Udacity Nano Degree

AI & ML

OOPs

* An object has characteristics and actions.
  → characteristics are called attributes (e.g = color)
  → actions are called methods (e.g = change color)
* A class is the generic version or blueprint of an object
  - Attributes don't have brackets (e.g = .shape)
  - Methods have brackets (e.g = .type())
  - A Method is inside of class
  - A function is outside of class
* Magic methods overrides the defaults behaviour of an operator.
* Inheritance = A child inherits from parent.

Module in Python

* Numpy is a module in python.
  → A module is a file containing classes, methods or functions etc. which can be accessed from other files/script

Project #1

Classifying Images

* Implement classify_image() function
  → use classifier in it to create labels
  → you have pet image labels already.
  → compare these labels

Submitted

Numpy

* **Attributes of array** = arr.dtype → tells datatype of arr elements
  = arr.shape → tells no. of rows and cols.
  = arr.size → tells no. of elements
  = arr.ndim → tells no. of dimension in array

Ranks

* We call 1d array as Rank 1 array
  " " 2d " " " 2 "

How to save np array

→ np.save("my_arr", arr)

- This will create a file my_arr.npy
  - Now, To load this file, use
  → arr = np.load("my_arr.npy")

Creating np array

- np.zeros()
- np.ones()
- np.full() → create an arr of a specific constant

* **Identity matrix** = $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ ⟹ np.eye(3)

* **Custom diagonal matrix** np.diag()

np.arange()	vs	np.linspace()
* works just like range() * end point is excluded		* used to get equidistant points b/w a range. * end point is inclusive by default but we can make is exclusive by endpoint = False

* np.linspace(0, 25, 10) = It will give 10 equidistant point b/w 0 — 25. If we don't specify 10, then default n will be 50

Reshaping arr

- np.reshape() → let's say you have an array of Rank 1, with 20 elements then it can be reshaped

$$\left.\begin{array}{l} 4 \times 5 = 20 \\ 10 \times 2 = 20 \\ 2 \times 10 = 20 \\ 5 \times 4 = 20 \end{array}\right\} \text{no. element should be same}$$

Array with random elements

* We use random values a lot, lets say you are building a neural network, so to initialize it's weights, you will require a random valued array.
  - np.random.random((3,3)) ⇒ gives 3,3 array of 9 elems in 0 — 1
  - np.random.randint() → random number b/w a range.
  - np.random.normal() ⇒ Normal distribution.

- np.insert()
- np.vstack()
- np.hstack()

(Slicing in nd-array)

- np.copy() ⇒ makes a seperate arr with slicing.
- np.unique() ⇒ give unique value from arr

boolean indexing

Pandas

* It is built on top of numpy, that's why it's fast

series
- Panda series is 1d array like object
- " " can hold different data types
- We can manually set indexes to panda series.

Access columns

* data_series.loc() => labelled index
* data_series.iloc() => numerical index

---

* inplace => Do operation on original series itself
* out of place => Do not operate on " " "

e.g drop() by default is out of place as it does not modify the original dataframe but if we give argument as inplace = True, then it will modify the original data.

Dataframe

* Dataframe is a 2-d object

Deal with NaN values

* Drop rows with nan values
* Drop columns with nan values
* fill nan values with 0
* fill nan values with previous or next values by col or row.
  - -> Forward filling
  - -> Backward filling
* Replace with mean, mode, median
  - -> don't use mean when data has outliers

Check if any columns has NaN values

df.isnull().any()

---

df.groupby()

PANDAS

Matplotlib & Seaborn

★ • **Univariate Visualization** = Visualization of a single variable
- Bar chart ⇒ use for qualitative data
- Histograms ⇒ use for quantitative data

★ **Bar Chart** ⟶ • On x-axis, it contains all categories
- categorical data
- On y-axis, it contain frequency (count) of these variable.
- Always start axis by 0.
- if there are many categories then you should use horizontal Bar chart

sns.countplot()

