09.2 - Deep Learning - Deep Neural Network

November 2, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 9.2 Deep Learning - Deep Neural Network

1.1.1 Readings

- [WEIDMAN] Ch3
- [CHARU] Ch2-3

As a recap, last time, we have inputted our data into a linear function wwhich we got some decent result. To improve, we inserted a non-linear function in between follow by a linear function, which obviously increase the result since it help model the non-linearity. We can summarize that neural newtork has basically the following key things that make it work:

1. **Activation function (we can also generalized as Operations)**: these functions help model input data into a non-linear relationship
2. **Chain rule / Backpropagation**: they are essential for us to improve the neural network
3. **Layers of neurons**: they are performing some sequential processes that split out desired output.

Putting together, the typial procedure of training a neural network is as followss:

1. Feed observations/samples/records (X) into the model. This step we called "**forward pass**"
2. Calculate the loss
3. Calculate gradients based on how each parameters (e.g., W, B) affect the loss by using chain rule. This step was called "**backward pass**"
4. Update the parameters (e.g., W, B) so that the loss will be hopefully be reduced in the next iteration. This step was called "**training**"
5. Stop when the loss does not decrease further by some tolerance level (e.g., 0.00001) or when it exceeds the specified maximum iteration. Sometimes we called this "**early stopping**"

In fact, you are now very close to understanding Deep Neural Networks. In this lesson, we have several objectives:

- From our low-level understandings of neural network, we shall code them up as a Python class, so they are resuable. They will be essential for understanding deep neural network, CNN, and RNN. You will be so surprised that all these fancy terms are simply layers after layers

- When we code our work, we want to make sure these classes resemble PyTorch as much as possible, so you will understand PyTorch right away.
- Of course, we shall also understand what is "deep" neural network. Here, we shall simply say that "deep" neural network is simply neural network that has more than "one" hidden layers (which we did not yet define what is "hidden" layers)

1.1.2 1. Operations

Let's first code up the first building block, the class Operation, which is the operations/functions.

Each function has a **forward** and **backward** methods. Forward methods for running the function and backward for calculating its gradients.

Each of these functions receives an ndarray as input and outputs an ndarray. In some operations such as matrix multiplication, we receive ndarray as params, thus we probably should have another class inheriting from Operation and allow for params as another instance variable.

We also need to note that the shape of the output may vary. For example, in matrix multiplication, the shape of output will be different from shape of input. In sigmoid, input and output shares the same shape. To make sure the shape is consistent, we can follow these facts:

1. Each Operation will send outputs forward on the forward pass and will receive an "output gradient" on the backward pass, which will represent the partial derivative of the loss with respect to every element of the Operation's output. Thus **The shape of the output gradient ndarray must match the shape of the output.**
2. On the backward pass, each Operation will send an "input gradient" backward, representing the partial derivative of the loss with respect to each element of the input. **The shape of the input gradient that the Operation sends backward during the backward pass must match the shape of the Operation's input.**

