Open Lab 1

Single-Layer Networks

CSCI 7850 - Deep Learning

Due: Sep. 12 @ 11:00pm

Assignment

Here are the details of what you need to do for this assignment:

- Create a python script named `OL1.py` that takes three command line arguments: `[Data] [Optimizer] [Standardize]`
  - `[Data]` can be either `iris` or `wdbc`: your script should use the appropriate data set selected using this argument
  - `[Optimizer]` can be either `adam` or `rmsprop` or `sgd` to select the optimizer used by the model (Adam(), RMSprop(), or SGD())
  - `[Standardize]` can be either `0` or `1` indicated whether to perform standardization on the data (`0` for unstandardized and `1` for standardized)
  - Your code should **always** perform a permutation (or shuffle) of the data set prior to training
  - Other settings should be maintained as specified in the tutorial below (number of epochs=101, etc.) so that your script prints the results line (validation accuracy) **exactly matching the format provided below**.
- Use your `OL1.py` script to run your models using all of the different combinations of command-line arguments - **perform 100 independent runs for each combination**.
- Compile your data into *uniquely named* text files for each combination of arguments that can be read in using `np.loadtxt`.
- Create an iPython Notebook file named `OL1.ipynb` which reads in the compiled results for Adam, RMSprop, and SGD to produce a learning curve with mean and standard error. An example of one such plot is provided below, but your notebook should contain four plots in the end: (WDBC-unstandardized), (WDBC-standardized),(Iris-unstandardized), (Iris-standardized). No code for model training/testing should be in this notebook file, it should only read in the results text files and plot them.
- At the end of your notebook file, create a Markdown cell and compile answers to the following questions:
  1. Which of the chosen optimizers seems to benefit the most from standardization of the data sets?
  2. Why do you think this may be happening?
  3. What other choices (hyperparameters, architecture changes, data prep, etc.) do you think might be explored which could impact the performance?
  4. What outcome would you expect from changing the code in this way (hypothesis)?
  5. What process would you use to attempt to confirm your hypothesis and what steps would testing it involve?
  6. Which part(s) of the lab/code/experiments are still *unclear* to you after finishing this assignment?
  7. Which parts are you interested in learning more about?

Example results plot (illustration for `OL1.ipynb`)

In [136…
```
results1 = np.loadtxt("wdbc-unstandardized-adam-results.txt")
results2 = np.loadtxt("wdbc-unstandardized-rmsprop-results.txt")
results3 = np.loadtxt("wdbc-unstandardized-sgd-results.txt")
```

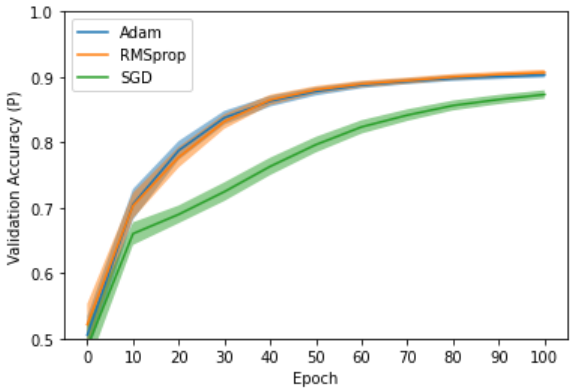
In [8]:
```
# All have this shape
results1.shape
```

Out[8]: (100, 11)

In [27]:
```
plt.plot(np.arange(0,results1.shape[1]),
         np.mean(results1,0),label='Adam')
plt.fill_between(np.arange(0,results1.shape[1]),
                 np.mean(results1,0)-(1.96*np.std(results1,0)/np.sqrt(results1.shape[0])),
                 np.mean(results1,0)+(1.96*np.std(results1,0)/np.sqrt(results1.shape[0])),
                 alpha=0.5)

plt.plot(np.arange(0,results2.shape[1]),
         np.mean(results2,0),label='RMSprop')
plt.fill_between(np.arange(0,results2.shape[1]),
                 np.mean(results2,0)-(1.96*np.std(results2,0)/np.sqrt(results2.shape[0])),
                 np.mean(results2,0)+(1.96*np.std(results2,0)/np.sqrt(results2.shape[0])),
                 alpha=0.5)

plt.plot(np.arange(0,results3.shape[1]),
         np.mean(results3,0),label='SGD')
plt.fill_between(np.arange(0,results3.shape[1]),
                 np.mean(results3,0)-(1.96*np.std(results3,0)/np.sqrt(results3.shape[0])),
                 np.mean(results3,0)+(1.96*np.std(results3,0)/np.sqrt(results3.shape[0])),
                 alpha=0.5)
plt.ylim([0.5,1.0])
plt.ylabel('Validation Accuracy (P)')
plt.xlabel('Epoch')
plt.legend()
plt.xticks(ticks=np.arange(0,results1.shape[1]),
           labels=np.arange(0,results1.shape[0]+1,10))
plt.show()
```



Submission

Create a zip archive which contains the following contents:

- `OL1.py`
- `OL1.ipynb`
- `*-results.txt` : **All** processed results text files (needed by np.loadtxt in your notebook)
- `sources.pdf` : **Optional**, if you used any electronic resources other than those provided in the course, include full documentation of how/why they were utilized

Upload your zip archive to the course assignment system by the deadline at the top of this document.

Computational Explorations Using JupyterLab: *Inside and Outside the Notebook*

JupyterLab is a great interface for prototyping, interactive coding, and explanatory documentation, so we will utilize it often for performing analysis and making plots. However, it's also not always the best tool for the job. Your work below will consist of utilizing a Jupyter Notebook to explore the creation and training of single-layer neural networks for binary- and multi-class data sets.

Let's start by loading a data set, and doing some preliminary exploration of what information seems to be in that data set - effectively trying to determine the task (T) that this particular data is aiming to solve. Next, we will parse the data into distinct

parts to make up the experiences (E) that our network will use to learn. These particular problems are examples of *supervised learning*, which means that the experiences consist of measurement vectors that are all *labeled*. So, we separate E into two groups (X,Y) where X is a set of measurement (input) vectors and Y is the corresponding set of classification (output) labels: one paired label for each vector in X. Later, we will determine the last piece of the machine learning puzzle: the performance measure (P) we will use to determine how good our neural network *model* is becoming at T through experience with E.

However, even before we begin, we will need some tools for the job...

In [1]:
```
import numpy as np
import torch
import lightning.pytorch as pl
from torchinfo import summary
from torchview import draw_graph
import matplotlib.pyplot as plt
import pandas as pd
```

- **Numpy:** (`numpy`) for manipulating multidimensional arrays a.k.a. tensors
- **Pytorch:** (`torch`) for building/training deep learning models
- **Lightning:** (`lightning.pytorch`) for building/training deep learning models quickly and easily
- **TorchInfo:** (`torchinfo`) for printing useful model details
- **TorchView:** (`torchview`) for visualizing deep learning models
- **Matplotlib:** (`matplotlib.pylot`) for visualizing the results of our simulations
- **Pandas:** (`pandas`) for loading tables with mixed data types

We also need to quickly configure PyTorch depending on the particular computing devices that we have available. Code and data have to be moved onto a device to use it, so often we need to specify which device we are placing/using certain parts of our code and data. We will see more on this below, but for now if we have an Nvidia GPU (and therefore CUDA) available then we will primarily use it. If we don't have such a device then we will just use the CPU.

