataIter*)
Perform predictions on test data using the target_model (repurposed neural network).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for

Returns Predicted labels

Return type numpy.ndarray

serialize (*file_prefix*)
Serialize repurposer to dictionary.

Returns Dictionary describing repurposer

Return type dict

deserialize (*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to

save_repurposer (*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

Repurposers:

xfer.LrRepurposer	Perform Transfer Learning through a Logistic Regression meta-model which repurposes the source neural network.
xfer.SvmRepurposer	Perform Transfer Learning through a Support Vector Machine (SVM) meta-model which repurposes the source neural network.
xfer.GpRepurposer	Repurpose source neural network to create a Gaussian Process (GP) meta-model through Transfer Learning.
xfer.BnnRepurposer	Perform Transfer Learning through a Bayesian Neural Network (BNN) meta-model which repurposes the source neural network.
xfer.NeuralNetworkFineTuneRepurposer	Class that creates a target neural network from a source neural network through Transfer Learning.
xfer.NeuralNetworkRandomFreezeRepurposer	Class that creates a target neural network from a source neural network through Transfer Learning.

5.4 xfer.LrRepurposer

class xfer.**LrRepurposer**(*source_model: mxnet.module.module.Module*, *feature_layer_names*, *context_function=<function cpu>*, *num_devices=1*, *tol=0.0001*, *c=1.0*, *n_jobs=-1*)

Bases: xfer.meta_model_repurposer.MetaModelRepurposer

Perform Transfer Learning through a Logistic Regression meta-model which repurposes the source neural network.

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do Transfer Learning from.
- **feature_layer_names** (*list[str]*) – Name of layer(s) in source_model from which features should be transferred.
- **context_function** (function(int)->:class:*mx.context.Context*) – MXnet context function that provides device type context.
- **num_devices** (*int*) – Number of devices to use to extract features from source_model.
- **tol** (*float*) – Tolerance for stopping criteria.
- **c** (*float*) – Inverse of regularization strength; must be a positive float.
- **n_jobs** (*int*) – Number of CPU cores used during the cross-validation loop. If given a value of -1, all cores are used.

Methods

__init__	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.
get_features_from_source_model	Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.
get_params	Get parameters of repurposer that are in the constructor.
predict_label	Predict class labels on test data using the target_model (repurposed meta-model).
predict_probability	Predict class probabilities on test data using the target_model (repurposed meta-model).
repurpose	Train a meta-model using features extracted from training data through the source neural network.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Saves repurposer (excluding source model) to file_prefix.json.

Attributes

feature_layer_names	Names of the layers to extract features from.
source_model	Model to extract features from.

get_params()
Get parameters of repurposer that are in the constructor.

Return type dict

serialize(*file_prefix*)
Saves repurposer (excluding source model) to file_prefix.json.

Parameters **file_prefix** (*str*) – Prefix to save file with.

deserialize(*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to.

feature_layer_names
Names of the layers to extract features from.

get_features_from_source_model(*data_iterator: mxnet.io.io.DataIter*)
Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.

In addition, return mapping of feature_layer_name to indices in feature array.

Parameters **data_iterator** (mxnet.io.DataIter) – Iterator for data to be passed through the source network and extract features.

Returns features, feature_indices_per_layer and labels.

Return type MetaModelData

predict_label(*test_iterator: mxnet.io.io.DataIter*)
Predict class labels on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted labels.

Return type numpy.ndarray

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Predict class probabilities on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted probabilities.

Return type numpy.ndarray

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Train a meta-model using features extracted from training data through the source neural network.

Set self.target_model to the trained meta-model.

Parameters **train_iterator** – Training data iterator to use to extract features from source_model.

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

source_model
Model to extract features from.

5.5 xfer.SvmRepurposer

class xfer.**SvmRepurposer**(*source_model: mxnet.module.module.Module*, *feature_layer_names*, *context_function=<function cpu>*, *num_devices=1*, *c=1.0*, *kernel='linear'*, *gamma='auto'*, *enable_probability_estimates=False*)
Bases: xfer.meta_model_repurposer.MetaModelRepurposer

Perform Transfer Learning through a Support Vector Machine (SVM) meta-model which repurposes the source neural network.

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer learning from.
- **feature_layer_names** (*list[str]*) – Name of layer(s) in source_model from which features should be transferred.
- **context_function** (function(int)->:class:*mx.context.Context*) – MXNet context function that provides device type context.
- **num_devices** (*int*) – Number of devices to use to extract features from source_model.
- **c** (*float*) – Penalty parameter C of the error term.
- **kernel** (*string*) – Specifies the kernel type to be used in the SVM algorithm in sklearn library. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.
- **gamma** (*float*) – Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma is 'auto' then 1/n_features will be used instead.
- **enable_probability_estimates** (*bool*) – Whether to enable probability estimates. This must be enabled for predict_probability to work and will slow down training.

