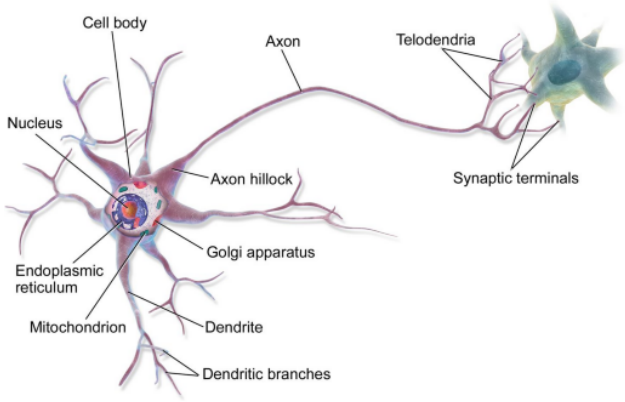
Lecture 3: Multi-Layer Perceptron (MLP)/ Neural Networks

Assoc Prof Tham Chen Khong
Dept of Electrical & Computer Engineering (ECE)
NUS
E-mail: eletck@nus.edu.sg

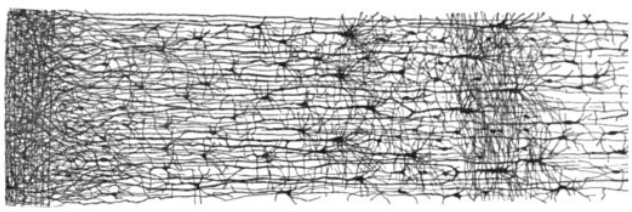
Acknowledgement: some materials from EE2211, NUS; Géron, Hands On ML

Neural Networks & Deep Learning

- Original inspiration: brain structure



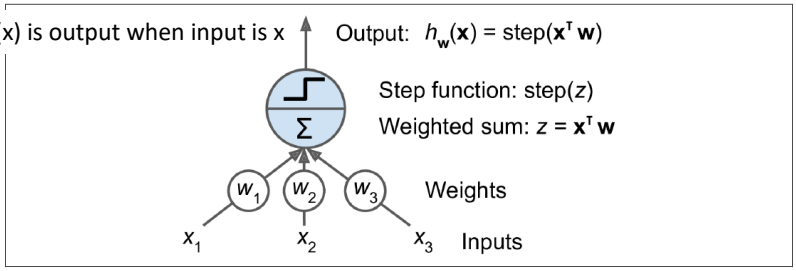
Biological neuron

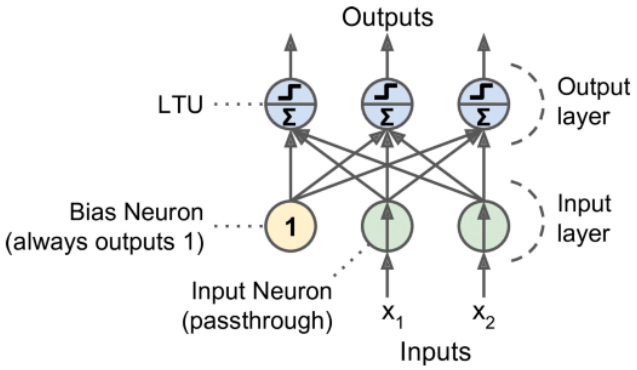


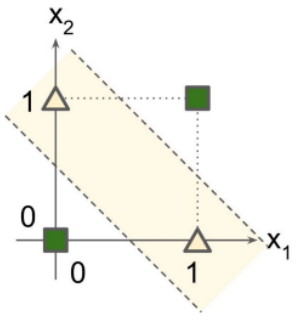
Multiple layers of neurons

Perceptron

McCulloch & Pitts (1943)
Rosenblatt (1950's); Minsky & Papert (1969)

