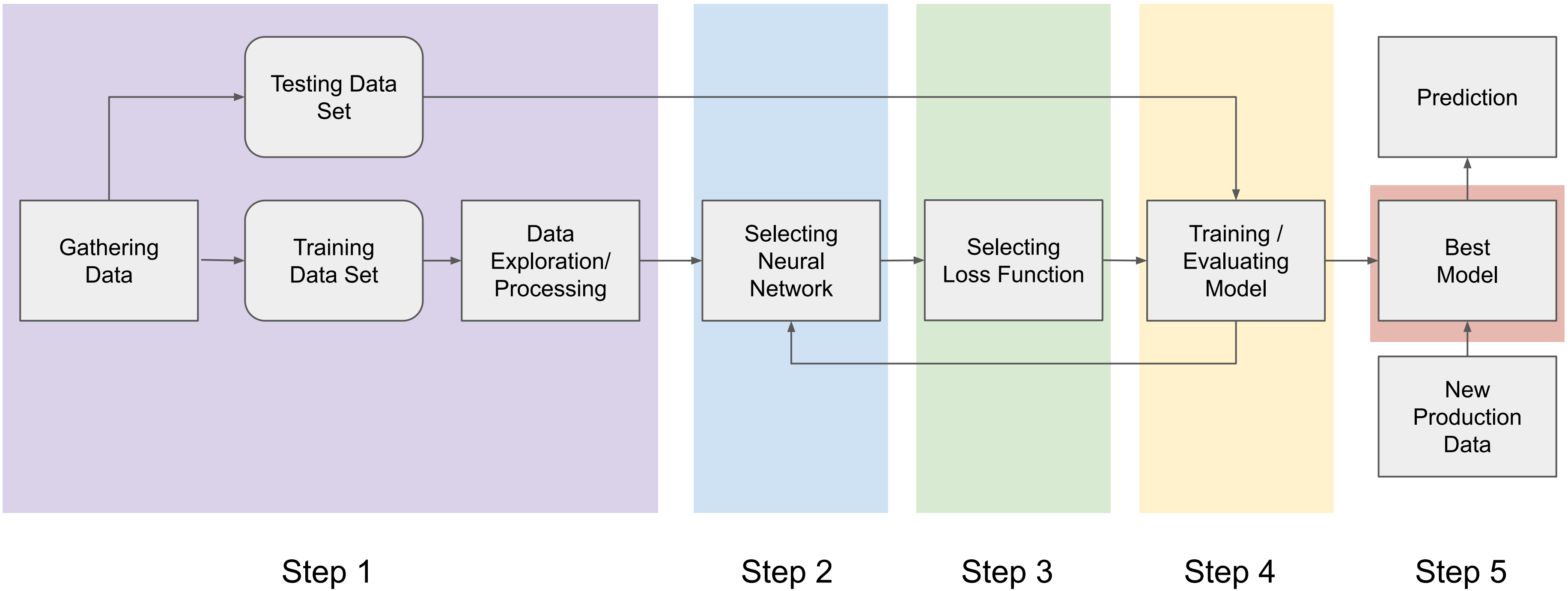
Learning Deep Learning with PyTorch

(2) Mechanics of Learning

Qiyang Hu
UCLA IDRE
October 31, 2019

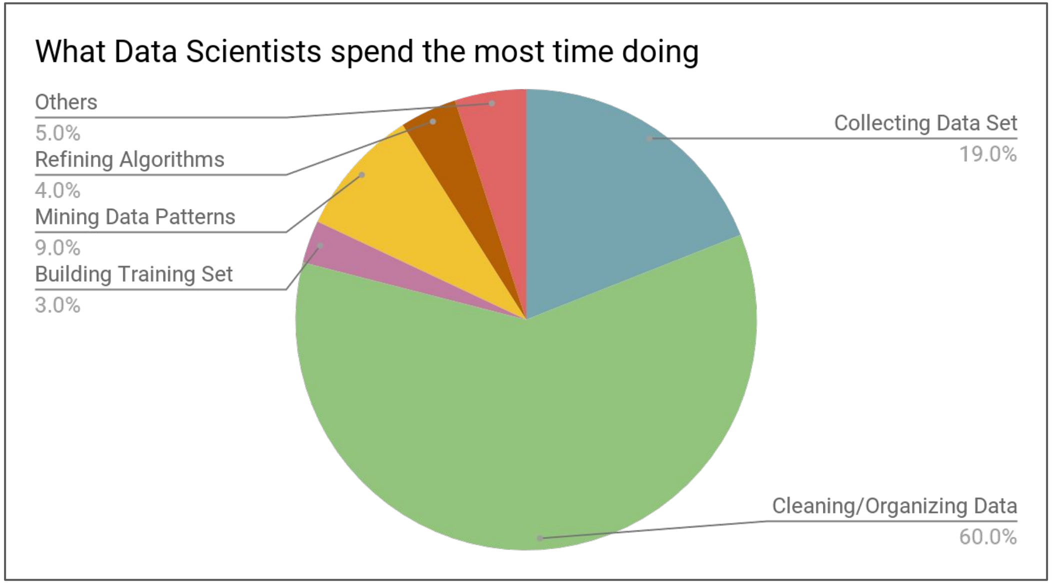
Workflow for a deep learning project



Qiyang Hu

Step 1. Data Prep

- The most time-consuming but the most creative job
    - Take > 80% time
    - Require experience
    - May need domain expertise
- Determines the upper limit for the goodness of DL
    - Models/Algorithms: just approach the upper limit