Methods

__init__	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.

Continued on next page

Table 9 – continued from previous page

get_features_from_source_model	Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.
get_params	Get parameters of repurposer that are in the constructor.
predict_label	Predict class labels on test data using the target_model (repurposed meta-model).
predict_probability	Predict class probabilities on test data using the target_model (repurposed meta-model).
repurpose	Train a meta-model using features extracted from training data through the source neural network.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Saves repurposer (excluding source model) to file_prefix.json.

Attributes

feature_layer_names	Names of the layers to extract features from.
source_model	Model to extract features from.

get_params()
Get parameters of repurposer that are in the constructor.

Return type dict

serialize(*file_prefix*)
Saves repurposer (excluding source model) to file_prefix.json.

Parameters **file_prefix** (*str*) – Prefix to save file with.

deserialize(*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to.

feature_layer_names
Names of the layers to extract features from.

get_features_from_source_model(*data_iterator: mxnet.io.io.DataIter*)
Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.

In addition, return mapping of feature_layer_name to indices in feature array.

Parameters **data_iterator** (mxnet.io.DataIter) – Iterator for data to be passed through the source network and extract features.

Returns features, feature_indices_per_layer and labels.

Return type MetaModelData

predict_label(*test_iterator: mxnet.io.io.DataIter*)
Predict class labels on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted labels.

Return type numpy.ndarray

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Predict class probabilities on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted probabilities.

Return type numpy.ndarray

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Train a meta-model using features extracted from training data through the source neural network.

Set self.target_model to the trained meta-model.

Parameters **train_iterator** – Training data iterator to use to extract features from source_model.

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

source_model
Model to extract features from.

5.6 xfer.GpRepurposer

class xfer.**GpRepurposer**(*source_model: mxnet.module.module.Module*, *feature_layer_names*, *context_function=<function cpu>*, *num_devices=1*, *max_function_evaluations=100*, *apply_l2_norm=False*)
Bases: xfer.meta_model_repurposer.MetaModelRepurposer

Repurpose source neural network to create a Gaussian Process (GP) meta-model through Transfer Learning.

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer learning from.
- **feature_layer_names** (*list[str]*) – Name of layer(s) in source_model from which features should be transferred.
- **context_function** (function(int)->:class:*mx.context.Context*) – MXNet context function that provides device type context.
- **num_devices** (*int*) – Number of devices to use to extract features from source_model.
- **max_function_evaluations** (*int*) – Maximum number of function evaluations to perform in GP optimization.

- **apply_l2_norm** (*bool*) – Whether to apply L2 normalization after extracting features from source neural network. If set to True, L2 normalization will be applied to features before passing to GP during training and prediction.

Methods

`__init__`	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.
get_features_from_source_model	Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.
get_params	Get parameters of repurposer that are in the constructor's argument list.
predict_label	Predict class labels on test data using the target_model (repurposed meta-model).
predict_probability	Predict class probabilities on test data using the target_model (repurposed meta-model).
repurpose	Train a meta-model using features extracted from training data through the source neural network.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Serialize the GP repurposer (model and supporting info) and save to file.

Attributes

feature_layer_names	Names of the layers to extract features from.
source_model	Model to extract features from.

get_params()
Get parameters of repurposer that are in the constructor's argument list.

Return type dict

serialize(*file_prefix*)
Serialize the GP repurposer (model and supporting info) and save to file.

Parameters **file_prefix** (*str*) – Prefix of file path to save the serialized repurposer to.

Returns None

deserialize(*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to.

Returns None

feature_layer_names
Names of the layers to extract features from.

get_features_from_source_model(*data_iterator: mxnet.io.io.DataIter*)
Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.

In addition, return mapping of feature_layer_name to indices in feature array.

Parameters **data_iterator** (mxnet.io.DataIter) – Iterator for data to be passed through the source network and extract features.

Returns features, feature_indices_per_layer and labels.

Return type MetaModelData

predict_label(*test_iterator: mxnet.io.io.DataIter*)

Predict class labels on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted labels.

Return type numpy.ndarray

predict_probability(*test_iterator: mxnet.io.io.DataIter*)

Predict class probabilities on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted probabilities.

Return type numpy.ndarray

repurpose(*train_iterator: mxnet.io.io.DataIter*)

Train a meta-model using features extracted from training data through the source neural network.

Set self.target_model to the trained meta-model.

Parameters **train_iterator** – Training data iterator to use to extract features from source_model.

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)

Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

source_model

Model to extract features from.

5.7 xfer.BnnRepurposer

class xfer.**BnnRepurposer**(*source_model: mxnet.module.module.Module*, *feature_layer_names*, *context_function=<function cpu>*, *num_devices=1*, *bnn_context_function=<function cpu>*, *sigma=100.0*, *num_layers=1*, *n_hidden=10*, *num_samples_mc=3*, *learning_rate=0.001*, *batch_size=20*, *num_epochs=200*, *start_annealing=None*, *end_annealing=None*, *num_samples_mc_prediction=100*, *verbose=0*)

Bases: xfer.meta_model_repurposer.MetaModelRepurposer

Perform Transfer Learning through a Bayesian Neural Network (BNN) meta-model which repurposes the source neural network.