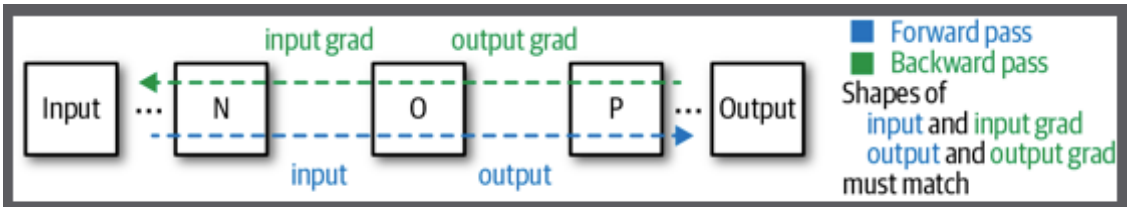


Figure 3-1. An Operation, with input and output

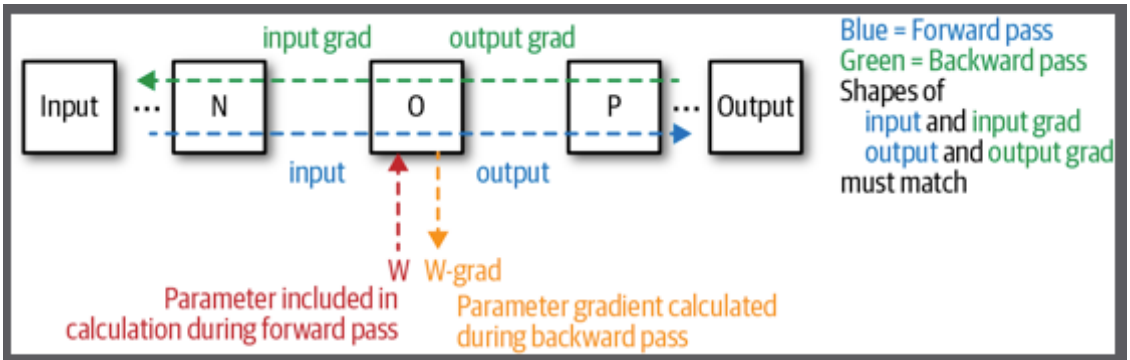


Figure 3-2. A ParamOperation, with input and output and parameters

Based on this, we can write the class Operation like this:

[1]:
```
from numpy import ndarray
from typing import List
import numpy as np
from time import time


class Operation(object):

    #nothing to init
    def __init__(self):
        pass

    #forward receive ndarray as input
    def forward(self, input_: ndarray) -> ndarray:
        #put trailing _ to avoid naming conflict
        self.input_ = input_

        #this _output will use self.input_ to calculate the ouput
        #_  here means internal use
        self.output = self._output()

        return self.output


    def backward(self, output_grad: ndarray) -> ndarray:

        #make sure output and output_grad has same shape
        assert self.output.shape == output_grad.shape

        #perform input grad based on output_grad
        self.input_grad = self._input_grad(output_grad)
```

```
        #input grad must have same shape as input
        assert self.input_.shape == self.input_grad.shape

        return self.input_grad

    def _output(self) -> ndarray:
        raise NotImplementedError()

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        raise NotImplementedError()
```

Let's add also another class that inherits from Operation that we'll use specifically for Operations that involve parameters.

[2]:
```
class ParamOperation(Operation):
    def __init__(self, param: ndarray):
        super().__init__()  #inherit from parent if any
        self.param = param  #this will be used in _output

    def backward(self, output_grad: ndarray) -> ndarray:

        #make sure output and output_grad has same shape
        assert self.output.shape == output_grad.shape

        #perform gradients for both input and param
        self.input_grad = self._input_grad(output_grad)
        self.param_grad = self._param_grad(output_grad)

        assert self.input_.shape == self.input_grad.shape
        assert self.param.shape == self.param_grad.shape

        return self.input_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        raise NotImplementedError()
```

Let's implement some functions that we implement in last class, including: 1. Matrix multiplication 2. Addition of bias term 3. Sigmoid activation function

Lets start with matrix multiplication. Since the input has two params, X and W, we inherit from ParamOperation.

[3]:
```
class WeightMultiply(ParamOperation):

    def __init__(self, W: ndarray):
        #initialize Operation with self.param = W
        super().__init__(W)
```

```
    def _output(self) -> ndarray:
        return self.input_ @ self.param

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad @ self.param.T  #same as last class

    def _param_grad(self, output_grad: ndarray)  -> ndarray:
        return self.input_.T @ output_grad  #same as last class
```

Next is the BiasAdd operation where the gradients are simply one. Since it is an operation between X and B, we inherit from ParamOperation.

