Deep Learning for COMP6714 – Part I

Wei Wang @ CSE, UNSW

October 8, 2018

Outline

- ML basics
- Feed forward Network

Problem Definition

The standard *supervised* classification/regression setting:

- Input:
  - Labelled data: $\{\mathbf{x}_{(i)}, y_{(i)}\}_{i \in [n]}$
  - Can be deemed as $[\mathbf{X}, \mathbf{y}]$, i.e., *Data Matrix* consisting of training samples, and the corresponding *Class Labels*.
  - Domain of $y_{(i)}$
    - Binary classification: $y_{(i)} \in \{-1, 1\}$, or $\{0, 1\}$.
    - $|C|$-class classification: $y_{(i)} \in \{0, 1, \ldots, |C| - 1\}$.
    - Regression: $y_{(i)} \in \mathbb{R}$.
- Output: a function/mapping (typically within a *function class*) from $\operatorname{dom} \mathbf{x} \to \operatorname{dom} y$ such that some loss function is minimized.
- Assumption:
  - Training and test data are drawn i.i.d. from the same (unknown) distribution (defined over $\operatorname{dom} \mathbf{X} \times \operatorname{dom} \mathbf{y}$).

Key Concepts

Ultimate goal:

- Generalization error: Errors (of the model) on unseen data

How to approximate it?

- Labelled datasets are divided into two/three subsets.
  - Training data:
  - (Optional) Development/validation data:
  - Test data:
- Use the errors on the test data

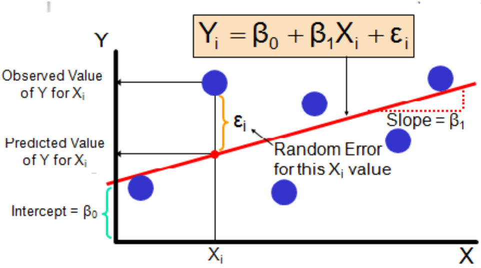
How to train a model?

- Minimize the loss function on the training data
- (Optionally) also considering some regularization measures.
  - To prevent overfitting

Loss Functions

Used to

- Characterize how bad a prediction is, compared with the ground truth.
- An important tuning knob: tradeoff of prediction accuracies among training examples.



Commonly Used Loss Functions

Loss functions $L(\{\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_n\}, \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n\})$:
Typically, $L = \sum_{i=1}^{n} \ell(\hat{\mathbf{y}}_i, \mathbf{t}_i)$

- Classification:
  - Cross entropy-loss: $\ell(\hat{\mathbf{p}}, \mathbf{t}) = \sum_{j=1}^{|C|} t_j \log(\hat{p}_j)$
  - For hard classification problems, it is just $-\log(\hat{p_{j^*}})$, where j^* is the correct class.
  - Exercise: write out the loss function for (hard) binary classification problems.
- Regression:
  - MSE (Mean Squared Error): $\ell(\hat{\mathbf{y}}, \mathbf{t}) = \frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{t}\|^2$

(Traditional) Machine Learning vs. Deep Learning

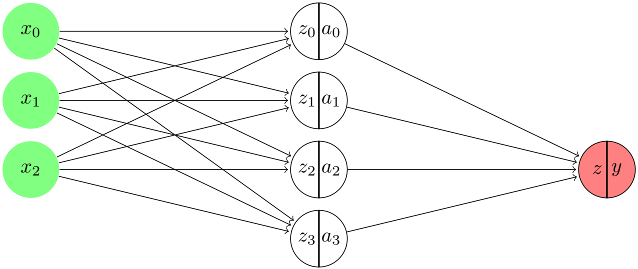
- ML: Features are defined/engineered.
- DL: Features are learned in an *end-to-end* fashion.

Examples

OCR

- ML: define *invariant* features. E.g., number of circles, number of (almost) horizontal strokes, . . .
  - Even with such features, usually a powerful (non-linear) model need to be used (e.g., SVM with non-linear kernels).
- DL: features are learned in a hierarchical fashion automatically by the model.
  - The final classifier is in fact a simple softmax classifier (i.e., a linear classifier).