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer learning from.
- **feature_layer_names** (*list[str]*) – Name of layer(s) in source_model from which features should be transferred.
- **context_function** (function(int)->:class:*mx.context.Context*) – MXNet context function that provides device type context. It is used to extract features with the source model.
- **num_devices** (*int*) – Number of devices to use to extract features from source_model.
- **bnn_context_function** (function(int)->:class:*mx.context.Context*) – MXNet context function used to train the BNN.
- **sigma** (*float*) – Standard deviation of the Gaussian prior used for the weights of the BNN meta-model (w_i sim N(0, sigma^2)).
- **num_layers** (*int*) – Number of layers of the BNN meta-model.
- **n_hidden** (*int*) – Dimensionality of the hidden layers of the BNN meta-model (all hidden layers have the same dimensionality).
- **num_samples_mc** (*int*) – Number of samples used for the Monte Carlo approximation of the variational bound.
- **learning_rate** (*float*) – Learning rate for the BNN meta-model training.
- **batch_size** (*int*) – Mini-batch size for the BNN meta-model training.
- **num_epochs** (*int*) – Number of epochs for the BNN meta-model training.
- **start_annealing** (*int*) – To help the training of the BNN meta-model, we anneal the KL term using a weight that varies fromm zero to one. start_annealing determines the epoch at which the annealing weight start to increase linearly until it reaches one in the epoch given by end_annealing.
- **end_annealing** (*int*) – Determines the epoch at which the annealing process of the KL term ends.
- **step_annealing_sample_weight** (*float*) – Amount that the annealing weight is incremented in each epoch (from start_annealing to end_annealing).
- **num_samples_mc_prediction** (*int*) – Number of Monte Carlo samples to use on prediction.
- **verbose** (*bool*) – Flag to control whether accuracy monitoring is logged during repurposing.

- **annealing_weight** (*float*) – Annealing weight in the current epoch.
- **train_acc** (*list[float]*) – Accuracy in training set in each epoch.
- **test_acc** (*list[float]*) – Accuracy in validation set in each epoch.
- **moving_loss_total** (*list[float]*) – Total loss (negative ELBO) smoothed across epochs.
- **average_loss** (*list[float]*) – Average loss (negative ELBO) per data point.
- **anneal_weights** (*list[float]*) – Annealing weight used in each epoch.

Methods

`__init__`	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.
get_features_from_source_model	Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.
get_params	Get parameters of repurposer that are in the constructor.
predict_label	Predict class labels on test data using the target_model (repurposed meta-model).
predict_probability	Predict class probabilities on test data using the target_model (repurposed meta-model).
repurpose	Train a meta-model using features extracted from training data through the source neural network.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Saves repurposer (excluding source model) to file_prefix.json, file_prefix_posterior.json, file_prefix_posterior_params.npz.

Attributes

end_annealing	Determines the epoch at which the annealing process of the KL term ends.
feature_layer_names	Names of the layers to extract features from.
source_model	Model to extract features from.
start_annealing	Determines the epoch at which the annealing process of the KL term starts.

get_params()
Get parameters of repurposer that are in the constructor.

Return type dict

start_annealing
Determines the epoch at which the annealing process of the KL term starts.

end_annealing
Determines the epoch at which the annealing process of the KL term ends.

feature_layer_names
Names of the layers to extract features from.

get_features_from_source_model (*data_iterator: mxnet.io.io.DataIter*)
Extract feature outputs from feature_layer_names in source_model, merge and return all features and labels.

In addition, return mapping of feature_layer_name to indices in feature array.

Parameters **data_iterator** (mxnet.io.DataIter) – Iterator for data to be passed through the source network and extract features.

Returns features, feature_indices_per_layer and labels.

Return type MetaModelData

predict_label (*test_iterator: mxnet.io.io.DataIter*)
Predict class labels on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted labels.

Return type numpy.ndarray

predict_probability (*test_iterator: mxnet.io.io.DataIter*)
Predict class probabilities on test data using the target_model (repurposed meta-model).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for.

Returns Predicted probabilities.

Return type numpy.ndarray

repurpose (*train_iterator: mxnet.io.io.DataIter*)
Train a meta-model using features extracted from training data through the source neural network.

Set self.target_model to the trained meta-model.

Parameters **train_iterator** – Training data iterator to use to extract features from source_model.

save_repurposer (*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

source_model
Model to extract features from.

serialize (*file_prefix*)
Saves repurposer (excluding source model) to file_prefix.json, file_prefix_posterior.json, file_prefix_posterior_params.npz.

Parameters **file_prefix** (*str*) – Prefix to save file with.

deserialize(*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to.

