**Deep Learning with PyTorch**

Classical machine learning relies on using statistics to determine relationships between features and labels, and can be very effective for creating predictive models. However, a massive growth in the availability of data coupled with advances in the computing technology required to process it has led to the emergence of new machine learning techniques that mimic the way the brain processes information in a structure called an artificial neural network.

PyTorch is a framework for creating machine learning models, including deep neural networks (DNNs). In this example, we'll use PyTorch to create a simple neural network that classifies penguins into species based on the length and depth of their culmen (bill), their flipper length, and their body mass.

**Citation**: The penguins dataset used in the this exercise is a subset of data collected and made available by [Dr. Kristen Gorman](https://www.uaf.edu/cfos/people/faculty/detail/kristen-gorman.php) and the [Palmer Station, Antarctica LTER](https://pal.lternet.edu/), a member of the [Long Term Ecological Research Network](https://lternet.edu/).

**Explore the Dataset**

Before we start using PyTorch to create a model, let's load the data we need from the Palmer Islands penguins dataset, which contains observations of three different species of penguin.

**Note**: In reality, you can solve the penguin classification problem easily using classical machine learning techniques without the need for a deep learning model; but it's a useful, easy to understand dataset with which to demonstrate the principles of neural networks in this notebook.

In [2]:



**import** pandas **as** pd

​

*# load the training dataset (excluding rows with null values)*

penguins **=** pd.read\_csv('data/penguins.csv').dropna()

​

*# Deep Learning models work best when features are on similar scales*

*# In a real solution, we'd implement some custom normalization for each feature, but to keep things simple*

*# we'll just rescale the FlipperLength and BodyMass so they're on a similar scale to the bill measurements*

penguins['FlipperLength'] **=** penguins['FlipperLength']**/**10

penguins['BodyMass'] **=** penguins['BodyMass']**/**100

​

*# The dataset is too small to be useful for deep learning*

*# So we'll oversample it to increase its size*

**for** i **in** range(1,3):

penguins **=** penguins.append(penguins)

​

*# Display a random sample of 10 observations*

sample **=** penguins.sample(10)

sample

​

Out[2]:

|  | **CulmenLength** | **CulmenDepth** | **FlipperLength** | **BodyMass** | **Species** |
| --- | --- | --- | --- | --- | --- |
| **0** | 39.1 | 18.7 | 18.1 | 37.50 | 0 |
| **1** | 39.5 | 17.4 | 18.6 | 38.00 | 0 |
| **2** | 40.3 | 18.0 | 19.5 | 32.50 | 0 |
| **4** | 36.7 | 19.3 | 19.3 | 34.50 | 0 |
| **5** | 39.3 | 20.6 | 19.0 | 36.50 | 0 |
| **...** | ... | ... | ... | ... | ... |
| **339** | 55.8 | 19.8 | 20.7 | 40.00 | 2 |
| **340** | 43.5 | 18.1 | 20.2 | 34.00 | 2 |
| **341** | 49.6 | 18.2 | 19.3 | 37.75 | 2 |
| **342** | 50.8 | 19.0 | 21.0 | 41.00 | 2 |
| **343** | 50.2 | 18.7 | 19.8 | 37.75 | 2 |

1368 rows × 5 columns

The **Species** column is the label our model will predict. Each label value represents a class of penguin species, encoded as 0, 1, or 2. The following code shows the actual species to which these class labels corrrespond.

In [3]:



penguin\_classes **=** ['Amelie', 'Gentoo', 'Chinstrap']

print(sample.columns[0:5].values, 'SpeciesName')

**for** index, row **in** penguins.sample(10).iterrows():

print('[',row[0], row[1], row[2],row[3], int(row[4]), ']',penguin\_classes[int(row[**-**1])])

['CulmenLength' 'CulmenDepth' 'FlipperLength' 'BodyMass' 'Species'] SpeciesName

[ 49.6 16.0 22.5 57.0 1 ] Gentoo

[ 49.2 18.2 19.5 44.0 2 ] Chinstrap

[ 46.5 14.5 21.3 44.0 1 ] Gentoo

[ 35.9 16.6 19.0 30.5 0 ] Amelie

[ 42.2 19.5 19.7 42.75 0 ] Amelie

[ 39.7 17.7 19.3 32.0 0 ] Amelie

[ 36.2 17.2 18.7 31.5 0 ] Amelie

[ 37.0 16.9 18.5 30.0 0 ] Amelie

[ 40.9 16.6 18.7 32.0 2 ] Chinstrap

[ 39.2 18.6 19.0 42.5 0 ] Amelie

As is common in a supervised learning problem, we'll split the dataset into a set of records with which to train the model, and a smaller set with which to validate the trained model.