- **Absolute frequency** = Actual count of categories
- **Relative frequency** = proportion of each category

★ **Pie Chart** ⟶ • Use it when you have less categories, otherwise use bar chart
- categorical data
- it uses relative frequencies

★ **Histograms** ⟶ • On y-axis, it contains frequency (counts)
- numerical
- Use it to check distribution of data
  - → ie = check in which range, the data has most

sns.distplot()
plt.hist()

★ • **Bivariate Visualization** = Visualization of two variable

Quantitative vs Quantitative = **Scater plot**
" vs Qualitative = **Violin plot**
Qualitative " Qualitative = **Clustered bar charts**

★ **Scatter plot** = • if your data has too many points, then use transparency overlapping.
- if your data for one or both variable is discrete then use jitter.

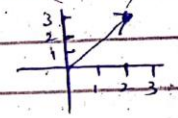
★ **Heatmaps** → • Use it for discrete vs discrete data
• It is also called 2-d histogram

Linear Algebra

Vectors

★ **Vector** = ordered list of numbers
• Vector in linear Algebra always origin at (0,0)

e.g vector = $\begin{bmatrix} 2 \\ 3 \end{bmatrix}$, can be written as (2,3), can be drawn as



★ Addition of vectors = $\begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 5 \\ 4 \end{bmatrix} = \begin{bmatrix} 7 \\ 7 \end{bmatrix}$
★ Multiplication of a vector by constant = $4 \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 8 \\ 12 \end{bmatrix}$
reversing the direction of a vector by multiplying it by —, called scaling

Unit Vectors

★ $\hat{i}$, called "eye hat", is unit vector in horizontal direction (x-axis)
e.g. in $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$, x value of this vectors consists of $3\hat{i}$
★ $\hat{j}$, called "jay hat", is unit vector in vertical direction (y-axis).
e.g. in $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$, y value of this vector consists of $2\hat{j}$

Note, i, j are also called basis vectors

Span of vectors

★ Suppose you have two vectors v and w. you can multiply these by any scalar and by scaling them you can get their all linear combinations $(a\vec{v} + b\vec{w})$ a, b = scalar.
★ and all of these linear combinations (possible) is called span of these vectors
★ **Linearly dependent** = when one vector can be removed without affecting span.
★ **Linearly independent** = when both of vectors effect on span

Basis of vector space

* It is a set of all linearly independent vectors that span the full space

Linear Transformation

- Transformation is also called as function.
- In linear Algebra, transformation takes a vector as input and give a vector as output
- A Linear Transformation is one, in which after transforming the origin and straightness of lines doesn't change

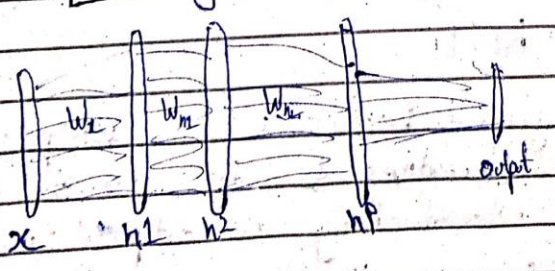
Neural Networks

* The inputs are each connected to neurons in hidden layer
* Then the neurons in hidden layers are each connected to neurons in output layer
* When each neuron layer represent a single output
* In short the inputs are multiplied with some weights to give output

* **Dimension of Weights** = no. of Rows = no. of inputs
  - no. of cols = no. of outputs

Input x Weights = output

Working of Neural Network



input(x) will be multiplied with some weights and produce outputs as first hidden layer (h1), the h1 will be multiplied by some weights and produce output as second hidden layer (h2) and so on untill it gives final output.

Calculus

* Area of circle = πr^2, but why?
* Many hard problems can be solved by doing sum of of many many small values and it can be solved by finding area under the curve.

* derivative and Integral are inverse to each other
* Integral is area under the curve

Derivative

* It is instantenous rate of change OR rate of change at single point

Neural Networks

* Cost error should decrease as epochs increases.