5.8 xfer.NeuralNetworkFineTuneRepurposer

class xfer.**NeuralNetworkFineTuneRepurposer**(*source_model: mxnet.module.module.Module*, *transfer_layer_name*, *target_class_count*, *context_function=<function cpu>*, *num_devices=1*, *batch_size=64*, *num_epochs=5*, *optimizer='sgd'*, *optimizer_params=None*)
Bases: xfer.neural_network_repurposer.NeuralNetworkRepurposer

Class that creates a target neural network from a source neural network through Transfer Learning. It transfers layers from source model and fine-tunes the network to learn from target data set.

Steps involved:

- Layers and weights of source model are transferred from input layer up to and including layer 'transfer_layer_name'
- A fully connected layer with nodes equal to number of classes in target data set is added after the transfer layer
- A softmax layer is added on top of the new fully connected layer
- This modified network represents the target model and is fine tuned to adapt weights to the target data set

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer leaning from. Can be None in a predict-only case
- **transfer_layer_name** (*str*) – Name of layer up to which layers/weights in source_model need to be transferred. For example, name of layer before the last fully connected layer in source network. Can be None in a predict-only case
- **target_class_count** (*int*) – Number of classes to train the target neural network for. Can be None in a predict-only case
- **context_function** (*function*) – MXNet context function that provides device type context
- **num_devices** (*int*) – Number of devices to use to train target neural network
- **batch_size** (*int*) – Size of data batches to be used for training the target neural network
- **num_epochs** (*int*) – Number of epochs to be used for training the target neural network
- **optimizer** (*str*) – Optimizer required by MXNet to train target neural network. Default: 'sgd'
- **optimizer_params** (*dict(str, float)*) – Optimizer params required by MXNet to train target neural network. Default: {'learning_rate': 1e-3}

Methods

`__init__`	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.
get_params	Get parameters of repurposer that are in the constructor
predict_label	Perform predictions on test data using the target_model (repurposed neural network).
predict_probability	Perform predictions on test data using the target_model (repurposed neural network).
repurpose	Train a neural network by transferring layers/weights from source_model.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Serialize repurposer to dictionary.

get_params()
Get parameters of repurposer that are in the constructor

Return type dict

deserialize(*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to

predict_label(*test_iterator: mxnet.io.io.DataIter*)
Perform predictions on test data using the target_model (repurposed neural network).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for

Returns Predicted labels

Return type numpy.ndarray

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Perform predictions on test data using the target_model (repurposed neural network).

Parameters **test_iterator** (mxnet.io.DataIter) – Test data iterator to return predictions for

Returns Predicted probabilities

Return type numpy.ndarray

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Train a neural network by transferring layers/weights from source_model. Set self.target_model to the repurposed neural network.

Parameters **train_iterator** (mxnet.io.DataIter) – Training data iterator to use to extract features from source_model

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.

- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

serialize(*file_prefix*)
Serialize repurposer to dictionary.

Returns Dictionary describing repurposer

Return type dict

5.9 xfer.NeuralNetworkRandomFreezeRepurposer

class xfer.**NeuralNetworkRandomFreezeRepurposer**(*source_model: mxnet.module.module.Module*, *target_class_count*, *fixed_layers*, *random_layers*, *num_layers_to_drop=2*, *context_function=<function cpu>*, *num_devices=1*, *batch_size=64*, *num_epochs=5*, *optimizer='sgd'*, *optimizer_params=None*)
Bases: xfer.neural_network_repurposer.NeuralNetworkRepurposer

Class that creates a target neural network from a source neural network through Transfer Learning. It transfers layers from source model and fine-tunes the network to learn from target data set.

Steps involved:

- Layers and weights of source model are transferred
- N layers are removed from the output of the model
- A fully connected layer with nodes equal to number of classes in target data set is added to the model output
- A softmax layer is added on top of the new fully connected layer
- Some layers are frozen and some randomly initialized and then model is fine-tuned

Parameters

- **source_model** (mxnet.mod.Module) – Source neural network to do transfer leaning from. Can be None in a predict-only case
- **target_class_count** (*int*) – Number of classes to train the target neural network for. Can be None in a predict-only case
- **fixed_layers** (*list[str]*) – List of layers to keep weights frozen for
- **random_layers** (*list[str]*) – List of layers to randomly reinitialise
- **num_layers_to_drop** (*int*) – Number of layers to remove from model output
- **context_function** (*function*) – MXNet context function that provides device type context
- **num_devices** (*int*) – Number of devices to use to train target neural network
- **batch_size** (*int*) – Size of data batches to be used for training the target neural network
- **num_epochs** (*int*) – Number of epochs to be used for training the target neural network

- **optimizer** (*str*) – Optimizer required by MXNet to train target neural network. Default: 'sgd'
- **optimizer_params** (*dict(str, float)*) – Optimizer params required by MXNet to train target neural network. Default: {'learning_rate': 1e-3}

Methods

`__init__`	Initialize self.
deserialize	Uses dictionary to set attributes of repurposer.
get_params	Get parameters of repurposer that are in the constructor
predict_label	Perform predictions on test data using the target_model (repurposed neural network).
predict_probability	Perform predictions on test data using the target_model (repurposed neural network).
repurpose	Train a neural network by transferring layers/weights from source_model.
save_repurposer	Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.
serialize	Serialize repurposer to dictionary.

get_params()
Get parameters of repurposer that are in the constructor

Return type dict

deserialize(*input_dict*)
Uses dictionary to set attributes of repurposer.