In [2]:
```
if (torch.cuda.is_available()):
    device = ("gpu")
else:
    device = ("cpu")
print(torch.cuda.is_available())
```

```
False
```

Note: False means we are -not- seeing a GPU and therefore we will have to use the CPU instead. You can use the above to diagnose problems because on Hamilton/Babbage, we expect this to be True since we should be using a GPU. However, on biosim/azuread we will expect only CPU support.

The data set we will be using as a starting point is the WDBC data set from the UC Irvine Machine Learning Repository: Original LINK or LINK. It contains data from 568 medical patient breast tissue samples. Each tissue sample contained a mass which was measured using 30 different types of measurements (radius, area, smoothness, etc.) and then later confirmed to be either benign (0) or malignant (1). So, our task is to be able to predict 0 or 1 based on the vector of 30 measurements alone. While it's possible to think that there might be some algorithmic way to combine these features together to make a good prediction, it's not clear how to proceed with determining such an algorithm. It's important to keep in mind that the techniques used to obtain the labels are more resource- and time-consuming, so using these 30 features instead would result in a practical benefit.

Load up the data using numpy and look at the first five rows...

In [3]:
```
data = np.loadtxt("https://www.cs.mtsu.edu/~jphillips/courses/CSCI7850/public/WDBC.txt")
data[0:5]
```

Out[3]:
```
array([[0.6766275 , 0.6260183 , 0.6472149 , 0.4302279 , 0.5525704 ,
        0.3491604 , 0.343955  , 0.4110835 , 0.6424342 , 0.5776888 ,
        0.1912635 , 0.1358444 , 0.13899   , 0.1063261 , 0.1243816 ,
        0.1360414 , 0.09368687, 0.2273158 , 0.248765  , 0.1118298 ,
        0.681465  , 0.6138474 , 0.6086783 , 0.3815233 , 0.5610961 ,
        0.3030246 , 0.4596645 , 0.6721649 , 0.5959626 , 0.4476145 ,
        1.        ],
       [0.5186766 , 0.5481161 , 0.5167639 , 0.2578169 , 0.6450428 ,
        0.5408222 , 0.33388   , 0.4365308 , 0.7407895 , 0.7105911 ,
        0.08858336, 0.2012692 , 0.09599636, 0.03882331, 0.1430132 ,
        0.2256278 , 0.06770202, 0.2561091 , 0.1841672 , 0.1243633 ,
        0.4889012 , 0.6703674 , 0.4872611 , 0.2108369 , 0.6850854 ,
        0.6278828 , 0.4424121 , 0.9281787 , 0.6423622 , 0.6144578 ,
        1.        ],
       [0.5190324 , 0.5773931 , 0.5113528 , 0.2627349 , 0.5185435 ,
        0.3850608 , 0.2410965 , 0.1856859 , 0.4782895 , 0.6308498 ,
        0.07845458, 0.2268168 , 0.1011829 , 0.03603836, 0.1362673 ,
        0.3426145 , 0.1661111 , 0.3042243 , 0.2074731 , 0.1476542 ,
        0.4295228 , 0.5504643 , 0.4215764 , 0.172426  , 0.4609164 ,
        0.2997164 , 0.292492  , 0.3797251 , 0.3401627 , 0.3857349 ,
        0.        ],
       [0.4087513 , 0.3714358 , 0.3925199 , 0.1618952 , 0.6401469 ,
        0.2382166 , 0.1243674 , 0.09786282, 0.5851974 , 0.6746716 ,
        0.07079708, 0.2386899 , 0.07129208, 0.02644781, 0.1592355 ,
        0.15613   , 0.1049495 , 0.1522637 , 0.2334389 , 0.1211126 ,
        0.3440622 , 0.442067  , 0.3265924 , 0.1099201 , 0.6073675 ,
        0.1899811 , 0.2073482 , 0.2553608 , 0.4430551 , 0.4424096 ,
        0.        ],
       [0.3728211 , 0.5056008 , 0.3539523 , 0.135026  , 0.6548348 ,
        0.172872  , 0.1131912 , 0.1525845 , 0.5713816 , 0.6609195 ,
        0.1294466 , 0.5346981 , 0.1145132 , 0.04282553, 0.5152586 ,
        0.1023634 , 0.04709596, 0.214624  , 0.4402787 , 0.1193029 ,
        0.318535  , 0.594671  , 0.2933121 , 0.09468735, 0.680593  ,
        0.09697543, 0.09432907, 0.2314777 , 0.4343176 , 0.3733976 ,
        0.        ]])
```

In [4]:
```
data.shape
```

Out[4]:
```
(568, 31)
```

So the first thirty columns contain the measurement data for each sample (one per row), and the labels are in the last column (for each corresponding row). Let's split this into X and Y since the Keras tools will expect this...

In [5]:
```
X = data[:,:-1]
X.shape
```

Out[5]:
```
(568, 30)
```

In [6]:
```
Y = data[:,-1:]
Y.shape
```

Out[6]:
```
(568, 1)
```

In [7]:
```
np.unique(Y)
```

Out[7]:
```
array([0., 1.])
```

This data is already preprocessed in several ways to make this lab straight-forward, so if you go back to the original data you will find it's not as clean and ready-to-go as what we are using here. We will save those difficulties for some other time.

For now, let's make a single-layer neural network for binary classification...

In [8]:
```
# Define model
class NeuralNetwork(torch.nn.Module):
    def __init__(self, input_size, output_size, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor

        # This is an actual neural layer...
        self.output_layer = torch.nn.Linear(input_size, output_size)
```

```
    def forward(self, x):
        # Here is where we use the layers to compute something...
        y = x # Start with the input
        y = self.output_layer(y) # y replaces y (stateful parameters)
        return y # Final calculation returned

    # Separate the final activation function out because
    # binary_cross_entropy assumes you are using a sigmoid
    # (outputs are considered logits - more later on this...)
    def predict(self, x):
        # Here is where we use the layers to compute something...
        y = x
        y = self.forward(y) # Start with the input
        y = torch.sigmoid(y) # Apply the activation function (stateless)
        return y # Final calculation returned

# Notice what numbers are used to set the input_size
# and output_size arguments to the contructor.
neural_net = NeuralNetwork(X.shape[-1],
                           Y.shape[-1]).to(device)

print(neural_net) # Not too useful, but some information can be gleaned...
```

```
NeuralNetwork(
  (output_layer): Linear(in_features=30, out_features=1, bias=True)
)
```

We will cover making models in lecture, but the template above shows how to build a basic neural network which contains a single `Linear` or densely connected layer which which matches the length of our data set input vectors (`in_features` - 30 for the WDBC data) and the length of our target vectors (`out_features` - just vectors of length 1 here). Specifying these as the *first* and *second* arguments to the layer object (`Layer()`), and then also making this neural unit utilize a sigmoid activation function, `torch.sigmoid` - since this function has an output range of (0,1) which analogously corresponds to the values of our targets {0,1} (more info in lecture).