- **Gradient Descent** = An Algorithm to reduce cost function.
- **Back propogation** = A way to update weights after calculating cost errors
- **Use of RELU or Sigmoid** : These are non-linear functions which gets wrapped around the output ((input)(weight)+bias)

e.g. $\sigma(x.w + b)$ → RELU

* **Conclusion**: In gradient Descent, we use Backpropagation and in this we use derivatives to adjust weights by checking out the rate of change in output caused by rate of change in weights.

AI in Python

* Neural Network at its core do classification
* The result of this classification is continuous (not discrete)
  → we move from discrete to continuous by changing Activation function from "Step" to "Sigmoid"
  → Continuous prediction tells likelihood/confidence or probability of prediction
    e.g. • Discrete prediction = It's a dog
    • Continuous prediction = It's 80% dog and 20% cat

• Activation function used for multi classification is "Softmax".
• Maximum Likelihood is a process of maximizing probabilities
  → The more/larger the product of all probabilities the better the model is.

---

* Good Model has low cross entropy
  → cross entropy is sum of negative logarithms of probabilities
* Higher cross entropy = lower possibility of event

---

* Good Model has low S.S.E (sum of squared error)
* Under fitting = Training and testing errors are both large
* Overfitting = Training error is very small but testing error is high
* Good fitting = Training and testing errors are both small
• We handle overfitting by using dropout layers
* To reach the global minimum from local minimum, we use momentum in gradient descent

Pytorch for DL → part of udacity degree (10)

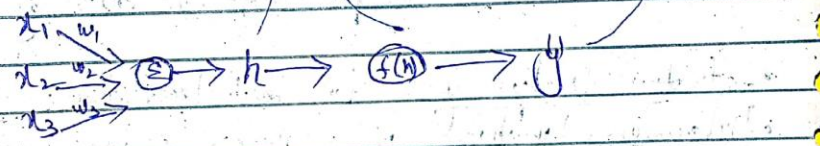
Neural Network in Pytorch

* The data type we use to build NN in torch is tensor (which can be vector of matrices)

→ Output of NN = σ(features(x) * weights(w) + bias)
activation function.

A Simple N.N.

* We have some input values, we multiply them with some weights, and then we pass the resultant value throught an activation function to get final output.



* In the 1st step, we can see that we have multiple inputs and their respected weights. So to make things simpler, we can represent all our inputs as a vector and all weights as another vector and then easily find their summed product as Dot product.

AND

to represent vectors and matrices in pytorch, we use tensors.

- A tensor with 1d is called a vector.
- " " " 2 or more d " matrices.

SO,

In Pytorch language we can write output of NN as.

y = f([Input] x [Weight] + bias)

→ any activation function.

OR,

y = activation(torch.sum(Inputs x weights) + bias)

Matrix Multiplication

* MM is process of multiplying two tensors and then adding up all values.

Condition ⇒ No. of columns of 1st tensor must ~~have~~ equal to no. of rows of 2nd tensor.

AND

In $y = \text{activation}(\underbrace{\text{torch.sum}(\text{Inputs} \times \text{weights})}_{\text{this is exactly MM.}} + \text{bias})$

So,

In pytorch, we can use: $\boxed{\text{torch.mm()}}$ for this. as

$y = \text{activation}(\text{torch.mm}(\text{Inputs}, \text{weights}) + \text{bias})$

Multilayer Neural Network

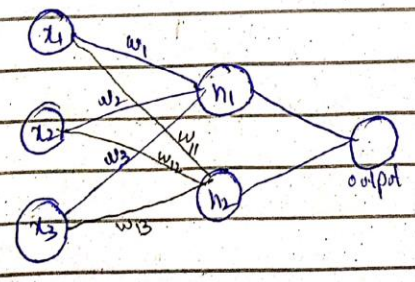
* In MNN, you have hidden layers.
  ↳ we create them to give ability to our model to understand correlations and patterns.
  - At first, you have a vector of inputs (x_1, x_2, x_3)
  - Now in MNN, we will create more than 2 vectors of weights. e.g = (w_1, w_2, w_3), (w_{11}, w_{12}, w_{13})

    and by 2 sets of weights we can create 2 hidden layers

$$h_1 = (x_1, x_2, x_3) \times (w_1, w_2, w_3)$$
$$h_2 = (x_1, x_2, x_3) \times (w_{11}, w_{12}, w_{13})$$

⇒ Recall: In Single/simple neural N, we only have 1 set of weight.

