Neural Networks & PyTorch

ICDSS - William Profit (wtp18)

Overview

Mathematics of NNs

PyTorch framework

Data pipelines

Implementation

Why?

Linear/Logistic regression can only model simple data distributions

Model more complicated functions

Desire to **loosely** mimic the brain

Problem

Given a set of input-output pairs we want to find a function mapping input to output

E.g given: $S = \{(x_1, y_1), (x_2, y_2)..(x_n, y_n)\}$

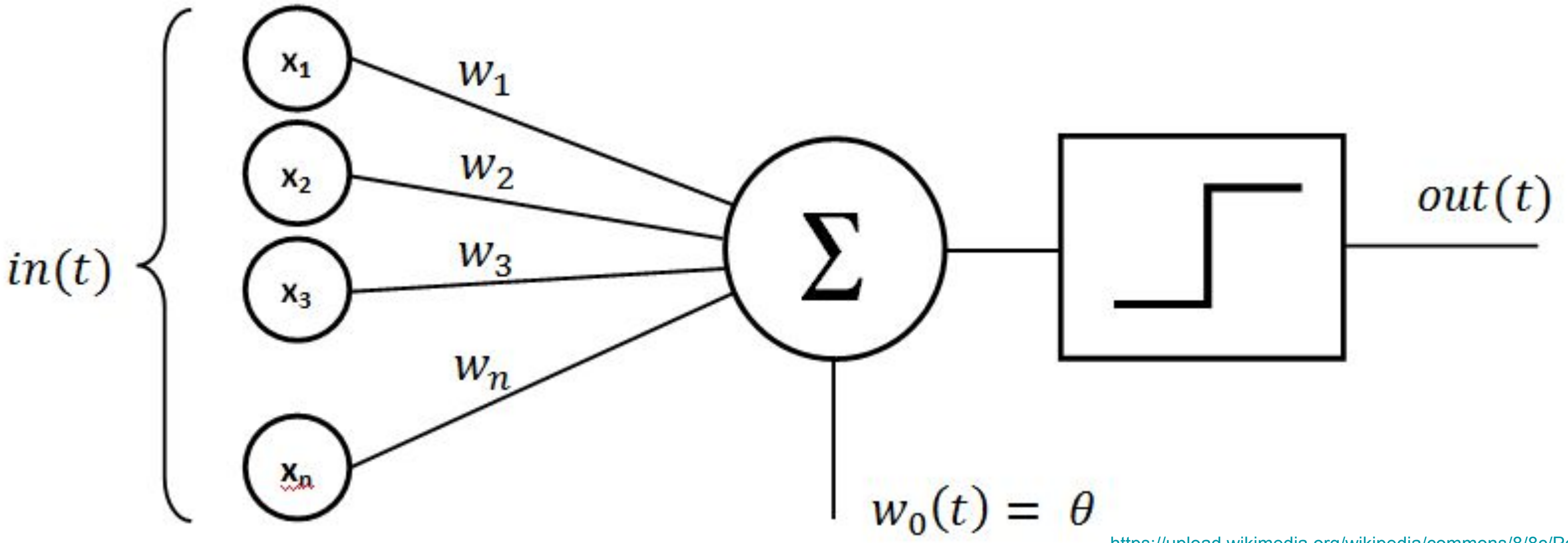
Find: $f : \mathcal{X} \rightarrow \mathcal{Y}$

Example: map image (pixels) of scan to whether there is a tumor or not (binary classification)

Perceptron

Example of a single neuron

Equivalent to logistic regression



https://upload.wikimedia.org/wikipedia/commons/8/8c/Perceptron_moj.png

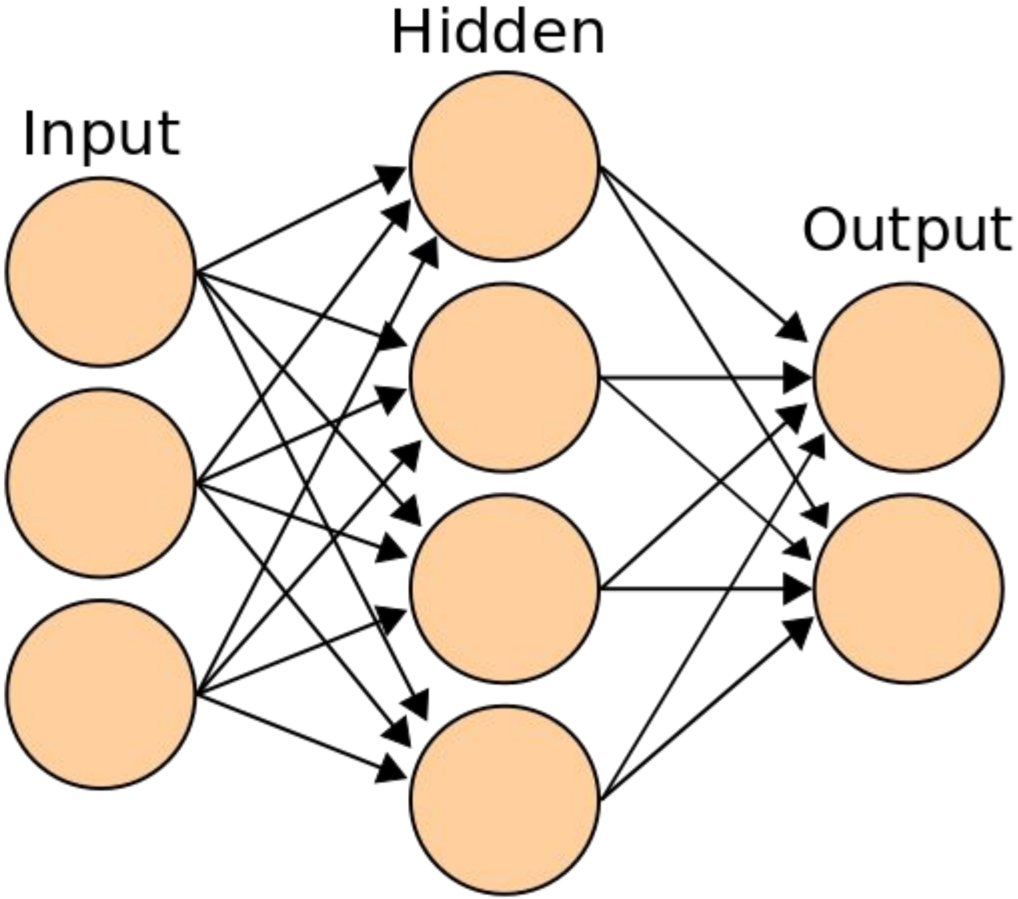
Neural Network

Stack perceptrons into layers

Stack layers into network

Weights now represented by matrix

All parameters (weights) represented by **θ**



https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Artificial_neural_network.svg

Feed Forward

Every neuron computes its activation and forwards it to the next layer

Runs until the last layer, which gives us output

Predicted output is $\hat{y} = Net(\theta, x)$

=> Now need to train NN to predict correct value

Loss

Loss function **L(θ)** measures how ŷ differs from y

Mean Squared Error for **regression**

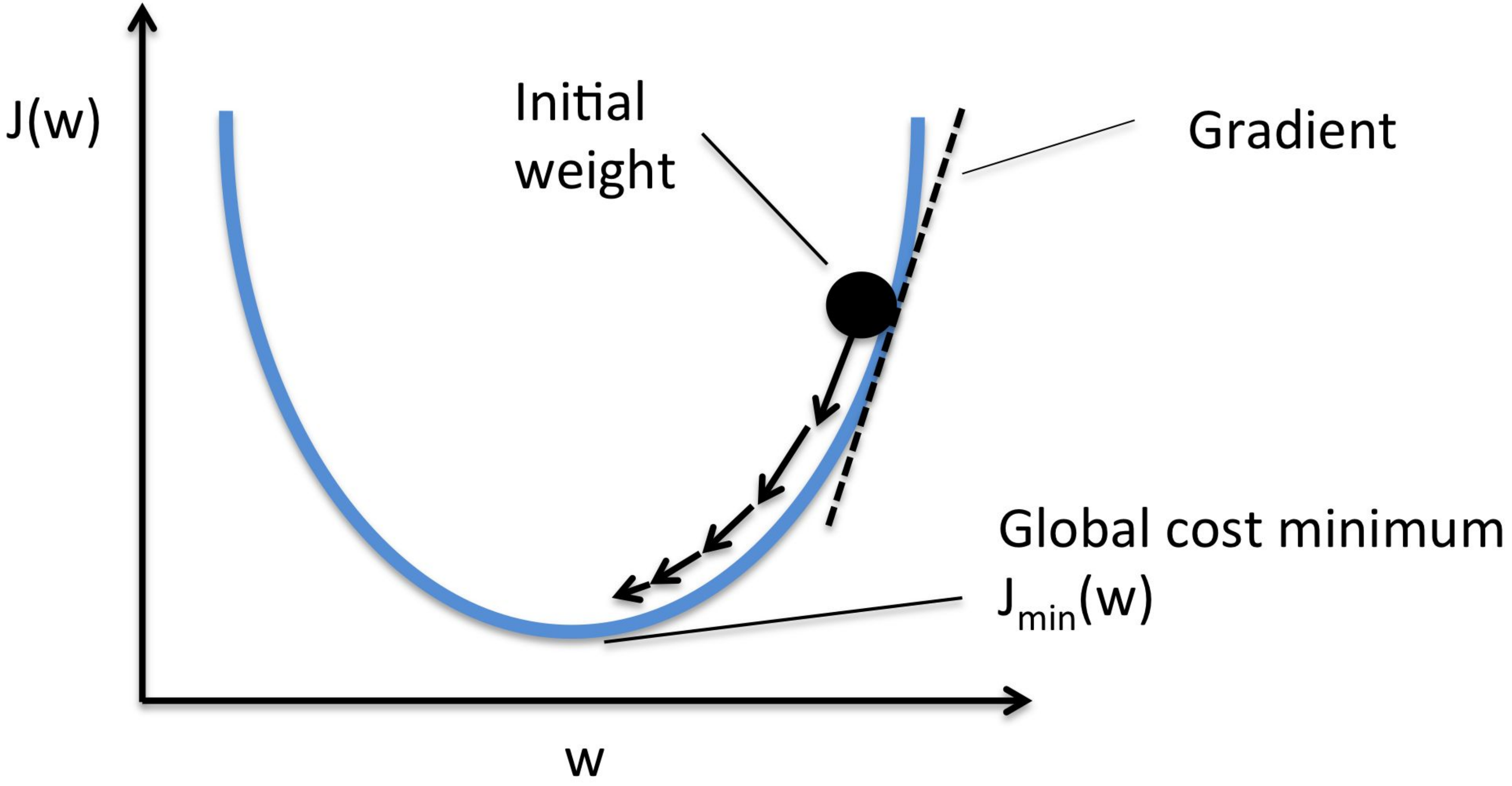
$$\mathtt{MSE} \triangleq \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Cross Entropy for **binary classification**

$$\mathtt{CE} \triangleq -\frac{1}{n}\sum_{i=1}^{n} y_i log(\hat{y}_i) + (1 - y_i)log(1 - \hat{y}_i)$$

Categorical Cross Entropy for **classification**

$$\mathtt{CCE} \triangleq -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} log(\hat{y}_{ij})$$



http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/

Backpropagation

We have now computed **L(θ)** using the predicted **ŷ** and the ground truth **y**

Now for every parameter, compute its gradient $\frac{\partial L(\theta)}{\partial \theta}$

Computed using chain rule from last layer to first layer (i.e. propagating gradients back -> backpropagation)

=> For every parameter, we measure how it influences the loss.

Goal is now to minimise that loss using **Gradient Descent**

Gradient Descent

For each parameter of **θ:**

- Compute its gradient w.r.t. the loss (backprop)
- Update its value using learning rate alpha
- Repeat

$$
repeat\{
\\
\theta := \theta - \alpha \frac{\partial L(\theta)}{\partial \theta}
\\
\}
$$

Full training process

Initialise NN randomly

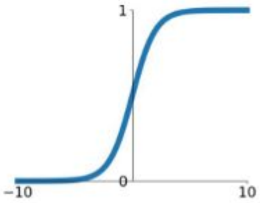
Repeat :

- Feed forward input
- Compute loss comparing output and ground truth
- Propagate gradients from loss back for each parameter in θ (backprop)
- Update parameters with gradients and learning rate (gradient descent step)

Activation Functions

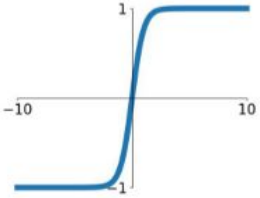
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



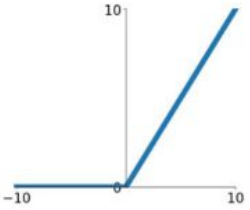
tanh

$\tanh(x)$



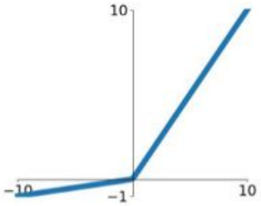
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

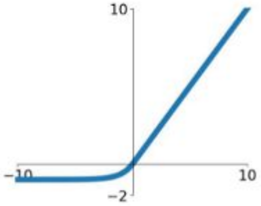


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

Softmax

Used to get probability distribution on output

All outputs sum to 1

2.0 → → 0.7

1.0 → $$Softmax(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}, y \in \mathbb{R}^n$$ → 0.2

0.1 → → 0.1

PyTorch

Tensors

Autograd

Neural Networks

Optimizers

Data Pipelines

Tensors

```python
import torch
import numpy as np

x = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
y = torch.from_numpy(np.array([1, 2, 3, 4]))

x.to('cuda')

print(x.shape)
```

Autograd

Used for automatic computation of gradients

```
x = torch.ones(2, 2, requires_grad=True)
y = x * x * 3
out = x.mean()

out.backward()
print(x.grad) # d(out)/dx
```

Neural Networks

```python
import torch.nn as nn
import torch.nn.functional as F

class NeuralNet(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(784, 300)
    self.fc2 = nn.Linear(300, 10)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = F.softmax(self.fc2(x), dim=1)
```

Optimising

```
import torch.optim as optim

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001)

optimizer.zero_grad()
output = net(input)
loss = loss_fn(output, ground_truth)
loss.backward()
optimizer.step()
```

Datasets

Inherit ***`Dataset`***

Override **`__len__(self)`**

and **`__getitem__(self, index)`** methods

```python
class MyDataset(Dataset):
  def __init__(self, file, transform=None):
    # Load data and do any preliminary operations
    self.data = loadfile(file)
    self.transform = transform

  def __len__(self):
    return len(self.data)

  def __getitem__(self, index):
    item = data[index]
    item = self.transform(item)
    return item
```

Data Transforms

We can define transforms to be applied to our data before processing it

-> Used for data preprocessing pipeline

Examples: normalising, scaling, formatting, augmenting

Create class and define **`__call__(self, sample)`** method

```python
class MyTransform():
  def __init__(self, params):
    some_initialisation_stuff(params)

  def __call__(self, sample):
    return transform_sample(sample)

composed = transforms.Compose([Tranform1, Transform2])
```

Data Loader

We now need to iterate through the dataset

Also want to shuffle the data (less biases)

Use batches for faster training

=> Use a data loader

```python
dataset = MyDataset(
  'path/to/data.csv',
  transform.Compose([
    Transform1,
    Transform2
  ])
)
dataloader = DataLoader(
  dataset,
  batch_size=32,
  shuffle=True,
  num_workers=4
)
```

Iterate through data using a DataLoader

`data` is a tuple of the form `[inputs, labels]`

```
for i, data in enumerate(dataloader):
    inputs, labels = data
```

Demo

MNIST Dataset - recognising handwritten digits

Further reading

How backprop is computed (chain rule, computation graphs)

Regularisation, dropout

Different optimisers (adam, adadelta..)

Overfitting, underfitting

Gradient explosion/vanishing