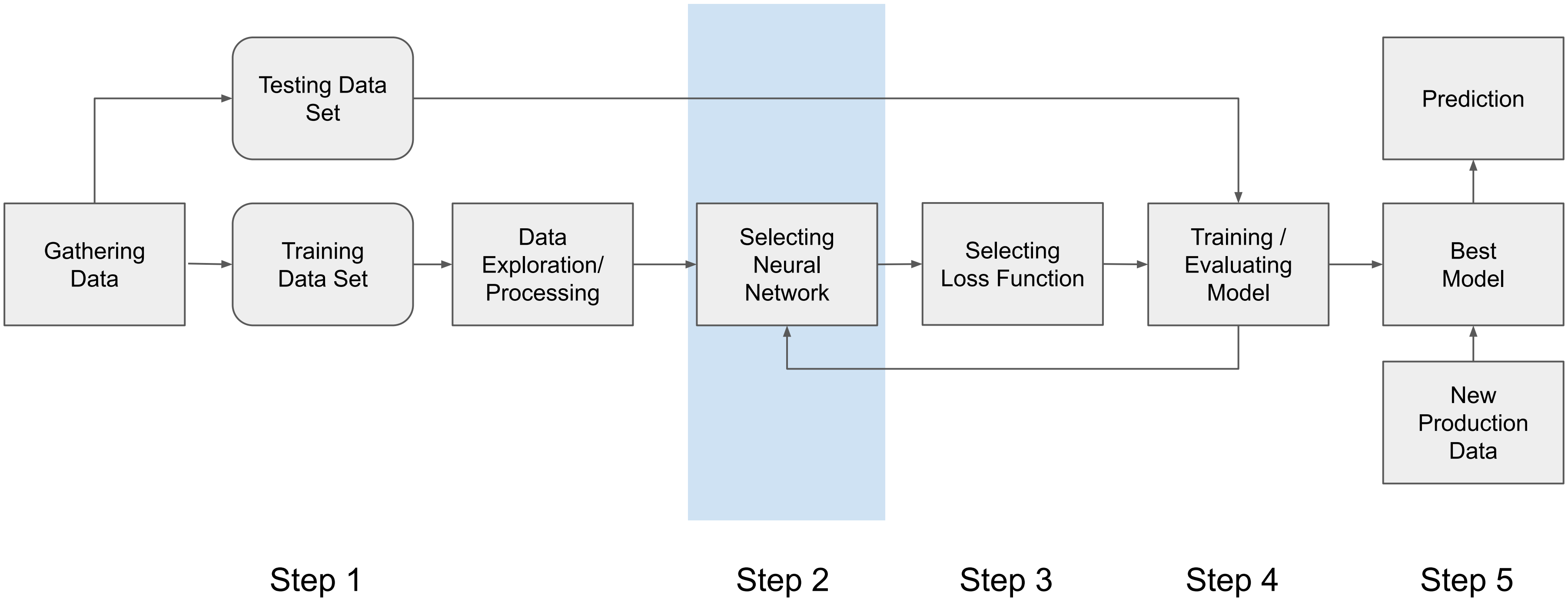


Survey from Forbes in 2017 (Data Source)

Feature Engineering

- Transforming raw data into features with a good representation
  - Deep learning can extract hierarchical features automatically
  - DL still needs the digitalization of the input raw data and some form of prior knowledge.
- Some common techniques:
  - Imputation (almost every column)
  - Label binarization (e.g. sex)
  - One-hot encoding (nominal categorical data)
  - Binning and grouping
  - Scaling: standardization and normalization (numerical values with different ranges)
  - Splitting the features (e.g. title from name, etc)