Parameters **input_dict** (*dict*) – Dictionary containing values for attributes to be set to

predict_label(*test_iterator: mxnet.io.io.DataIter*)
Perform predictions on test data using the target_model (repurposed neural network).

Parameters **test_iterator** (*mxnet.io.DataIter*) – Test data iterator to return predictions for

Returns Predicted labels

Return type numpy.ndarray

predict_probability(*test_iterator: mxnet.io.io.DataIter*)
Perform predictions on test data using the target_model (repurposed neural network).

Parameters **test_iterator** (mxnet.io.DataIter) – Test data iterator to return predictions for

Returns Predicted probabilities

Return type numpy.ndarray

repurpose(*train_iterator: mxnet.io.io.DataIter*)
Train a neural network by transferring layers/weights from source_model. Set self.target_model to the repurposed neural network.

Parameters **train_iterator** (mxnet.io.DataIter) – Training data iterator to use to extract features from source_model

save_repurposer(*model_name*, *model_directory=''*, *save_source_model=None*)
Serialize the repurposed model (source_model, target_model and supporting info) and save it to given file_path.

Parameters

- **model_name** (*str*) – Name to save repurposer to.
- **model_directory** (*str*) – File directory to save repurposer in.
- **save_source_model** (*boolean*) – Flag to choose whether to save repurposer source model. Will use default if set to None. (MetaModelRepurposer default: True, NeuralNetworkRepurposer default: False)

serialize(*file_prefix*)
Serialize repurposer to dictionary.

Returns Dictionary describing repurposer

Return type dict

CHAPTER 6

Model Handler

xfer.model_handler.ModelHandler	Class for model manipulation and feature extraction.
xfer.model_handler.exceptions	Exceptions for Model Handler.
xfer.model_handler.consts	Model Handler constants.

6.1 xfer.model_handler.ModelHandler

class xfer.model_handler.**ModelHandler**(*module*, *context_function=<function cpu>*, *num_devices=1*, *data_name='data'*)

Bases: object

Class for model manipulation and feature extraction.

Parameters

- **module** (mx.module.Module) – MXNet module to be manipulated.
- **context_function** (*function*) – MXNet context function.
- **num_devices** (*int*) – Number of devices to run process on.
- **data_name** (*str*) – Name of input layer of model.

Methods

__init__	Initialize self.
add_layer_bottom	Add layer to input of model.
add_layer_top	Add layer to output of model.
drop_layer_bottom	Remove layers from input of model.
drop_layer_top	Remove layers from output of model.
get_layer_names_matching_type	Return names of layers of specified type.

Continued on next page

Table 2 – continued from previous page

get_layer_output	Function to extract features from data iterator with model.
get_layer_parameters	Get list of layer parameters associated with the the layer names given.
get_layer_type	Return type of named layer.
get_module	Return MXNet Module using the model symbol and parameters.
save_symbol	Serialise model symbol graph.
update_sym	Update symbol attribute, layer names, and layer types dict and clean parameters.
visualize_net	Display computational graph of model.

Attributes

layer_names	Get list of names of model layers.

drop_layer_top(*num_layers_to_drop=1*)
Remove layers from output of model.

Parameters **n** (*int*) – Number of layers to remove from model output.

drop_layer_bottom(*num_layers_to_drop=1*)
Remove layers from input of model.

Parameters **n** (*int*) – Number of layers to remove from model input.

add_layer_top(*layer_list*)
Add layer to output of model. model layers = (layer1, layer2, layer3), layer_list = [layerA, layerB] -> model layers = (layer1, layer2, layer3, layerA, layerB)

Parameters **layer_list** (list(`mx.symbol`)) – List of MxNet symbol layers to be added to model output.

add_layer_bottom(*layer_list*)
Add layer to input of model. model layers = (layer1, layer2, layer3), layer_list = [layerA, layerB] -> model layers = (layerA, layerB, layer1, layer2, layer3)

Parameters **layer_list** (list(`mx.symbol`)) – List of MxNet symbol layers to be added to model input.

get_module(*iterator*, *fixed_layer_parameters=None*, *random_layer_parameters=None*)
Return MXNet Module using the model symbol and parameters.

Parameters

- **iterator** (`mxnet.io.DataIter`) – MXNet iterator to be used with model.
- **fixed_layer_parameters** (*list(str)*) – List of layer parameters to keep fixed.
- **random_layer_parameters** (*list(str)*) – List of layer parameters to randomise.

Returns MXNet module

Return type `mx.module.Module`

get_layer_type(*layer_name*)
Return type of named layer.

Parameters **name** (*str*) – Name of layer being inspected.

Returns Layer type

Return type str

get_layer_names_matching_type(*layer_type*)
Return names of layers of specified type.

Parameters **layer_type** (*str*) – Return list of layers of this type.

Returns Names of layers with specified type

Return type list(str)

get_layer_output(*data_iterator*, *layer_names*)
Function to extract features from data iterator with model. Returns a dictionary of layer_name -> numpy array of features extracted (flattened).

Parameters

- **data_iterator** (mxnet.io.DataIter) – Iterator containing input data.
- **layer_names** (*list(str)*) – List of names of layers to extract features from.

Returns Ordered Dictionary of features ({layer_name: features}), list of labels Layer names in the ordered dictionary follow the same order as input list of layer_names

Return type OrderedDict[str, numpy.array], list(int)

get_layer_parameters(*layer_names*)
Get list of layer parameters associated with the the layer names given.

Parameters **layer_names** (*list(str)*) – List of layer names.

Returns List of layer parameters

Return type list(str)

visualize_net()
Display computational graph of model.

save_symbol(*model_name*)
Serialise model symbol graph.

Parameters **model_name** (*str*) – Prefix to file name (model_name-symbol.json).

layer_names
Get list of names of model layers.

Returns List of layer names

Return type list[str]

update_sym(*new_symbol*)
Update symbol attribute, layer names, and layer types dict and clean parameters.

Parameters **new_symbol** (mx.symbol.Symbol) – Symbol with which to update ModelHandler.

6.2 xfer.model_handler.exceptions

Exceptions for Model Handler.

Exceptions

ModelArchitectureError	Exception type for errors caused when model architecture is incorrect or mismatched
ModelError	Exception type for errors caused by invalid model actions.

exception xfer.model_handler.exceptions.**ModelError**
Bases: Exception

Exception type for errors caused by invalid model actions.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception xfer.model_handler.exceptions.**ModelArchitectureError**
Bases: Exception

Exception type for errors caused when model architecture is incorrect or mismatched

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

6.3 xfer.model_handler.consts

Model Handler constants.

Classes

LayerType	An enumeration.

class xfer.model_handler.consts.**LayerType**
Bases: enum.Enum

An enumeration.

FULLYCONNECTED = 'FullyConnected'

CONVOLUTION = 'Convolution'

ACTIVATION = 'Activation'

POOLING = 'Pooling'

FLATTEN = 'Flatten'

SOFTMAXOUTPUT = 'SoftmaxOutput'

DROPOUT = 'Dropout'

CONCAT = 'Concat'

BATCHNORM = 'BatchNorm'

```
PLUS = '_Plus'
SVMOUTPUT = 'SVMOutput'
LRN = 'LRN'
DATA = 'Data'
MEAN = 'mean'
EMBEDDING = 'Embedding'
```

CHAPTER 7

Writing a custom Repurposer

Xfer implements and supports two kinds of Repurposers:

- **Meta-model Repurposer** - this uses the source model to extract features and then fits a meta-model to the features
- **Neural network Repurposer** - this modifies the source model to create a target model

Below are examples of creating custom Repurposers for both classes

7.1 Setup

First import relevant modules, define data iterators and load a source model

```
In [1]: import warnings
        warnings.filterwarnings("ignore")

        import logging
        logging.disable(logging.WARNING)

        import xfer

        import os
        import glob
        import mxnet as mx
        import random
        from sklearn.model_selection import StratifiedShuffleSplit
        from sklearn.metrics import classification_report

        random.seed(1)
```

```
In [2]: def get_iterators_from_folder(data_dir, train_size=0.6, batchsize=10, label_name='softmax_lab
            """
            Method to create iterators from data stored in a folder with the following structure:
            /data_dir
                /class1
```

```
                class1_img1
                class1_img2
                ...
                class1_imgN
            /class2
                class2_img1
                class2_img2
                ...
                class2_imgN
            ...
            /classN
        """
        # assert dir exists
        if not os.path.isdir(data_dir):
            raise ValueError('Directory not found: {}'.format(data_dir))
        # get class names
        classes = [x.split('/')[-1] for x in glob.glob(data_dir+'/*')]
        classes.sort()
        fnames = []
        labels = []
        for c in classes:
                # get all the image filenames and labels
                images = glob.glob(data_dir+'/'+c+'/*')
                images.sort()
                fnames += images
                labels += [c]*len(images)
        # create label2id mapping
        id2label = dict(enumerate(set(labels)))
        label2id = dict((v,k) for k, v in id2label.items())

        # get indices of train and test
        sss = StratifiedShuffleSplit(n_splits=2, test_size=None, train_size=train_size, random_st
        train_indices, test_indices = next(sss.split(labels, labels))

        train_img_list = []
        test_img_list = []
        train_labels = []
        test_labels = []
        # create imglist for training and test
        for idx in train_indices:
            train_img_list.append([label2id[labels[idx]], fnames[idx]])
            train_labels.append(label2id[labels[idx]])
        for idx in test_indices:
            test_img_list.append([label2id[labels[idx]], fnames[idx]])
            test_labels.append(label2id[labels[idx]])

        # make iterators
        train_iterator = mx.image.ImageIter(batchsize, (3,224,224), imglist=train_img_list, label
                                            path_root='')
        test_iterator = mx.image.ImageIter(batchsize, (3,224,224), imglist=test_img_list, label_n
                                           path_root='')

        return train_iterator, test_iterator, train_labels, test_labels, id2label, label2id
```