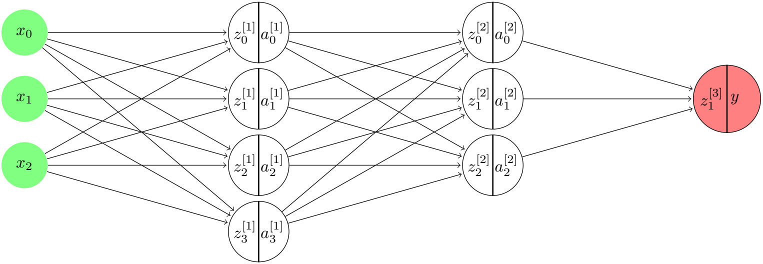
Feed Forward Network / Multilayer Perceptron (MLP)



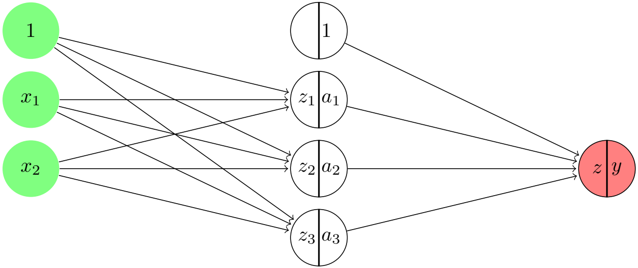
Concepts:

- Neurons
- Input / hidden / output layers
- Activitation function

NN with Multiple Hidden Layers



NN with One Hidden Layer and Biases



- $\mathbf{a}_n = \sigma_n(\underbrace{\mathbf{W}_n\mathbf{a}_{n-1} + \mathbf{b}_n}_{\mathbf{z}_n})$
- $\mathbf{y} = \mathbf{a}_n$ and $\mathbf{x} = \mathbf{a}_1$
- σ_ns are typically non-linear functions, applied element-wise to the input vector.

Non-linearalities /1

- sigmoid (aka. logistic): $\sigma(z) = \frac{1}{1+\exp(-z)}$
  - Special case of $\mathrm{Softmax}([z, 0])$, where $\mathrm{Softmax}([z_1, z_2, \ldots, z_m]) = [\frac{\exp(z_1)}{Z}, \frac{\exp(z_2)}{Z}, \ldots, \frac{\exp(z_m)}{Z}]$
  - Intuition:
    - Squashing $\mathbb{R}$ to $[0, 1]$, and differentiable every where.
    - A smooth approximation of the step function.
  - $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Logit and Logistic Functions

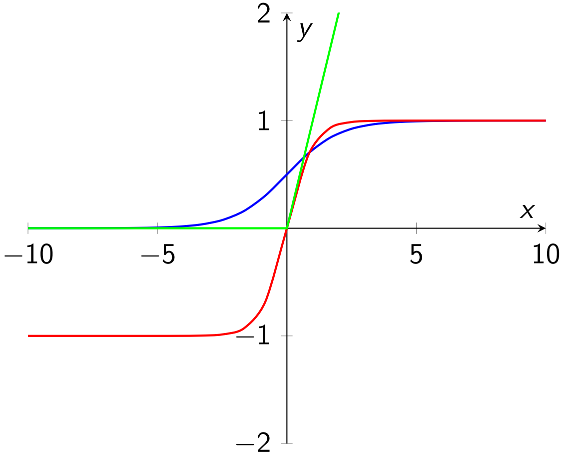
Recall that $\mathrm{logit}(p) = \log \frac{p}{1-p}$. It follows that

$$\mathrm{logit}(p) = z \qquad \Longleftrightarrow \qquad \mathrm{logistic}(z) = p$$

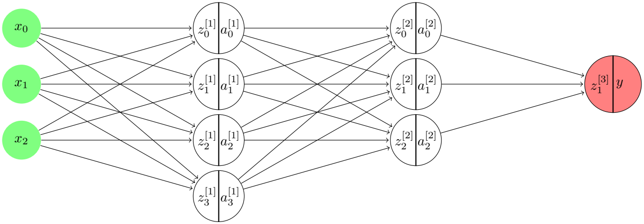
Non-linearalities /2

- tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
  - It is a rescaled sigmoid: $\tanh(z) = 2\sigma(2z) - 1$
  - Squashing $\mathbb{R}$ to $[-1, 1]$, and differentiable every where.
  - $\tanh'(z) = 1 - \tanh^2(z)$
- ReLU (Rectified linear unit): $\mathrm{ReLU}(z) = \max(0, z)$.
  - Inexpensive to calculate derivatives, and alleviates gradient vanishing problems. Hence, popular for DL models.
  - There exist many slight variants.
  - $\mathrm{ReLU}'(z) = \begin{cases} z & \text{, if } z >= 0 \\ 0 & \text{, otherwise.} \end{cases}$