In [ ]:



**from** sklearn.model\_selection **import** train\_test\_split

​

features **=** ['CulmenLength','CulmenDepth','FlipperLength','BodyMass']

label **=** 'Species'

*# Split data 70%-30% into training set and test set*

x\_train, x\_test, y\_train, y\_test **=** train\_test\_split(penguins[features].values,

penguins[label].values,

test\_size**=**0.30,

random\_state**=**0)

​

print ('Training Set: %d, Test Set: %d \n' **%** (len(x\_train), len(x\_test)))

print("Sample of features and labels:")

​

*# Take a look at the first 25 training features and corresponding labels*

**for** n **in** range(0,24):

print(x\_train[n], y\_train[n], '(' **+** penguin\_classes[y\_train[n]] **+** ')')

The *features* are the measurements for each penguin observation, and the *label* is a numeric value that indicates the species of penguin that the observation represents (Amelie, Gentoo, or Chinstrap).

**Install and import the PyTorch libraries**

Since we plan to use PyTorch to create our penguin classifier, we'll need to run the following two cells to install and import the PyTorch libraries we intend to use. The specific installation of of PyTorch depends on your operating system and whether your computer has graphics processing units (GPUs) that can be used for high-performance processing via *cuda*. You can find detailed instructions at <https://pytorch.org/get-started/locally/>.

In [ ]:



**!**pip install torch**==**1.7.1**+**cpu torchvision**==**0.8.2**+**cpu torchaudio**==**0.7.2 **-**f https:**//**download.pytorch.org**/**whl**/**torch\_stable.html

In [ ]:



**import** torch

**import** torch.nn **as** nn

**import** torch.utils.data **as** td

​

*# Set random seed for reproducability*

torch.manual\_seed(0)

​

print("Libraries imported - ready to use PyTorch", torch.\_\_version\_\_)

**Prepare the data for PyTorch**

PyTorch makes use of *data loaders* to load training and validation data in batches. We've already loaded the data into numpy arrays, but we need to wrap those in PyTorch datasets (in which the data is converted to PyTorch *tensor* objects) and create loaders to read batches from those datasets.

In [ ]:



*# Create a dataset and loader for the training data and labels*

train\_x **=** torch.Tensor(x\_train).float()

train\_y **=** torch.Tensor(y\_train).long()

train\_ds **=** td.TensorDataset(train\_x,train\_y)

train\_loader **=** td.DataLoader(train\_ds, batch\_size**=**20,

shuffle**=False**, num\_workers**=**1)

​

*# Create a dataset and loader for the test data and labels*

test\_x **=** torch.Tensor(x\_test).float()

test\_y **=** torch.Tensor(y\_test).long()

test\_ds **=** td.TensorDataset(test\_x,test\_y)

test\_loader **=** td.DataLoader(test\_ds, batch\_size**=**20,

shuffle**=False**, num\_workers**=**1)

print('Ready to load data')

**Define a neural network**

Now we're ready to define our neural network. In this case, we'll create a network that consists of 3 fully-connected layers:

* An input layer that receives an input value for each feature (in this case, the four penguin measurements) and applies a *ReLU* activation function.
* A hidden layer that receives ten inputs and applies a *ReLU* activation function.
* An output layer that uses a *Softmax* activation function to generate an output for each penguin species (which represent the classification probabilities for each of the three possible penguin species). Softmax functions produce a vector with probability values that sum to 1.

In [ ]:



*# Number of hidden layer nodes*

hl **=** 10

​

*# Define the neural network*

**class** PenguinNet(nn.Module):

**def** \_\_init\_\_(self):

super(PenguinNet, self).\_\_init\_\_()

self.fc1 **=** nn.Linear(len(features), hl)

self.fc2 **=** nn.Linear(hl, hl)

self.fc3 **=** nn.Linear(hl, len(penguin\_classes))

​

**def** forward(self, x):

x **=** torch.relu(self.fc1(x))

x **=** torch.relu(self.fc2(x))

x **=** torch.softmax(self.fc3(x),dim**=**1)

**return** x

​

*# Create a model instance from the network*

model **=** PenguinNet()

print(model)

**Train the model**

To train the model, we need to repeatedly feed the training values forward through the network, use a loss function to calculate the loss, use an optimizer to backpropagate the weight and bias value adjustments, and validate the model using the test data we withheld.

To do this, we'll create a function to train and optimize the model, and function to test the model. Then we'll call these functions iteratively over 50 epochs, logging the loss and accuracy statistics for each epoch.