Workflow for a deep learning project

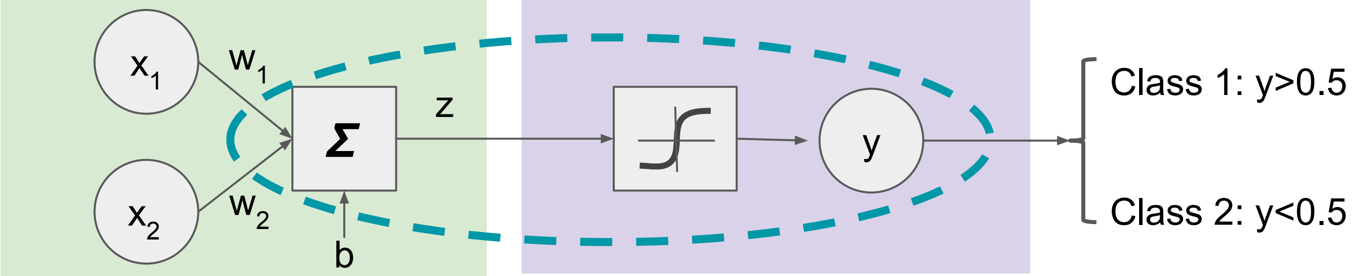


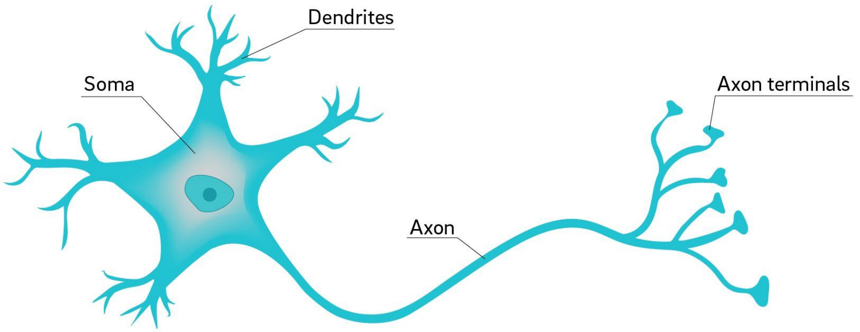
Qiyang Hu

Recap: A linear classifier ~ one artificial neuron

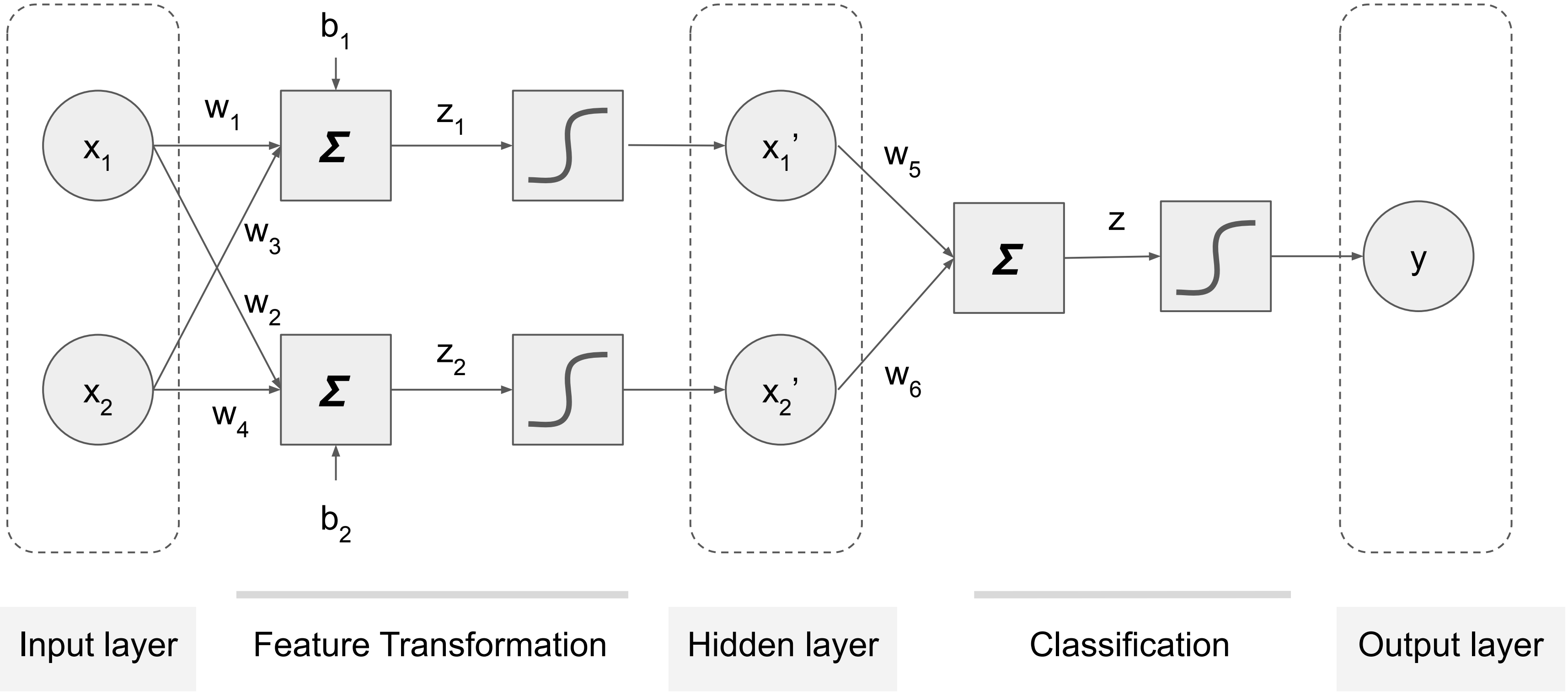
$$z = w_1 x_1 + w_2 x_2 + b$$

$$y = \sigma(w_1 x_1 + w_2 x_2 + b)$$



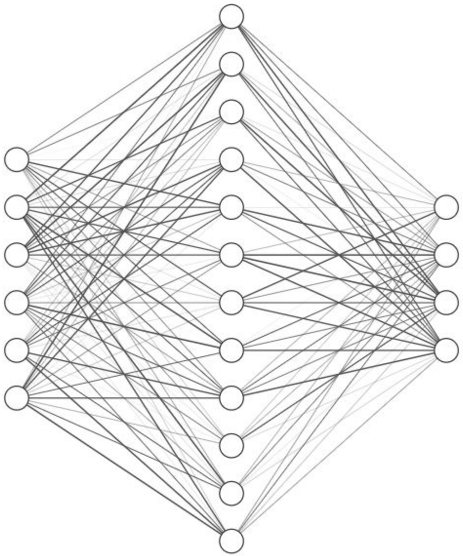


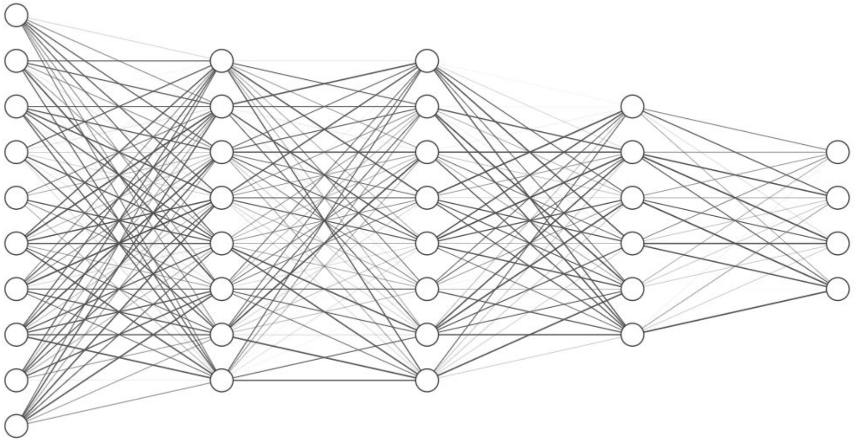
(Deep) Neural Networks ~ piling/stacking logistic-regression classifiers



Why deep?

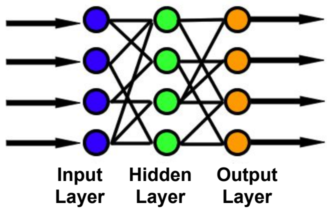
- Shallow network can fit any function
  - Has less number of hidden layers
  - Has to be really “fat”
- Deep network is more efficient.
  - It can extract/build better features
  - Exponentially fewer parameters ([2017])



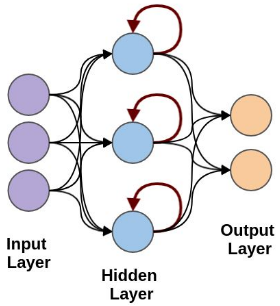


Types of Neural Network Architectures

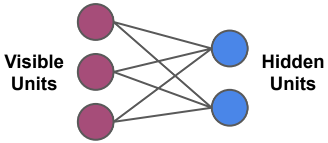
- Feed forward neural networks (No cycle in node connections)
  - Fully connected network
  - Convolutional networks (CNNs)



- Recurrent networks (w/ directed cycle in node connections)
  - Fully recurrent NN
  - Recursive NN
  - Long short-term memory (LSTM)
  - Hopfield network (w/o hidden nodes)

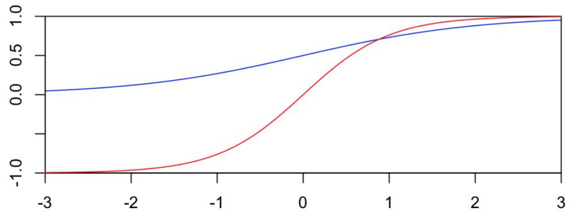


- Symmetric networks (no directions in node connections)
  - Boltzmann Machines
    - RBM, DBM

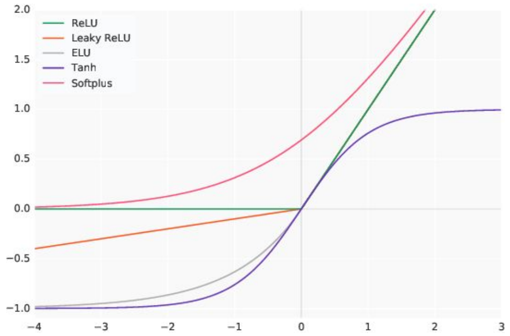


Activation Function

- Sigmoid function: $$\sigma(z) = \frac{1}{1+\exp(-z)}$$
- tanh function: $$tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

