DAT494_Lab2B_Fix

October 8, 2024

1 Lab 2, Part B: Simple Neural Nets in PyTorch

- Our goal in this lab is to introduce/review the basic PyTorch data prep process and training loop, and then train several variations on regression and classification models.
- We will create some DataSets and DataLoaders as well.
- We will also implement a few custom Layers, Activation functions, etc.
- Finally, we'll implement a simple autoencoder and practice transfer learning.

[2]:
```
## Basic libraries needed:

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


#Some PyTorch stuff:
import torch
import torch.nn as nn

from torch.utils.data import Dataset, DataLoader
```

1.1 Part 1: DataSets and DataLoaders

- We will use the processed `mpg` dataset provided: There are several continuous and several one-hot encoded categorical features; the outputs are `cty` and `hwy`. As a somewhat contrived exercise, do the following:

Create a class called `FileDataset` that is a child of the parent class `Dataset`. This class should open a csv file of a given name, and return the features and target for the given (integer) index. Your template is:

```
class FileDataset(Dataset):
    def __init__(self, filename, feature_cols, target_cols, mean=None, sd=None):
        ...

    def __getitem__(self, index):
        ...

    def __len__(self):
```

```
    ...
```

Construct the `__init__` method such that the Dataset is created like so:

```
my_dataset = FileDataset(filename, feature_columns, target_columns, mean, sd)
```

That is, pass the name of the file, a list of column names for the features, and a column name for the single target. `mean` should be a list of feature means, and `sd` is a list of feature standard deviations. *You will need to calculate appropriate means and standard devations for the continous features. Use `mean=0` and `sd=1` for one-hot encoded categorical features.*

Given this setup, construct `__getitem(self, index)__` such that a float32 tensor of features and a float32 tensor of target values is returned, as a tuple. Moreover, the features should be normalized according to the `mean` and `sd` arrays.

1.1.1 Create training, validation, and testing `FileDatasets` from `mpg_processed.csv`

1. Create a `FileDataset` from `mpg_processed.csv`, using all columns except `cty` and `hwy` as features, and `hwy` as your target. *You will need to determine appropriate means and standard deviations for the continuous features.*

Confirm it loaded correctly by inspecting the first few entries.

2. Divide your dataset into training, validation, and testing datasets. Use 20% of the dataset for testing. Of the 80% allocated to training, split off 10% of this for validation. Be sure to randomize the train/validation/test split, but use a fixed random number seed so your work is reproducible.

1.2 Make DataLoaders:

- Create a DataLoader for each of the datasets above. Set the batch size for the training loader to 16, and `shuffle=True`.
- Set the batch size for the validation and training dataloaders to the length of the associated dataset; set `shuffle=False`
- Check the first few batches of the training DataLoader to confirm it works

Name your DataLoaders `train_loader`, `valid_loader`, and `test_loader`.

1.3 Create a Fully Connected Regression Model

- Let's build a simple regression model using PyTorch
- Create a class for a regerssion model inherited from the nn.Module class: name it `MyRegressor`; the class should also have two functions as in the following format:

```
class MyRegressor(nn.Module):
    def __init__(...):
        ...

    def forward(...):
        ...
```

- This model should consist of three hidden Linear layers of size 128, 64, and 32, an appropriate output layer.

- Use a ReLU activation function for all hidden Linear layers.