[4]:
```
class BiasAdd(ParamOperation):
    def __init__(self, B: ndarray):
        #initialize Operation with self.param = B.
        assert B.shape[0] == 1  #make sure it's only B
        super().__init__(B)

    def _output(self) -> ndarray:
        return self.input_ + self.param

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return np.ones_like(self.input_) * output_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        param_grad = np.ones_like(self.param) * output_grad
        return np.sum(param_grad, axis=0).reshape(1, param_grad.shape[1])
```

Finally, let's do sigmoid. Since sigmoid is simply a operation that maps to another value, it inherits from Operation:

[5]:
```
class Sigmoid(Operation):
    def __init__(self):
        super().__init__()

    def _output(self) -> ndarray:
        return 1.0/(1.0 + np.exp(-1.0 * self.input_))

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        sigmoid_backward = self.output * (1.0 - self.output)
        input_grad = sigmoid_backward * output_grad
        return input_grad
```

Let's also code up Linear activation function which does nothing. We can use this for Linear Regression since it does not have any activation function:

[6]:
```
class Linear(Operation):
    def __init__(self):
```

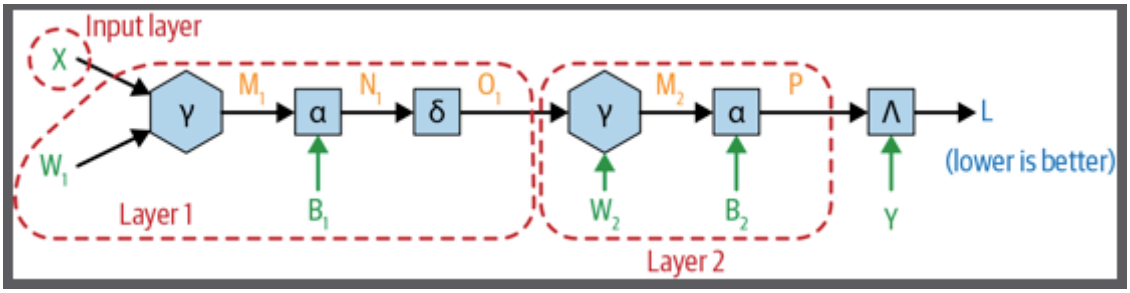
```
        super().__init__()

    def _output(self) -> ndarray:
        return self.input_

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad
```

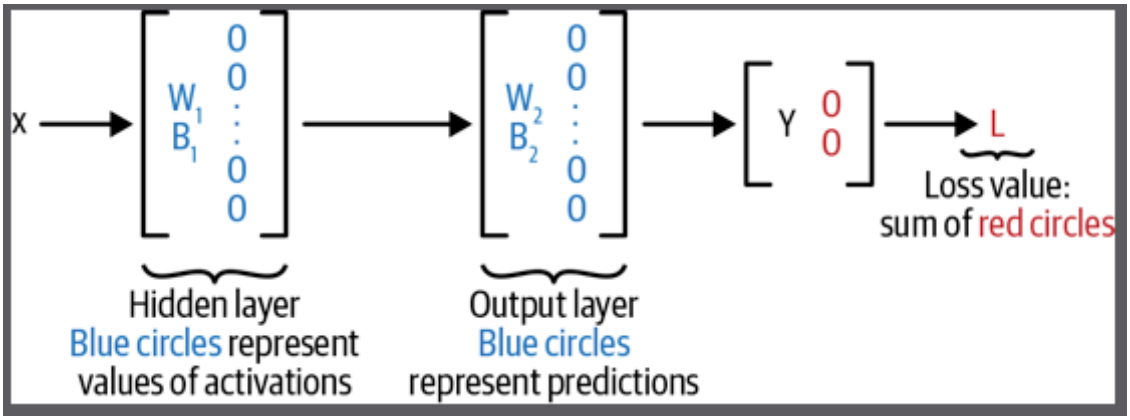
1.1.3 2. Layers

In terms of Operations, layers are a series of linear operations followed by a nonlinear operation. For example, our neural network from the last chapter could be said to have had five total operations: two linear operations — a weight multiplication and the addition of a bias term — followed the sigmoid function and then two more linear operations.



Here, we define the input as **input layer**, Layer 1 is typically called **hidden layer** because it is the only layer whose values we don't typically see explicitly during the course of training. Layer 2 is typically called **output layer** which outputs the desired value.