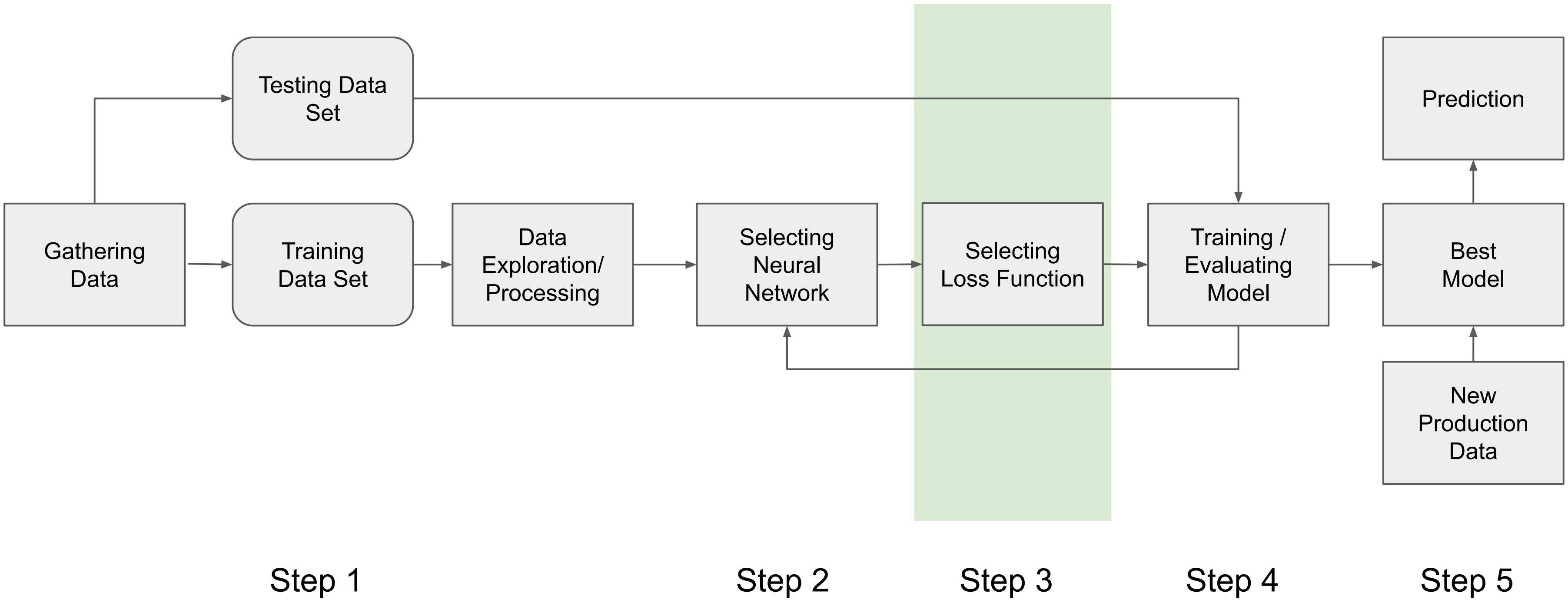


- Rectified linear unit (ReLU) $$f(x) = x^+ = \max(0, x)$$
  - Softplus
  - Leaky ReLU
  - Exponential LU (ELUs)
  - GELU, etc.



- Softmax function: $$y_i = \frac{e^{z^{(i)}}}{\sum_{j=0}^{K} e^{z^{(j)}}}$$
- Maxout Network:
  - *Learnable* activation function

Workflow for a deep learning project



Qiyang Hu

How to measure the performance of the model?

- General name: objective function

- Measure the misfit of the model as a function of parameters
  - Criterion is to *minimize* the error functions
  - Loss Function: for a single training example
  - Cost Function: over the entire or mini-batch training set

- Evaluate the probability of generating training set
  - Criterion is to *maximize* the distribution likelihood as a function of parameters
  - Maximum (log)-likelihood estimation

- Regression losses and classification losses

Loss functions

- Regression Loss
  - Mean Square Error / Quadratic Loss / L2 Loss:

$$L_{MSE} = \frac{1}{n}\sum_{i}^{n}(t_i - s_i)^2$$

  - Mean Absolute Error / L1 Loss:

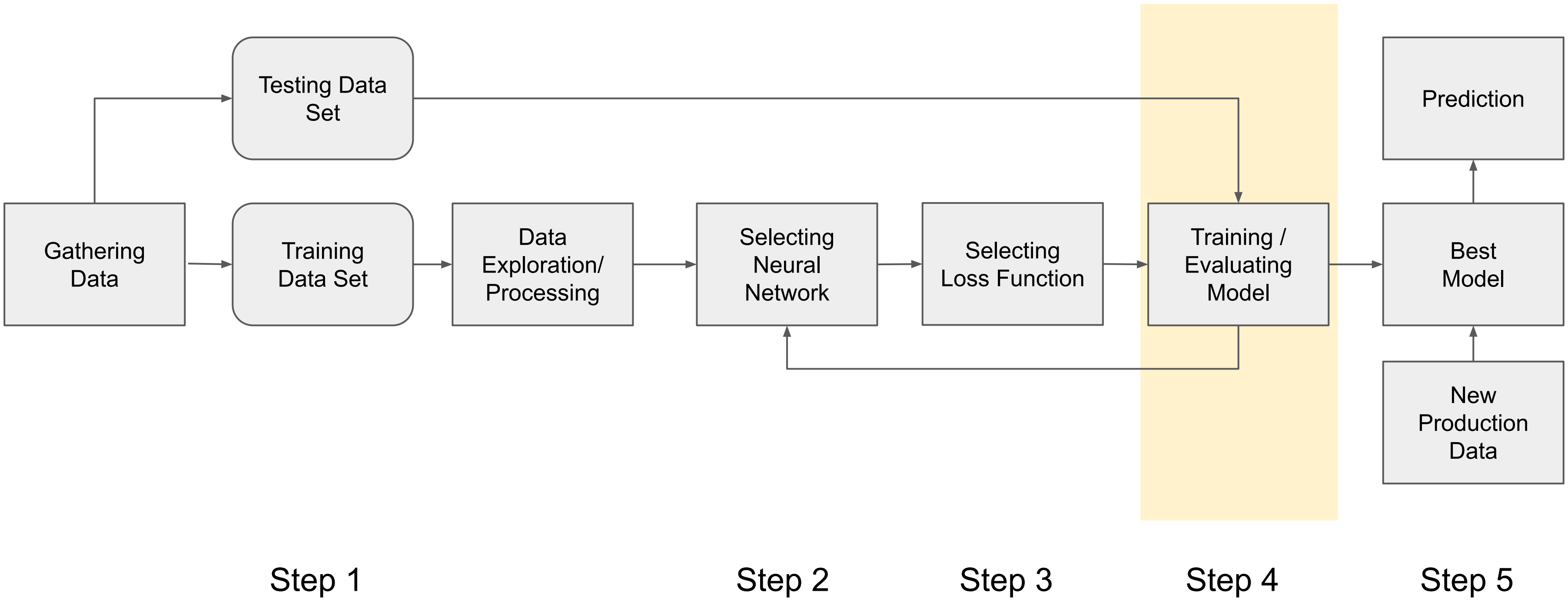
$$L_{MAE} = \frac{1}{n}\sum_{i}^{n}|t_i - s_i|$$

- Cross-Entropy Loss and variations

$$L_{CE} = -\sum_{i}^{C} t_i \log(s_i)$$

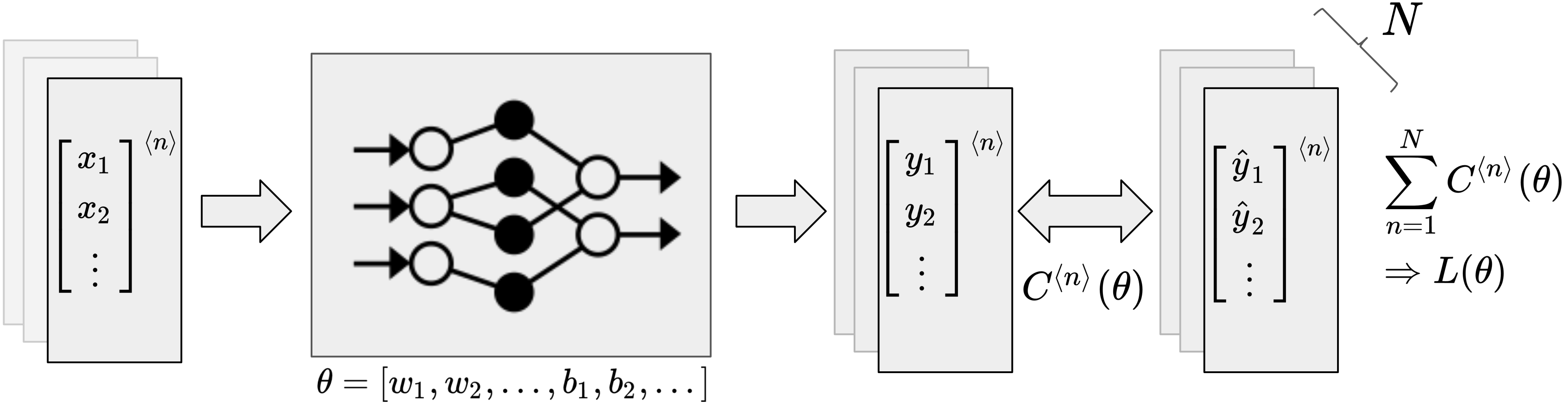
  - Softmax Loss / Log Loss / Negative Log Likelihood
  - Focal Loss
  - Dice Loss / IOU Loss / Tversky Loss
- Ranking Losses
  - Ranking Loss/Margin Loss/Contrastive Loss/Triplet Loss
  - Hinge Loss

Workflow for a deep learning project



Qiyang Hu

Training a DNN is an optimization problem

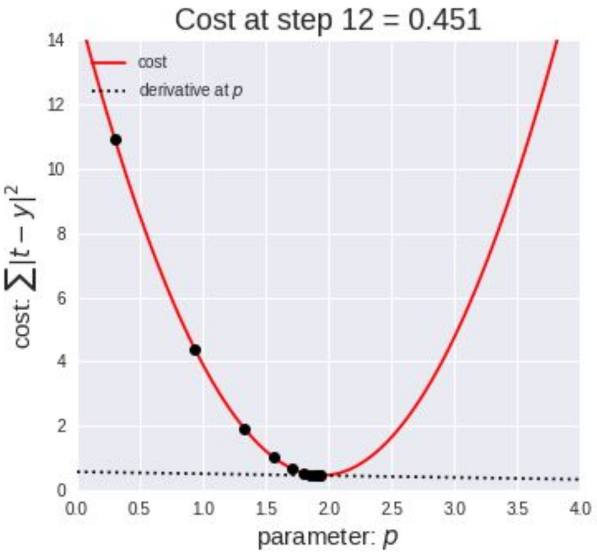


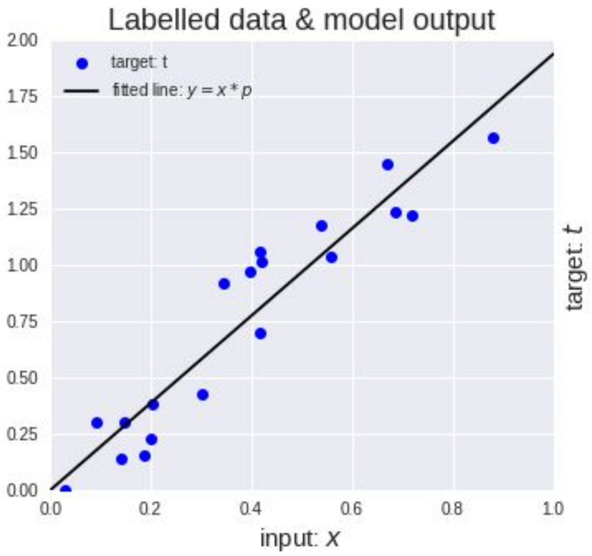
- We know how to compute L(θ), analytically or numerically.
- Start from an arbitrary initialization of θ_0, and get an initial $L_0(\theta)$
- Apply optimization algorithm to minimize L(θ)

Qiyang Hu

DL Optimization Algorithm

- Gradient Descent (a 1st-order approach) $\theta \longleftarrow \theta - \eta \nabla L(\theta)$
  - Most popular algorithm
    - Pros: simple and fast
    - Cons: sometimes hard to tune



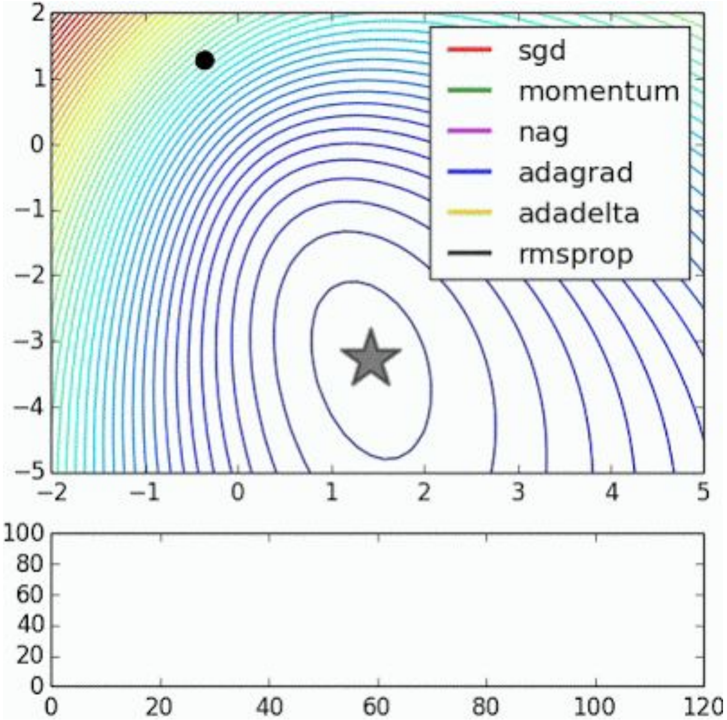


Source Link

Qiyang Hu

Gradient-Descent Optimizers

- Stochastic GD / Mini-Batch GD
- Adding momentum:
  - Classical Momentum (CM)
  - Nesterov's Accelerated Gradient (NAG)
- Adaptive learning rate:
  - AdaGrad, AdaDelta, ...
  - RMSprop (animation source) →
- Combining the two
  - **ADAM** (as **default** in many libs)
- Beyond Adam:
  - Lookahead (2019)
  - RAdam (2019)
  - AdaBound/AmsBound (ICLR 2019)
  - Range (2019)



Higher Order Optimization Algorithm

- Newton-like methods (2nd-order methods)

$$\theta \longleftarrow \theta - \frac{\ell'(\theta)}{\ell''(\theta)}$$

  - Prons:
    - Fewer iterations (quadratic convergence)
    - Fewer hyperparameters
  - Cons:
    - Much more **costly** in each iteration
    - Need more storing
  - BFGS/L-BFGS: a quasi-newton one
    - Good for low dimensional models
  - CG (Conjugate gradient)
    - moderately high dimensional models

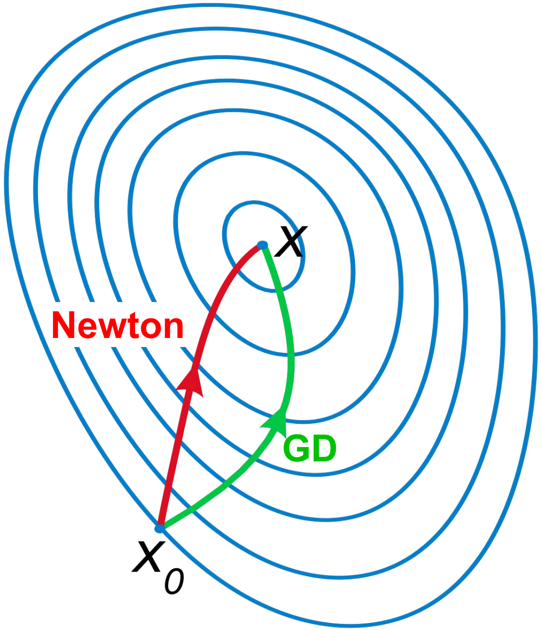
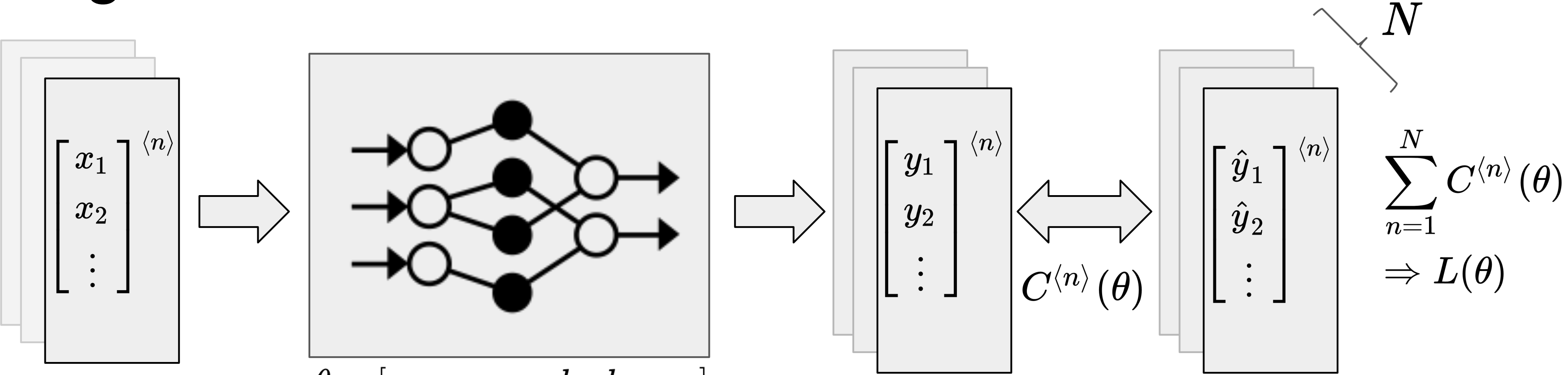


Figure from Wikipedia

Using Gradient Descent to train DNN



$$\theta = \underbrace{[w_1, w_2, \ldots, b_1, b_2, \ldots]}_{\text{Millions of parameters!}}$$

$$\theta_0 \to \nabla L(\theta_0) \to \theta_1 \to \nabla L(\theta_1) \to \theta_2 \to \cdots$$

$$\theta_1 = \theta_0 - \eta \nabla L(\theta_0)$$

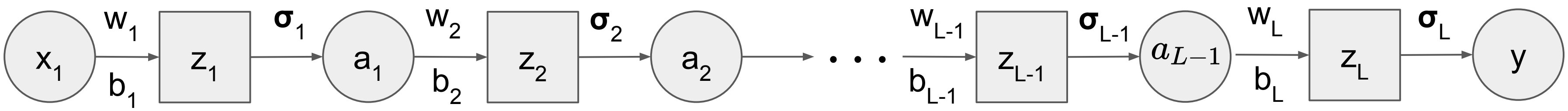
$$\theta_2 = \theta_1 - \eta \nabla L(\theta_1)$$

$$\vdots$$

$$\nabla L(\theta) = \sum_{n=1}^{N} \begin{bmatrix} \frac{\partial C^{\langle n \rangle}(\theta)}{\partial w_1} \\ \frac{\partial C^{\langle n \rangle}(\theta)}{\partial w_2} \\ \vdots \\ \frac{\partial C^{\langle n \rangle}(\theta)}{\partial b_1} \\ \vdots \end{bmatrix}$$

How to compute the gradient vector with millions of elements **efficiently?**

Backpropagation: a game of chain rule



$$y = \sigma_L\Big(w_L \cdot \sigma_{L-1}\big(\cdots w_2 \cdot \underbrace{\sigma_1(\underbrace{w_1 \cdot x + b_1}_{z_1})}_{a_1} + b_2\big) + b_L\Big)$$

$$\frac{\partial C\big(y(w)-\hat{y}\big)}{\partial w} = \frac{\partial z}{\partial w}\frac{\partial C}{\partial z} = \frac{\partial z}{\partial w}\left[\frac{\partial a}{\partial z}\frac{\partial C}{\partial a}\right] = \frac{\partial z}{\partial w}\left[\sigma' \cdot \left(\frac{\partial z_{(+1)}}{\partial a}\frac{\partial C}{\partial z_{(+1)}}\right)\right]$$

① Forward Pass

$$\frac{\partial z_1}{\partial w_1} = x_1 \longrightarrow \frac{\partial z_2}{\partial w_2} = a_1 \longrightarrow \cdots \longrightarrow \frac{\partial z_{L-1}}{\partial w_{L-1}} = a_{L-2} \longrightarrow \frac{\partial z_L}{\partial w_L} = a_{L-1}$$

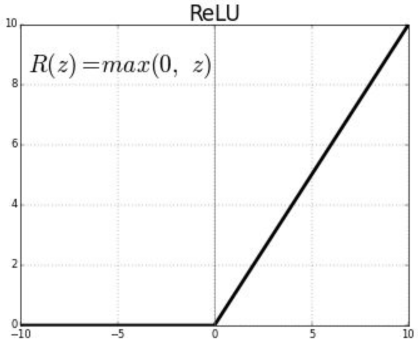
② Backward Pass

$$\frac{\partial C}{\partial z_1} = \sigma'_1\left[w_2\frac{\partial C}{\partial z_2}\right] \longleftarrow \cdots \longleftarrow \frac{\partial C}{\partial z_{L-1}} = \sigma'_{L-1}\left[w_L\frac{\partial C}{\partial z_L}\right] \longleftarrow \frac{\partial C}{\partial z_L} = \sigma'_L\frac{\partial C}{\partial y} \longleftarrow \frac{\partial C}{\partial y}$$