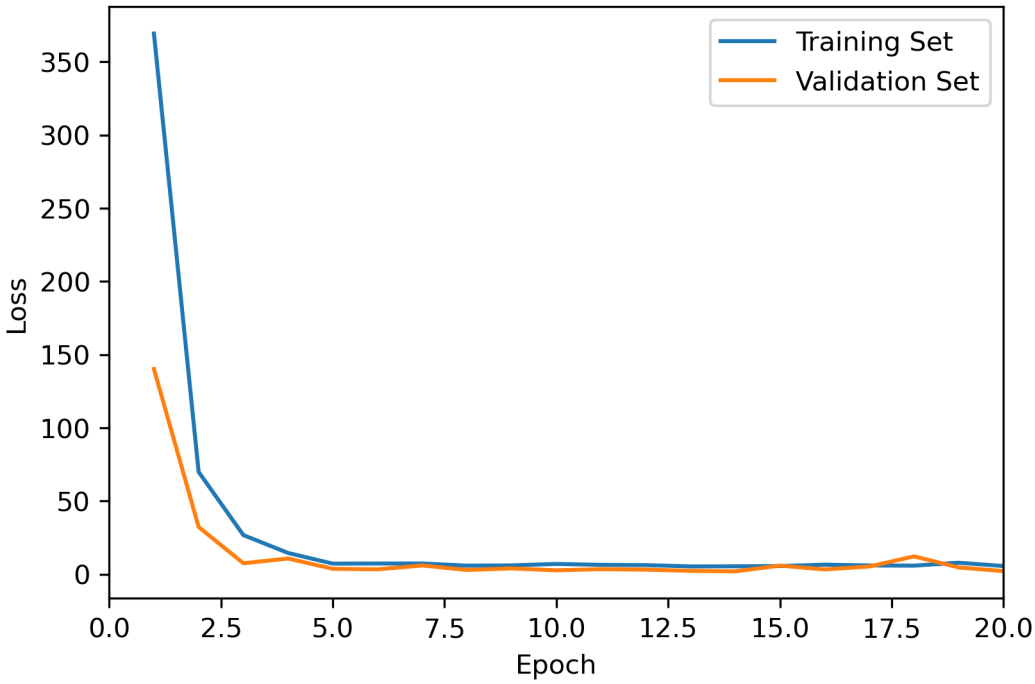
1.4 Construct model, define training loop, and train

Now, you'll need to create a model of class `MyRegressor`, define a loss function and optimizer, and then train the model for a given number of epochs.

- Use Mean Square Error for your loss function
- Use an Adam Optimizer with a learning rate of .01.

Then:

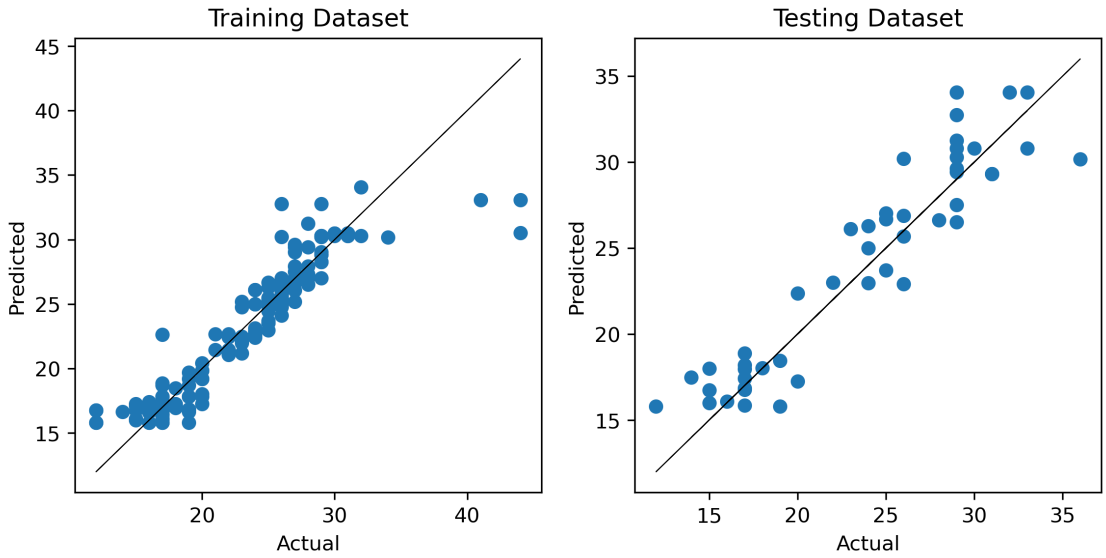
- Train the model for 20 epochs
- Calculate the validation loss and training loss for each epoch: Note that the training loss should be calculated as the average for each batch, while the validation loss is calculated at the end of the epoch
- Plot these losses as a function of epoch number; you should get something like the following:



1.4.1 Plot and score predictions on both training and test sets

- Generate predictions for both the training and testing datasets
- Plot the predicted values against the true values
- Report both R^2 values (these should both be around 85%)

Your results should resemble the following:



1.5 Initialization Schemes

Let's try a few different initialization schemes. By default, PyTorch initializes a Linear layer (`torch.nn.Linear()`) by drawing both weights and biases from:

$$\text{Uniform}\left(-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}\right),$$

where n_{in} is the number of input features/neurons.

Our goal in backpropagating neural nets is to avoid the vanishing and exploding gradients problems. When running a net in the forward direction, we want the variance of the outputs of each layer is similar to the variance of the inputs. When backpropagating the error signal, we want the variance of the gradients to be similar before and after passing through each layer.

Glorot and Bengio (2010) showed that an alternative weight initialization scheme, known as Glorot or Xavier initialization, can better achieve these goals:

$$\text{Uniform}\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right),$$

where n_{in} is the number of input features/neurons, and n_{out} is the number of neurons in the current layer (equal to number of outputs). This is also often re-written as a uniform distribution between $-r$ and r, with

$$r = \sqrt{\frac{3}{fan_{avg}}},$$

and where

$$fan_{avg} = \frac{fan_{in} + fan_{out}}{2}$$

We can also use *normally distributed* weights, with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$.

In PyTorch, the weights for a given layer can be calculated using Glorot/Xavier initialization using:

- `nn.init.xavier_uniform_`
- `nn.init.xavier_normal_`

Named for Xavier Glorot.

Paper: Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256). JMLR Workshop and Conference Proceedings.

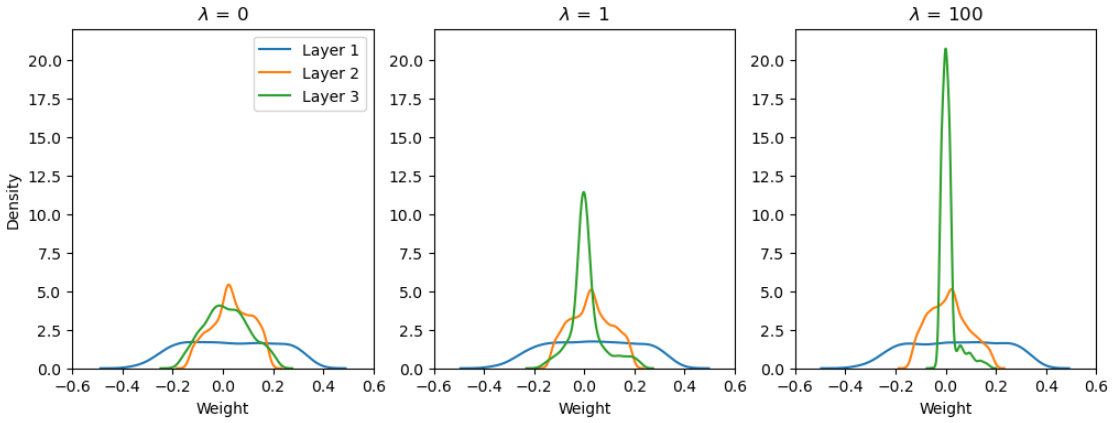
1.5.1 Your Task:

1. Create a `MyRegressor()` object.
2. Plot the distribution of weights (as either a histogram or KDE plot) for the three hidden layers that you get initially.
3. Then, initialize all hidden layers using Glorot/Xavier uniform initialization, and plot the new distributions. Furthermore, initialize the biases for these layers to 0.
4. Train this new model for 20 epochs using Glorot/Xavier uniform initialization for the weights, and biases initialized to 0. You should observe similar performance as before.