By abstraction, we can make neural network look much simpler as follows:



Each layer can be said to have a certain number of neurons equal to the dimensionality of the vector that represents each observation in the layer's output. The neural network from the last class can thus be thought of as having 13 neurons in the input layer (i.e., 13 features), then 13 neurons (again) in the hidden layer, and one neuron in the output layer.

Neurons in the brain have the property that they can receive inputs from many other neurons and will "fire" and send a signal forward only if the signals they receive cumulatively reach a certain

"activation energy." Neurons in the context of neural networks have a loosely analogous property: they do indeed send signals forward based on their inputs, but the inputs are transformed into outputs simply via a nonlinear function. Thus, this nonlinear function is called the activation function, and the values that come out of it are called the activations for that layer.

Building on the context of layers, deep learning models are simply neural networks with more than one hidden layer.

Now leaving all the theory behind, let's code the Layer together:

[7]:
```
class Layer(object):
    def __init__(self, neurons: int):
        self.neurons = neurons
        self.first = True   #first layer is true for init
        self.params: List[ndarray] = []
        self.param_grads: List[ndarray] = []
        self.operations: List[Operation] = []

    def _setup_layer(self, num_in: int):
        #setup the series of operations
        raise NotImplementedError()

    def forward(self, input_: ndarray) -> ndarray:
        #setup self.operations if haven't
        if self.first:
            self._setup_layer(input_)
            self.first = False

        self.input_ = input_

        #run the series of operations
        for operation in self.operations:
            input_ = operation.forward(input_)

        self.output = input_

        return self.output

    def backward(self, output_grad: ndarray) -> ndarray:

        assert self.output.shape == output_grad.shape

        for operation in reversed(self.operations):
            output_grad = operation.backward(output_grad)

        input_grad = output_grad

        self._param_grads()
```

```
        return input_grad

    #if the operation is a subclass of ParamOperatio
    #append param_grad to self.param_grads
    def _param_grads(self):
        self.param_grads = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.param_grads.append(operation.param_grad)

    def _params(self):
        self.params = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.params.append(operation.param)
```

Now, let's create our layer. Remember that we have three layers:

1. Input layer
2. Hidden layer
3. Output layer

We don't really need to implement the input layer since it's only the input.

As for our hidden layer, it composes of WeightMultiply, then BiasAdd, then sigmoid. What name should we give to this layer? How about LinearNonLinear layer. In fact, there is a common name for this is "**Dense/Fully-Connected Layer**" which refers to layer where each output neuron is a function of all of the input neurons. Imagine thirteen circles, each circle connected to all circles…(that's why it's called fully-connected)

Our output layer is very similar to the hidden layer but without the hidden layer. We consider this still as a **Dense** layer because each output neuron is again connected to all input neurons.

To code this is simple, we simply inherit **Layers** and define the series of operations in _setup_layer function

[8]:
```
class Dense(Layer):
    def __init__(self, neurons: int,
                 activation: Operation = Sigmoid()):
        #define the desired non-linear function as activation
        super().__init__(neurons)
        self.activation = activation

    def _setup_layer(self, input_: ndarray):
        #in case you want reproducible results
        if self.seed:
            np.random.seed(self.seed)

        self.params = []
```

```
        # randomize weights of shape (num_feature, num_neurons)
        self.params.append(np.random.randn(input_.shape[1], self.neurons))

        # randomize bias of shape (1, num_neurons)
        self.params.append(np.random.randn(1, self.neurons))

        self.operations = [WeightMultiply(self.params[0]),
                           BiasAdd(self.params[1]),
                           self.activation]
```

1.1.4 3. Loss Class

The next thing we have to code up is the loss function (forward) and its gradients (backward). We gonna make a parent class called Loss and a child class called MeanSquaredError The code is quite straightforward, similar to Layers