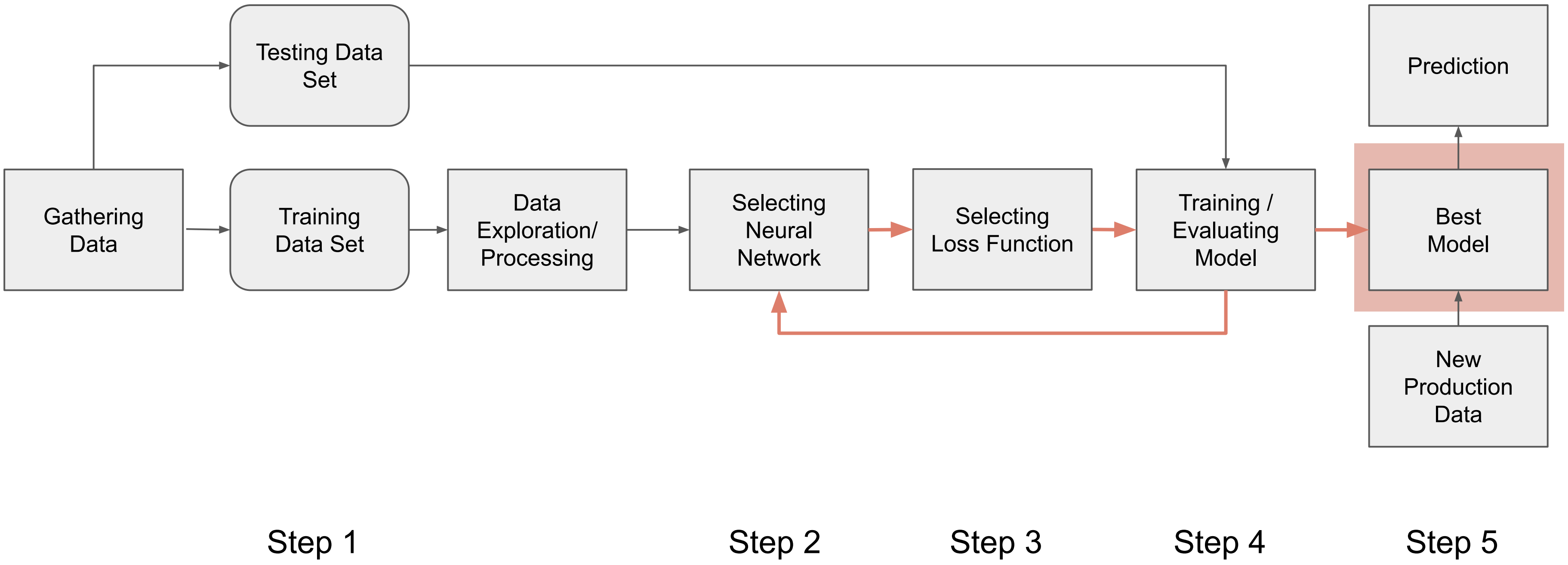
Qiyang Hu

Wait, here is a catch...

- ReLU as one of the most popular activation functions: $f(x) = x^+ = \max(0, x)$
- ReLU is not differentiable at x=0
- Why we can use it in gradient based DNN training?
    - NN training *rarely* arrives at a local minimum of the cost function
    - Software implementations of NN training usually return one of the one-sided derivatives (*sub-gradient*)
- In practice we can safely disregard the non-differentiability of the hidden unit activation functions.