Illustration of Non-linearalities



Forward Computation



Notations: $w_{i \to j}^{[l]}$: the weight on the edge from the i-th neuron in layer $l-1$ to the j-th neuron in layer l.

Things to ponder:

- Which weights influence $z_1^{[2]}$?
- What's the impact to y if x_1 increases by a tiny amount ϵ?

Function Approximation

- ANN can well approximate any function (despite potentially huge size requirement)
- Learning: find $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_i \ell(\mathbf{y}_i, \mathbf{t}_i)$, where $\mathbf{y}_i = f(\mathbf{x}_i; \boldsymbol{\theta})$

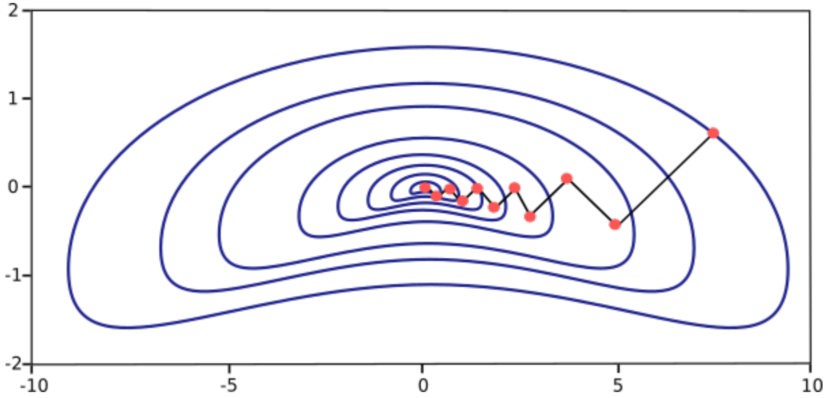
Function Minimization

- Typically, NP-hard to minimize a general function.
- However, we can find a good-quality local minima instead of the global minimum.
- Gradient Descent:
  1. Start at a random $\mathbf{x}$.
  2. Approximate a tiny neighborhood around $\mathbf{x}$ using a linear function
  3. Based on this approximation, find the best direction to move $\mathbf{x}$ within the tiny neighborhood. Then, **goto** Step 2.
- Extending Taylor series to functions with vector input.

$$f(x_0 + \epsilon) \approx f(x_0) + f'(x_0)\epsilon$$
$$f(\mathbf{x}_0 + \boldsymbol{\epsilon}) \approx f(\mathbf{x}_0) + \langle \boldsymbol{\nabla} f(\mathbf{x}_0), \boldsymbol{\epsilon} \rangle$$

Which $\boldsymbol{\epsilon}$ can minimize $f(\mathbf{x}_0 + \boldsymbol{\epsilon})$ subject to $\|\boldsymbol{\epsilon}\| \leq$ some small constant?

Illustration of GD



Variants of GD

- Gradient descent (GD):

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \alpha \cdot \boldsymbol{\nabla}_L(\boldsymbol{\theta}^{(t)})$$

- Stochastic gradient descent (SGD):
  - $\boldsymbol{\nabla}_L(\boldsymbol{\theta})$ is evalauted only on a randomly chosen training sample.
  - Inexpensive to compute the $\boldsymbol{\nabla}$, but bringing in much variance.
- Mini batch SGD:
  - $\boldsymbol{\nabla}_L(\boldsymbol{\theta})$ is evalauted only on a mini-batch of training sample.
  - Tuning minibatach sizes may achieve good results.
- SGD with momentum:
  - Think of the gradient as the velocity, and $\boldsymbol{\theta}$ as the position. Then this method keeps a portion of the last velocity value together with new gradient.
  - Helps to get over some difficult regions quickly (e.g., avoid too much oscillation).