[9]:
```
class Loss(object):

    def __init__(self):
        pass

    def forward(self, prediction: ndarray, target: ndarray) -> float:
        assert prediction.shape == target.shape

        self.prediction = prediction
        self.target = target

        #self._output will hold the loss function
        loss_value = self._output()

        return loss_value

    def backward(self) -> ndarray:

        self.input_grad = self._input_grad()

        assert self.prediction.shape == self.input_grad.shape

        #input_grad will hold the gradient of the loss function
        return self.input_grad

    def _output(self) -> float:
        raise NotImplementedError()

    def _input_grad(self) -> ndarray:
        raise NotImplementedError()
```

Now we have the Loss/Objective/Cost function, let's make the concrete loss function. Here we will be using the MeanSquaredError

[10]:
```
class MeanSquaredError(Loss):

    def __init__(self):
        super().__init__()

    def _output(self) -> float:
        loss = (
            np.sum(np.power(self.prediction - self.target, 2)) /
            self.prediction.shape[0]
        )

        return loss

    def _input_grad(self) -> ndarray:
        return 2.0 * (self.prediction - self.target) / self.prediction.shape[0]
```

1.1.5 4. NeuralNetwork

Now that we have abstracted low-level Operations into Layers, we can also further abstract bunch of Layers into a NeuralNetwork class. We can also plug in our Loss into our NeuralNetwork class.

For coding implementation:

The structure can be summarized as follows:

1. A NeuralNetwork will have a list of Layers as an attribute. The Layers would be as defined previously, with forward and backward methods. These methods take in ndarray objects and return ndarray objects.
2. Each Layer will have a list of Operations saved in the operations attribute of the layer during the _setup_layer function.
3. These Operations, just like the Layer itself, have forward and backward methods that take in ndarray objects as arguments and return ndarray objects as outputs.
4. In each operation, the shape of the output_grad received in the backward method must be the same as the shape of the output attribute of the Layer. The same is true for the shapes of the input_grad passed backward during the backward method and the input_ attribute.
5. Some operations have parameters (stored in the param attribute); these operations inherit from the ParamOperation class. The same constraints on input and output shapes apply to Layers and their forward and backward methods as well—they take in ndarray objects and output ndarray objects, and the shapes of the input and output attributes and their corresponding gradients must match.
6. A NeuralNetwork will also have a Loss. This class will take the output of the last operation from the NeuralNetwork and the target, check that their shapes are the same, and calculate both a loss value (a number) and an ndarray loss_grad that will be fed into the output layer, starting backpropagation.

In terms of processes, we can describe as follows:

1. Receive X and y as inputs, both ndarrays.
2. Feed X successively forward through each Layer.
3. Use the Loss to produce loss value and the loss gradient to be sent backward.
4. Use the loss gradient as input to the backward method for the network, which will calculate the param_grads for each layer in the network.
5. Call the update_params function on each layer, which will use the overall learning rate for the NeuralNetwork as well as the newly calculated param_grads.

Without further ado, let's code it now!

[11]:
```
class NeuralNetwork(object):
    def __init__(self,
                 layers: List[Layer],
                 loss: Loss,
                 seed: int = 1):
        self.layers = layers
        self.loss = loss
        self.seed = seed
        if seed:
            for layer in self.layers:
                setattr(layer, "seed", self.seed)

    def forward(self, x_batch: ndarray) -> ndarray:
        x_out = x_batch
        for layer in self.layers:
            x_out = layer.forward(x_out)

        return x_out

    def backward(self, loss_grad: ndarray):
        grad = loss_grad
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

            #you may wonder why I did not return anything
            #it's because in Layer.backward, it is appending this value to␣
↪param_grads to each layer
            #this return "grad" is simply something it returns

    def train_batch(self,
                    x_batch: ndarray,
                    y_batch: ndarray) -> float:

        predictions = self.forward(x_batch)
```

```
        loss = self.loss.forward(predictions, y_batch)
        self.backward(self.loss.backward())

        return loss

    def params(self):
        #get the parameters for the network
        #use for updating w and b
        for layer in self.layers:
            #yield is different from return is that
            #it will return a generator
            #what's amazing is that you can manipulate
            #the values and change
            yield from layer.params

    def param_grads(self):
        #get the gradient of the loss with respect to the parameters
        #for the network
        #use for updating w and b
        for layer in self.layers:
            yield from layer.param_grads
```