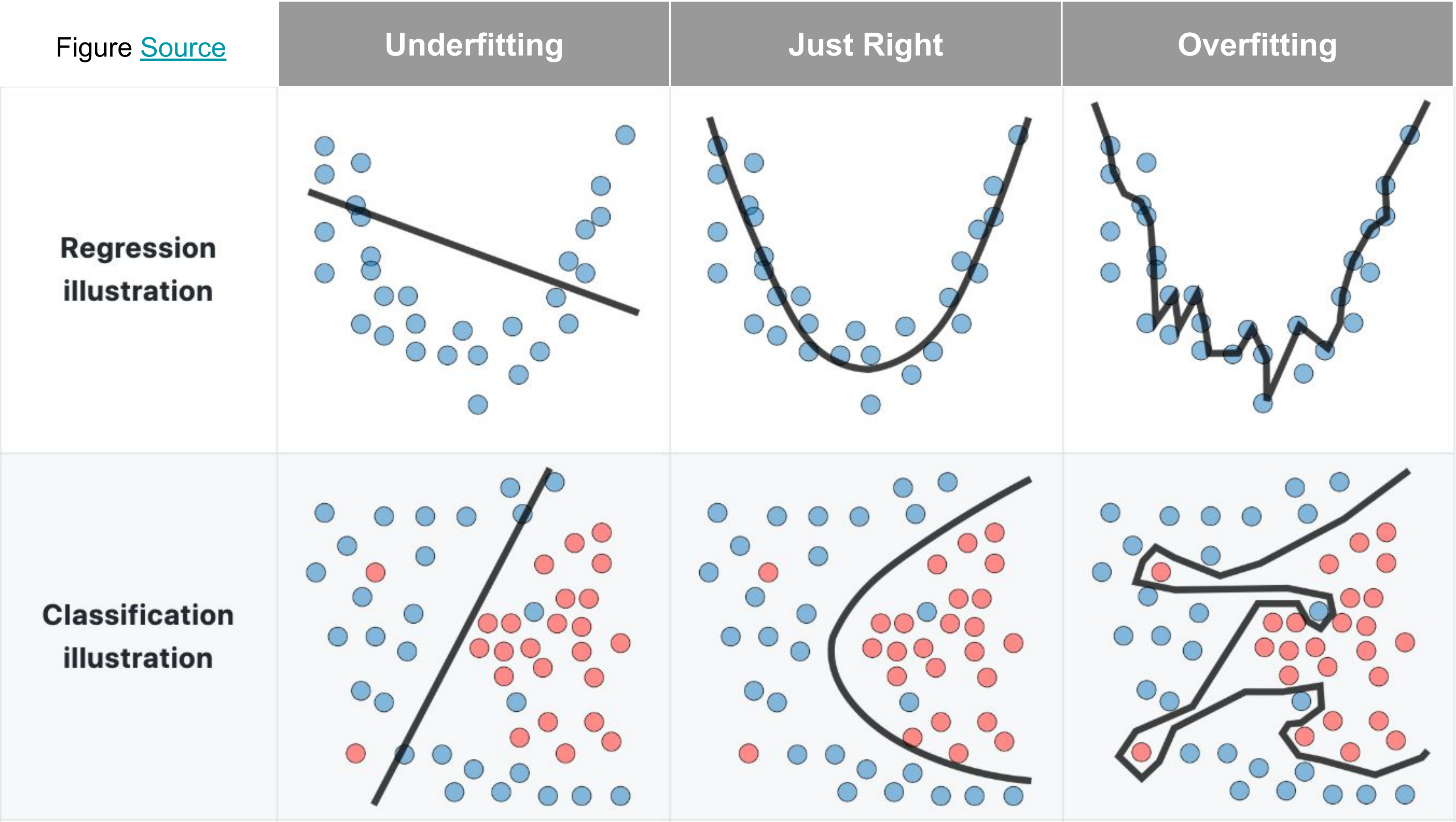


Workflow for a deep learning project



Qiyang Hu

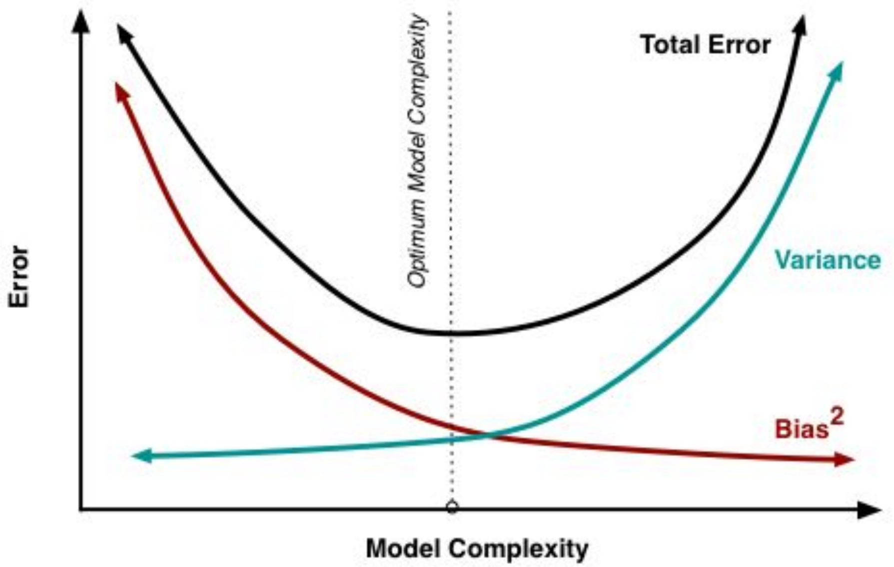
Figure Source



Underfitting and Overfitting

- Underfitting: model too simple:
  - Diagnose:
    - cannot even fit the training data
    - training error ~ testing error
  - Ignore the variance in training data
  - Higher prediction bias

- Overfitting: model too complex
  - Diagnose:
    - well-fit for training data
    - large error for testing data
  - Over-interpret training data
  - More deviation from new data



How to prevent underfitting?

- Redesign the model
- Increase model’s complexity
- Add more features as input
- Training longer
- More data will <u>not</u> help

How to prevent overfitting?

- Get more data
  - Collect more data
  - Data augmentation

- Reduce the model's complexity

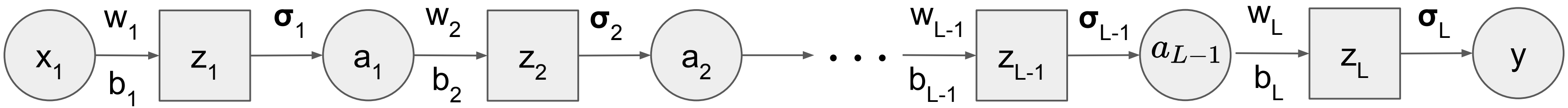
- Regularization
  - Weight Regularization to make the model smoother (L1, L2, Elastic net)

$$\hat{L}(x, y) = L(x, y) + \lambda \sum_{i=1}^{n} \theta_i^2$$

  - Early stopping

Gradient vanishing/exploding in DL training

- Causes

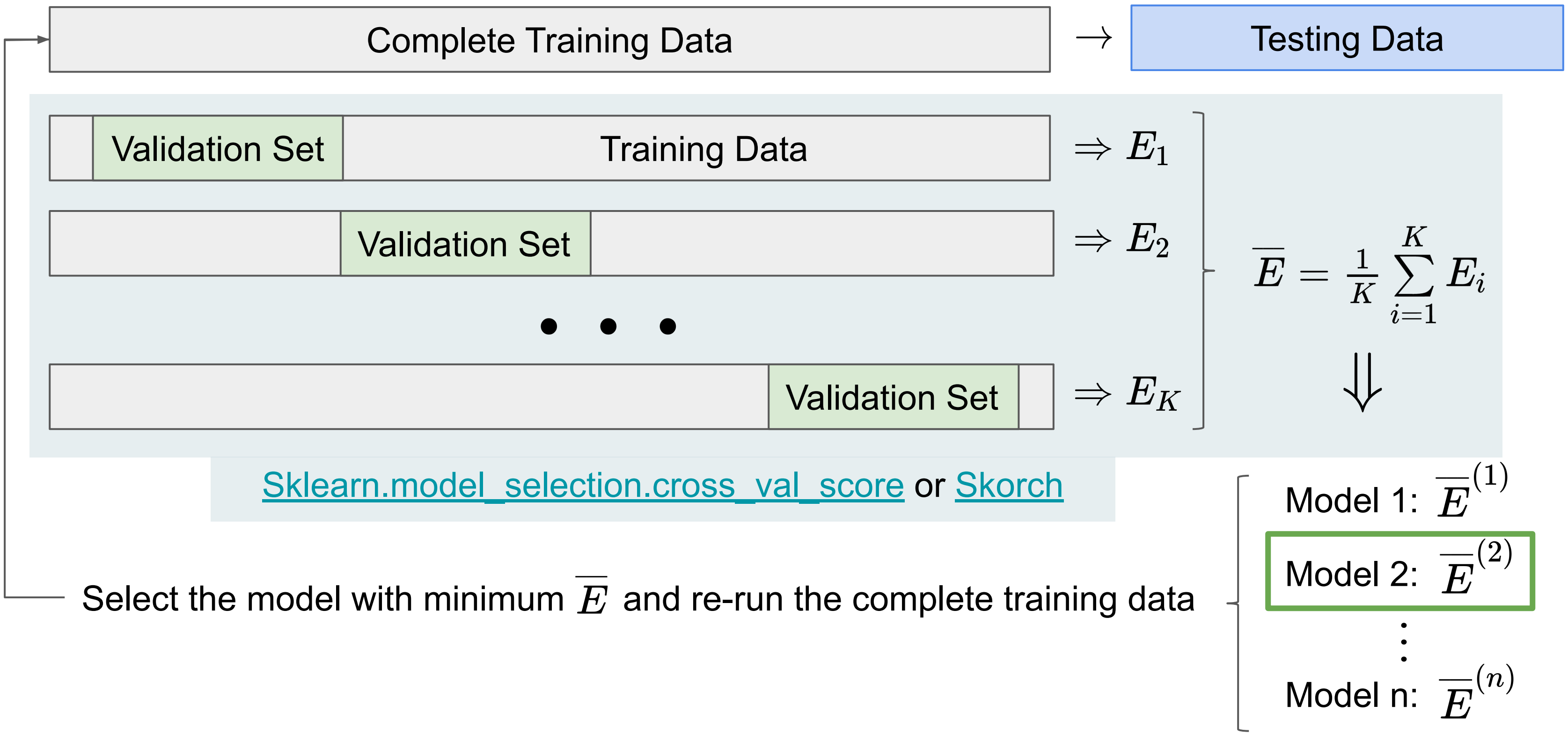


$$y = \sigma_L\Big(w_L \cdot \sigma_{L-1}\big(\cdots w_2 \cdot \sigma_1(w_1 \cdot x + b_1) + b_2\big) + b_L\Big)$$

  - Gradients in initial layers = Multiplication of Gradients at prior layers
  - Small variation around 1 results in vanishing/exploding

- Techniques to resolve:
  - General: adjusting learning rate, dropout, batch normalization, layer normalization
  - For gradient exploding: gradient clipping, weight regularization
  - For gradient vanishing: activation function, proper initialization parameters, LSTM, skip connections

Model Selection: K-fold Cross Validation



Errors/scores in practice

Training Set	Validation Set		Public Testing Set	Private Testing Set

Error: $E^{val} < E^{Pub} < E^{Pri}$

Score: $S^{val} > S^{Pub} > S^{Pri}$

Don’t forget to

- Sign in your info to the class
  - To get the email notifications

- Contact me for questions or discussions
  - huqy@idre.ucla.edu
  - Office: Math Sci #3330
  - Phone: 310-825-2011