In [ ]:



**def** train(model, data\_loader, optimizer):

*# Set the model to training mode*

model.train()

train\_loss **=** 0

**for** batch, tensor **in** enumerate(data\_loader):

data, target **=** tensor

*#feedforward*

optimizer.zero\_grad()

out **=** model(data)

loss **=** loss\_criteria(out, target)

train\_loss **+=** loss.item()

​

*# backpropagate*

loss.backward()

optimizer.step()

​

*#Return average loss*

avg\_loss **=** train\_loss **/** (batch**+**1)

print('Training set: Average loss: {:.6f}'.format(avg\_loss))

**return** avg\_loss

**def** test(model, data\_loader):

*# Switch the model to evaluation mode (so we don't backpropagate)*

model.eval()

test\_loss **=** 0

correct **=** 0

​

**with** torch.no\_grad():

batch\_count **=** 0

**for** batch, tensor **in** enumerate(data\_loader):

batch\_count **+=** 1

data, target **=** tensor

*# Get the predictions*

out **=** model(data)

​

*# calculate the loss*

test\_loss **+=** loss\_criteria(out, target).item()

​

*# Calculate the accuracy*

\_, predicted **=** torch.max(out.data, 1)

correct **+=** torch.sum(target**==**predicted).item()

*# Calculate the average loss and total accuracy for this epoch*

avg\_loss **=** test\_loss**/**batch\_count

print('Validation set: Average loss: {:.6f}, Accuracy: {}/{} ({:.0f}%)\n'.format(

avg\_loss, correct, len(data\_loader.dataset),

100. **\*** correct **/** len(data\_loader.dataset)))

*# return average loss for the epoch*

**return** avg\_loss

​

*# Specify the loss criteria (CrossEntropyLoss for multi-class classification)*

loss\_criteria **=** nn.CrossEntropyLoss()

​

*# Use an "Adam" optimizer to adjust weights*

*# (see https://pytorch.org/docs/stable/optim.html#algorithms for details of supported algorithms)*

learning\_rate **=** 0.001

optimizer **=** torch.optim.Adam(model.parameters(), lr**=**learning\_rate)

optimizer.zero\_grad()

​

*# We'll track metrics for each epoch in these arrays*

epoch\_nums **=** []

training\_loss **=** []

validation\_loss **=** []

​

*# Train over 50 epochs*

epochs **=** 50

**for** epoch **in** range(1, epochs **+** 1):

​

*# print the epoch number*

print('Epoch: {}'.format(epoch))

*# Feed training data into the model to optimize the weights*

train\_loss **=** train(model, train\_loader, optimizer)

*# Feed the test data into the model to check its performance*

test\_loss **=** test(model, test\_loader)

*# Log the metrics for this epoch*

epoch\_nums.append(epoch)

training\_loss.append(train\_loss)

validation\_loss.append(test\_loss)

​

While the training process is running, let's try to understand what's happening:

1. In each *epoch*, the full set of training data is passed forward through the network. There are four features for each observation, and four corresponding nodes in the input layer - so the features for each observation are passed as a vector of four values to that layer. However, for efficiency, the feature vectors are grouped into batches; so actually a matrix of multiple feature vectors is fed in each time.
2. The matrix of feature values is processed by a function that performs a weighted sum using initialized weights and bias values. The result of this function is then processed by the activation function for the input layer to constrain the values passed to the nodes in the next layer.
3. The weighted sum and activation functions are repeated in each layer. Note that the functions operate on vectors and matrices rather than individual scalar values. In other words, the forward pass is essentially a series of nested linear algebra functions. This is the reason data scientists prefer to use computers with graphical processing units (GPUs), since these are optimized for matrix and vector calculations.
4. In the final layer of the network, the output vectors contain a probability value for each possible class (in this case, classes 0, 1, and 2). This vector is processed by a *loss function* to determine how far the values calculated by the network are from the actual values - so for example, suppose the output for a Gentoo penguin (class 1) observation is [0.3, 0.4, 0.3]. The correct prediction should be [0.0, 1.0, 0.0], so the variance between the predicted and actual values (how far away the each predicted value is from what it should be) is [0.3, 0.6, 0.3]. This variance is aggregated for each batch and maintained as a running aggregate to calculate the overall level of error (*loss*) incurred by the training data for the epoch.
5. At the end of each epoch, the validation data is passed through the network, and its loss and accuracy (proportion of correct predictions based on the highest probability value in the output vector) are also calculated. It's important to do this because it enables us to compare the performance of the model using data on which it was not trained, helping us determine if it will generalize well for new data or if it's *overfitted* to the training data.
6. After all the data has been passed forward through the network, the output of the loss function for the *training* data (but not the *validation* data) is passed to the opimizer. The precise details of how the optimizer processes the loss vary depending on the specific optimization algorithm being used; but fundamentally you can think of the entire network, from the input layer to the loss function as being one big nested (*composite*) function. The optimizer applies some differential calculus to calculate *partial derivatives* for the function with respect to each weight and bias value that was used in the network. It's possible to do this efficiently for a nested function due to something called the *chain rule*, which enables you to determine the derivative of a composite function from the derivatives of its inner function and outer functions. You don't really need to worry about the details of the math here (the optimizer does it for you), but the end result is that the partial derivatives tell us about the slope (or *gradient*) of the loss function with respect to each weight and bias value - in other words, we can determine whether to increase or decrease the weight and bias values in order to decrease the loss.
7. Having determined in which direction to adjust the weights and biases, the optimizer uses the *learning rate* to determine by how much to adjust them; and then works backwards through the network in a process called *backpropagation* to assign new values to the weights and biases in each layer.
8. Now the next epoch repeats the whole training, validation, and backpropagation process starting with the revised weights and biases from the previous epoch - which hopefully will result in a lower level of loss.
9. The process continues like this for 50 epochs.