Derivative

Let $y(x, a) = \sin(a \cdot x + 3\exp(x))$. Compute $\frac{\partial y}{\partial x}$.

- $\frac{\partial y}{\partial x} =$

Rewrite y in a verbose manner:

- $y = \sin(z_1)$
- $z_1 = z_2 + 3z_3$
- $z_2 = a \cdot x$
- $z_3 = 3\exp(x)$

Then:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_1} \frac{\partial z_1}{\partial x} \qquad \frac{\partial z_1}{\partial x} = \frac{\partial z_2}{\partial x} + 3\frac{\partial z_3}{\partial x}$$

Rules

Important rules about (partial) derivatives (useful for NN):

- Chain rule: $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_1} \frac{\partial z_1}{\partial z_2} \dots \frac{\partial z_k}{\partial x}$
- Sum rule: $\frac{\partial (z_1 + z_2)}{\partial x} = \frac{\partial z_1}{\partial x} + \frac{\partial z_2}{\partial x}$

These rules still hold for functions with vector/matrix input(s).

Note:

- We require that $\frac{\partial y}{\partial \mathbf{x}}$ has the same shape as $\mathbf{x}$.
- We can use this as a cue to work out which term needs a transposition.

Computational Graph

$$y(x, a) = \sin(a \cdot x + 3\exp(x))$$

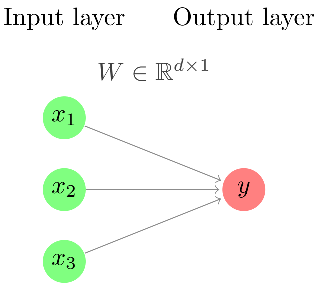
Baby Network

Model:

- For single $\mathbf{x} \in \mathbb{R}^d$:
  $y = \mathbf{W}\mathbf{x} + b$
- For many **x**s: $y = \mathbf{x}\mathbf{W} + \mathbf{b}$
- **not** the same **x**, **W** above

Shapes:

- y is a *scalar*
- **x** is a row vector, $\mathbb{R}^{1 \times d}$
  ($d = 3$ here)
- **W** is a matrix, $\mathbb{R}^{d \times 1}$
- b (plot as x_0) is a *scalar*



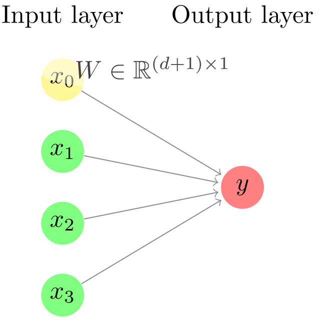
Simplifying the Bias Terms

Model:

- Extend **x** to $\mathbb{R}^{d+1}$ and let $x_0 := 1$.
- $y = \mathbf{xW}$
- i.e., $y = \sum_{i=0}^{d} x_i W_{i1}$. Therefore, W_{01} is the bias term.

Shapes:

- y is a *scalar*
- **x** is a row vector, $\mathbb{R}^{1\times(d+1)}$ ($d = 3$ here)
- **W** is a matrix, $\mathbb{R}^{(d+1)\times 1}$



Add the Non-linear Transformation

Model:

- For simplicity, ignore the bias terms from now on in this lecture **only**.
- $y = \sigma(\underbrace{\mathbf{Wx}}_{z})$
- Let σ be the sigmoid function, then $\sigma'(u) =$

Shapes:

Exercise:

- $\frac{\partial y}{\partial \mathbf{W}} =$

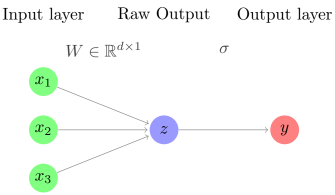


Figure: NN1

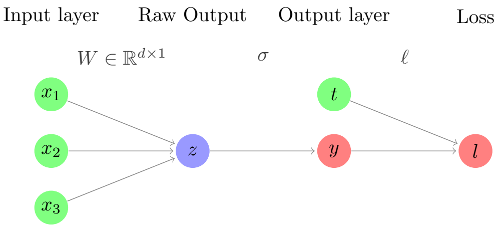
Add the Loss Function

Model:

- $l = \ell(\underbrace{\sigma(\underbrace{\mathbf{Wx}}_{z})}_{y}, t)$
- $\ell(u, v) = \frac{1}{2}(u - v)^2$

Exercise:

- $\frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{W}} =$



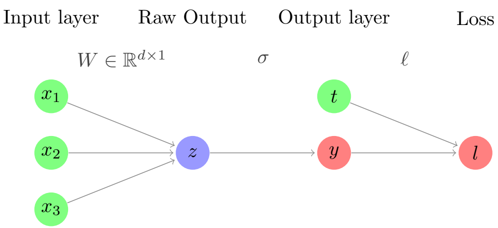
Vectorized Version

Model:

- $l = \underbrace{\ell(\sigma(\underbrace{\mathbf{WX}}_{\mathbf{z}}), \mathbf{t})}_{\mathbf{y}}$
- $\ell(\mathbf{u}, \mathbf{v}) = \frac{1}{2}\|\mathbf{u} - \mathbf{v}\|^2$

Exercise:

- $\frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{W}} =$



Computational Graph

Model:

- $l = \ell(\underbrace{\sigma(\underbrace{\mathbf{WX}}_{\mathbf{z}})}_{\mathbf{y}}, \mathbf{t})$
- $\ell(\mathbf{u}, \mathbf{v}) = \frac{1}{2}\|\mathbf{u} - \mathbf{v}\|^2$

Exercise:

- $\frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{W}} =$

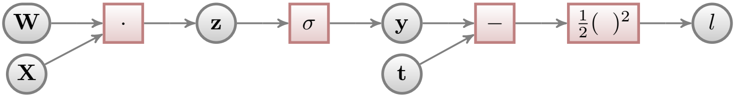
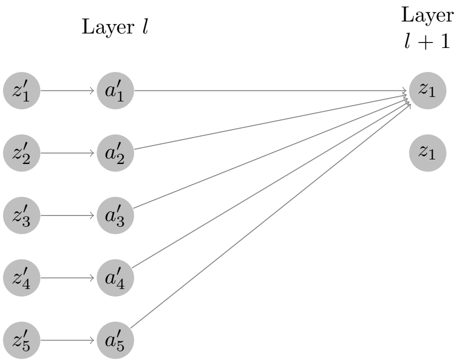


Figure: NN2

Model Learning

- Backpropagation (BP) is the prevalent training method for NNs.
  - Forward pass/propagation
  - Backward pass/propagation
- Supported by Autograd modules in most DL frameworks by building the computational graph (explicitly or implicitly).

Forward Propagation

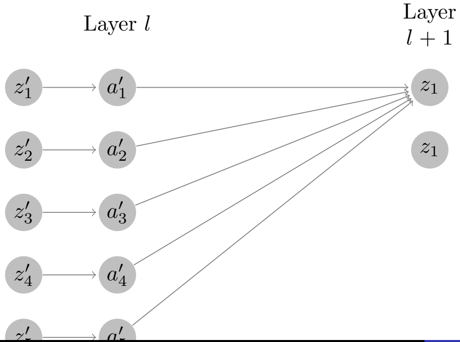


- $z_1 =$

Backward Propagation /1

Define $\boldsymbol{\delta}^{(l)} := \frac{\partial \ell}{\partial \mathbf{z}^{(l)}}$; intuitively, meaning "how much each $\mathbf{z}_i^{(l)}$ needs to move".

- For a non-final layer: $\boldsymbol{\delta}^{(l)} = \left((\mathbf{W}^{(l+1)})^\top \boldsymbol{\delta}^{(l+1)}\right) * \sigma'(\mathbf{z}^{(l)})$
- For the final layer: $\boldsymbol{\delta}^{(l)}$ can be calculated directly (by chain rules).



- $\frac{\partial z_1}{\partial a_i^l} =$

Backward Propagation /2

$\frac{\partial \ell}{\partial \mathbf{W}^{(l)}}$ can be easily obtained from $\boldsymbol{\delta}^{(l)}$ as: $\boldsymbol{\delta}^{(l)}(\mathbf{a}^{l-1})^{\top}$.

References

TBA