1.6 L2 Regularization

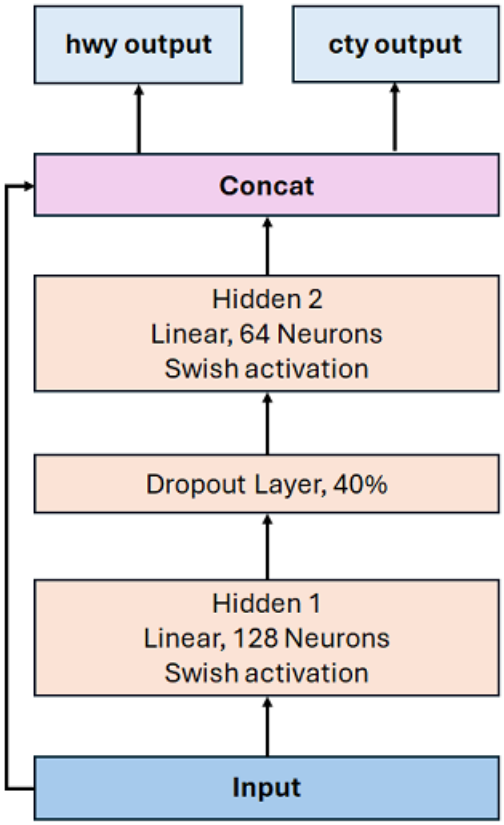
Now, explicitly add L2 Regularization to your model (you will need to modify your loss term):

- Regularize the weights of hidden layer 3 (the shallowest hidden layer), using standard L2 regularization and a λ factor of either 0, 1, or 100.
- Train your regressor, using the default weight initialization, for 20 epochs using both λ values
- Plot the distribution of weights for each layer under either λ value at the end of training. Your results should resemble the following:



1.7 Two Outputs

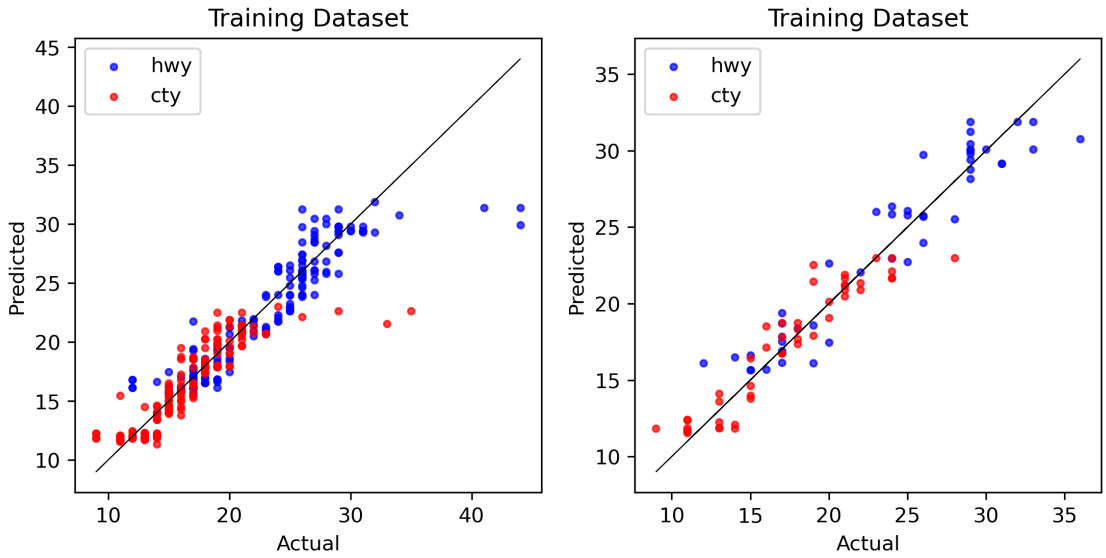
- Now let's extend our work to predict two outputs: `hwy` and `cty`
- Using your `FileDataset` class from before, create appropriate training, testing, and validation datasets and DataLoaders, using the same split sizes.
- Create a model with the following structure:



Using mean square error as your loss function, weight both outputs equally in terms of the overall error. Note that it would be more appropriate to normalize both outputs, but they are similar enough in scale that we can get away with omitting this step.