Diagram



∴ we can think of two vectors of weights as 1 matrices with 2 coloms, each colum is representing 1 set of weights

$$\begin{bmatrix} w_1 & w_{11} \\ w_2 & w_{12} \\ w_3 & w_{13} \end{bmatrix}$$

Output of this MNN

h = activation(torch.mm(features, W1) + B1)
y = activation(torch.mm(h , W2) + B2)

Using nn in pytorch to build neural networks

* We have to use a proper syntax :

```
class My_network(nn.Module):
    def __init__(self):
        super().__init__()
            # Add Layers.
    def forward(self, x):
        # Mention order of execution.
        return x
```

by this pytorch will know to register all layers.

Note: We mostly use a different activation function for hidden layers and a different for output
e.g = Sigmoid for hidden
Softmax for output

* We can also use nn.functional to make network more easy to understand.

Training a N.N

* By training, we mean that we wanna adjust the weights of our network to be set to give correct predictions.
  → Updation of weights will be dependent on a a loss/cost function (e.g MSE) and it basically error($y - \hat{y}$).
  → The Algorithm to find the configuration of weights which minimize this error is called Gradient Descent or Back Propagation (in MNN)

★ **Back-propagation**: In this we have 2 things.
① Forward pass , ② Backward pass

→ At first, we do a forward pass (randomly intialize weights and do a prediction.

→ then we do a backward pass in which we find gradient w.r.t weights and subtract this value from the previously used weights.

Note: We scale this gradient value by multiplying it with an learning rate (alpha) which is already set by us.

→ Now after updating weights, we again do forward pass and the again backward and so on. until our loss is minimized.

An overview of an epoch in training

① Optimizer.zero_grad()

→ Optimizer is used to update weights and we are setting it to zero, so that it clears out weights from previous epoch.

② output = model.forward(data)

→ It is a forward pass (to make a prediction).

③ loss = criterion(output, labels)

→ Here, we are calculating loss as respect to prediction

④ loss.backward()

→ Here, we are calculating gradients (the value which we have to subtract from previously used weights

⑤ optimizer.step()

→ finally, we are updating our weights by subtracting the value calculated in 4th step from weights.

Note: We do these steps in a loop where limit is the no. of epochs.

Testing the Model

* Use "with torch.no_grad():" while testing or making prediction bcz it will make things faster.
* To avoid overfitting you can use Dropout layers
* If you use dropout in model training then use,
  → model.eval() while testing to ignore dropout in doing prediction
  → model.train() to activate dropouts

Saving a Pytorch model

1 → Create a dictionary which include/contain parameters
- input_size in model
- output_size in model
- hidden layers information
- model.state_dict() ⇒ it has trained weights and bias

2 → Now, save the model
- torch.save(param_dictionary, "my_model.pth")

3 → Now, To again load the saved model
- At first, rebuild the model with same architecture using the param_dictionary by loading it as,
  param_dict = torch.load("my_model.pth")
- Now load the saved weights and bias in model
  - model.load_state_dict(param_dict["state_dict"])

Loading Image Data in Pytorch

* Use torchvision.datasets.ImageFolder("my_data")
  1 → with some transformations to make dataset
* To load this dataset use,
  torch.utils.data.DataLoader()

Note: To get better validation accuracy use data augmentation (Rotate, Resize, Flip etc) on training set but use only resize on test data.

Transfer learning

* In this technique, we use pretrained models.
* Basically we use them as feature detector on our custom data.