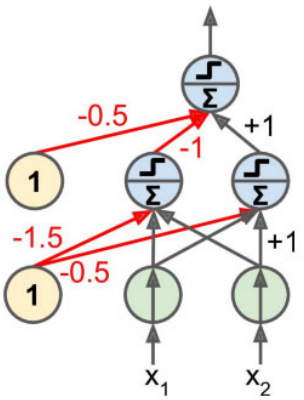






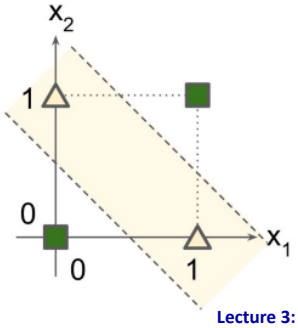
unable to learn the XOR function

Multi-Layer Perceptron (MLP)

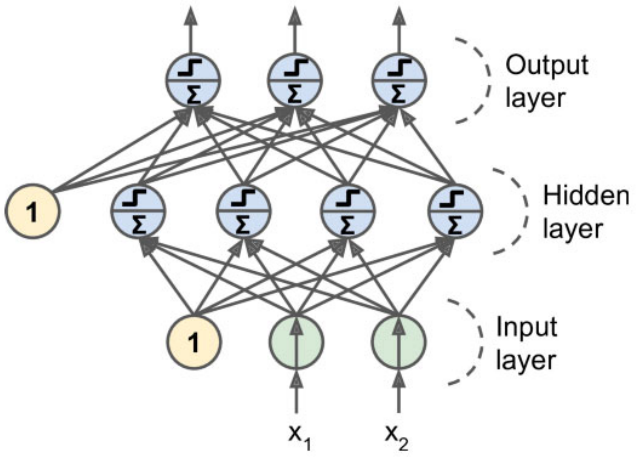


Other connections have weight=1

Solves the XOR problem!



Extend to more outputs and nodes in hidden layer (also, more layers (later))

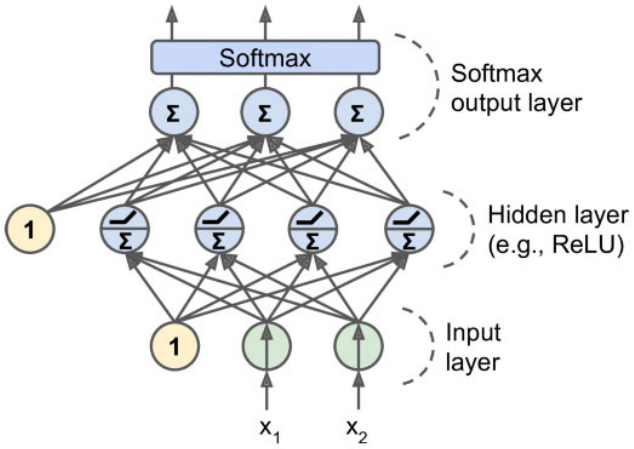


Modern MLP / Neural Network

- Able to do classification or regression

Forward Pass does Inference

ensures output probabilities sum to 1.0



Multi-class classification: softmax

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

A modern MLP (including ReLU and softmax) for classification

Machine Learning Big Picture

- Building blocks of machine learning algorithms
  - Learning model: reflects our belief about the relationship between features and the target or label we want to predict
  - Loss function: penalty for wrong prediction
  - Regularization: penalizes complex models or overfitting
  - Optimization routine: find minimum of overall cost function
- Different learning models, e.g. linear, polynomial, sigmoid, ReLU, exponential, etc.
- Different loss functions, e.g. squared error, cross entropy, etc.
- Different optimizers, e.g. stochastic gradient descent (SGD), Adam, etc.

Training in Supervised Learning

- Supervised learning: given feature(s) x, we want to predict target y
- Most supervised learning algorithms can be formulated as the following optimization problem

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, \mathbf{Data\text{-}Loss}(\mathbf{w}) + \lambda \mathbf{Regularization}(\mathbf{w})$$

- **Data-Loss(w)** quantifies fitting error to training set given parameters **w:** smaller error => better fit to training data
- **Regularization(w)** penalizes more complex models and prevents overfitting

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} L(f(\mathbf{x}_i, \mathbf{w}), y_i) + \lambda R(\mathbf{w})$$

m training samples

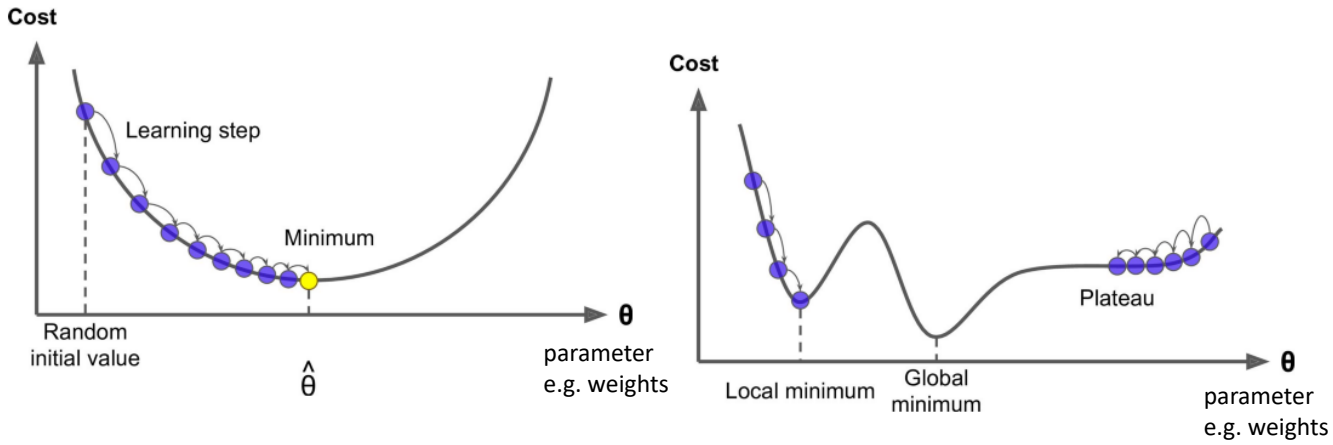
Cost Function — Loss Function — Learning Model (e.g. the whole NN) — Regularization (a.k.a. Penalty term)

Training in Supervised Learning

- An ML model usually has parameters that need to be determined, e.g. weights
- These parameters can be initialised to zero or small random values
- The model output is computed with these parameter values
- The output is compared with the label or target output
- The difference is the *error* which forms the *loss* term, which is part of the *cost*
- The learning algorithm determines updates to the parameters to reduce the error
- The process continues until some stopping criterion is met, e.g. the error is sufficiently small

Pictorial view of the Training process

- e.g. minimize sum of squared error



Gradient Descent algorithm

- Suppose we want to minimize $C(\mathbf{w})$ with respect to $\mathbf{w} = [w_1, \cdots, w_d]^T$
- Gradient $\nabla_{\mathbf{w}} C(\mathbf{w}) = \begin{pmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_d} \end{pmatrix}$
  - $\nabla_{\mathbf{w}} C(\mathbf{w})$ is vector & function of $\mathbf{w}$
  - $\nabla_{\mathbf{w}} C(\mathbf{w})$ is direction at $\mathbf{w}$ where C is increasing most rapidly, so $-\nabla_{\mathbf{w}} C(\mathbf{w})$ is direction at $\mathbf{w}$ where C is decreasing most rapidly
- Gradient Descent:

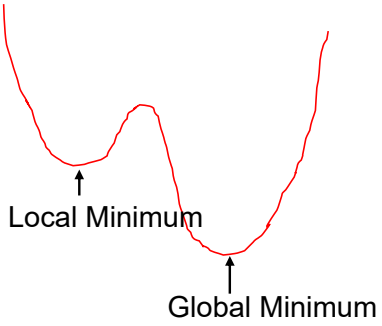
Initialize $\mathbf{w}_0$ and learning rate η;
while *true* **do**
  Compute $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$
  if *converge* **then**
    return $\mathbf{w}_{k+1}$
  end
end

At each iteration, compute gradient and update model parameters in direction opposite to gradient.
According to multi-variable calculus, if eta is not too big, then $C(\mathbf{w}_{k+1}) < C(\mathbf{w}_k)$ => we get a better $\mathbf{w}$ after each iteration

Gradient Descent algorithm

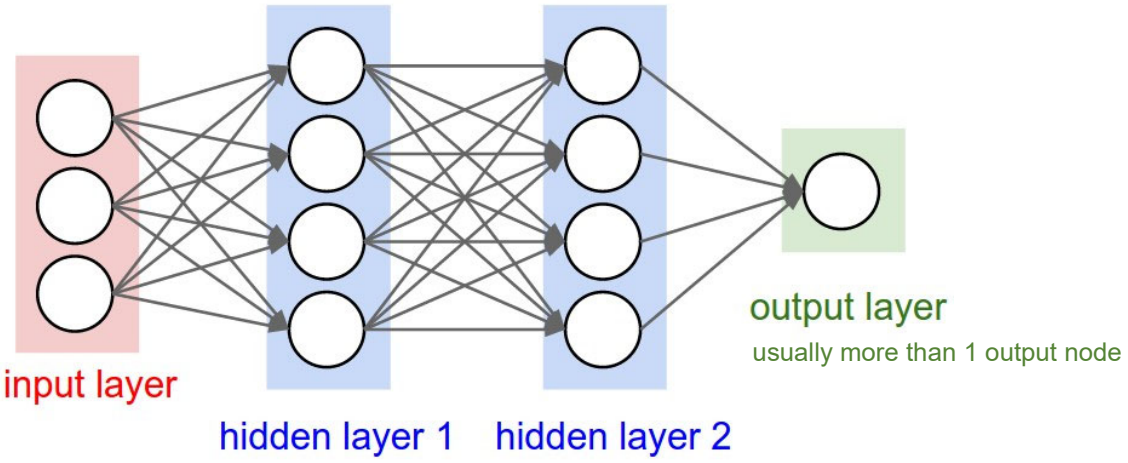
- Gradient Descent:

Initialize $\mathbf{w}_0$ and learning rate η;
while *true* **do**
  Compute $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$
  if *converge* **then**
    return $\mathbf{w}_{k+1}$
  end
end

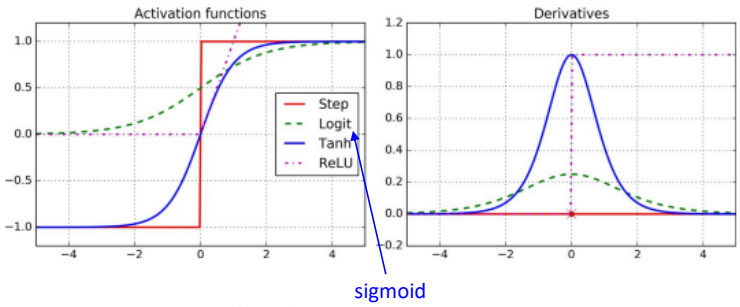


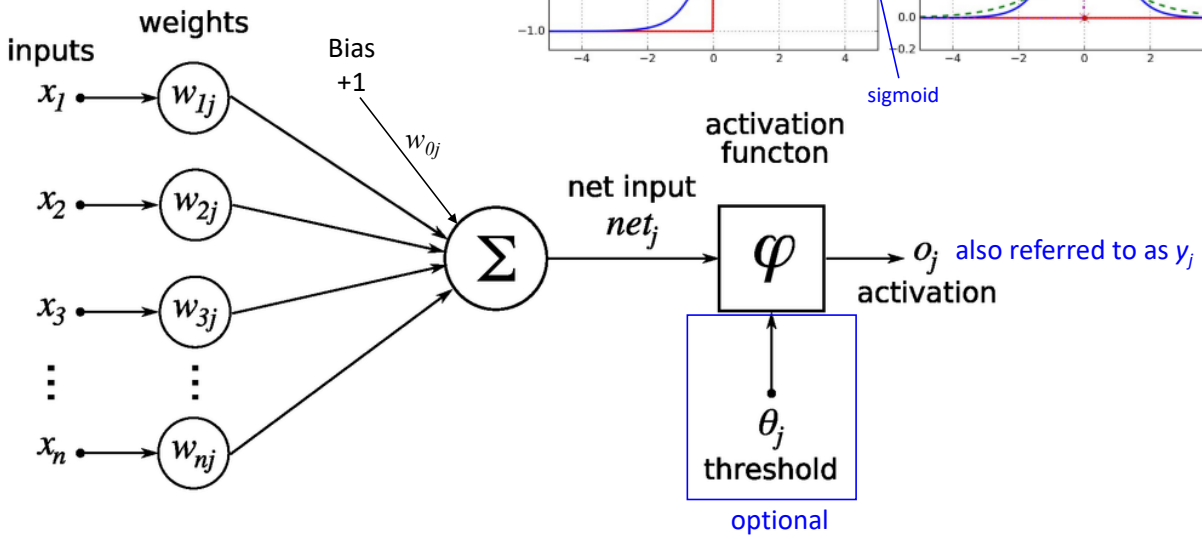
- Possible convergence criteria
  - Set maximum iteration k
  - Check percentage or absolute change in C below a threshold
  - Check percentage or absolute change in $\mathbf{w}$ below a threshold
- Gradient descent can only find local minimum
  - Because gradient = 0 at local minimum, so $\mathbf{w}$ will not change after that
- Many variations of gradient descent, e.g. change how gradient is computed or learning rate η decreases with increasing k
- Need to use appropriate value of η (too large → unstable; too small → slow convergence)

Multi-Layer Perceptron (MLP)



Single Neuron (node *j*)





Back Propagation algorithm

Rumelhart, Hinton, Williams (1986)

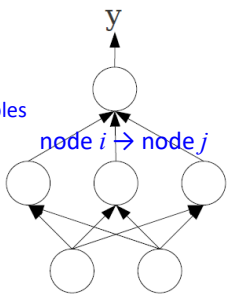
How Do We Train A Multi-Layer Network?

Loss function

Define sum-squared error:

p is index to samples

$$E = \frac{1}{2}\sum_p (d^p - y^p)^2$$



Notes: - can have more than 1 output (same manner)
- assume no regularization term

Use gradient descent error minimization:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

Works if the nonlinear activation function is differentiable.

Switch to Smooth Nonlinear Units

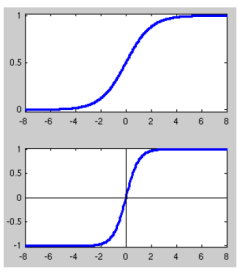
$$\text{net}_j = \sum_i w_{ij} y_i$$ "network input"

$$y_j = g(\text{net}_j)$$ *g must be differentiable* (φ)

Common choices for g:

$$g(x) = \frac{1}{1+e^{-x}}$$ logistic fn

$$g'(x) = g(x)\cdot(1-g(x))$$

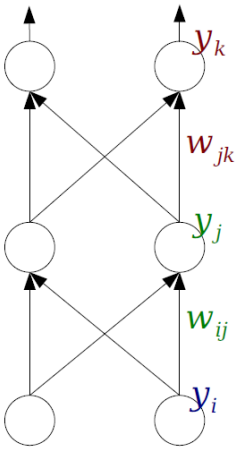


$$g(x) = \tanh(x)$$
$$g'(x) = 1/\cosh^2(x)$$

Back Propagation algorithm

Rumelhart, Hinton, Williams (1986)

Now We Can Use The Chain Rule



$$\frac{\partial E}{\partial y_k} = (y_k - d_k)$$

$$\delta_k = \frac{\partial E}{\partial net_k} = (y_k - d_k)\cdot g'(net_k)$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial net_k}\cdot\frac{\partial net_k}{\partial w_{jk}} = \frac{\partial E}{\partial net_k}\cdot y_j$$

$$\frac{\partial E}{\partial y_j} = \sum_k \left(\overbrace{\frac{\partial E}{\partial net_k}}^{\delta_k} \cdot \overbrace{\frac{\partial net_k}{\partial y_j}}^{w_{jk}} \right)$$

$$\delta_j = \frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial y_j}\cdot g'(net_j)$$

$$\frac{\partial E}{\partial w_{ij}} = \overbrace{\frac{\partial E}{\partial net_j}}^{\delta_j}\cdot y_i$$

Weight Updates

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial net_k}\cdot\frac{\partial net_k}{\partial w_{jk}} = \delta_k \cdot y_j$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j}\cdot\frac{\partial net_j}{\partial w_{ij}} = \delta_j \cdot y_i$$

$$\Delta w_{jk} = -\eta\cdot\frac{\partial E}{\partial w_{jk}} \qquad \Delta w_{ij} = -\eta\cdot\frac{\partial E}{\partial w_{ij}}$$

The same principle is used in DNN/CNN and RNN/LSTM (BPTT) as we shall see later in this course

Optimizers

Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.
SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Image 1.

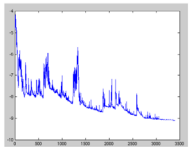


Image 1: SGD fluctuation (Source: Wikipedia)

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Adaptive Moment (Adam)

Adam,[17] which stands for *adaptive moment estimation*, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-8).[18]

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1)\nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$ (element wise multiplication)
3. $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\theta \leftarrow \theta + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$ (element wise division)

17 Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint arXiv: 1412.6980 (2014).

18 These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment* while the variance is often called the *second moment*, hence the name of the algorithm.

source: Géron, Hands-on ML

Others: momentum, RMSprop, Adadelta, Adamax etc.

Hyperparameters

- A **hyperparameter** is a parameter whose value is used to control the learning process (it is different from "weights" or "parameters")
- e.g. the batch size is a hyperparameter of gradient descent that controls the number of training samples to work through before the model's internal parameters or weights are updated
- e.g. the number of epochs is a hyperparameter that controls the number of complete passes through the training dataset
- Other hyperparameters: optimizer (e.g. SGD, Adam) and its parameter settings, no. of layers and neurons in each layer, activation functions etc.

Training, Validation and Test sets

- **Basic method**
- Split entire dataset into **training set** and **validation sets** (we may also wish to set aside a **test set**)
  - The training set is used for training
  - The validation set (fixed) is to **simulate the unseen test data** for **model tuning** during the **training stage**
- Use validation set to decide when to stop training and prevent overfitting

- also used for hyperparameter tuning (see later)

- *Strictly speaking, the **test set** is untouched until the final evaluation. note: sometimes, the "test set" is used as the validation set*
- Performance metrics such as mean-squared error, classification accuracy etc.

Performance Metrics: Binary Classification

Confusion Matrix for Binary Classification

	$\hat{P}$ (predicted)	$\hat{N}$ (predicted)	
P (actual)	TP	FN	Sensitivity/ Recall TP/(TP+FN)
N (actual)	FP	TN	Specificity TN/(TN+FP)
	Precision TP/(TP+FP)	Accuracy (TP+TN)/(TP+TN+FP+FN)	

Evaluation is usually done with validation or test samples, but sometimes we evaluate for training samples

Refer to sklearn.metrics and Keras metrics

Performance Metrics: Regression

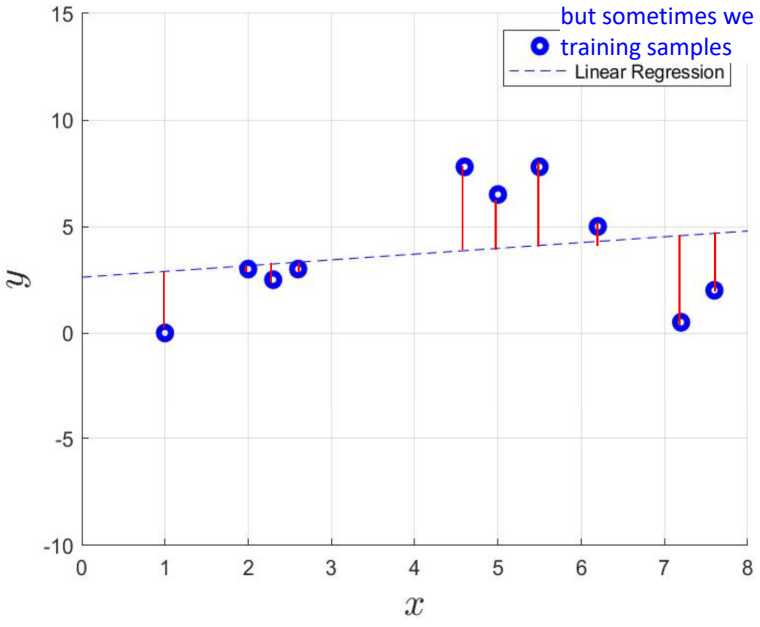
Mean Square Error

$$(\text{MSE} = \frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{n})$$

Mean Absolute Error

$$(\text{MAE} = \frac{\Sigma_{i=1}^{n}|y_i - \hat{y}_i|}{n})$$

where y_i denotes the target output and $\hat{y}_i$ denotes the predicted output for sample i.

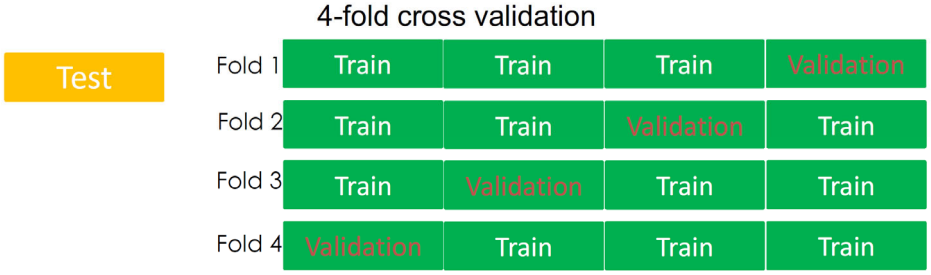


Evaluation is usually done with validation or test samples, but sometimes we evaluate for training samples

Refer to sklearn.metrics and Keras metrics

Training, Validation and Test sets

- **Advanced method with *K*-fold Cross Validation**
- Split entire dataset into **training set** and **test set**
- The **training set** is used for training and validation
- *The **test set** is untouched until the final evaluation*
- Partition the training set and select one partition to be the validation set
  - next, the validation set is rotated until all the training data have been used for validation
  - in this example, since we partition the training data into 4 equal parts, i.e. *K*=4



Hyperparameter Optimization / Tuning

- The validation set is used for tuning the hyperparameters during the training stage
  - evaluate the model with different hyperparameters on the validation set

Which hyperparameter value(s) should we use?

	Acc. on Val Set 1 **Fold 1**	Acc. on Val Set 2 **Fold 2**	Acc. on Val Set 3 **Fold 3**	Acc. on Val Set 4 **Fold 4**	Avg. Acc on Val. Set
Classifier with hyper-Param1	88%	89%	93%	92%	90.5%
Classifier with hyper-Param2	90%	88%	91%	91%	90%

- Hence, select hyperparameter 1 and retrain using the entire training set to obtain the final model, which can then be evaluated on the test set
- Note: Hyperparameter optimization will be covered in greater detail later

Thank You

Questions?

Assoc Prof Tham Chen Khong

E-mail: eletck@nus.edu.sg