Note that this time you are to use *Swish* activation functions for all hidden layers.

Train the model for 20 epochs and graphically characterize performance with respect to both outputs, as follows.



1.8 Custom Layers

Create three custom layers:

1. `LinearClone()`
2. `NoisyLinear()`
3. `DropoutClone()`

All layers should inherit from `nn.Module`, and all should implement `__init()__` and `forward` methods.

For the particulars:

1. `LinearClone(in_dim, out_dim, bias=True)`

Basically clone the standard PyTorch Linear layer, but change the initialization scheme: Upon initialization, create a weight tensor that is initialized using **Glorot/Xavier Normal** initialization; create a bias tensor initialized to zero.

2. `NoisyLinear(in_dim, out_dim, bias=True, noise_sigma=1.0)`

Base this layer on your `LinearClone()`, but add normal random noise with mean zero and with a specified (initial) standard deviation, to the layer outputs **only when in training mode**.

Moreover, make the standard deviation of noise a *trainable parameter.*

3. `DropoutClone(p=.4)`

This layer should accept an input of arbitrary size, and set randomly set fraction p to zero (default value of `p` is `0.4`). In training mode, the output of this layer is the randomly zeroed input, scaled by a factor of $1/(1-p)$. In evaluation mode, no inputs are zeroed, and the layer simply acts as the identity function.

Confirm that your DropoutClone() works correctly on some test input tensor

1.9 Custom Activation Functions

Create the following three custom activation functions as classes that inherit from the `nn.Module` class:

- `LeakyReLUClone(negative_slope=0.01)`
- `Swish()` (Sigmoid Linear Unit)
- `ParameterizedSwish(beta=0.6)` with a *trainable* β that starts with default $\beta = 0.6$
- You should implement an `__init()__` and `forward` method for all three classes.

Confirm that all activation functions give the expected behavior by plotting the outputs over a reasonable range of input values.

1.10 Regressor With Custom Layers, Activation Functions

- Now, implement a regressor that takes in our inputs as before, and has the single output `hwy`.
- Call the class `CustomRegressor()`
- Implement the following architecture:

1. (Input)
2. `LinearClone` layer with 128 units, use `LeakyReLUClone(negative_slope=0.1)` as activation function
3. `DropoutClone(p=.2)`
4. `NoisyLinear(noise_sigma=10.0)` layer with 64 units, use `ParameterizedSwish(beta=.6)` as activation function
5. `DropoutClone(p=.2)`
6. Appropriate output layer

Then:

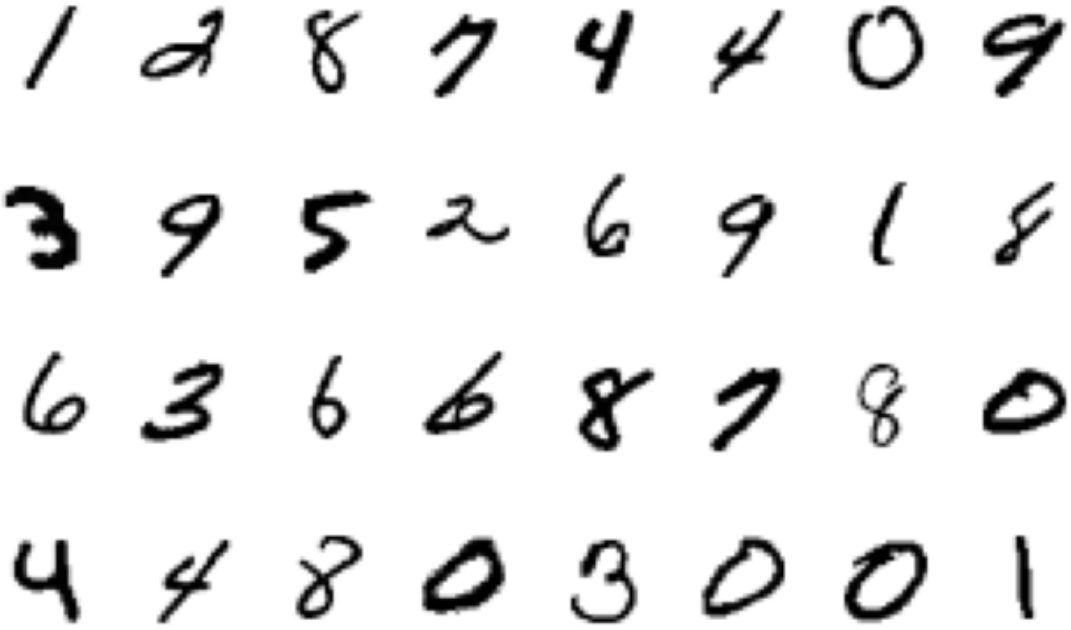
1. Train the model for 20 epochs using an appropriate loss function and Adam optimizer with `lr=.01`.

