Lab 4 introduces PyTorch and focuses on applying the perceptron and multilayer perceptron from Week 4 lecture to automatic detection of the number of days of ground frost and snow based on other weather variables.

Lab again uses the *curated_data_1month_2010-2022_nonans.csv*

The first 3 sections follow on from the previous labs: reading in the data, splitting it and scaling the features.

The notebook implements a perceptron from scratch. It won't be required to do this in the assessment, instead we will focus on using pytorch.

The target frost label which is continuous is split into two with the target labels being -1 or 1

The split point is made using the distribution of the given data and takes into account any domain knowledge such as max number of days in a month

A perceptron model is then trained using stochastic gradient descent.

A looping/iteration setup is implemented to achieve this.

The training is based on max epochs though usually the limit would be based on the loss function

Updating the weights based on a learning rate of 0.0001

The weights and bias are initialized randomly though there are a few different approaches to this - of note there is the option of using a pre-train model where the weights and biases are optimized based on a large corpus of data

This section of the notebook details setting up the parameters and hyperparameters prior to any training

```
# Initialize parameters (i.e. weights and bias) and hyperparameters
lr = 0.0001
random_seed = 1
rng =  numpy.random.default_rng(random_seed)
w_perc = rng.random((scaled_train_data.shape[1]+1, 1)) #includes the bias
max_epochs = 30
num_iters = scaled_train_data.shape[0]

x_perc = numpy.concatenate([scaled_train_data, numpy.ones((scaled_train_data.shape[0], 1))], axis=1) #bias column added
y_perc = deepcopy(numpy.reshape(train_ground_frost_labels_class, (-1, 1)))
```

X_perc is adding in a 1 to the data “rows” to init the bias

Y_perc is just converting the array into a vertical vector for the network shape

Num_iters is the shape of the data and allows us to feed in the data row by row

When coding up network it is import to spend the time making sure the inputs and outputs of layers are the correct shape for the next layer

The gradient descent method where you update the model's parameters after processing **every single training instance** is called **Stochastic Gradient Descent (SGD)**.

Other options could be mini-batch or batch (whole) gradient descent.

There are some alternative methods such as AdaGrad and Adam. In the assessment it will be important to clarify which method of optimization we are using a why.

The choice of which one to use often depends on the specific problem, dataset, and model architecture.

Section 4 focused on training a single layer perceptron with no hidden layers

Section trains a multilayer perceptron but focuses on training it via PyTorch an nn.Module and nn.Linear (the layers)

The notebook sets up a good, basic class for training an MLP which includes an init where the layers are set up and parameterized and then a function for the feedforward network

A function is set up to calculate a selection of evaluation metrics which seems like a a good practice

A class for managing the data is set up which includes recoding the labels and reshaping for the modelling. It doesn’t seem to have any code to handle the scaling though this seems like a good place for it

This section of code was just for setting up classes and functions for repeatability. The next section executes the code setup as well as ad-hoc lines.

PyTorch has its down random number generator for seeding

hidden_layer_sizes = [10, 10] tells the code we are going to run an mlp with 2 hidden layers for 10 nodes each

The hypers and params are the plugged into the mlp class to init
three_layer_MLP(feature_count, hidden_layer_sizes, class_count)

The code utilised a PyTorch function called **from torch.utils.data import DataLoader** to set the training set up into batches **DataLoader(train_set, batch_size=batch_size)**

Test and validation set don't need batches so just set as the length of the subset **len(val_set)**

Notebook uses the **from torch import optim** for optimization
optim.SGD(MLP_model.parameters(), lr=learning_rate)

The code uses the Cross Entropy Loss as its performance metric **nn.CrossEntropyLoss()**

This code once again illustrates a double loop. The outer for the training epochs and the inner this run through the training batches (instances)

The implementation of this code uses 100 epochs and will run until finished but a more common approach is to use an early finishing metric which the next section explores

Early stopping is based on the epochs, not the inner iterations

The early stop method used in the notebook is a checkpointing method. It runs through all 100 epochs as stores the best performing model to be used on the training set

The core idea of **early stopping** is to prevent overfitting by stopping the training process (or selecting the best model) at the point where the model's performance on a **validation set** starts to degrade, even if its performance on the training set is still improving.

In this method all training epochs are still run but this is mostly for the ability to plot and visual at the end

Plotting the performance between the two is an important part of the process

The model selected for training is the one that validation set decided was the best

Remember, the validation sets purpose is the optimize hyperparameters, of which epochs is one

The output size of a network needs to be the same as the output categories

A softmax activation recodes the outputs such that the sum of the outputs equals 1:
torch.softmax(out, 1)

The recoded can be interpreted as the probability of the corresponding class, and so, the class corresponding to the highest of the outputs can be taken to be the class predicted by the model.

An argmax numpy function is generally used to select the highest category

All hyperparameters can be changed and suited to the model and task at hand. Some params should be optimized through testing, others could be selected based on theoretic basis

The notebook then demonstrates some clear methods of constructing a full MLP model in pytorch

Then implementing that model with given hyperparams, opt method and loss function

Most of the model is pre-initalized before being looped into the epochs