Advanced Topics on ML - 17/09/2022

Neural Networks + Deep Learning

Logistic Regression

- Linear regression:
  - Can be used in ML for prediction tasks
  - $y = \alpha + \beta x$, Least Squares for finding β
- Multiple Lin reg.
  - $y = \alpha + \sum_i \beta_i x_i$

For discrete output var., it doesn't make sense to use LinReg

Try to approximate $p(y|x)$ instead.

Logistic function

$$p(y|x) = \frac{e^{\alpha+\beta x}}{1 + e^{\alpha+\beta x}}$$

S-shaped, bounded $p \in [0, 1]$

How do we fit the function to our data?

Maximum Likelihood Estimation (MLE)

(see handwritten notes #1)

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \mathcal{L}(\theta|y)$$

where θ are the parameters we want to estimate

- We can choose any underlying distribution
- Our θ can be their respective moments
  - E.g For a Gaussian, $\theta = (\mu, \sigma)$

We usually use negative log-likelihood for convenience since most prob. dist. are from the exponential family

NOTE: Refresh MLE, exponential family by reading Casella's Statistical Inference

Training LogReg

(see handwritten notes #2)

The results are - MLE for α, β - Estimates of $p(y|x)$: this allows us to perform classification once we define a decision boundary

How to define decision boundary? $p(y|x) = 0.5$, assuming the negative/positive classes are equivalent.

If they are not, then we can weight them by picking a boundary other than 0.5

Multiple LogReg

For many independent varaibles:

$$\ln \frac{p}{1-p} = \alpha + \sum_{i}^{N} \beta_i x_i$$

Mathematical details are in the slides

Regularization

Penalize large coefficients to avoid overfitting

i.e we also try to minimize one of these:

- L1 Reg:
  - $|\alpha| + \sum_i^N |\beta_i|$
- L2 Reg:
  - $\alpha^2 + \sum_i^N \beta_i^2$
- Elastic-net Reg:
  - $c_1 L1 + c_2 L2$

Notice that the larger (α, β_i) are, the larger these cost functions are

Notes on decision boundary

For Logistic Reg., the decision boundary is always linear

Neural Networks

Mimic the brain: Nodes $\rightarrow$ Neurons, Links $\rightarrow$ Synapses

Perceptron

Weights w and bias b

$$\sum w_i x_i = y_i = \begin{cases} +1, y_i > 0 \\ -1, \text{ else} \end{cases} = \text{sgn}(\sum w_i x_i)$$

This is just a linear model, restricted to linear problems

$$y = \text{sgn}(w_T x)$$

Error function:

- 0-1 Loss:

Only update weights when we fail:

$$w_{t+1} = w_t + \Delta w_t, \Delta w_t = \eta(y_n - y(x_n))x_n$$

η is called **learning rate**: how much we change weights at each step $\rightarrow$ can be tuned to speed training

Note: η needs not to be constant!

A good strategy is to start with larger η and then decrease it as we approach the local minimum to avoid overshooting and oscillating around it.

Notice that this is a **hyperparameter**

Sign/Step function

Bad: - Not differentiable - Small change in input leads to large change in output

Better activation functions

- Sigmoid

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

- Tanh

These are good because they are smooth, differentiable.

These have limitations when it comes to Gradient Descent since they have vanishing gradient at the boundary of their domains $\rightarrow$ problematic when training with GD

- ReLU

$$\max(0, x)$$

Tries to fix vanishing gradient problem.

- etc. (See slides)

We can look at all these activation function options as **hyperparameters** of the network.

Think about the output layer in our network:

- For binary classification, a sigmoid makes sense
- For regression, we need a linear function!
- Regression with bounded output (e.g pixel intensity): use a sigmoid and then map to desired range

Multi-layer Perceptron (MLP)

Network architecture: layers + connections

The accepted idea is that **deeper is better than wider**: increase number of layers, not their size

Note: these models usually **work better with normalized inputs**! This avoids problems with numeric representation in computers

Usually we want the network to **become wider as we get deeper**. The starting layers encode **low level features**; the intermediate layers combine those to form **high-level features**, which are more specific and detailed. Thus, it is useful to have more of them.

This model is basically an **universal function approximator**.

All the operations between layers are just matrix multiplications. GPUs are special hardware designed for fast/large matrix multiplication

Network structure

- Feed-forward network
  - No internal state: there is no memory of past predictions used for the current prediction
  - Directed *acyclic* graph: simply computes outputs from inputs
- Recurrent network
  - Directed *cyclic* graph: can memorize information since it is a dynamical system with internal states

Training Neural Networks

General approach is to move along parameters space in the direction of negative gradient

$$w_n = w_{n-1} - \gamma \nabla f(w_n), \text{stop if max it or } \nabla f = 0$$

For multi-variate functions the gradient is generalized as the Jacobian matrix

Lets train our NN with GD

$x^i, y^i \rightarrow$ n training samples$f(\mathbf{x}) \rightarrow$ feed-forward NN$L(\mathbf{x}, y; \theta) \rightarrow$ loss function

Apply GD for loss function: calculate error and update weights

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial L}{\partial w_{ji}}$$

Each complete pass on our training set is called **epoch**

Backpropagation is the algorithm allows us to efficiently perform GD

There are different tactics for updating weights:

- **Online learning:** every input, update
- **Batch learning:** accumulate error of all input samples and perform a single update
- **Mini-batch learning:** mix both strategies

Online learning has our curve ever changing, and this can lead to convergence problems

Batch learning is essentially impossible to implement due to resources limitation: we would need to load all our data at once in the GPU, but we usually don't have enough memory for that!

Mini-batches are the way to go.

e.g 100 inputs, 10 mini-batches:

- process 10 inputs at a time, an epoch completes after processing every mini-batch

How to choose our loss function?

Depends on task at hand.

- Regression:
  - MSE (L2)
  - Mean absolute error (L1)
- Binary classification:
  - Binary Cross-Entropy (BCE)
  - Hinge Loss
- Multi-Class classification:
  - Multi-class Cross-Entropy (CE)

Output layer: - Regression: linear - Binary class.: sigmoid - Multi-class class.: softmax

Softmax

Collapses C-dimensional vector O into C-dim. vector $\sigma(O) = [\cdots c_k \cdots]$ such that:

$$\begin{cases} c_k \in [0,1] \\ \sum_k c_k \end{cases}$$

What we do here is **probability estimation**

Backpropagation

1. Forward phase

Compute $z_j = h(a_j)$, $a_j = \sum_i w_{ji} z_i$

2. Backward phase

Note: see handwritten note #3

$$w_{ji} \to \frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$$

Vanishing gradient: if gradient is too small, consecutive multiplication during backpropagation makes training impossible

Vanishing / exploding gradients

- Exploding
  - Model does not learn, cost oscillates, weights diverge
- Vanishing
  - Learning is slow / stops early, weights closer to the input don't change

Solutions: see slides

Overfitting

Solutions: - Early stopping - Regularization: - L1 / L2 penalty - Dropout: randomly "drop" some units from the network when training a given step

i.e add noise to network

- Data augmentation: create synthetic data by applying transformations to dataset

e.g rotating/stretching/displacing images

i.e add noise (variability) to data