2. Report R^2 on the training and testing datasets. Moreover, make predictions in both training and evaluation mode: Confirm you get different predictions/R^2 in the two modes.

3. Confirm that both the `noise_sigma` parameter for the noisy linear layer, and the `beta` parameter for the parameterized Swish activation function have changed with after training.

1.11 Classifiers and Autoencoders on MNIST Numbers

- Use `torchvision.datasets` to import the MNIST numbers (as we have done previously)
- Holding the last 5,000 instances in the training dataset aside for validation, create DataLoaders for the training and validation datasets; use a batch size of `32` for both.
- Shuffle the training DataLoader; do NOT shuffle the validation DataLoader
- Plot the numbers in the first validation batch to confirm you have loaded the data correctly (these should be the same as below)



1.12 Basic Classifier

- Implement and train a (multi-) classifier using PyTorch to classify all 10 digits
- Call your class `MNISTClassifier()`, inherit from `nn.Module`, and define appropriate `__init__` and `forward` methods
- Use three hidden layers (use the default weight/bias initialization), all with 64 neurons, all using a *Swish* activation function
- Put a **batch normalization** layer after each of the three hidden layers
- Use an appropriate output layer and appropriate loss function
- Using the optimizer of your choice, train the model for 2 epochs

Report the final accuracy on both the training and validation datasets. ***Be sure you are in evaluation mode when calculating final accuracy.***

Plot the first batch of the validation set, along with the model prediction. Highlight any misclassifications.

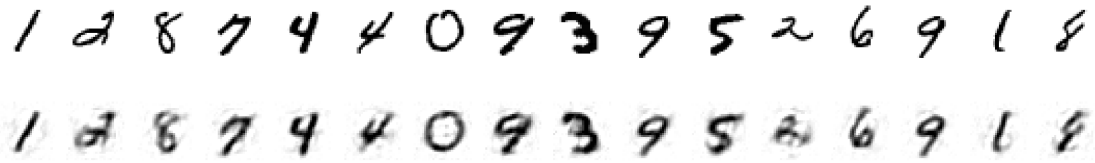
1.13 Stacked Autoencoder

Implemenent a stacked autoencoder, called `MNISTAutoencoder()`, with the following architecture:

1. (Input layer with 784 units)
2. Hidden layer with 128 units
3. Hidden layer with 32 units
4. Hidden layer with 128 units
5. Output layer with 784 units

Use Swish activation functions for the hidden layers.

- Train the autoencoder to reproduce the MNIST input numbers as output.
- Train for 2 epochs, using an appropriate loss function and an Adam optimizer, with learning rate `.01`
- Plot the actual input and your output for the first 16 items of the first validation batch. Your results should resemble the following (note that output is clipped to $[0, 1]$ range in the following):



1.14 Transfer Learning

Use the same basic architecture as above for a classifier. However, replace the output layer with one appropriate for classifying the MNIST numbers.

1. Use the weights and biases trained by the autoencoder for all hidden layers.
2. Freeze the weights/biases of all layers, except the output layer.
3. Using an appropriate loss function and an Adam optimizer with learning rate `0.01`, train the model for 2 epochs.
4. Explicitly confirm that the weights and biases of the hidden layers did not change during training.
5. Report your final accuracy on the validation dataset.