Notice that the `Linear` layer object is created in the constructor, but then we *functionalize* it (see the `(y)` after `self.output_layer`) in the `forward` method. Most layer objects work in this way, taking an input tensor, like `x` , and (so long as it's shape agrees with what the layer object is expecting) transforms the results of the neural layer into an output tensor (often of a different shape like in this example). A reference to the output tensor is then stored (and therefore overwrites) `y` . We take a similar approach with the activation function, which accepts a tensor and returns back a tensor (in this case of the same shape with the function applied to all elements in the tensor one-by-one). This tensor could be passed to other layer objects, but since we only need this *single layer*, then we instantiate a model using it's contructor function (`NeuralNetwork()`), and by providing both shape arguments, `input_size` and `output_size` . This completes construction of the neural network, and allows us to pass data through it to see what it computes.

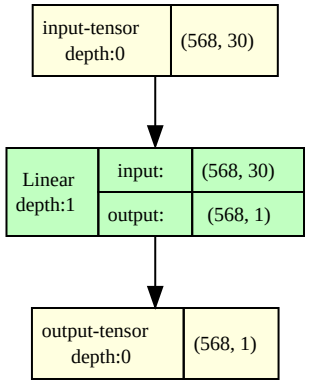
To see what's in the created model, there are two very useful tools (there are many others as well, but these work nicely in a notebook): `summary()` from `torchinfo` and `draw_graph()` from `torchview` . Here're the results of those functions, and how we will typically need to use them:

In [9]:
```
summary(neural_net, input_size=X.shape)
```

Out[9]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
NeuralNetwork                            [568, 1]                  --
├─Linear: 1-1                            [568, 1]                  31
==========================================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.02
==========================================================================================
Input size (MB): 0.07
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.07
==========================================================================================
```

In [10]:
```
model_graph = draw_graph(neural_net, input_size=X.shape,
                         device=device, depth=1)
model_graph.visual_graph
```

Out[10]:



Remember that our input vector set was shaped `(568, 30)` but the input expected by the InputLayer is generally `(None, 30)`. You can think of `None` in this context as a flexible placeholder for how much data we want the network to process (more-or-less in parallel since the weights are reused for all vectors). If you just want to pass in one input vector, `(1,30)`, or maybe 5 vectors, `(5,30)`, then you could easily emply a *slice* of X accordingly and do so. For example:

In [11]:
```
X[:5,:].shape
```

Out[11]:
```
(5, 30)
```

In [12]:
```
predictions = neural_net.predict(torch.Tensor(X[:5,:]))
predictions = predictions.detach().numpy()
predictions
```

Out[12]:
```
array([[0.48700395],
       [0.46988097],
       [0.49984998],
       [0.50783885],
       [0.5448232 ]], dtype=float32)
```

It's important to note that the model requires a `torch.Tensor` object instead of a numpy array. Conversion is typically as simple as calling the `torch.Tensor` constructor on the numpy array. However, the returned `predictions` is also a `torch.Tensor` object. While we can often work with PyTorch functions on these tensors, converting back into numpy arrays is possible and sometimes convenient using `.detach().numpy()`. The detachment operation pulls the tensor off of the device which can then be converted to numpy. This is a bit tedious at times, but it also helps keep the memory load down the device we are using (since GPUs often have less memory than our CPU).

In [13]:
```
predictions.shape
```

Out[13]:
```
(5, 1)
```

You can probably see how the `None` value above became a placeholder for 5, such that the output tensor at the final dense layer is of shape `(None,1)`. This is important to understand: the summary and graph can help you see and understand how data passes through a neural network in most modern frameworks in the form of tensors (multidimensional arrays). You are current observing what the **untrained** neural network thinks about these samples: it's generating values near 0.5 for each of them, than that's basically at chance. Let's check the corresponding targets....

In [14]:
```
Y[:5]
```

Out[14]:
```
array([[1.],
       [1.],
       [0.],
       [0.],
       [0.]])
```

In [15]:
```
import torchmetrics
binary_accuracy = torchmetrics.classification.Accuracy(task='binary')
```

In [16]:
```
binary_accuracy(torch.Tensor(predictions), # predicted
                torch.Tensor(Y[:5]))       # target
```

Out[16]:
```
tensor(0.2000)
```

The `binary_accuracy()` function above was built using the `Accuracy` constructor from the `torchmetrics` package and will compare the predictions to the targets using 0.5 as the comparison threshold. It produces a value of false (specifically, 0.) when the particular prediction/target pair *do not match* and a value of true (specifically, 1.) when the particular prediction/target pair *do match*, and returns the **mean** or the fraction of the predictions which were correct. We are typically interested in the **mean** accuracy value for summary purposes - to quickly see how well the predictions/targets match across all of the data that we provided to the model bove. For that, we usually instantiate an `Accuracy` object and then functionalize it similar to the above.

Note that the performance is likely not too good (1.0 is best, 0.0 is worst) for the random model we made above, but it *can* get lucky on occasion!

We need a model that does a good job, and for that we need to train it produce answers that are more correct then the ones obtained above (which are essentially random because the initial parameters/weights in the neural network are selected randomly from a particular pseudorandom distribution). In order to do that, we will utilize some additional data structures and functions when building the model.

In [17]:
```
# Define Trainable Module
class PLModel(pl.LightningModule):
    def __init__(self, module, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor
        self.module = module

        # This creates an accuracy function
        self.network_acc = torchmetrics.classification.Accuracy(task='binary')
        # This creates a loss function
        self.network_loss = torch.nn.BCEWithLogitsLoss()

    def forward(self, x):
        return self.module.forward(x)

    def predict(self, x):
        return self.module.predict(x)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x, y_true = train_batch
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    def validation_step(self, val_batch, batch_idx):
        x, y_true = val_batch
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    def testing_step(self, test_batch, batch_idx):
        x, y_true = test_batch
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss
```

There is a decent amount of new boiler-plate code here that we need to create to make the model *trainable*. Most important of these is the **optimizer** which we specify in the `configure_optimizers()` function. However, more information will be provided below on this topic. For now, let's look at three other main parts of the code:

1. The super class used this time was `pl.LightningModule` which is the way that you make a trainable network using the PyTorch Lightning framework. This framework is compatible with all other PyTorch modules (like, for example, the earlier network we made and what we will use as part of this model). Even though it involves a bit of boiler-plate code above, it's actually easier that writing PyTorch directly and offers a lot of nice features for collecting model output, saving and loading models, and options for using GPU resources effectively.
2. The `network_loss` and `network_acc` functions are defined in the constructor. Technically, only a `loss` function of some kind is needed to train a network, but it's also helpful to attach other metrics (like binary accuracy) in order to also collect those statistics during the training process.
3. The `forward` function now leaves off the final `sigmoid` calculation. Instead, a new `predict` function now uses the `forward` function and then implements a `sigmoid` transform on the results. This change is needed because the loss function we are using assumes the sigmoid computation *would* normally be performed, but it is computationally more efficient to leave it off (and compute loss directly from the *logits* which would be provided to the sigmoid). We will briefly discuss this in lecture for now, and then get to the formal reasons later in the semester. However, for binary or multiclass loss functions, we typically prefer the numerically more stable and efficient versions which bypass the need to explicitly compute the final layer's activation function (it's only implicitly computed).

Let's test this model just to be sure all's well here...

In [18]:
```
# Notice what numbers are used to set the input_size
# and output_size arguments to the contructor.
model = PLModel(neural_net).to(device)

print(model) # Not too useful, but some information can be gleaned...
```

```
PLModel(
  (module): NeuralNetwork(
    (output_layer): Linear(in_features=30, out_features=1, bias=True)
  )
  (network_acc): BinaryAccuracy()
  (network_loss): BCEWithLogitsLoss()
)
```

This is a new model object that works pretty much like the original `neural_net` that lacked the boiler plate, so let's see how it performs on the first 5 examples...

In [19]:
```
predictions = model.predict(torch.Tensor(X[:5,:]))
predictions = predictions.detach().numpy()
predictions
```

Out[19]:
```
array([[0.48700395],
       [0.46988097],
       [0.49984998],
       [0.50783885],
       [0.5448232 ]], dtype=float32)
```

In [20]:
```
binary_accuracy(torch.Tensor(predictions), # predicted
                torch.Tensor(Y[:5]))       # target
```

Out[20]:
```
tensor(0.2000)
```

Back to the optimizer...

An **optimizer** is a method for calculating (usually approximate) derivatives (a.k.a. the gradient) of the chosen *loss function* with respect to the parameters/weights (we choose `Adam()` above but we will discuss other options in lecture and below) and then updating the model parameters by making a step in the opposite direction of the gradient (in order to lower the loss). The **loss function** typically specifies how quantitatively dissimilar the current model predictions are from the provided targets (we choose `binary_cross_entropy()` here, but, again, alternatives will be discussed later). Finally, we add an appropriate instantiation of a **metric** which helps us understand performance (but which the optimizer *does not use* for fitting the weights/parameters (this is the same metric we explored above, `binary_accuracy()`.

We are now ready to train our model, but we will make a couple of typical but important choices.

1. We randomly permute the vectors in our data set, so that they do not have the same order in the X_shuffled and Y_shuffled arrays which we will use for training the neural net (`model`).

2. We will be separating these vectors into a **training set** and a **validation set**. The training set will be used to adjust the parameters in the model to try to minimize the loss and (hopefully, by extension) increase accuracy. However, the validation set will **never** be used to adjust the parameters: this will assess how well the model should perform on *new* vectors in the future after it has been trained.

3. Below, we will choose a `validation_split` which determines what fraction of the data is used for training and validation. We will use `0.3` which means that the top rows (first 70%) of the data in X_shuffled,Y_shuffled will be used for *training*, while the remaining bottom rows (last 30%) will be used for *validation*. The model will calculate loss and metric values for both sets during the training process. (The `training_step()` and `validation_step()` functions calculate these quantities for the training data and the validation data, respectively).

4. While there are many other choices to make for now, we also need to specify how much practice we would like to give the model. This is accomplished by choosing some number of training `epochs` for the fitting process. An epoch is one complete pass through all of the training data (and validation data). In other words, the network "practices" by going through the the vectors, making a prediction, and then adjusting weights to make the predictions closer to the targets in the future (by minimizing the loss function). The network learns "slowly" by following the negative gradient of the loss function with each subsequent epoch, or pass through the data. We will use 100 epochs for now, and explain other ways to determine this value later in the semester.

We will need a couple of other components from PyTorch Lightning: a `Logger` and a `Trainer`. Below you will see the `CSVLogger()` constructor being used to prepare a place to store statistics and the final parameters for the trained model. The appropriate hooks are provided by the `self.log()` calls in the `training_step()` and `validation_step()` function defined in the `PLModel` code above. The `Trainer()` constructor will create a training loop (automatically) for our model and connect our logger to this process. The trainer's `fit()` function can then be used to perform the learning process for us. However, we will need to provide the input vectors, `x_train` and `x_val`, and corresponding targets, `y_train` and `y_val`, as PyTorch tensors which can be collated using a PyTorch `DataLoader` object.

In [21]:
```
# Define a permutation needed to shuffle both
# inputs and targets in the same manner....
shuffle = np.random.permutation(X.shape[0])
X_shuffled = X[shuffle,:]
Y_shuffled = Y[shuffle,:]
```

In [22]:
```
# Keep 70% for training and remaining for validation
split_point = int(X_shuffled.shape[0] * 0.7)
x_train = X_shuffled[:split_point]
y_train = Y_shuffled[:split_point]
x_val = X_shuffled[split_point:]
y_val = Y_shuffled[split_point:]
```

The `DataLoader` class will take tensors and create data sets which can be broken up into `batch_size` chunks for training and/or validation. The `shuffle` argument is used here to indicate that the vectors should be resuffled (similar to what we did above) for *each new epoch*. Notice we don't use this feature for *validation* data, and this is typically best-practice since otherwise small trailing batches may randomly bias the results when averaging.

In [23]:
```
# The dataloaders handle shuffling, batching, etc...
xy_train = torch.utils.data.DataLoader(list(zip(torch.Tensor(x_train).type(torch.float),
                                                torch.Tensor(y_train).type(torch.float))),
                                       shuffle=True, batch_size=32)
xy_val = torch.utils.data.DataLoader(list(zip(torch.Tensor(x_val).type(torch.float),
                                              torch.Tensor(y_val).type(torch.float))),
                                     shuffle=False, batch_size=32)
```

The `CSVLogger` will generate data files in the `logs` subdirectory, and also in a subdirectory `Single-Layer-Network` under that. So, we will probably just use `logs` for most projects, but change this second argument for other projects/labs/models to make sure we don't overwrite any data. We will look at what is produced after we execute the training process below...

As for the `Trainer`, it will run for *at most* 100 epochs - even additional calls to the `fit()` function below will not allow this to proceed any farther to make sure we don't overwrite any of our logged data. There is also a nice progress bar, but we have to control how often statistics get logged (`log_every_n_steps=0` so we will only log at the end of an epoch) and we slow down the progress bar `refresh_rate` in order for it to behave well with JupyterLab.

In [24]:
```
logger = pl.loggers.CSVLogger("logs",name="Single-Layer-Network",)
trainer = pl.Trainer(max_epochs=100,logger=logger,
                     enable_progress_bar= True,
                     log_every_n_steps=0,
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

We can use the `validate()` function on the trainer to check the current performance of the model on the validation data *before* training, and then use it *after* training again to see if there was some improvement...

In [25]:
```
preliminary_result = trainer.validate(model, dataloaders=xy_val)
```

```
/opt/conda/lib/python3.11/site-packages/lightning/pytorch/trainer/connectors/data_connector.py:432: PossibleUserWarning: The dataloader, val_dataloader, does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` (try 32 which is the number of cpus on this machine) in the `DataLoader` init to improve performance.
  rank_zero_warn(
Validation: 0it [00:00, ?it/s]
```

Runningstage.validating metric	DataLoader 0
val_acc val_loss	0.2982456088066101 0.7105503082275391

This model isn't doing very well, right? So, let's perform 100 epochs of training and see if that changes...

In [26]:
```
trainer.fit(model, train_dataloaders=xy_train, val_dataloaders=xy_val)
```

```
  | Name         | Type              | Params
-----------------------------------------------------
0 | module       | NeuralNetwork     | 31
1 | network_acc  | BinaryAccuracy    | 0
2 | network_loss | BCEWithLogitsLoss | 0
-----------------------------------------------------
31        Trainable params
0         Non-trainable params
31        Total params
0.000     Total estimated model params size (MB)
Sanity Checking: 0it [00:00, ?it/s]
/opt/conda/lib/python3.11/site-packages/lightning/pytorch/trainer/connectors/data_connector.py:432: PossibleUserWarning: The dataloader, train_dataloader, does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` (try 32 which is the number of cpus on this machine) in the `DataLoader` init to improve performance.
  rank_zero_warn(
Training: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
```

```
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
```

```
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
`Trainer.fit` stopped: `max_epochs=100` reached.
```

In [27]:
```
final_result = trainer.validate(model, dataloaders=xy_val)
```

```
Validation: 0it [00:00, ?it/s]
```

Runningstage.validating metric	DataLoader 0
val_acc	0.9181286692619324
val_loss	0.3701685070991516

So, after training and re-evaluating the model, we are seeing some improvement (i.e. accuracy has increased and loss has decreased, compared to the earlier validation calculation).

A lot of detailed statistics are stored in the `logs/Single-Layer-Network` directory which we picked earlier...

In [28]:
```
! ls logs/Single-Layer-Network
```

```
version_0  version_2  version_4  version_6
version_1  version_3  version_5  version_7
```

Depending on how many times you have run the code above and retrained the model, you might see a differing number of directories listed. You will typically find your most recent run in the highest version directory seen above. Let's look in there...

In [29]:
```
! ls logs/Single-Layer-Network/version_7
```

```
checkpoints  hparams.yaml  metrics.csv
```

There is a `checkpoints` directory which actually contains a copy of the final parameters/weights of the model. While we still have those weights in-memory and can use them, if we needed to shut down our Python kernel for some reason we could load those paramaters/weights from the checkpoint. We will use this feature later-on, but for now we are more interested in the `metrics.csv` file which contains all of the logged statistics from the run above. Let's load that data and explore it a little bit..

In [30]:
```
results = pd.read_csv("logs/Single-Layer-Network/version_7/metrics.csv",
                      delimiter=',')
results
```

Out[30]:

	val_acc	val_loss	epoch	step	train_acc	train_loss
0	0.298246	0.710550	0	0	NaN	NaN
1	0.409357	0.700073	0	12	NaN	NaN
2	NaN	NaN	0	12	0.297229	0.708748
3	0.584795	0.691089	1	25	NaN	NaN
4	NaN	NaN	1	25	0.473552	0.699841
...	...	...	...	...	...	...
197	0.923977	0.371517	98	1286	NaN	NaN
198	NaN	NaN	98	1286	0.891688	0.373132
199	0.918129	0.370169	99	1299	NaN	NaN
200	NaN	NaN	99	1299	0.894207	0.371473
201	0.918129	0.370169	100	1300	NaN	NaN

202 rows × 6 columns

Quick example: The accuracy values for each epoch are being logged in the `val_acc` column, and the corresponding epoch number for each value can be obtained from the `epoch` column. Just use the correponding column name above to get those data. You can also see the names with Python...

In [31]:
```
results.columns
```

Out[31]:
```
Index(['val_acc', 'val_loss', 'epoch', 'step', 'train_acc', 'train_loss'], dtype='object')
```

In [32]:
```
val_acc = results["val_acc"]
val_acc
```

Out[32]:
```
0      0.298246
1      0.409357
2           NaN
3      0.584795
4           NaN
         ...
197    0.923977
198         NaN
199    0.918129
200         NaN
201    0.918129
Name: val_acc, Length: 202, dtype: float64
```

In [33]:
```
epoch = results["epoch"]
epoch
```

Out[33]:
```
0        0
1        0
2        0
3        1
4        1
      ...
197     98
198     98
199     99
200     99
201    100
Name: epoch, Length: 202, dtype: int64
```

Validation and loss statistics (the means across all batches for each epoch) are logged alternately. A validation epoch is performed right at the beginning. Training performance and then validation performance gets retested and logged at the end of each training epoch. Since these operations may or may not be performed depending on how you set up your training process, it can leave `NaN` values throughout the data columns.

Numpy can easily be used to clean out the `NaN` values from a particular column and match it with the corresponding entries in the `epoch` column. We typically do this for both training and validation so that both can be plotted on the same graph using Matplotlib.

In [34]:
```
# Just to get a clean vector for epochs and train_loss
results["epoch"][np.logical_not(np.isnan(results["train_loss"]))]
```

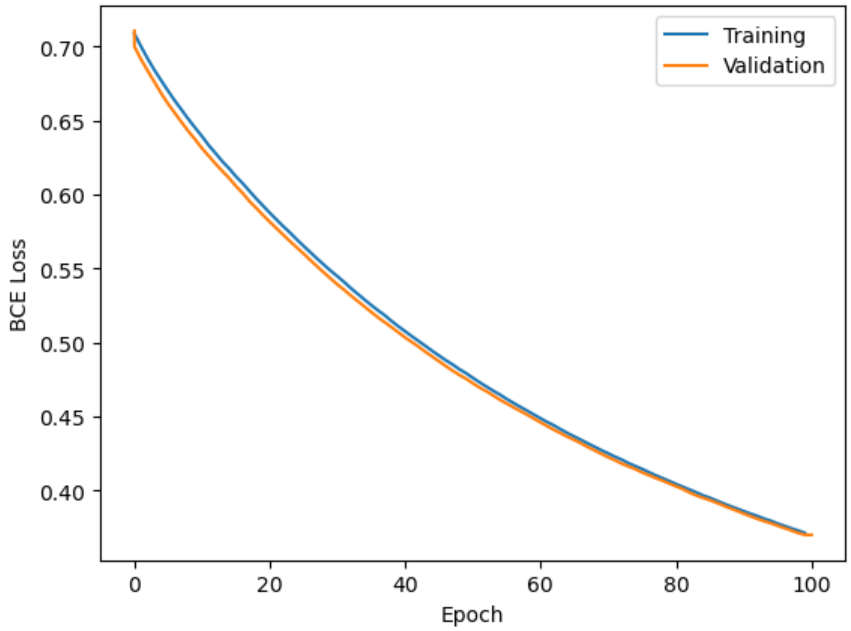
Out[34]:
```
2       0
4       1
6       2
8       3
10      4
       ..
192    95
194    96
196    97
198    98
200    99
Name: epoch, Length: 100, dtype: int64
```

In [35]:
```
results["train_loss"][np.logical_not(np.isnan(results["train_loss"]))]
```

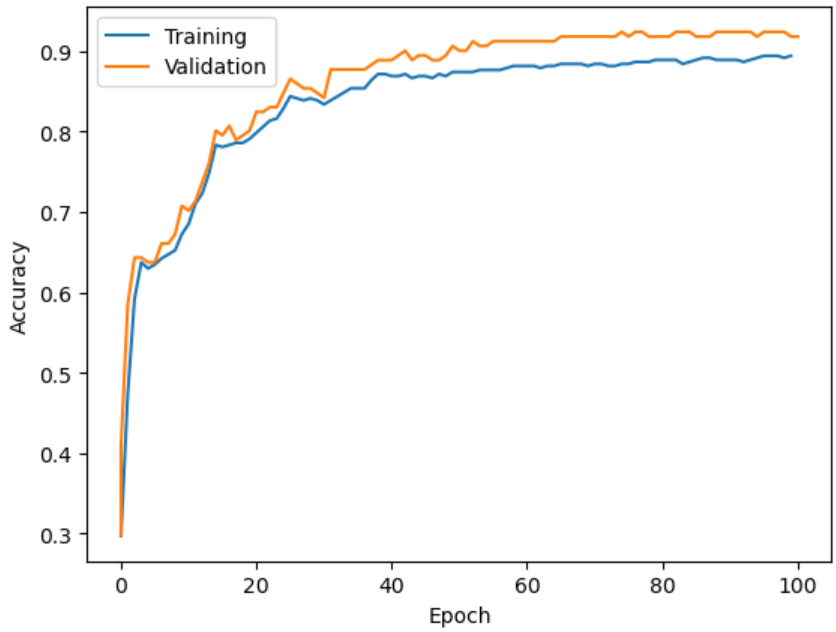
Out[35]:
```
2      0.708748
4      0.699841
6      0.691512
8      0.683780
10     0.676774
         ...
192    0.377775
194    0.376224
196    0.374618
198    0.373132
200    0.371473
Name: train_loss, Length: 100, dtype: float64
```

Now that we have two vectors of the same length, we can plot `epoch` on the x-axis and `train_loss` on the y-axis (and any other quantities can be treated similarly).

In [36]:
```
plt.plot(results["epoch"][np.logical_not(np.isnan(results["train_loss"]))],
         results["train_loss"][np.logical_not(np.isnan(results["train_loss"]))],
         label="Training")
plt.plot(results["epoch"][np.logical_not(np.isnan(results["val_loss"]))],
         results["val_loss"][np.logical_not(np.isnan(results["val_loss"]))],
         label="Validation")
plt.legend()
plt.ylabel("BCE Loss")
plt.xlabel("Epoch")
plt.show()
```



In [37]:
```
plt.plot(results["epoch"][np.logical_not(np.isnan(results["train_acc"]))],
         results["train_acc"][np.logical_not(np.isnan(results["train_acc"]))],
         label="Training")
plt.plot(results["epoch"][np.logical_not(np.isnan(results["val_acc"]))],
         results["val_acc"][np.logical_not(np.isnan(results["val_acc"]))],
         label="Validation")
plt.legend()
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
plt.show()
```



In the assignment information below, you will need to sample these vectors every 10 epochs for generating some statistical plots and figures. Here is one method to subsample the `val_acc` list which is the expected accuracy for the network on the *validation data* (or how we expect it to perform in the future). Note that even though the other statistics are informative in their own ways, it's really good performance on future data that is important. Performing well on future data means that the network can **generalize** it's learning knowledge or apply it to future examples - **all gained from the experience (E) with the training data by minimizing loss which is the performance measure (P) in order to perform the cancer classification task (T)**.

In [38]:
```
print("Validation accuracy:",*["%.8f"%(x) for x in
                                results['val_acc'][np.logical_not(np.isnan(results["val_acc"]))][0::10]])
```

```
Validation accuracy: 0.29824561 0.70760232 0.80116957 0.84795320 0.88888890 0.90643275 0.91228068 0.9181286
7 0.91812867 0.92397660 0.91812867
```

Single-layer with Multi-class Targets

There are a few more tools in the toolbox needed if the chosen task has more than two classes to predict, so let's walk through those tools as well much more quickly...

We will use the Iris data set, which contains measurement vectors of length 4 taken from 3 different species of Iris. The input data vector length will be smaller than WDBC above, but the target vectors can contain a value from the set {0, 1, 2}, corresponding to the three iris species. We need to make a network capable of handling three classes this time around.

More details on the properties of the measurements can be found here or here (remember, I have preprocessed the raw data a little to make it easier to work with from the start for this lab).

In [39]:
```
data = np.loadtxt("https://www.cs.mtsu.edu/~jphillips/courses/CSCI7850/public/iris-data.txt")
data[0:5]
```

Out[39]:
```
array([[5.8, 2.7, 3.9, 1.2, 1. ],
       [6.9, 3.1, 5.4, 2.1, 2. ],
       [7.7, 3. , 6.1, 2.3, 2. ],
       [5.7, 3. , 4.2, 1.2, 1. ],
       [5.1, 3.7, 1.5, 0.4, 0. ]])
```

In [40]:
```
data.shape
```

Out[40]:
```
(150, 5)
```

In [41]:
```
X = data[:,:-1]
X.shape
```

Out[41]:
```
(150, 4)
```

In [42]:
```
Y = data[:,-1:]
Y.shape
```

Out[42]:
```
(150, 1)
```

In [43]:
```
len(np.unique(Y))
```

Out[43]:
```
3
```

Carefully note the number of units in the output layer and the choice of activation function...

In [44]:
```
# Define neural network
class NeuralNetwork(torch.nn.Module):
    def __init__(self, input_size, output_size, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor

        # This is an actual neural layer...
        self.output_layer = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        # Here is where we use the layers to compute something...
        y = x # Start with the input
        y = self.output_layer(y) # y replaces y (stateful parameters)
        return y # Final calculation returned

    # Separate the final activation function out because
    # CrossEntropyLoss assumes you are using a softmax
    # (outputs are considered logits - more later on this...)
    def predict(self, x):
        # Here is where we use the layers to compute something...
        y = x
        y = self.forward(y) # Start with the input
        y = torch.softmax(y,-1) # Apply the activation function (stateless)
        return y # Final calculation returned

# Notice what numbers are used to set the input_size
# and output_size arguments to the contructor.
neural_net = NeuralNetwork(X.shape[-1],
                           len(np.unique(Y))).to(device)

print(neural_net) # Not too useful, but some information can be gleaned...
```

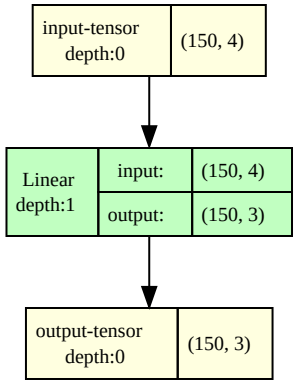
```
NeuralNetwork(
  (output_layer): Linear(in_features=4, out_features=3, bias=True)
)
```

In [45]:
```
summary(neural_net, input_size=X.shape)
```

Out[45]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
NeuralNetwork                            [150, 3]                  --
├─Linear: 1-1                            [150, 3]                  15
==========================================================================================
Total params: 15
Trainable params: 15
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.01
==========================================================================================
```

In [46]:
```
model_graph = draw_graph(neural_net, input_size=X.shape,
                         device=device, depth=1)
model_graph.visual_graph
```

Out[46]:



In [47]:
```
predictions = neural_net.predict(torch.Tensor(X[:5])).detach().numpy()
predictions
```

Out[47]:
```
array([[0.10801835, 0.01615885, 0.8758228 ],
       [0.07654689, 0.00558362, 0.91786957],
       [0.05564503, 0.00349226, 0.9408627 ],
       [0.11239303, 0.01458577, 0.87302125],
       [0.1398396 , 0.03459132, 0.82556903]], dtype=float32)
```

In [48]:
```
predictions.shape
```

Out[48]:
```
(5, 3)
```

In [49]:
```
Y[:5,:]
```

Out[49]:
```
array([[1.],
       [2.],
       [2.],
       [1.],
       [0.]])
```

Outputs represent probabilities, so integer targets are mapped to the corresponding output unit (highest probability among the three predictions is assummed to the the class chosen by the network). The vectors, Y, are therefore much more *sparse* (length 1) than the prediction vectors (length 3), so we employ a **sparse version of the accuracy and loss below**.

In [50]:
```
mca = torchmetrics.classification.Accuracy(task='multiclass',
                                           num_classes=len(np.unique(Y)))
```

In [51]:
```
mca(torch.Tensor(predictions),
    torch.Tensor(Y[:5,0])) # (predicted, target)
```

Out[51]:
```
tensor(0.4000)
```

In [52]:
```
# Define Trainable Module
class PLModel(pl.LightningModule):
    def __init__(self, module, num_classes, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor
        self.module = module

        # This creates an accuracy function
        self.network_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                                 num_classes=num_classes)
        # This creates a loss function
        self.network_loss = torch.nn.CrossEntropyLoss()

    def forward(self, x):
        return self.module.forward(x)

    def predict(self, x):
        return self.module.predict(x)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.01)
        return optimizer
```

```
    def training_step(self, train_batch, batch_idx):
        x, y_true = train_batch
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    def validation_step(self, val_batch, batch_idx):
        x, y_true = val_batch
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    def testing_step(self, test_batch, batch_idx):
        x, y_true = test_batch
        y_pred = self(x)
        acc = self.network_acc(y_pred,y_true)
        loss = self.network_loss(y_pred,y_true)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss
```

Instantiate a trainable model...

In [53]:
```
model = PLModel(neural_net,len(np.unique(Y)))
```

Hyperparameters: while the 15 parameters in the model above will be adjusted using the error backpropagation algorithm (specifically, a modified variant of stochastic gradient descent - SGD) during the learning process, there are some other settings which control how this learning process proceeds. The weights in the model are **parameters**, but any other settings which impact the behavior of the model are **hyperparameters**.

You can think of hyperparameters as having an indirect impact on the weights/parameters instead of a direct impact like SGD. We (or perhaps some meta-optimizer) are required to tune these hyperparameters for the given task - and every task is a little different. However, a good rule of thumb is that simple problems can be solved more quickly and so the rate of convergence can be increased to find good solutions quickly. Above, I have updated the `learning_rate` parameter to `0.01` since the default value of `0.001` is too slow and can actually *hinder* the model in it's search to solve the task. Be mindful of your tasks and understand that, to our current knowledge, there is no computationally tractable approach for hyperparameter tuning that will guarantee successful learning.

In [54]:
```
# Define a permutation needed to shuffle both
# inputs and targets in the same manner....
shuffle = np.random.permutation(X.shape[0])
X_shuffled = X[shuffle,:]
Y_shuffled = Y[shuffle,:]
```

In [55]:
```
# Keep 70% for training and remaining for validation
split_point = int(X_shuffled.shape[0] * 0.7)
x_train = X_shuffled[:split_point]
y_train = Y_shuffled[:split_point,0]
x_val = X_shuffled[split_point:]
y_val = Y_shuffled[split_point:,0]
```

In [56]:
```
# The dataloaders handle shuffling, batching, etc...
xy_train = torch.utils.data.DataLoader(list(zip(torch.Tensor(x_train).type(torch.float),
                                                torch.Tensor(y_train).type(torch.long))),
                                       shuffle=True, batch_size=32)
xy_val = torch.utils.data.DataLoader(list(zip(torch.Tensor(x_val).type(torch.float),
                                              torch.Tensor(y_val).type(torch.long))),
                                       shuffle=False, batch_size=32)
```

In [57]:
```
logger = pl.loggers.CSVLogger("logs",name="Iris-Network",)
trainer = pl.Trainer(max_epochs=100,logger=logger,
                     enable_progress_bar= True,
```

```
                    log_every_n_steps=0,
                    callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=50)])
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

In [58]:
```
preliminary_result = trainer.validate(model, dataloaders=xy_val)
```

```
/opt/conda/lib/python3.11/site-packages/lightning/pytorch/trainer/connectors/data_connector.py:432: Possibl
eUserWarning: The dataloader, val_dataloader, does not have many workers which may be a bottleneck. Conside
r increasing the value of the `num_workers` argument` (try 32 which is the number of cpus on this machine)
in the `DataLoader` init to improve performance.
  rank_zero_warn(
Validation: 0it [00:00, ?it/s]
```

Runningstage.validating metric	DataLoader 0
val_acc	0.24444444477558136
val_loss	2.6612234115600586

In [59]:
```
trainer.fit(model, train_dataloaders=xy_train, val_dataloaders=xy_val)
```

```
  | Name         | Type               | Params
----------------------------------------------------
0 | module       | NeuralNetwork      | 15
1 | network_acc  | MulticlassAccuracy | 0
2 | network_loss | CrossEntropyLoss   | 0
----------------------------------------------------
15        Trainable params
0         Non-trainable params
15        Total params
0.000     Total estimated model params size (MB)
Sanity Checking: 0it [00:00, ?it/s]
/opt/conda/lib/python3.11/site-packages/lightning/pytorch/trainer/connectors/data_connector.py:432: Possibl
eUserWarning: The dataloader, train_dataloader, does not have many workers which may be a bottleneck. Consi
der increasing the value of the `num_workers` argument` (try 32 which is the number of cpus on this machin
e) in the `DataLoader` init to improve performance.
  rank_zero_warn(
Training: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
```

```
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
```

```
`Trainer.fit` stopped: `max_epochs=100` reached.
```

In [60]:

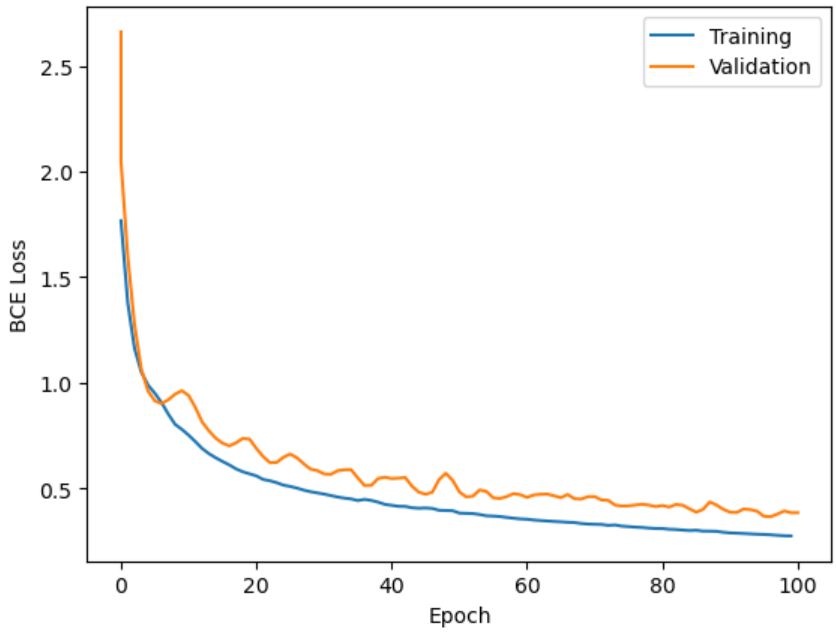
```python
final_result = trainer.validate(model, dataloaders=xy_val)
```

```
Validation: 0it [00:00, ?it/s]
```

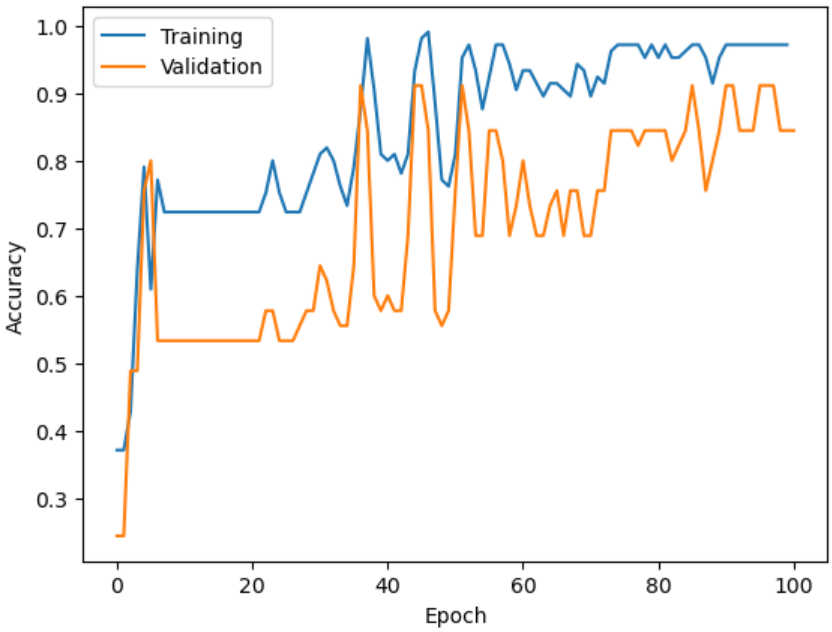
Runningstage.validating metric	DataLoader 0
val_acc	0.8444444537162781
val_loss	0.38433724641799927

In [63]:
```
results = pd.read_csv("logs/Iris-Network/version_10/metrics.csv",
                      delimiter=',')
```

In [64]:
```
plt.plot(results["epoch"][np.logical_not(np.isnan(results["train_loss"]))],
         results["train_loss"][np.logical_not(np.isnan(results["train_loss"]))],
         label="Training")
plt.plot(results["epoch"][np.logical_not(np.isnan(results["val_loss"]))],
         results["val_loss"][np.logical_not(np.isnan(results["val_loss"]))],
         label="Validation")
plt.legend()
plt.ylabel("BCE Loss")
plt.xlabel("Epoch")
plt.show()
```



In [65]:
```
plt.plot(results["epoch"][np.logical_not(np.isnan(results["train_acc"]))],
         results["train_acc"][np.logical_not(np.isnan(results["train_acc"]))],
         label="Training")
plt.plot(results["epoch"][np.logical_not(np.isnan(results["val_acc"]))],
         results["val_acc"][np.logical_not(np.isnan(results["val_acc"]))],
         label="Validation")
plt.legend()
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
plt.show()
```



In [66]:
```
print("Validation accuracy:",*["%.8f"%(x) for x in
                                results['val_acc'][np.logical_not(np.isnan(results["val_acc"]))][0::10]])
```

```
Validation accuracy: 0.24444444 0.53333336 0.53333336 0.57777780 0.57777780 0.57777780 0.73333335 0.6888889
1 0.84444445 0.84444445 0.84444445
```

We have learned to predict different species of Iris!

Standardization

While many data sets consist of measurements which use the same units or operate in similar ranges, this is not always the case. Some data therefore should be standardized to ensure that the model doesn't underutilize or ignore small-magnitude measurements/features and also doesn't overutilize or be overwhelmed by large-magnitude measurements/features.

While many techniques for performing standardization exist, one common method is mean-centering each measurement type (subtracting the mean value by column), and then scaling the magnitudes of the measurements (dividing by the standard deviation by column). This means that the mean value of each measurement type in the data set will be zero, and the standard deviation of each measurement type will be one. This allows for roughly equal attention to all of the features in the data set, and can possibly improve performance.

This standardization can be performed on the input vectors of a data set, X, using the following:

In [67]:
```
def preprocess(x):
    return (x - np.mean(x)) / np.std(x)
```

Note that we create a function for performing the process on a single *column vector* above. This can be applied to all columns in X by using `apply_along_axis`.

In [68]:
```
X_preprocessed = np.apply_along_axis(preprocess,0,X)
X_preprocessed.shape
```

Out[68]:
```
(150, 4)
```

We can validate that the mean-centering and scaling has had the intended effect by examing the mean and standard deviation of the first column of the data before (X) and after (X_preprocessed) standardization.

In [69]:
```
np.mean(X[:,0])
```

Out[69]:
```
5.843333333333334
```

In [70]:
```
np.mean(X_preprocessed[:,0])
```

```
Out[70]: -4.1966430330830915e-16
```

```
In [71]: np.std(X[:,0])
```

```
Out[71]: 0.8253012917851409
```

```
In [72]: np.std(X_preprocessed[:,0])
```

```
Out[72]: 1.0
```

Is there a better way than standardization?

Yes, there are approaches for including standardization procedures directly into our models which we will study later in the semester. Ideally, we would prefer to not have to perform such standardization ourselves since we can't be sure if we are standardizing our data in a way that will aid learning. Deep learning is often preferred to other machine learning approaches specifically because it is a flexible framework which can include ways to *learn* the appropriate transformations without direct programmer interference (and potential bias).

However, it is important to demonstrate how different standardization techniques are useful by applying them in-practice. Once we know that a particular strategy is useful in certain cases, then we can add this to our models in a way that lets them learn to use these techniques when they are helpful, but also in a manner which allows the model to ignore those suggestions when they are unhelpful.

Optimizers

There are other **optimizers** (part of the `torch.optim` module) which may also be used to minimize the loss function. Some options are `Adam()` (like we used above), `RMSprop()`, or `SGD()`. We will look at them more in class, but you can utilize any of them with default parameters in any of the above examples.

Important:

Only use the default `learning_rate` settings (and any other arguments) for all of these optimizers for the problems below. We are going to try to see how they behave "out of the box" which should give us some insights about how important (and difficult) hyperparameter selection can actually be.

Scripting this process

You can always utilize the tools above outside of the notebook environment, but it's wise to remove the matplotlib tools and other interactive/exploratory parts of your code before doing so since a script will be run from the command line and therefore does not have the same level of interactivity as what we are using above. An example of doing this for the WDBC data would look similar to the following (not all details above are included, nor all details needed for the assignment below):

```
#!/usr/bin/env python3

import numpy as np
import torch
import lightning.pytorch as pl
from torchinfo import summary
from torchview import draw_graph
import matplotlib.pyplot as plt
import pandas as pd

data = np.loadtxt("https://www.cs.mtsu.edu/~jphillips/courses/CSCI7850/public/WDBC.txt")
X = data[:,:-1]
Y = data[:,-1:]

# Note that other code goes here...

print("Validation accuracy:",*["%.8f"%(x) for x in
                                results['val_acc'][np.logical_not(np.isnan(results["val_acc"]))]
[0::10]])
```

If you put this in a file named `OL1.py`, then you can use the `python3 OL1.py` command from the terminal to run the script. Alternatively, you can add executable permissions to the script using `chmod +x OL.py` and then you can simply run `./OL1.py` assuming the script is in the current directory.

Also, you can run this as a batch job on the SLURM scheduler on the cluster if you wrap it in a shell script which passes it through singularity.

```
#!/bin/sh

singularity exec /home/shared/sif/csci-2023-Fall.sif python3 OL1.py
```

One other reminder is that you can provide command-line arguments to a script and utilize the `sys` module to parse them appropriately:

```
#!/usr/bin/env python3

import sys

print('Number of arguments:', len(sys.argv))
print('List of arguments:', str(sys.argv))
print('The first argument:', sys.argv[0])
```