**Review training and validation loss**

After training is complete, we can examine the loss metrics we recorded while training and validating the model. We're really looking for two things:

* The loss should reduce with each epoch, showing that the model is learning the right weights and biases to predict the correct labels.
* The training loss and validation loss should follow a similar trend, showing that the model is not overfitting to the training data.

Let's plot the loss metrics and see:

In [ ]:



**%**matplotlib inline

**from** matplotlib **import** pyplot **as** plt

​

plt.plot(epoch\_nums, training\_loss)

plt.plot(epoch\_nums, validation\_loss)

plt.xlabel('epoch')

plt.ylabel('loss')

plt.legend(['training', 'validation'], loc**=**'upper right')

plt.show()

**View the learned weights and biases**

The trained model consists of the final weights and biases that were determined by the optimizer during training. Based on our network model we should expect the following values for each layer:

* Layer 1: There are four input values going to ten output nodes, so there should be 10 x 4 weights and 10 bias values.
* Layer 2: There are ten input values going to ten output nodes, so there should be 10 x 10 weights and 10 bias values.
* Layer 3: There are ten input values going to three output nodes, so there should be 3 x 10 weights and 3 bias values.

In [ ]:



**for** param\_tensor **in** model.state\_dict():

print(param\_tensor, "\n", model.state\_dict()[param\_tensor].numpy())

**Evaluate model performance**

So, is the model any good? The raw accuracy reported from the validation data would seem to indicate that it predicts pretty well; but it's typically useful to dig a little deeper and compare the predictions for each possible class. A common way to visualize the performace of a classification model is to create a *confusion matrix* that shows a crosstab of correct and incorrect predictions for each class.

In [ ]:



*#Pytorch doesn't have a built-in confusion matrix metric, so we'll use SciKit-Learn*

**from** sklearn.metrics **import** confusion\_matrix

**import** numpy **as** np

​

*# Set the model to evaluate mode*

model.eval()

​

*# Get predictions for the test data*

x **=** torch.Tensor(x\_test).float()

\_, predicted **=** torch.max(model(x).data, 1)

​

*# Plot the confusion matrix*

cm **=** confusion\_matrix(y\_test, predicted.numpy())

plt.imshow(cm, interpolation**=**"nearest", cmap**=**plt.cm.Blues)

plt.colorbar()

tick\_marks **=** np.arange(len(penguin\_classes))

plt.xticks(tick\_marks, penguin\_classes, rotation**=**45)

plt.yticks(tick\_marks, penguin\_classes)

plt.xlabel("Actual Species")

plt.ylabel("Predicted Species")

plt.show()

The confusion matrix should show a strong diagonal line indicating that there are more correct than incorrect predictions for each class.

**Save the trained model**

Now that we have a model we believe is reasonably accurate, we can save its trained weights for use later.

In [ ]:



*# Save the model weights*

model\_file **=** 'models/penguin\_classifier.pt'

torch.save(model.state\_dict(), model\_file)

**del** model

print('model saved as', model\_file)

**Use the trained model**

When we have a new penguin observation, we can use the model to predict the species.

In [ ]:



*# New penguin features*

x\_new **=** [[50.4,15.3,20,50]]

print ('New sample: {}'.format(x\_new))

​

*# Create a new model class and load weights*

model **=** PenguinNet()

model.load\_state\_dict(torch.load(model\_file))

​

*# Set model to evaluation mode*

model.eval()

​

*# Get a prediction for the new data sample*

x **=** torch.Tensor(x\_new).float()

\_, predicted **=** torch.max(model(x).data, 1)

​

print('Prediction:',penguin\_classes[predicted.item()])

**Learn more**

This notebook was designed to help you understand the basic concepts and principles involved in deep neural networks, using a simple PyTorch example. To learn more about PyTorch, take a look at the [tutorials on the PyTorch web site](https://pytorch.org/tutorials/).