With this NeuralNetwork class, we can implement the models in a more modular, flexible way and define other models to represent complex nonlinear relationships between input and output. For example, here's how to easily instantiate the two models we covered in the last chapter— the linear regression, the neural network, and the deep neural network like this:

[12]:
```
lr = NeuralNetwork(
    layers=[Dense(neurons=1,
                  activation=Linear())],
    loss=MeanSquaredError(),
    seed=20200720
)

nn = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=Linear())],
    loss=MeanSquaredError(),
    seed=20200720
)

dl = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=13,
```

```
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=Linear())],
    loss=MeanSquaredError(),
    seed=20200720
)
```

1.1.6 5. Trainer and Optimizer

To make this process cleaner and easier to extend to the more complicated deep learning scenarios, it will help us to define another class Trainer that carries out the **training**, as well as an additional class Optimizer that carries out the **learning** or the actual updating of the NeuralNetwork parameters given the gradients computed on the backward pass. Let's quickly define these two classes.

Let's start with the easier one - the Optimizer

1.1.7 Optimizer

By making a separate Optimizer, it allows us to incorporate different ways of updating the parameters based on gradients in the future.

For now, we will make something simple, which is basically a -learning_rate multiplies with the gradient. Since we want to use many different Optimizer in the future, we shall make a parent class as well.

The code is simple like this:

[13]:
```
#parent class
class Optimizer(object):
    def __init__(self, lr: float = 0.01):
        #learning rate
        self.lr = lr

    def step(self):
        #how parameters are updated
        pass
```

[14]:
```
#Stochasitc gradient descent optimizer.
class SGD(Optimizer):
    def __init__(self, lr: float = 0.01):
        super().__init__(lr)

    def step(self):
        #params hold w and b
        #param_grads hold their gradients
        for (param, param_grad) in zip(self.net.params(),
                                       self.net.param_grads()):
            param -= self.lr * param_grad
```

1.1.8 Trainer

Let's create a Trainer. Here this class will be wrapping everything, with NeuralNetwork and Optimizer as attributes in this class.

For training, we gonna keep this simple where the training works like this:

1. Shuffle the data at the beginning of the epoch (epoch is like iterations)
2. Feed the data through the newtork in batches, updating the parameters after each batch has been fed through
3. Epoch ends when we have fed the entire training set through the Trainer

[15]:
```
from copy import deepcopy
from typing import Tuple

class Trainer(object):
    #NeuralNetwork and Optimizer as attributes
    def __init__(self,
                 net: NeuralNetwork,
                 optim: Optimizer):
        #Requires a neural network and an optimizer in order for
        #training to occur.
        self.net = net
        self.optim = optim
        self.best_loss = 1e9  #use for comparing the least amount of loss

        #Assign the neural network as an instance variable to
        #the optimizer when the code runs
        setattr(self.optim, 'net', self.net)


    # helper function for shuffling
    def permute_data(self, X, y):
        perm = np.random.permutation(X.shape[0])
        return X[perm], y[perm]

    # helper function for generating batches
    def generate_batches(self,
                         X: ndarray,
                         y: ndarray,
                         size: int = 32) -> Tuple[ndarray]:
        #X and y should have same number of rows
        assert X.shape[0] == y.shape[0]

        N = X.shape[0]

        for i in range(0, N, size):
            X_batch, y_batch = X[i:i+size], y[i:i+size]
            #return a generator that can be loop
```

```
            yield X_batch, y_batch


    def fit(self, X_train: ndarray, y_train: ndarray,
            X_test: ndarray, y_test: ndarray,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32,
            seed: int = 1,
            restart: bool = True):

        np.random.seed(seed)

        #for resetting
        if restart:
            for layer in self.net.layers:
                layer.first = True

            self.best_loss = 1e9

        #Fits the neural network on the training data for a certain
        #number of epochs.
        for e in range(epochs):

            if (e+1) % eval_every == 0:

                # for early stopping
                # deepcopy is a hardcopy function that make sure it construct a␣
↪new object (copy() is a shallow copy)
                last_model = deepcopy(self.net)

            X_train, y_train = self.permute_data(X_train, y_train)

            batch_generator = self.generate_batches(X_train, y_train,
                                                    batch_size)

            for (X_batch, y_batch) in batch_generator:

                self.net.train_batch(X_batch, y_batch)

                self.optim.step()

            #Every "eval_every" epochs, it evaluated the neural network
            #on the testing data.
            if (e+1) % eval_every == 0:

                test_preds = self.net.forward(X_test)
```

```
                loss = self.net.loss.forward(test_preds, y_test)

                if loss < self.best_loss:
                    print(f"Validation loss after {e+1} epochs is {loss:.3f}")
                    self.best_loss = loss
                #if the validation loss is not lower, it stop and perform early␣
 ↪stopping
                else:
                    print(f"""Loss increased after epoch {e+1}, final loss was␣
 ↪{self.best_loss:.3f}, using the model from epoch {e+1-eval_every}""")
                    self.net = last_model
                    # ensure self.optim is still updating self.net
                    setattr(self.optim, 'net', self.net)
                    break
```