In [3]:
```
dataset = 'test_images' # options are: 'test_sketches', 'test_images_sketch', 'mnist-50', 'te
num_classes = 4

train_iterator, test_iterator, train_labels, test_labels, id2label, label2id = get_iterators_
```

In [4]:
```
# Download vgg19 (trained on imagenet)
```

```
        path = 'http://data.mxnet.io/models/imagenet/'
        [mx.test_utils.download(path+'vgg/vgg19-0000.params'),
        mx.test_utils.download(path+'vgg/vgg19-symbol.json')]
```

```
Out[4]: ['vgg19-0000.params', 'vgg19-symbol.json']
```

```
In [5]: # This will be the source model we use for repurposing later
        source_model = mx.module.Module.load('vgg19', 0, label_names=['prob_label'])
```

7.2 Custom Meta-model Repurposer

We will create a new Repurposer that uses the KNN algorithm as a meta-model. The resulting Meta-model Repurposer will classify the features extracted by the neural network source model.

```
In [6]: from sklearn.neighbors import KNeighborsClassifier
```

7.2.1 Definition

```
In [7]: class KNNRepurposer(xfer.MetaModelRepurposer):
            def __init__(self, source_model: mx.mod.Module, feature_layer_names, context_function=mx
                         n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metri
                # Call init() of parent
                super(KNNRepurposer, self).__init__(source_model, feature_layer_names, context_functi

                # Initialise parameters specific to the KNN algorithm
                self.n_neighbors = n_neighbors
                self.weights = weights
                self.algorithm = algorithm
                self.leaf_size = leaf_size
                self.p = p
                self.metric = metric
                self.metric_params = metric_params
                self.n_jobs = n_jobs

            # Define function that takes a set of features and labels and returns a trained model.
            # feature_indices_per_layer is a dictionary which gives the feature indices which corresp
            # to each layer's features.
            def _train_model_from_features(self, features, labels, feature_indices_per_layer=None):
                lin_model = KNeighborsClassifier(n_neighbors=self.n_neighbors,
                                                 weights=self.weights,
                                                 algorithm=self.algorithm,
                                                 leaf_size=self.leaf_size,
                                                 p=self.p,
                                                 metric=self.metric,
                                                 metric_params=self.metric_params)
                lin_model.fit(features, labels)
                return lin_model

            # Define a function that predicts the class probability given features
            def _predict_probability_from_features(self, features):
                return self.target_model.predict_proba(features)

            # Define a function that predicts the class label given features
            def _predict_label_from_features(self, features):
                return self.target_model.predict(features)

            # In order to make your repurposer serialisable, you will need to implement functions
```

```
# which convert your model's parameters to a dictionary.
def get_params(self):
    """
    This function should return a dictionary of all the parameters of the repurposer that
    are in the repurposer constructor arguments.
    """
    param_dict = super().get_params()
    param_dict['n_neighbors'] = self.n_neighbors
    param_dict['weights'] = self.weights
    param_dict['algorithm'] = self.algorithm
    param_dict['leaf_size'] = self.leaf_size
    param_dict['p'] = self.p
    param_dict['metric'] = self.metric
    param_dict['metric_params'] = self.metric_params
    param_dict['n_jobs'] = self.n_jobs
    return param_dict

# Some repurposers will need a get_attributes() and set_attributes() to get and set the p
# of the repurposer that are not in the constructor argument. An example is shown below:

# def get_attributes(self):
#     """
#     This function should return a dictionary of all the parameters of the repurposer th
#     are NOT in the constructor arguments.
#
#     This function does not need to be defined if the repurposer has no specific attribu
#     """
#     param_dict = super().get_attributes()
#     param_dict['example_attribute'] = self.example_attribute
#     return param_dict

# def set_attributes(self, input_dict):
#     super().set_attributes(input_dict)
#     self.example_attribute  = input_dict['example_attribute']

def serialize(self, file_prefix):
    """
    Saves repurposer (excluding source model) to file_prefix.json.
    This method converts the repurposer to dictionary and saves as a json.


    :param str file_prefix: Prefix to save file with
    """
    output_dict = {}
    output_dict[repurposer_keys.PARAMS] = self.get_params()
    output_dict[repurposer_keys.TARGET_MODEL] = target_model_to_dict()  # This should be
    output_dict.update(self.get_attributes())

    utils.save_json(file_prefix, output_dict)

def deserialize(self, input_dict):
    """
    Uses dictionary to set attributes of repurposer

    :param dict input_dict: Dictionary containing values for attributes to be set to
    """
    self.set_attributes(input_dict)  # Set attributes of the repurposer from input_dict
    self.target_model = target_model_from_dict()  # Unpack dictionary representation of t
```

7.2.2 Use

```
In [8]: repurposerKNN = KNNRepurposer(source_model, ['fc8'])

In [9]: repurposerKNN.repurpose(train_iterator)

In [10]: results = repurposerKNN.predict_label(test_iterator)

In [11]: print(classification_report(y_pred=results, y_true=test_labels))
             precision    recall  f1-score   support

          0       1.00      0.50      0.67         2
          1       0.67      1.00      0.80         2
          2       1.00      1.00      1.00         2
          3       1.00      1.00      1.00         2

avg / total       0.92      0.88      0.87         8
```

7.3 Custom Neural Network Repurposer

Now we will define a custom Neural Network Repurposer which performs transfer learning by:

1. taking the original source neural network and keeping all layers up to `transfer_layer_name`
2. adding two fully connected layers on the top
3. fine-tuning with any conv layers frozen

7.3.1 Definition

```
In [12]: class Add2FullyConnectedRepurposer(xfer.NeuralNetworkRepurposer):
             def __init__(self, source_model: mx.mod.Module, transfer_layer_name, num_nodes, target_c
                          context_function=mx.context.cpu, num_devices=1, batch_size=64, num_epochs=5
                 super().__init__(source_model, context_function, num_devices, batch_size, num_epochs

                 # initialse parameters
                 self.transfer_layer_name = transfer_layer_name
                 self.num_nodes = num_nodes
                 self.target_class_count = target_class_count

             def _get_target_symbol(self, source_model_layer_names):
                 # Check if 'transfer_layer_name' is present in source model
                 if self.transfer_layer_name not in source_model_layer_names:
                     raise ValueError('transfer_layer_name: {} not found in source model'.format(seli

                 # Create target symbol by transferring layers from source model up to 'transfer_laye
                 transfer_layer_key = self.transfer_layer_name + '_output'  # layer key with output s
                 source_symbol = self.source_model.symbol.get_internals()
                 target_symbol = source_symbol[transfer_layer_key]
                 return target_symbol

             # All Neural Network Repurposers must implement this function which takes a training ite
             def _create_target_module(self, train_iterator: mx.io.DataIter):
                 # Create model handler to manipulate the source model
                 model_handler = xfer.model_handler.ModelHandler(self.source_model, self.context_func
```

```
            # Create target symbol by transferring layers from source model up to and including
            target_symbol = self._get_target_symbol(model_handler.layer_names)

            # Update model handler by replacing source symbol with target symbol
            # and cleaning up weights of layers that were not transferred
            model_handler.update_sym(target_symbol)

            # Add a fully connected layer (with nodes equal to number of target classes) and a s
            fully_connected_layer1 = mx.sym.FullyConnected(num_hidden=self.num_nodes, name='fc_1
            fully_connected_layer2 = mx.sym.FullyConnected(num_hidden=self.target_class_count, n
            softmax_output_layer = mx.sym.SoftmaxOutput(name=train_iterator.provide_label[0][0]
            model_handler.add_layer_top([fully_connected_layer1, fully_connected_layer2,  softma

            # Get fixed layers
            conv_layer_names = model_handler.get_layer_names_matching_type('Convolution')
            conv_layer_params = model_handler.get_layer_parameters(conv_layer_names)

            # Create and return target mxnet module using the new symbol and params
            return model_handler.get_module(train_iterator, fixed_layer_parameters=conv_layer_pa

        # To be serialisable, Neural Network Repurposers require get_params, get_attributes, set
```

7.3.2 Use

```
In [13]: # instantiate repurposer
         repurposer2Fc = Add2FullyConnectedRepurposer(source_model, transfer_layer_name='fc7', num_no

In [14]: train_iterator.reset()
         repurposer2Fc.repurpose(train_iterator)

In [15]: results = repurposer2Fc.predict_label(test_iterator)

In [16]: print(classification_report(y_pred=results, y_true=test_labels))

             precision    recall  f1-score   support

          0       1.00      0.50      0.67         2
          1       1.00      1.00      1.00         2
          2       1.00      1.00      1.00         2
          3       0.67      1.00      0.80         2

avg / total       0.92      0.88      0.87         8
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

x

Index

L

M

N

P

R

S

U

V

W

X