1.1.9 6. Let's run it!

Let's load the boston data and test our code

[16]:
```
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

boston = load_boston()
X = boston.data
y = boston.target
features = boston.feature_names
s = StandardScaler()
X = s.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

#since our train function assumes y to be shape of (n, 1)
y_train, y_test = y_train.reshape(-1, 1), y_test.reshape(-1, 1)
```

[17]:
```
trainer = Trainer(lr, SGD(lr=0.01))

trainer.fit(X_train, y_train, X_test, y_test,
       epochs = 50,
       eval_every = 10,
       seed=20200720)
```

```
Validation loss after 10 epochs is 26.549
Validation loss after 20 epochs is 24.722
Validation loss after 30 epochs is 22.859
```

```
Validation loss after 40 epochs is 22.766
Validation loss after 50 epochs is 22.648
```

[18]:
```
from sklearn.metrics import mean_squared_error

#getting the MSE with testing data
preds = lr.forward(X_test)
mean_squared_error(y_test, preds)
```

[18]:
```
22.647618498359957
```

[19]:
```
#Let's try neural network and deep neural network
trainer = Trainer(nn, SGD(lr=0.01))

trainer.fit(X_train, y_train, X_test, y_test,
       epochs = 50,
       eval_every = 10,
       seed=20200720)

#getting the MSE with testing data
preds = nn.forward(X_test)
print("NN MSE: ", mean_squared_error(y_test, preds))

trainer = Trainer(dl, SGD(lr=0.01))

trainer.fit(X_train, y_train, X_test, y_test,
       epochs = 50,
       eval_every = 10,
       seed=20200720)

#getting the MSE with testing data
preds = dl.forward(X_test)
print("DL MSE: ", mean_squared_error(y_test, preds))
```

```
Validation loss after 10 epochs is 23.608
Validation loss after 20 epochs is 21.098
Validation loss after 30 epochs is 15.430
Validation loss after 40 epochs is 13.955
Validation loss after 50 epochs is 13.393
NN MSE:  13.39263889450387
Validation loss after 10 epochs is 31.474
Validation loss after 20 epochs is 19.227
Validation loss after 30 epochs is 15.901
Validation loss after 40 epochs is 14.381
Validation loss after 50 epochs is 12.854
DL MSE:  12.85354351825392
```

So that's it! My intention is to make sure you are no longer "scared" of the complexity of neural

networks. In fact, this is all about neural network. CNN, RNN and other improvements are simply variants of this codebase (adding more layers for normalization, for preventing overfitting, of course, we can do some dimensionality here as well)

In the next class, let's work out how we can further improve this codebase. Obviously, our deep neural net is only good enough for teaching. If you try to change the seed or change some hyper-parameters, you may be amazed that the deep neural net cannot beat the simple neural net.

See you next class!

[ ]: