What are neural networks?

An artificial neural network (ANN) is a digital architecture that mimics human cognitive processes to model complex patterns, develop predictions, and react appropriately to external stimuli. Structured data is required for many types of machine learning, versus neural networks, which are capable of interpreting events in the world around them as data that can be processed.

Whenever you read a report, watch a movie, drive a car, or smell a flower, billions of neurons in your brain process the information through tiny electric signals. Each neuron processes inputs, and the results are output to the next neuron for subsequent processing, ultimately and instantly producing a business insight, a chuckle, a foot on the brake, or a little joy. In machine learning, neural networks allow digital systems to interpret and react to situations in much the same way.

An ANN is like a brain full of digital neurons, and while most ANNs are rudimentary imitations of the real thing, they can still process large volumes of nonlinear data to solve complex problems that might otherwise require human intervention. For example, bank analysts can use an ANN to process loan applications and predict an applicant's likelihood of default.

What you can do with neural networks?

In machine learning, neural networks are used for learning and modeling complex, volatile inputs and outputs, inferring unseen relationships, and making predictions without data distribution restrictions. Neural network models are the foundation for many deep learning applications, such as computer vision and natural language processing, which can help support fraud protection, facial recognition, or autonomous vehicles.

Most businesses rely on forecasting to inform business decisions, sales strategies, financial policies, and resource utilization. But the limitations of traditional forecasting often make it difficult to predict complex, dynamic processes with multiple and often hidden underlying factors, such as stock market prices. Deep learning neural network models help expose complex nonlinear relationships and model unseen factors so that businesses can develop accurate forecasts for most business activities.

Common neural networks

There are dozens of different types of AI neural networks, and each is suitable for different deep learning applications. Use an ANN that's appropriate for your business and technology requirements. Here are some examples of common AI neural networks:

Convolutional neural network (CNN)

Developers use a CNN to help AI systems convert images to digital matrices. Used primarily for image classification and object recognition, CNNs are appropriate for facial recognition, topic detection, and sentiment analyses.

Deconvolutional neural network (DNN)

If complex or high-volume network signals get lost or convoluted with other signals, a DNN will help find them. DNNs are useful for processing high-resolution images and optical flow estimates.

Generative adversarial network (GAN)

Engineers use a GAN to train models how to generate new information or material that mimics the specific properties of the training data. GANs help the models distinguish subtle differences between originals and copies to make more authentic copies. GAN applications include high-fidelity image and video generation, advanced facial recognition, and super resolution.

Recurrent neural network (RNN)

An RNN inputs data to hidden layers with specific time-delays. Network computing accounts for historical information in current states, and higher inputs don't change the model size. RNNs are a good choice for speech recognition, advanced forecasting, robotics, and other complex deep learning workloads.

Transformers

Transformers are designed to handle sequential input data. However, they aren't restricted to processing that data in sequential order. Instead, transformers use attention—a technique that allows models to assign different levels of influence to different pieces of input data and to identify the context for individual pieces of data in an input sequence. This allows for an increased level of parallelization, which can reduce model training times.

Machine learning vs. neural networks

Although neural networks are considered a subset of machine learning, there are some significant differences between neural networks and regular machine learning models.

For one, neural networks are generally more complex and capable of operating more independently than regular machine learning models. For example, a neural network is able to determine on its own whether its predictions and outcomes are accurate, while a machine learning model would require the input of a human engineer to make that distinction.

Additionally, neural networks are structured so that the neural network can continue to learn and make intelligent decisions all on its own. Machine learning models, on the

other hand, are limited to decision-making based only on what it has specifically been trained on.

Neural network examples

A neural network simulates the way humans think. It's no surprise that neural networks are versatile since our brains are also so versatile. Below, you will find examples of different technologies that neural networks contribute to, applications in specific industries, and use cases for companies using neural networks to solve problems.

Neural network examples: Technology

A neural network acts as a framework, supporting how artificial intelligence will operate and what it will do with the data presented to it. As a framework, it powers specific technologies like computer vision, speech recognition, natural language processing, and recommendation engines, giving us specific use cases for neural network technology. Let's take a closer look at each of these AI fields.

Computer vision

Computer vision allows artificial intelligence to "look" at an image or video and process the information to understand and make decisions. Neural networks make computer vision faster and more accurate than was previously possible because a neural network can learn from data in real time without needing as much prior training. Much like human vision, artificial intelligence can use computer vision to observe and learn, classifying visual data for a broad range of applications.

Speech recognition

Speech recognition allows AI to "hear" and understand natural language requests and conversations. Scientists have been working on speech recognition for computers since at least 1962. But today, advancements in neural networks and deep learning make it possible for artificial intelligence to have an unscripted conversation with a human, responding in ways that feel natural to a human ear. You can also use neural networks to enhance human speech, for example, during recorded teleconferencing or for hearing aids.

Natural language processing

Natural language processing (NLP) is similar to speech recognition. In addition to understanding and interpreting spoken requests, NLP focuses on understanding text. This technology enables AI chatbots like ChatGPT to have a written conversation with you. Neural networks allow computer scientists to train NLP systems much faster because they do not have to hand code and train the algorithm.

Recommendation engines

A recommendation engine is an AI tool that suggests other products or media you might like based on what you've browsed, purchased, read, or watched. With neural networks, a recommendation engine can gain a deeper understanding of consumer behavior and offer further targeted results that are likely to interest consumers. Recommendation tools can help encourage customers to stay more engaged on a website and make it easier for them to find items they like.

Neural network examples: Applications

All the technologies mentioned above benefit from neural network artificial intelligence. In practice, these areas of artificial intelligence offer many uses. A few specific neural network examples include:

- **Medical imaging:** Healthcare professionals can use neural networks to read medical images, such as X-rays or MRIs. Artificial intelligence can analyze a medical image incredibly fast compared to a human professional and can continuously analyze images night and day, unlike a person constrained by human needs like hunger and fatigue.
- **Self-driving cars:** Neural networks power self-driving cars. While on the road, these cars must be aware of many different variables happening simultaneously and randomly. In this environment, artificial intelligence also needs to make decisions based on the information it receives. A neural network enables the complex thinking a self-driving vehicle requires.
- **Public safety and security:** Neural networks also offer various solutions for public safety and security. For example, artificial intelligence can be used for fraud detection, traffic accident detection, or predicting suspicious or criminal behavior.
- **Agriculture:** In agriculture, farmers can use artificial intelligence for tasks like irrigation, pest control, predicting weather patterns, and choosing seeds optimized for their growing area. For these tasks, the artificial intelligence will need sensors to help it gain more information about the growing conditions—for example, a sensor to detect moisture levels in soil.

- **Online content moderation:** Neural networks can detect online content that goes against community standards, acting as a quick and effective content moderator that never stops working. In fact, Meta reported in 2021 that it uses artificial intelligence to flag 97 percent of the content it removes from Facebook for community standards violations [2].
- **Voice-activated virtual assistants:** Using speech recognition technology, the neural network at the center of your voice-activated virtual assistant can understand what you say to it and respond accordingly. With the advanced ability of neural networks, voice-activated virtual assistants can also understand the tone and context of what you say.
- **AI subtitles:** Speech recognition and natural language processing together make it possible for artificial intelligence to automatically subtitle a video by listening to and understanding speech, and then translating it into a text caption.

Neural network use cases

We've discussed technologies and applications for neural networks, but what are some examples of companies using neural networks for solutions specific to their industries? Let's take a look at some solutions from Google and IBM:

- You can use Google Translate to automatically translate the text contained in an image. For example, you could take a picture of a street sign or handwritten note, and Google Translate will scan it and provide a translation.
- In 2018, IBM Watson used neural networks to create customized highlight reels of the Masters golf tournament. Users could curate the highlights they saw based on their preferences, taking advantage of a spoiler-free mode that would avoid ruining the cliffhanger moments.
- In a partnership between IBM Watson, Quest Diagnostics, and Memorial Sloan Kettering Cancer Center, artificial intelligence bolstered by neural networks began reviewing lab results from cancer patients to provide genetic testing. Comparing the results against a vast library of cancer-related research, the AI then suggests the best course of individualized treatment. An AI agent can complete this work in a fraction of the time it takes a human health care professional.

Multi-Objective Optimization for Software Testing Effort Estimation

Solomon Mensah[1], Jacky Keung[1], Kwabena Ebo Bennin[1] and Michael Franklin Bosu[2]
[1]Department of Computer Science, City University of Hong Kong, Hong Kong, China
{smensah2-c, kebennin2-c}@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk
[2]Centre for Business, Information Technology and Enterprise
Wintec, Hamilton, New Zealand
michael.bosu@wintec.ac.nz

***Abstract*— Software Testing Effort (STE), which contributes about 25-40% of the total development effort, plays a significant role in software development. In addressing the issues faced by companies in finding relevant datasets for STE estimation modeling prior to development, cross-company modeling could be leveraged. The study aims at assessing the effectiveness of cross-company (CC) and within-company (WC) projects in STE estimation. A robust multi-objective Mixed-Integer Linear Programming (MILP) optimization framework for the selection of CC and WC projects was constructed and estimation of STE was done using Deep Neural Networks. Results from our study indicate that the application of the MILP framework yielded similar results for both WC and CC modeling. The modeling framework will serve as a foundation to assist in STE estimation prior to the development of new a software project.**

Keywords- Software Testing Effort; Cross-Company; Within-Company; Optimization; Deep Neural Networks

I. INTRODUCTION

Software Testing Effort (STE) is one of the critical phases in software development as it is estimated to contribute about 25-40% [1] of the total development cost. STE is the effort in relation to duration, resources and source code for testing the software before deployment to the client or market [1]. The acquisition of relevant data for effort estimation has been a focus for prior work [2] concerning the effective use of within-company (WC) and cross-company (CC) datasets for predictive modeling. Researchers and practitioners who lack the relevant local data rely on data imported from other repositories or companies. The challenge is that this approach might not always be useful for WC estimation. In order for WC estimation to benefit from CC datasets, the selection of an optimal "mix" of project datasets and features is required for efficient estimation.

The WC and CC modeling have been considered in the domains of software development effort estimation and defect prediction [2][4]. To the best of our knowledge, this is the first study to employ the use of CC projects in the estimation of STE. A study by Burak et al. [4] indicates that models that used CC datasets for defect prediction yielded similar performance to WC models when normalization and Nearest Neighbor filtering techniques were applied to the datasets.

In this study, we incorporated a *z*-score normalization technique into our proposed project selection approach after it proved superior to log transformation and box cox transformation.

The two research questions addressed by this study are as follows: ***RQ1.*** Do models constructed from CC projects yield similar STE results to models from WC projects with/without normalization? ***RQ2.*** Does STE estimated from "prior" features yield similar results to STE estimated from "posterior" features?

In order to minimize the cost (effort) associated with project and feature selection in STE estimation, this study proposes a multi-objective Mixed-Integer Linear Programming (MILP) optimization framework. The selection of an optimal "mix" of N^* projects with their respective f^* features from m companies to estimate STE is subjected to the following constraints: 1) allocation of projects constraint; 2) allocation of features constraint; 3) size of project constraint.

The rest of the paper is organized as follows: Section II describes the related work. Section III presents the methodology. Section IV describes the results from the study. Finally, Section V presents the conclusion and future work.

II. RELATED WORK

Bareja and Singhal [1] studied various effort estimation techniques in order to minimize STE. They realized that machine learning and data mining testing techniques adopted by developers assist in the reduction of effort expended in the software development process. A study by Turhan et al. [4] investigated how cross-company (CC) projects can be used for defect prediction modeling. They found that within-company (WC) modeling performed better than CC modeling. In the field of effort estimation, Kocaguneli and Menzies [2] in the quest for finding relevant dataset for effort estimation concluded that there is little significant difference in the use of CC or WC datasets for modeling. Ansari et al. [7] proposed a regression test case optimization approach to assist in the reduction of error cost and testing time to achieve a more quality software product. Their test selection approach makes use of prioritized test cases for testing riskier components of the product in order to enhance system reliability and stability.

This study differs from previous works since to the best of our knowledge this is the first study to apply a multi-objective MILP for project and feature selection for STE estimation.

DOI reference number: 10.18293/SEKE2016-017

III. METHODOLOGY

A. Datasets

For the purpose of this study, 45 *PHP* projects from 5 software development companies were used. We present the projects with the means of their respective sampled features - Function Points (FP), Development Duration (DD) in months, Lines of Codes (LOC) and STE in person-months in Table I. Other features from the projects are Number of Development Personnel (DP), Test Cases (TC), Project Cost (PC), number of defects, user stories and marketers. For confidentiality reasons, we denote the names of the companies with $\boldsymbol{X_j}$ and their respective project names as $\boldsymbol{X_{ij}}$.

TABLE I. DESCRIPTIVE SUMMARY OF COMPANY PROJECTS

Company	No. of Projects	Mean			
		FP	DD	LOC	STE
X_1	15	47.2	13.7	14589	151.7
X_2	7	44.4	8.4	3942	25.0
X_3	4	301.0	16.0	719786	123.3
X_4	4	58.8	4.8	14654	18.6
X_5	15	238.0	7.3	65543	25.5

B. Within-Company and Cross-Company Modeling

Given $\{\boldsymbol{X_1, X_2 \ldots X_m}\}$ denoting a set of $\boldsymbol{m}$ companies and each $\boldsymbol{X_j}$ consisting of a set of $\{\boldsymbol{X_{1j}, X_{2j} \ldots X_{mj}}\}$ projects, then we formulate a within-company (WC) project modeling as a combination of selected $\boldsymbol{X_{ij}}$ projects from a single $\boldsymbol{X_j}$ company. Similarly, cross-company (CC) project modeling as a combination of projects from a number of companies. In each case of the modeling, the selected projects by the MILP were used for constructing the predictive model for estimating the STE of a target project.

Based on findings by Turhan et al. [4], we also found the need of preprocessing datasets prior to model construction. We applied a 3-step preprocessing approach. 1) Data Cleaning: In order to assess if the datasets were normally distributed, we made use of a graphical analytical technique namely Quantile-Quantile (QQ) plot. It should be noted that, no missing values were observed in the datasets used. 2) Data Transformation: Because of variations within the datasets, we used the *z*-score normalization to transform the data points into specific range in order to leverage the outweighing discrepancies. 3) Data Reduction: The MILP framework did not select three projects and this was due to the weak correlation between the independent features and the dependent feature (STE).

C. Proposed MILP Algorithm

The MILP optimization framework is formulated for the project and feature selection. Deep Neural Network (DNN) is incorporated into the framework for estimating the STE. A 7-step procedure of the proposed framework is presented below.

Step 1: Preprocess project datasets.

Step 2: Given *m* companies each with an archival set of projects, construct the MILP for the project selection.

Step 3: For every selected optimal set of X_i^* projects from each j^{th} company, compute the multivariate Spearman rank correlation between the respective independent features and the dependent feature (STE).

Step 4: Select the projects based on the project selection optimization method and based on the higher correlation values in a non-increasing order until the optimal number of projects for every j^{th} company is achieved.

Step 5: Combine the selected projects from each j^{th} company. For the given project datasets applied, it was assumed that each project has similar set of features for estimating STE.

Step 6: Select the features based on the feature selection optimization method.

Step 7: Construct both within-company and cross-company STE models in the domain of DNN and evaluate the performance with MdMRE, a robust metric free from outliers [9].

D. Optimization Model Formulation

A generalized optimization model formulation specifically MILP is composed of three main components namely; the objective function, the constraints and decision variable bounds. The optimization model minimizes or maximizes an objective function subject to certain constraints, which can be in the form of equality or inequality constraints. The decision variable (normally independent) is said to be discrete if it can assume a finite set values and continuous if it can assume values within a specified interval. Optimization problems with both discrete and continuous decision variables are said to be MILP. Based on Mridul and Rana [6] categorization of software size into functional and technical size, we constructed the generalized objective function for the MILP incorporating the functional size in the form of FP. Thus, since our ultimate aim is to estimate STE prior to development, we eliminated LOC which is a technical size from developers' perspective and made use of FP from users' perspective together with other prior input features - estimated project duration, estimated project cost and number of development personnel. FP is defined as the sum of five components – external inputs, external outputs, external query, external interface files and internal logical files with their respective weights [6]. Posterior features from the projects include LOC, coding time, test cases and number of defects.

In the formulation of the objective functions, we define a probability weight function, α_{ij} for each i^{th} project from the j^{th} company as shown in (1).

$$\alpha_{ij} = \frac{\left(\sum_{j=1}^{p} f_j\right)_i}{\sum_{i=1}^{n}\left(\sum_{j=1}^{p} f_j\right)_i} \quad (1)$$

where $\boldsymbol{p}$ denotes the total number of $\boldsymbol{f_j}$ features for every project and $\boldsymbol{n}$ denotes the total number of features from all companies. Each of the probability weights is multiplied by the respective prior input features. We incorporated two unique identifiers, $\boldsymbol{\lambda_{ij}}$ to uniquely label every i^{th} project from the j^{th} company and $\boldsymbol{\beta_j}$ to uniquely identify each of the companies as shown in (3) and (4).

Objective Function

In this study, we formulated two main objective functions for the MILP problem – WC and CC objective functions. Each objective function is further categorized for project and feature selection. All objective functions are minimization of project cost and feature selection problems from the respective companies.

A mathematical model based on Parkinson's Law [5] is formulated in (2) for computing the STE for each unsupervised project prior to the setting up of the MILP model. Here, unsupervised project refers to dataset without STE.

$$\text{STE} = \left(\sum_{i=1}^{t} DD \times \sum_{j=1}^{t} DP\right) \times TP\% \quad (2)$$

where DD = Development Duration in months; DP = Development Personnel; TP = Testing Proportion. For

consistency in the computation of the STE for each unsupervised project dataset, we chose a threshold of 40% for TP [1]. DP and DD are parameters denoting the total number of development personnel and duration for requirement gathering, design and testing respectively.

WC Objective Function for Project Selection

We first define the cost function, $Cost^{WCP}$ in relation to project selection for WC in (3) as a cost minimization involving the following design variables – DD, DP, Function Points (FP), Test Cases (TC), Project Cost (PC), Probability Weight Function (α), Project Identification Code (λ)

$$Cost^{WCP} = \alpha_{ij}\left(\sum_t \sum_n DP_n\, DD_t + \sum_u TC_u + \sum_i \sum_j FP_{ij} + \sum_c PC_c\right) + \sum_i \sum_j \lambda_{ij} \quad (3)$$

CC Objective Function for Project Selection

The objective cost function, $Cost^{CCP}$ for the CC project selection is defined in (4).

$$Cost^{CCP} = \alpha_{ij}\left(\left(\sum_t \sum_n DP_n\, DD_t + \sum_u TC_u + \sum_i \sum_j FP_{ij} + \sum_c PC_c\right) + \sum_i \sum_j \lambda_{ij}\right) + \sum_j \beta_j \quad (4)$$

WC Objective Function for Feature Selection

We define an expression for the objective function, $Cost^{WCF}$ for the optimal subset of features for WC in (5). The $Cost^{WCF}$ function incorporates the Akaike Information Criteria (AIC) for the optimal selection of features for the cross modeling process.

$$Cost^{WCF} = \underset{r,\delta,k\epsilon\Re}{\arg}\left\{\min\left\{rlog\left(\hat{\delta}^2\right) + 2k\right\}_f\right\}\lambda_{ij} \quad (5)$$

where r = number of records in the cross projects, $\hat{\delta}^2$= mean squared error, k = number of estimated parameters in AIC. Here, we consider STE as the dependent variable and independent variables are the rest of the features to be selected. We therefore constructed an ordinary least squares regression and used AIC incorporating both forward and backward selection with the aim of selecting the optimal subset of features. AIC is very good at handling much more complex models and achieves a better bias-variance tradeoff [10].

CC Objective Function for Feature Selection

The objective function, $Cost^{CCF}$ for the optimal subset of features for the CC approach is defined in (6).

$$Cost^{CCF} = \underset{r,\delta,k\epsilon\Re}{\arg}\left\{\min\left\{rlog\left(\hat{\delta}^2\right) + 2k\right\}_f\right\}\beta_j \quad (6)$$

Constraints

We considered three constraints namely; allocation of projects, allocation of features and size of projects constraints.

Allocation of Projects Constraint

In order to minimize cost, we considered an inequality constraint for the allocation of projects defined as $\sum_i \sum_j X_{ij} < Y_N$ whereby not all the company projects can be selected. The variable X_{ij} denotes the i^{th} project selected from the j^{th} company. Y_N denotes the total number of N projects from the companies.

Secondly, we considered the inequality constraint, $\sum_i X_i \leq X^*_{.j} < Y_N$ for the selection of projects from within-company. $X^*_{.j}$ denotes the total number of projects from a given j^{th} company. We assumed that at the worst case scenario, all projects can be selected from most of the companies but not all companies.

Allocation of Features Constraint

We define an inequality constraint for the allocation of features in terms of Variance Inflation Factor (VIF) to deal with the multicollinearity issues among the predictor (independent) features. After experimental fine tuning of the parameters, we considered multicollinearity as an issue if VIF is more than a specified threshold, k = 10 in each AIC model. Hence, for optimal predictor features to be obtained we needed correlation between predictor features to be very minimal.

An adjusted R^2 of 0.7 was chosen after parameter fine tuning. Hence, for optimal feature subset, we needed at least 70% of the total variation in STE to be explained by the predictor features.

Lastly, we considered an inequality constraint in the form of $\sum_i \sum_j \sum_p f_{ijp} < f_N$ in order not to select all features from the total projects. Thus, in minimizing cost, we made the assumption that, there exist a subset of features (f_{ijp}) that will equally play a significant role in estimating STE as compared to all features (f_N). f_{ijp} denotes the selected p^{th} feature from the i^{th} project selected from the j^{th} company.

Size of Projects Constraint

We define two inequality constraints, $\sum_i \sum_j FP_{ij} \leq FP_N$ and $\sum_i \sum_j \sum_p FP_{ijp} < FP_N$ for the project sizes in relation to FP [6]. Here, we considered the sum of FP in the selected projects in a given j^{th} company to be at most equal to the total number of FP in all projects from that j^{th} company. On the other hand, the sum of FP from the selected projects was considered to be strictly less than the total FP in all the projects from the selected companies.

E. Deep Neural Networks (DNN) Model

In the estimation of STE for the selected projects using their respective prior and posterior features, we made use of DNN. DNN was considered for the within-company (WC) and cross-company (CC) modeling approach since it makes use of multiple layers to automatically learn from a set of features and gives better predictive results [8]. After series of fine tuning of network architecture parameters, we considered the DNN for the STE estimation to be best at 3 hidden layers with 5, 2 and 1 neuron(s) respectively and an output layer with a single neuron. The Levenberg-Marquardt backpropagation optimization training function was used to update the weights and the hyperbolic tangent activation function was used in each of the neurons for giving the respective outputs. We employed the *k*-fold cross validation approach for setting up the DNN model. The formation of the training set and test set differ slightly for the CC and WC projects. For CC modeling, we used all projects from four of the five companies to form the training set whilst the fifth company project formed the test set. This is repeated in a leave-one-out (LOO) cross validation manner till all projects from the companies were part of the training set and test set respectively. For WC modeling, we used all but one project from a single company to form the training set whilst the remaining project formed the test set using the LOO method similar to the CC modeling. We then used the Median Mean Relative Error (MdMRE) [9] in evaluating the DNN model. Experiment was conducted in MATLAB toolkit (version R2014b).

IV. Results and Discussion

A. Project and Feature Selection by MILP

After applying the MILP to the project datasets, a subset of 42 projects and five features were selected. These features were project duration in months (PD), number of development personnel (DP), number of test cases (TC), number of function points (FP) and the cost of each project (PC) in USD.

RQ1: Do models constructed from CC projects yield similar STE results to models from WC projects with/without normalization?

In order to compare the evaluation performance of STE estimated from both approaches, we used the MdMRE accuracy measure [9] and presented results in Tables II-III. The results in relation to normalized and un-normalized datasets are presented using the win/tie/loss metric. The win/tie/loss metric enabled us to make an evaluation and comparative performance using MdMRE for the within-company (WC) and cross-company (CC) modeling [2]. For example, in Table II, in relation to MdMRE evaluation, we realized that WC and CC modeling with *z*-score normalization yielded similar predictive results in 3 cases (that is 3 ties in the 2nd, 3rd and 5th cases where one project from X_4, X_3 and X_1 were used for testing respectively). Without the normalization technique, we realized that WC dominated in estimating STE (Table III). Thus, results from our optimization selection framework and DNN estimation modeling approach reveal that, models constructed from WC and CC yield approximately similar STE results when datasets were subjected to the *z*-score normalization technique.

TABLE II. MDMRE EVALUATION (NORMALIZED)

CC Train set	WC Train set	Test set	Win	Tie	Loss
$X_{i1}, X_{i2}, X_{i3}, X_{i4}$	$X_{i-1,5}$	LOO	WC		CC
$X_{i1}, X_{i2}, X_{i3}, X_{i5}$	$X_{i-1,4}$	LOO		✓	
$X_{i1}, X_{i2}, X_{i4}, X_{i5}$	$X_{i-1,3}$	LOO		✓	
$X_{i1}, X_{i3}, X_{i4}, X_{i5}$	$X_{i-1,2}$	LOO	WC		CC
$X_{i2}, X_{i3}, X_{i4}, X_{i5}$	$X_{i-1,1}$	LOO		✓	

TABLE III. MDMRE EVALUATION (UN-NORMALIZED)

CC Train set	WC Train set	Test set	Win	Tie	Loss
$X_{i1}, X_{i2}, X_{i3}, X_{i4}$	$X_{i-1,5}$	LOO	WC		CC
$X_{i1}, X_{i2}, X_{i3}, X_{i5}$	$X_{i-1,4}$	LOO		✓	
$X_{i1}, X_{i2}, X_{i4}, X_{i5}$	$X_{i-1,3}$	LOO	WC		CC
$X_{i1}, X_{i3}, X_{i4}, X_{i5}$	$X_{i-1,2}$	LOO	WC		CC
$X_{i2}, X_{i3}, X_{i4}, X_{i5}$	$X_{i-1,1}$	LOO		✓	

LOO – Leave one out; WC – within company; CC – cross company

RQ2: Does STE estimated from "prior" features yield similar results to STE estimated from "posterior" features?

Here, we compared performance of STE estimation from both "prior" and "posterior" features with respect to WC and CC modeling. Due to space limitation, the MdMRE evaluation of results are presented in Tables IV-V available in the link[1]. In relation to cross-company (CC) modeling approach, the average MdMRE value was 79.7%. In relation to within-company (WC), the average MdMRE value was 82.0%. Result from Table IV shows that, in relation to the application of the normalization technique, STE estimated from CC modeling yielded similar results for both prior and posterior features. We further confirmed this result using Friedman's test statistic [3] which yielded *p*-values of 0.4240 and 0.0597 at 5% significance level for the prior and posterior features respectively. This indicates that there is no statistical difference in the STE estimation results from the prior and posterior features. In the WC modeling, prior features dominated best in the STE estimation as illustrated in Table V.

This means that, our proposed multi-objective MILP optimization selection framework can select optimal prior features which yield similar STE results as compared to posterior features provided a *z*-score normalization is applied.

V. CONCLUSION AND FUTURE WORK

A Mixed-Integer Linear Programming (MILP) optimization framework has been proposed for the selection of desirable number of projects with their respective features from within-company and cross-company projects. The five input features selected by the MILP for Software Testing Effort (STE) estimation prior to development are Project Duration, Development Personnel, Test Cases, Function Points and Project Cost. We subjected the selected projects and features to train a Deep Neural Network (DNN) model for estimating STE using the *k*-fold cross validation approach. Results show that the DNN model for estimating STE from cross-company projects yielded similar results to within-company projects provided that the *z*-score normalization method was applied.

Going forward, we intend to incorporate a filter in our MILP framework to further improve the CC project and feature selection approach for STE estimation.

ACKNOWLEDGEMENT

This research is supported by the City University of Hong Kong research funds (Project No. 7200354, 7004222, 7004474).

REFERENCES

[1] K. Bareja and A. Singhal. "A Review of Estimation Techniques to Reduce Testing Efforts in Software Development." *Advanced Computing & Communication Technologies (ACCT), 2015 Fifth International Conference on*. IEEE, 2015.

[2] E. Kocaguneli and T. Menzies. "How to find relevant data for effort estimation?." *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011.

[3] M. Friedman. "The use of ranks to avoid the assumption of normality implicit in the analysis of variance." *Journal of the american statistical association* 32.200 (1937): 675-701.

[4] B. Turhan, T. Menzies, A. B. Bener and J. D. Stefano. "On the relative value of cross-company and within-company data for defect prediction." *Empirical Software Engineering* 14.5 (2009): 540-578.

[5] C. N. Parkinson. *Parkinson's law, and other studies in administration*. Vol. 24. Boston: Houghton Mifflin, 1957.

[6] B. Mridul and A. Rana. "Impact of Size and Productivity on Testing and Rework Efforts for Web-based Development Projects." ACM SIGSOFT Software Engineering Notes 40.2 (2015): 1-4.

[7] A. S. A. Ansari, K. K. Devadkar and P. Gharpure. "Optimization of test suite-test case in regression test." *Computational Intelligence and Computing Research (ICCIC), 2013 IEEE International Conference on*. IEEE, 2013.

[8] K. Sangwook, M. Lee and J. Shen. "A novel deep learning by combining discriminative model with generative model." *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015.

[9] T. Foss, E. Stensrud, B. Kitchenham and I. Myrtveit. "A simulation study of the model evaluation criterion MMRE." Software Engineering, IEEE Transactions on 29.11 (2003): 985-995.

[10] D. Ruppert. *Statistics and finance: An introduction*. Springer Science & Business Media, 2004.

[1] MdMRE evaluation results - http://tinyurl.com/WinTieLoss

CSCD618: Deep Learning & Neural Networks

Session 1 – Fundamentals of Deep Learning

By
Solomon Mensah (PhD)

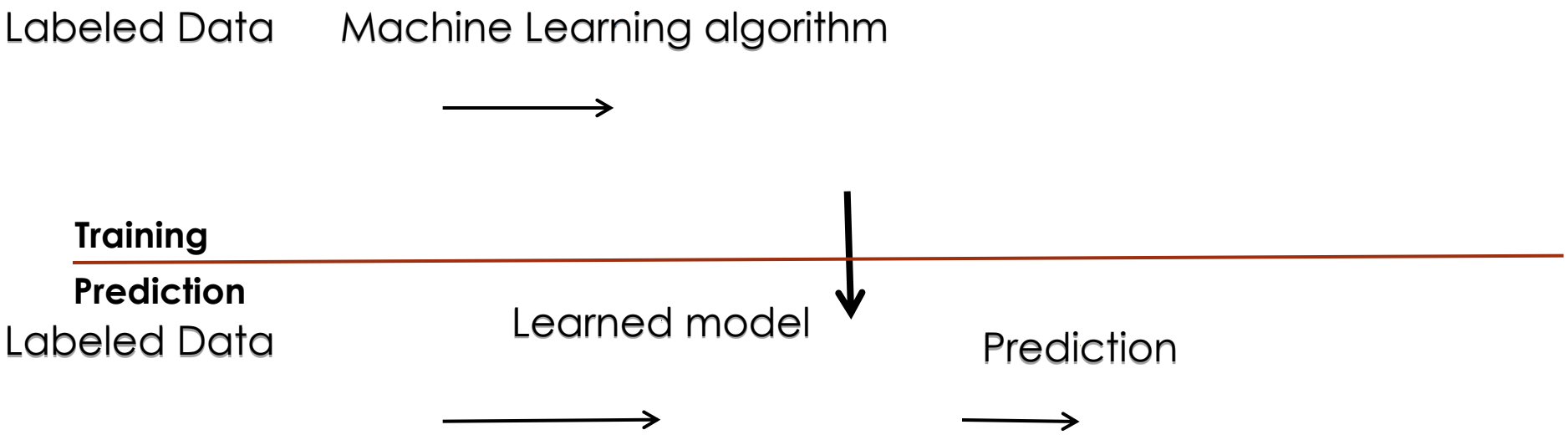
UNIVERSITY OF GHANA

Outline

- Machine Learning basics
- Types of Learning
- Introduction to Deep Learning
  - what is Deep Learning
  - why is it useful
- Deep learning vs ML
- Main components/hyper-parameters:
  - activation functions
  - overfitting vs underfitting
  - optimizers, cost functions and training
  - tuning
  - classification vs. regression tasks
- Applications

Machine Learning Basics

Machine learning is a field of computer science that gives computers the ability to **learn without being explicitly programmed**



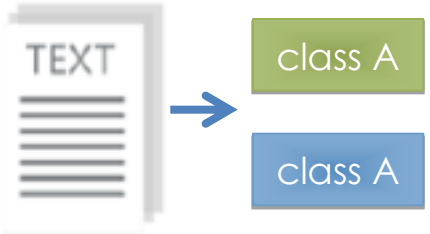
Methods that can learn from and make predictions on data

Types of Learning

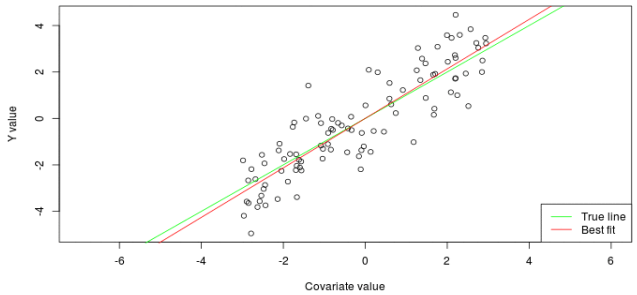
Supervised: Learning with a **labeled training** set
Example: email *classification* with already labeled emails

Unsupervised: Discover **patterns** in **unlabeled** data
Example: *cluster* similar documents based on text

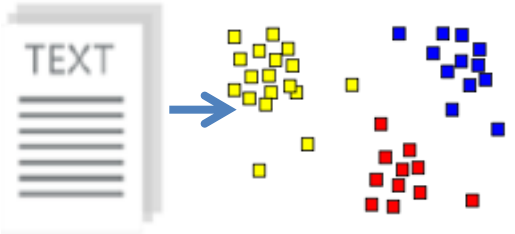
Reinforcement learning: learn to **act** based on **feedback/reward**
Example: learn to play Go, reward: *win or lose*



Classification



Regression

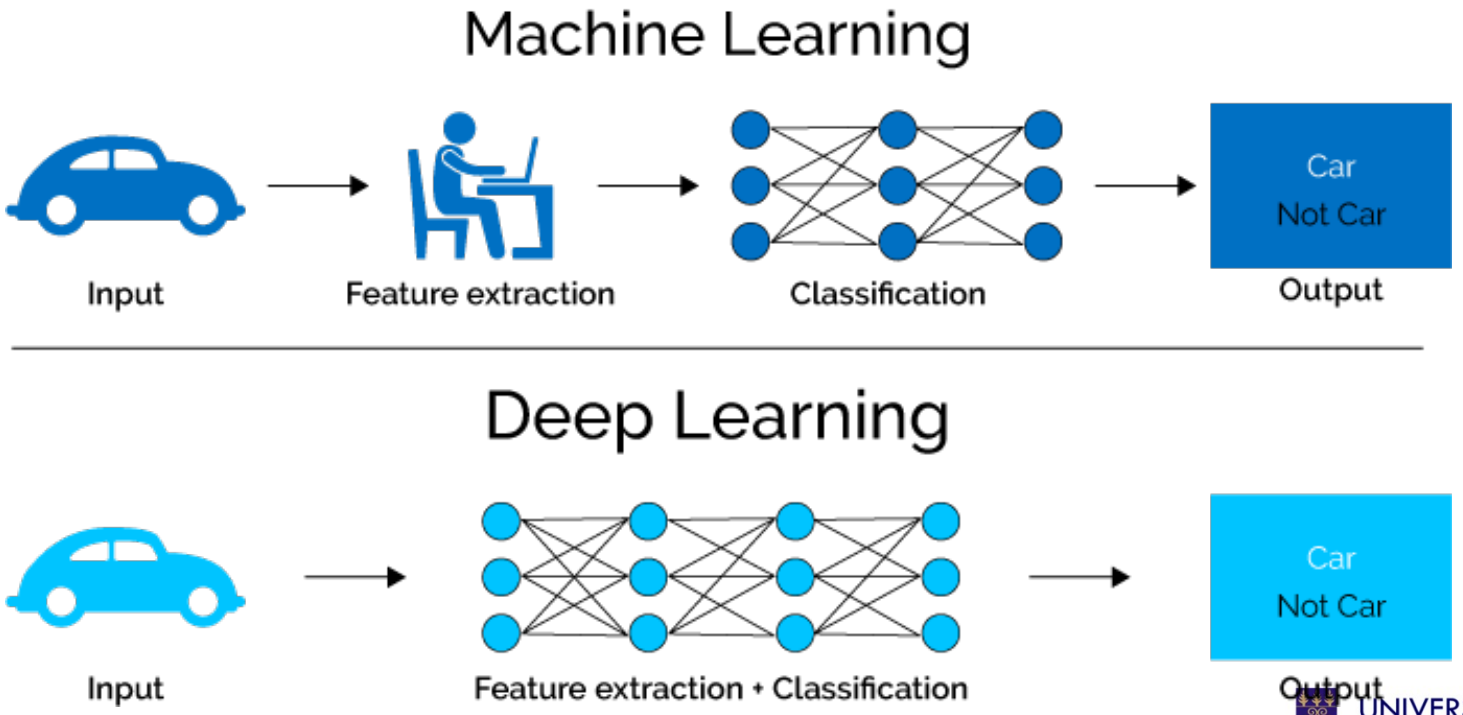


Clustering

Anomaly Detection
Sequence labeling
…

What is Deep Learning (DL)?

A machine learning subfield of learning **representations** of data.
Exceptional effective at **learning patterns**.
Deep learning algorithms attempt to learn (multiple levels of) representation by using a **hierarchy of multiple layers**
If you provide the system **tons of information**, it begins to understand it and respond in useful ways.

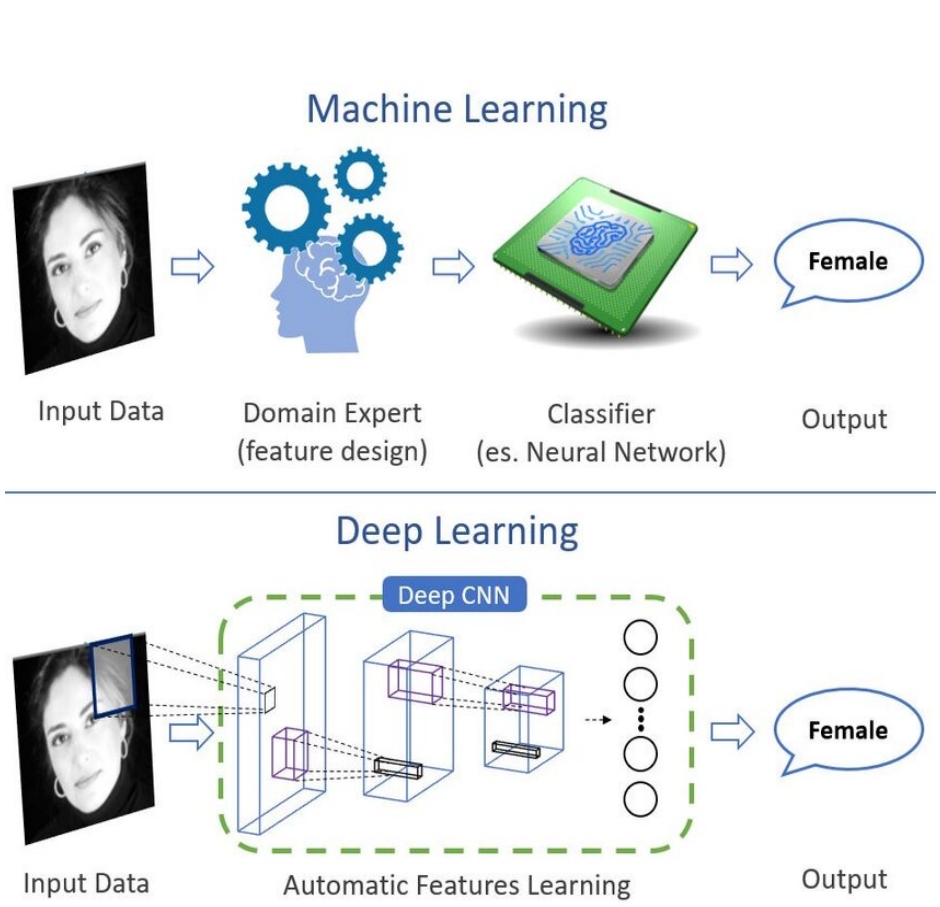


Why is DL useful?

- Manually designed features are often **over-specified**, **incomplete** and take a **long time to design** and validate
- Learned Features are **easy to adapt**, **fast** to learn
- Deep learning provides a very **flexible**, (almost?) **universal**, learnable framework for representing world, visual and linguistic information.
- Can learn both unsupervised and supervised
- Effective **end-to-end** joint system learning
- Utilize large amounts of training data

In ~2010 DL started outperforming
other ML techniques
first in speech and vision, then NLP

ML vs. Deep Learning



Machine learning

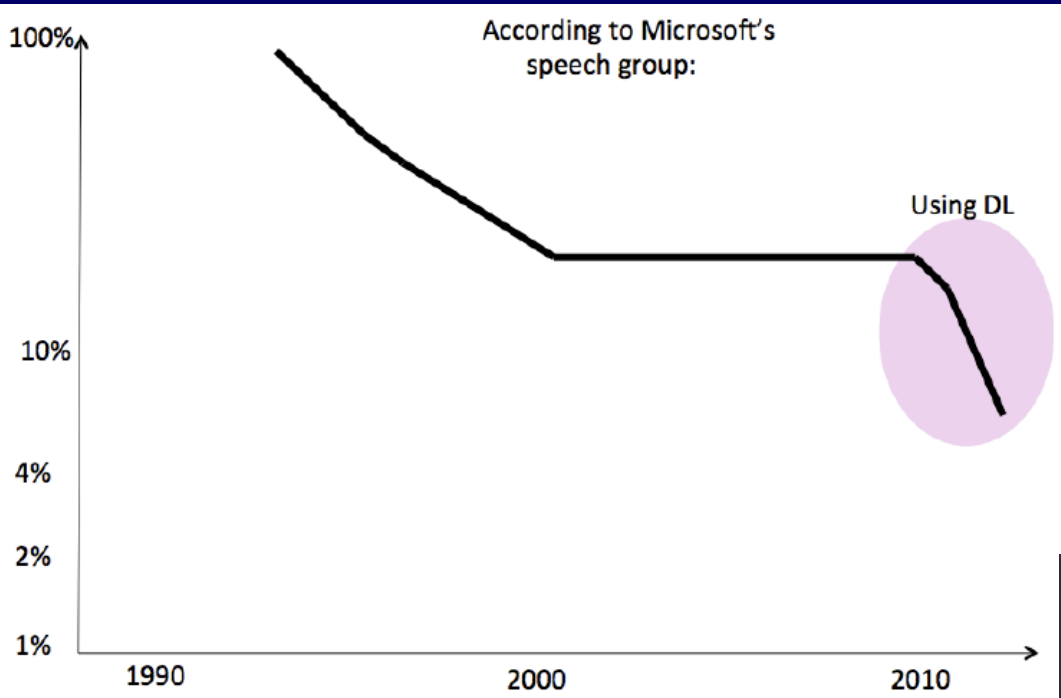
Uses algorithms and learns on its own but may need human intervention to correct errors

Deep learning

Uses advanced computing, its own neural network, to adapt with little to no human intervention

Machine Learning	Deep Learning
Quick to train a model	Intensive computation
Accuracy plateaus	Unlimited accuracy
Good results with small data sets	Requires Millions of data points
Trains on CPU	Trains better on GPU
Try different features to achieve best results	Learns features automatically

State of the art in …



Deep Learning in Speech Recognition

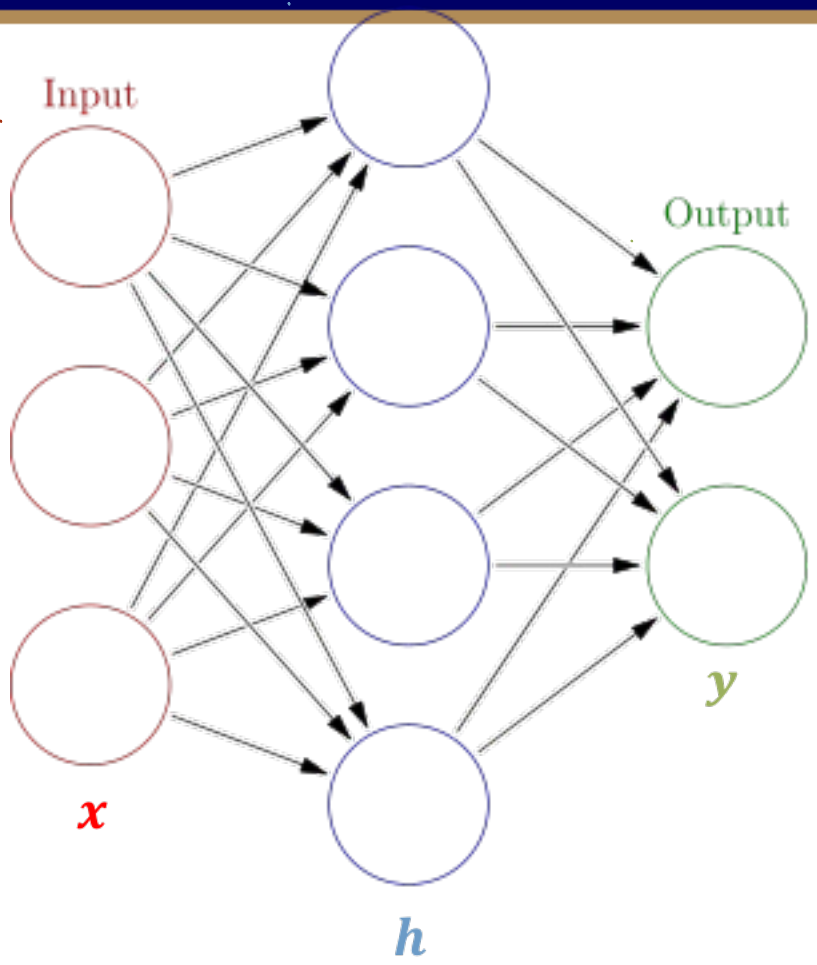
Several big improvements in recent years in NLP

- ✓ Machine Translation
- ✓ Sentiment Analysis
- ✓ Dialogue Agents
- ✓ ChatGPT
- ✓ Text Classification …

Leverage different levels of representation

- words & characters
- syntax & semantics

Neural Network Intro



Weights

$$h = \sigma(W_1 x + b_1)$$

$$y = \sigma(W_2 h + b_2)$$

Activation functions

How do we train?

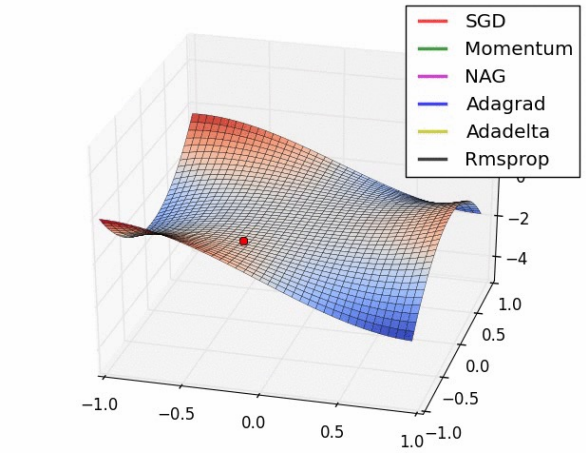
4 + 2 = 6 neurons (not counting inputs)
[3 x 4] + [4 x 2] = 20 weights
4 + 2 = 6 biases

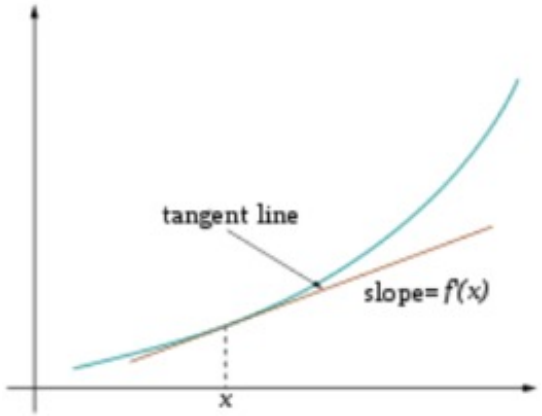
26 learnable **parameters**

Training

Sample labeled data (**batch**) → **Forward** it through the network, get predictions → **Back-propagate** the errors → **Update** the network weights

Optimize (min. or max.) **objective/cost function**
Generate **error signal** that measures difference between predictions and target values



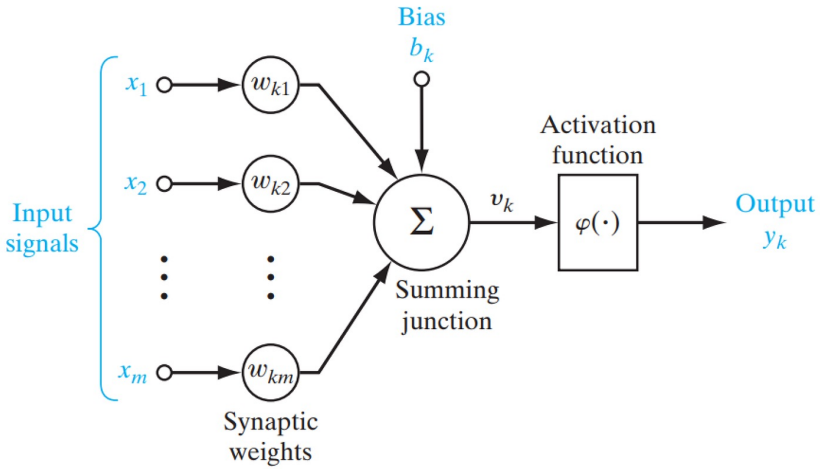


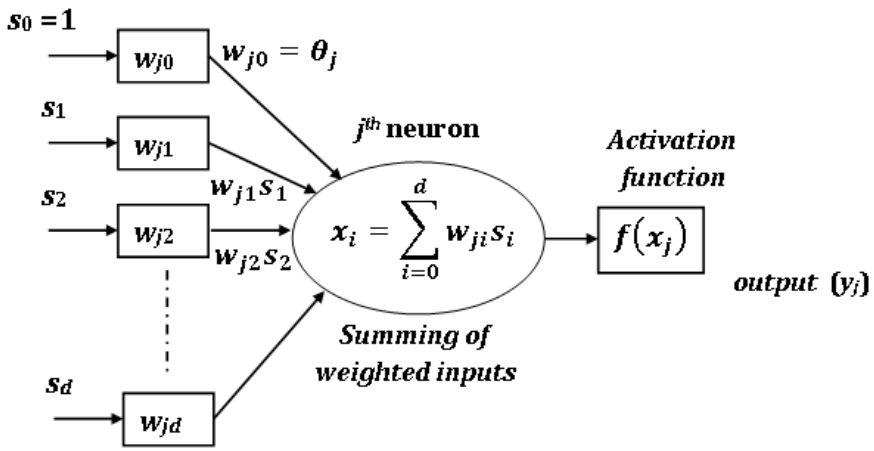
Use error signal to change the **weights** and get more accurate predictions
Subtracting a fraction of the **gradient** moves you towards the **(local) minimum of the cost function**

https://medium.com/@ramrajchandradevan/the-evolution-of-gradient-descend-optimization-algorithm-4106a6702d39

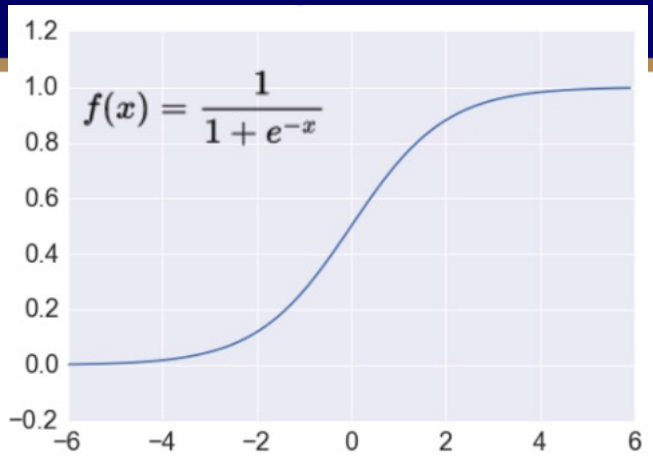
Activation functions

- The main purpose of an activation function is to transform the summed weighted input from a node into an output value that is passed on to the next hidden layer or used as the final output.
- An artificial neuron calculates the 'weighted sum' of its inputs and adds a bias, as shown in the figure below by the net input.





Activation: Sigmoid



http://adilmoujahid.com/images/activation.png

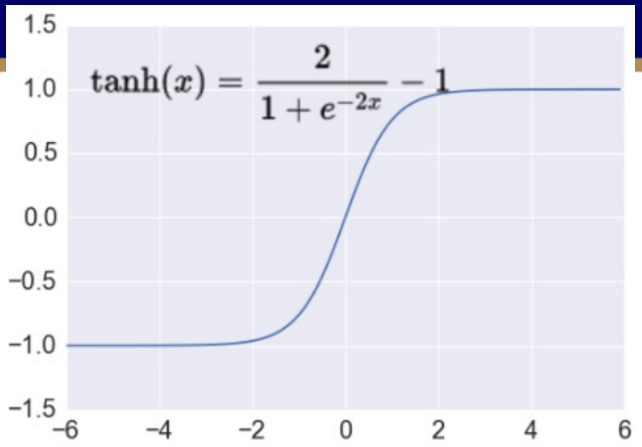
Takes a real-valued number and "squashes" it into range between 0 and 1.

$$R^n \rightarrow [0,1]$$

\+ Nice interpretation as the **firing rate** of a neuron
- 0 = not firing at all
- 1 = fully firing

\- Sigmoid neurons **saturate** and **kill gradients**, thus NN will barely learn
- when the neuron's activation are 0 or 1 (saturate)
  - gradient at these regions almost zero
  - almost no signal will flow to its weights
  - if initial weights are too large then most neurons would saturate

Activation: Tanh



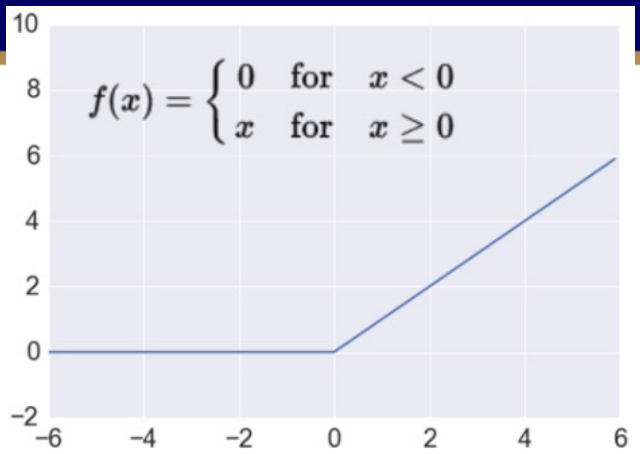
http://adilmoujahid.com/images/activation.png

Takes a real-valued number and "squashes" it into range between -1 and 1.

$$R^n \rightarrow [-1,1]$$

- Like sigmoid, tanh neurons **saturate**
- Unlike sigmoid, output is **zero-centered**
- Tanh is a **scaled sigmoid**:

Activation: ReLU



http://adilmoujahid.com/images/activation.png

Takes a real-valued number and thresholds it at zero

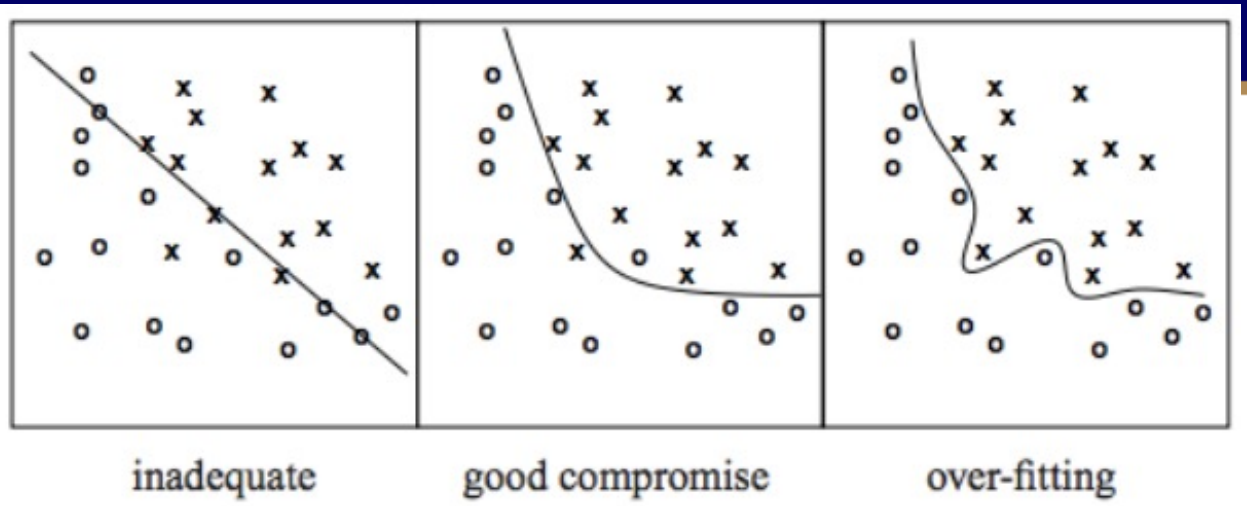
$$f(x) = \max(0, x)$$

$$R^n \rightarrow R^n_+$$

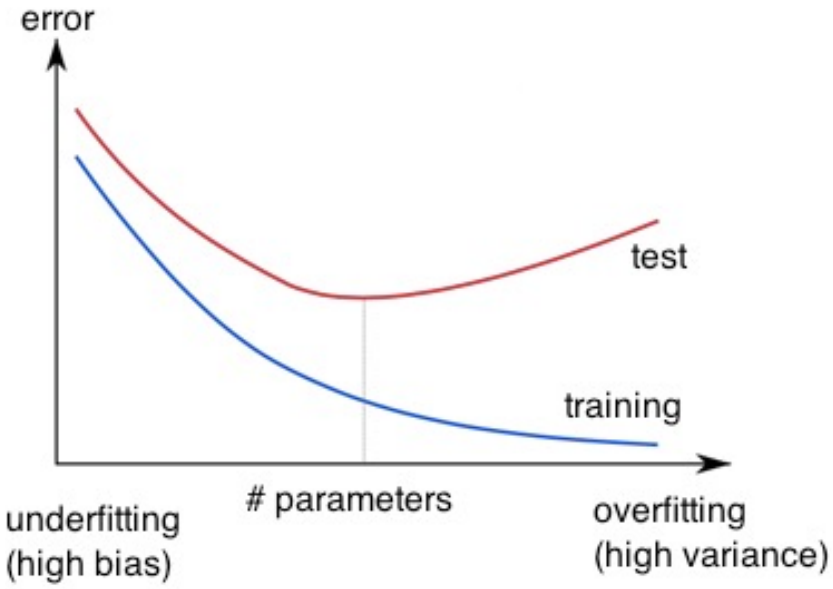
Most Deep Networks use ReLU

- Trains much **faster**
  - accelerates the convergence of SGD
  - due to linear, non-saturating form
- Less expensive operations
  - compared to sigmoid/tanh (exponentials etc.)
  - implemented by simply thresholding a matrix at zero
- More **expressive**
- Prevents the **gradient vanishing problem**

Overfitting



http://wiki.bethanycrane.com/overfitting-of-data



https://www.neuraldesigner.com/images/learning/selection_error.svg

Learned hypothesis may **fit** the training data very well, even outliers (**noise**) but fail to **generalize** to new examples (test data)

Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains sampling error.
  - There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model to the training set it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity. If the model is very flexible it can model the sampling error really well.
- If you fitted the model to another training set drawn from the same distribution over cases, it would make different predictions on the test data. This is called “variance”.

Preventing overfitting

- Approach 1: Get more data!
  - Almost always the best bet if data is cheap and you have enough compute power to train on more data.
- Approach 2: Use a model that has the right capacity:
  - enough to fit the true regularities.
  - not enough to also fit spurious regularities (if they are weaker).
- Approach 3: Average many different models.
  - Use models with different forms.
  - Or train the model on different subsets of the training data (this is called "bagging").
- Approach 4: (Bayesian) Use a single neural network architecture, but average the predictions made by many different weight vectors.

Some ways to limit the capacity of a neural net

- The capacity can be controlled in many ways:
  - Architecture: Limit the number of hidden layers and the number of units per layer.
  - Early stopping: Start with small weights and stop the learning before it overfits.
  - Weight-decay: Penalize large weights using penalties or constraints on their squared values (L2 penalty) or absolute values (L1 penalty).
  - Noise: Add noise to the weights or the activities.
- Typically, a combination of several of these methods is used.

How to choose meta parameters that control capacity
(like the number of hidden units or the size of the weight penalty)

- The wrong method is to try lots of alternatives and see which gives the best performance on the test set.
  - This is easy to do, but it gives a false impression of how well the method works.
  - The settings that work best on the test set are unlikely to work as well on a new test set drawn from the same distribution.

- An extreme example: Suppose the test set has random answers that do not depend on the input.
  - The best architecture will do better than chance on the test set.
  - But it cannot be expected to do better than chance on a new test set.

Cross-validation: A better way to choose meta parameters

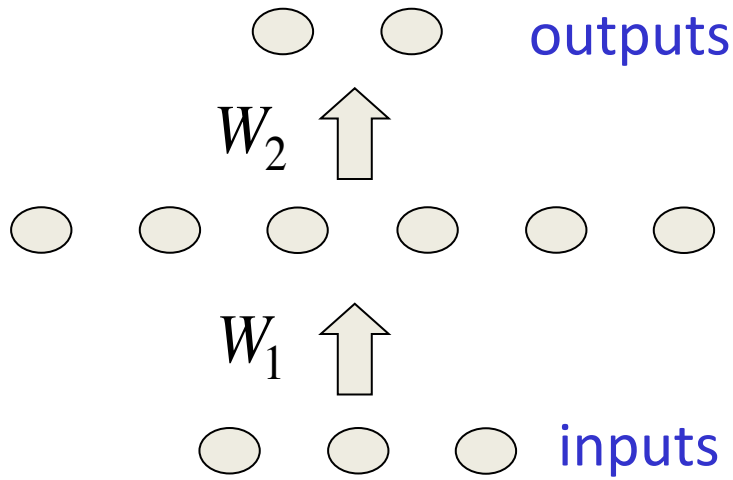
- Divide the total dataset into three subsets:
  - Training data is used for learning the parameters of the model.
  - Validation data is not used for learning but is used for deciding what settings of the meta parameters work best.
  - Test data is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could divide the total dataset into one final test set and N other subsets and train on all but one of those subsets to get N different estimates of the validation error rate.
  - This is called N-fold cross-validation.
  - The N estimates are not independent.

Preventing overfitting by early stopping

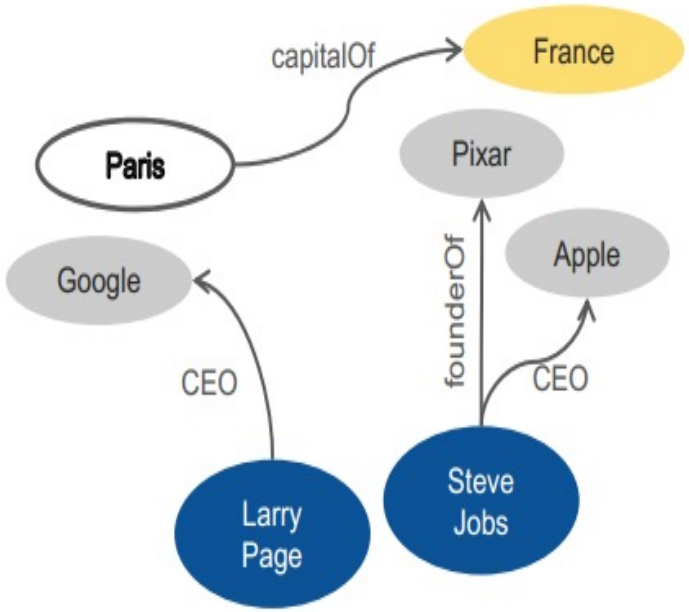
- If we have lots of data and a big model, its very expensive to keep re-training it with different sized penalties on the weights or different architectures.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse.
  - But it can be hard to decide when performance is getting worse.
- The capacity of the model is limited because the weights have not had time to grow big.
  - Smaller weights give the network less capacity. Why?

Why early stopping works

- When the weights are very small, every hidden unit is in its linear range.
  - So even with a large layer of hidden units it's a linear model.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



Application Example: Relation Extraction from text



Google CEO Larry Page announced that...

Steve Jobs has been Apple for a while...

Pixar lost its co-founder Steve Jobs...

I went to Paris, France for the summer...

http://www.mathcs.emory.edu/~dsavenk/slides/relation_extraction/img/distant.png

Useful for:

- knowledge base completion
- social media analysis
- question answering
- …

Applications of Deep learning

- Computer Vision
- Natural Language Processing (NLP)
- Speech Recognition and Synthesis
- Healthcare
- Finance and Trading
- Autonomous Vehicles
- Recommendation Systems
- Gaming and Entertainment
- Manufacturing and Industry
- Environmental Monitoring

Thank you

CSCD618: Deep Learning & Neural Networks

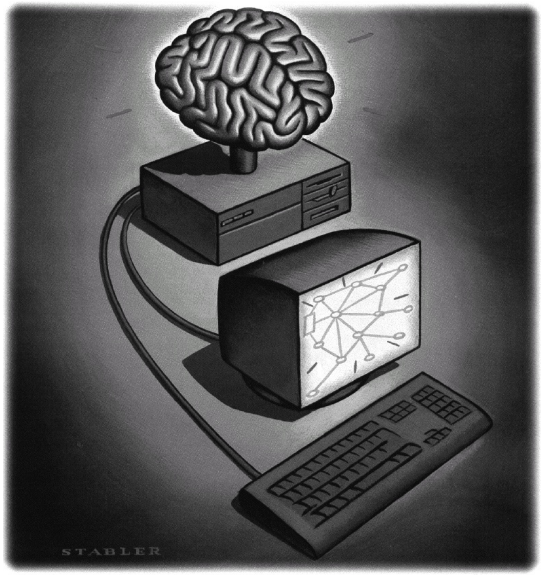
Session 2 – Artificial Neural Networks

By
Solomon Mensah (PhD)

UNIVERSITY OF GHANA

What is a Neural Network?

•Biologically motivated approach to machine learning



Similarity with biological network

Fundamental processing elements of a neural network is a neuron

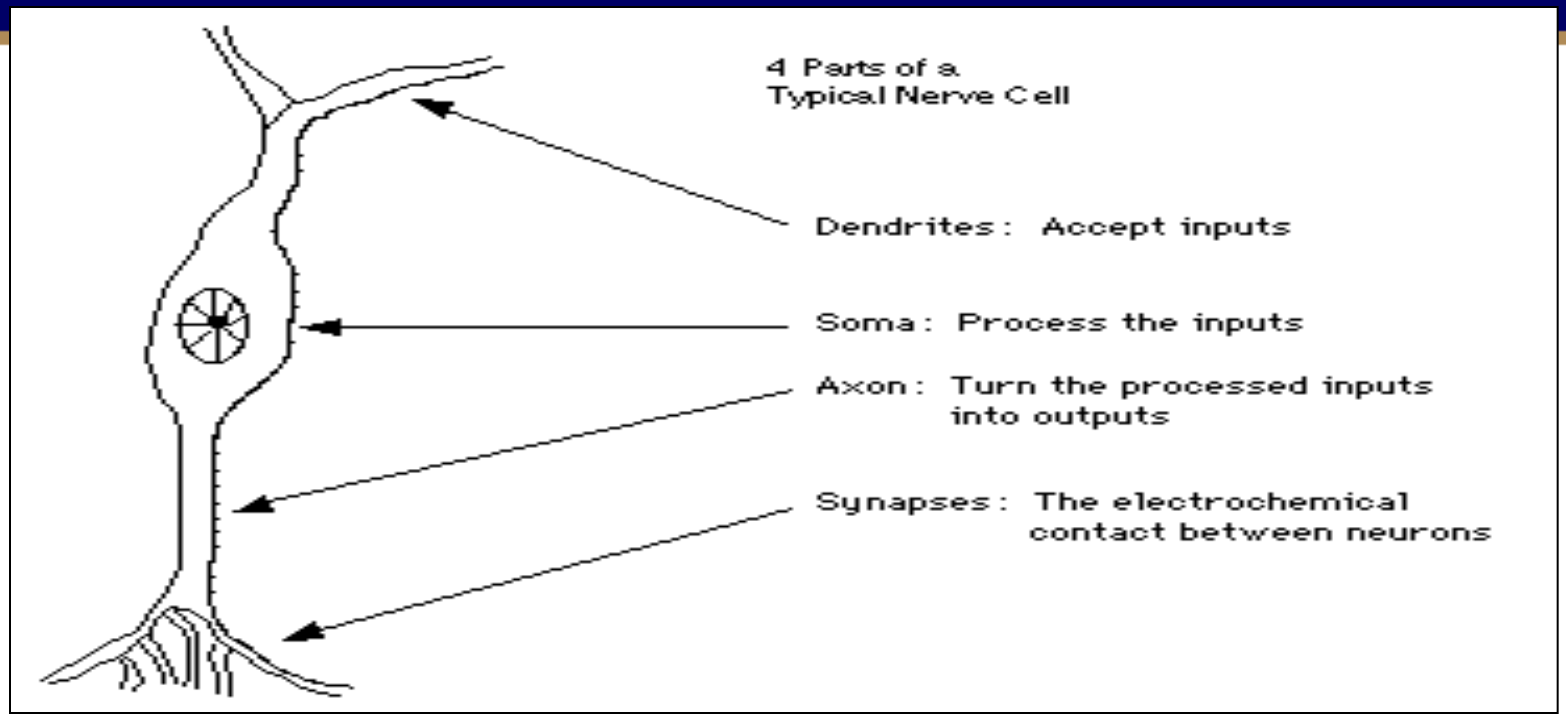
1.Receives inputs from other source

2.Combines them in someway

3.Performs a generally nonlinear operation on the result

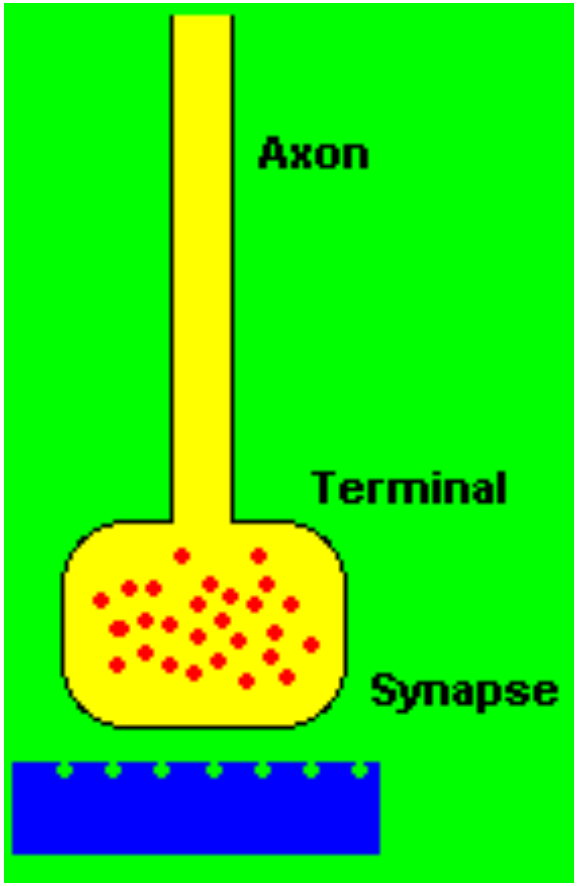
4.Outputs the final result

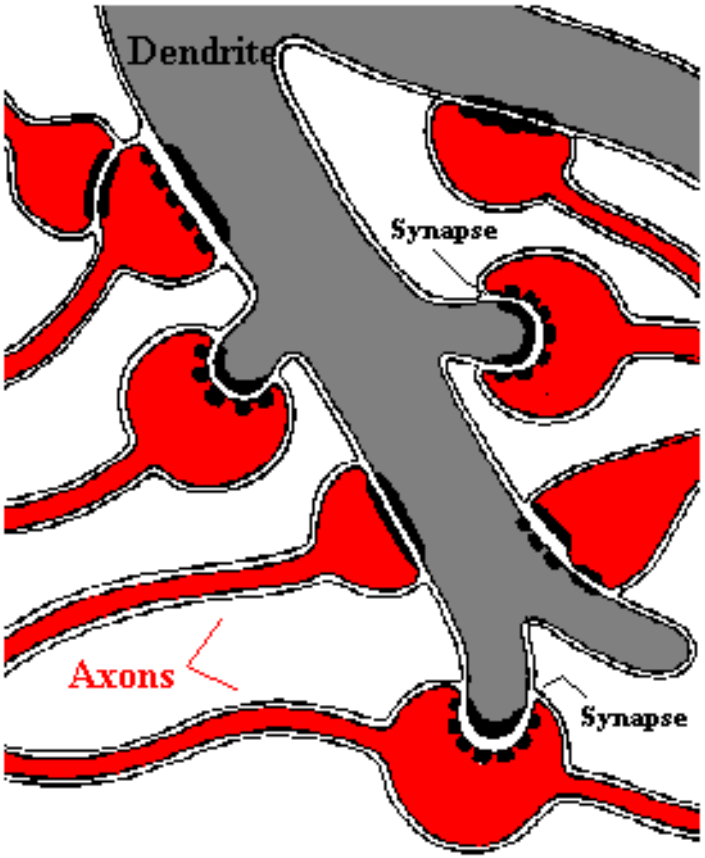
Similarity with Biological Network



- Fundamental processing element of a neural network is a neuron
- A human brain has 100 billion neurons
- An ant brain has 250,000 neurons

Synapses, the basis of learning and memory





Neural Network

- Neural Network is a set of connected INPUT/OUTPUT UNITS, where each connection has a WEIGHT associated with it.

- Neural Network learning is also called CONNECTIONIST learning due to the connections between units.

Neural Network

- Neural Network learns by adjusting the weights so as to be able to correctly classify the training data and hence, after testing phase, to classify unknown data.
- Neural Network needs long time for training.
- Neural Network has a high tolerance to noisy and incomplete data

Neural Network Classifier

- **Input: Classification data**

  It contains classification attribute
- Data is divided, as in any classification problem.

  [Training data and Testing data]

- **All data must be normalized.**

  (i.e. all values of attributes in the database are changed to contain values in the internal [0,1] or[-1,1])

  Neural Network can work with data in the range of (0,1) or (-1,1)

- **Two basic normalization techniques**

  [1] Max-Min normalization

  [2] Decimal Scaling normalization

Data Normalization

[1] Max-Min normalization formula is as follows:

$$v' = \frac{v - \min A}{\max A - \min A}(new\_\max A - new\_\min A) + new\_\min A$$

**[minA, maxA , the minimun and maximum values of the attribute A
max-min normalization maps a value v of A to v' in the range {new_minA, new_maxA}]**

Example of Max-Min Normalization

Max- Min normalization formula

$$v' = \frac{v - \min A}{\max A - \min A}(new\_\max A - new\_\min A) + new\_\min A$$

Example: **We want to normalize data to range of the interval [0,1].**
We put: **new_max A= 1, new_minA =0.**

Say, max A was 100 and min A was 20 (That means maximum and minimum values for the attribute).

Now, if v = 40 (If for this particular pattern , attribute value is 40), v' will be calculated as , v' = (40-20) x (1-0) / (100-20) + 0

=> v' = 20 x 1/80

=> v' = 0.4

Decimal Scaling Normalization

[2]Decimal Scaling Normalization

Normalization by decimal scaling normalizes by moving the decimal point of values of attribute A.

$$v' = \frac{v}{10^j}$$

Here j is the smallest integer such that max|v'|<1.

Example :

A – values range from -986 to 917. Max |v| = 986.

v = -986 normalize to v' = -986/1000 = -0.986

One Neuron as a Network

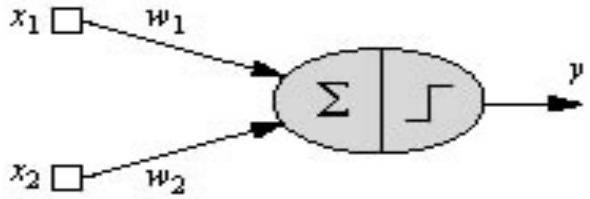


Fig1: an artificial neuron

- Here x1 and x2 are normalized attribute value of data.
- y is the output of the neuron , i.e the class label.
- x1 and x2 values multiplied by weight values w1 and w2 are input to the neuron x.
- Value of x1 is multiplied by a weight w1 and values of x2 is multiplied by a weight w2.
- Given that
  - w1 = 0.5 and w2 = 0.5
  - Say value of x1 is 0.3 and value of x2 is 0.8,
  - So, weighted sum is :
  - sum= w1 x x1 + w2 x x2 = 0.5 x 0.3 + 0.5 x 0.8 = 0.55

.

One Neuron as a Network

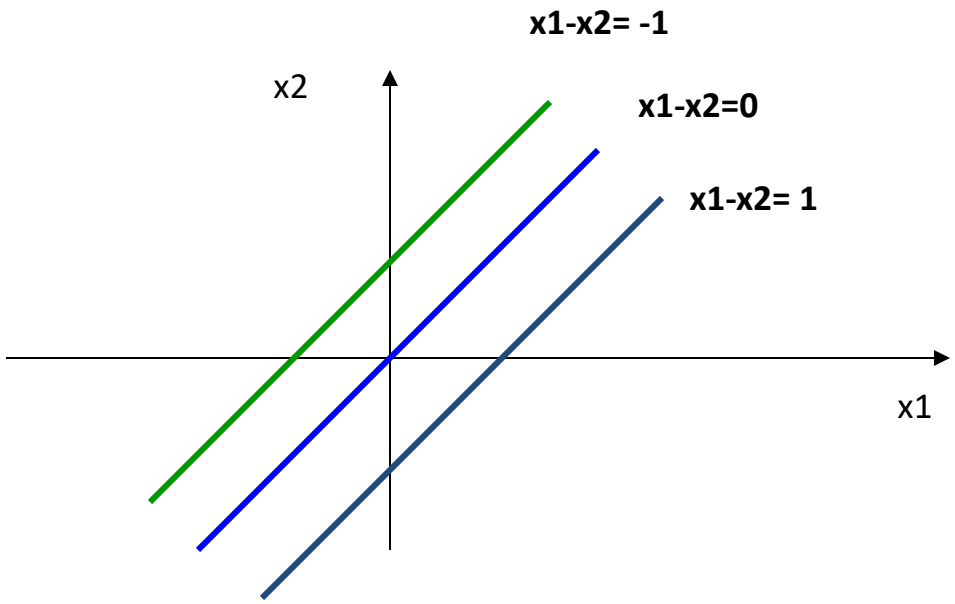
- **The neuron receives the weighted sum as input and calculates the output as a function of input as follows :**
- **y = f(x) , where f(x) is defined as**
- **f(x) = 0 { when x< 0.5 }**
- **f(x) = 1 { when x >= 0.5 }**
- **For our example, x (weighted sum) is 0.55, so y = 1 ,**
- That means corresponding input attribute values are classified in class 1.
- **If for another input values , x = 0.45 , then f(x) = 0,**
- **so we could conclude that input values are classified to class 0.**

.

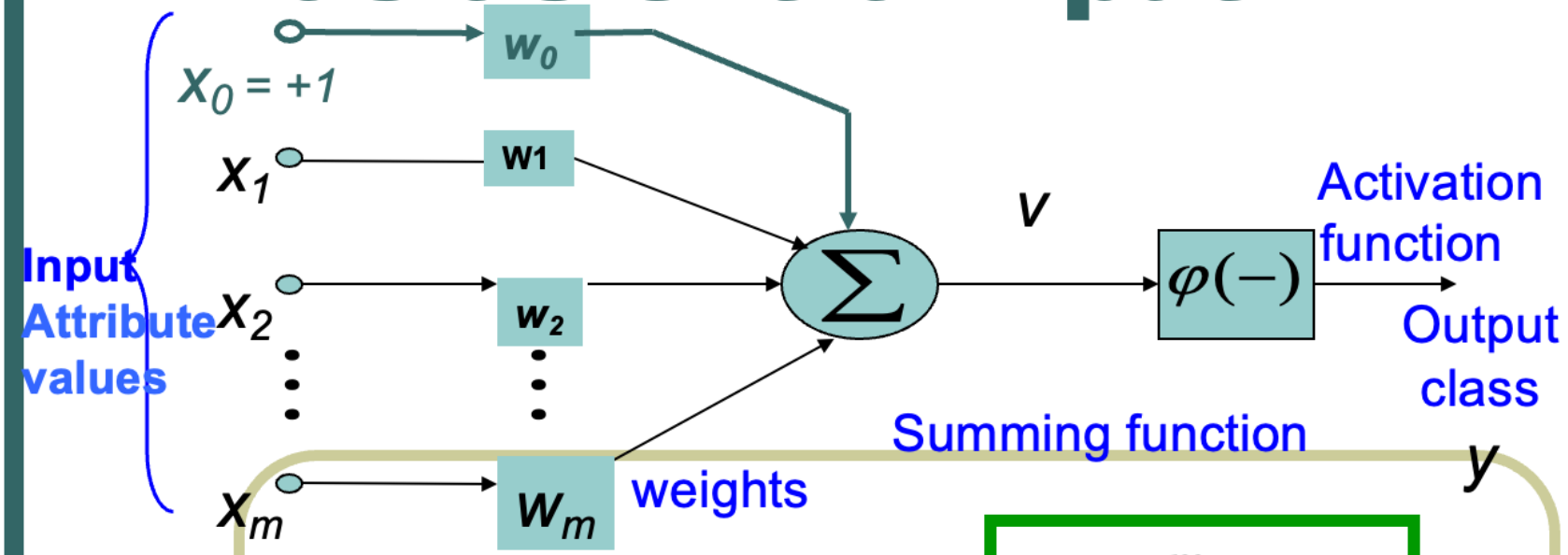
Bias of a Neuron

- We need the bias value to be added to the weighted sum ∑wixi so that we can transform it from the origin.

v = ∑wixi + b, here b is the bias



Bias as extra input



$$v = \sum_{j=0}^{m} w_j x_j$$

$$w0 = b$$

Neuron with Activation

- The neuron is the basic information processing unit of a NN. It consists of:

1 A set of links, describing the neuron inputs, with weights W_1, W_2, …, W_m

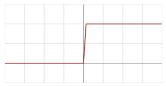
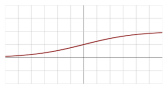
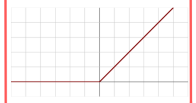
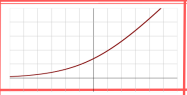
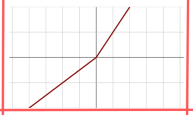
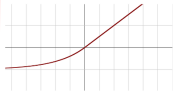
2. An adder function (linear combiner) for computing the weighted sum of the inputs (real numbers):

$$u = \sum_{j=1}^{m} w_j x_j$$

3 Activation function : for limiting the amplitude of the neuron output.

$$y = \varphi(u + b)$$

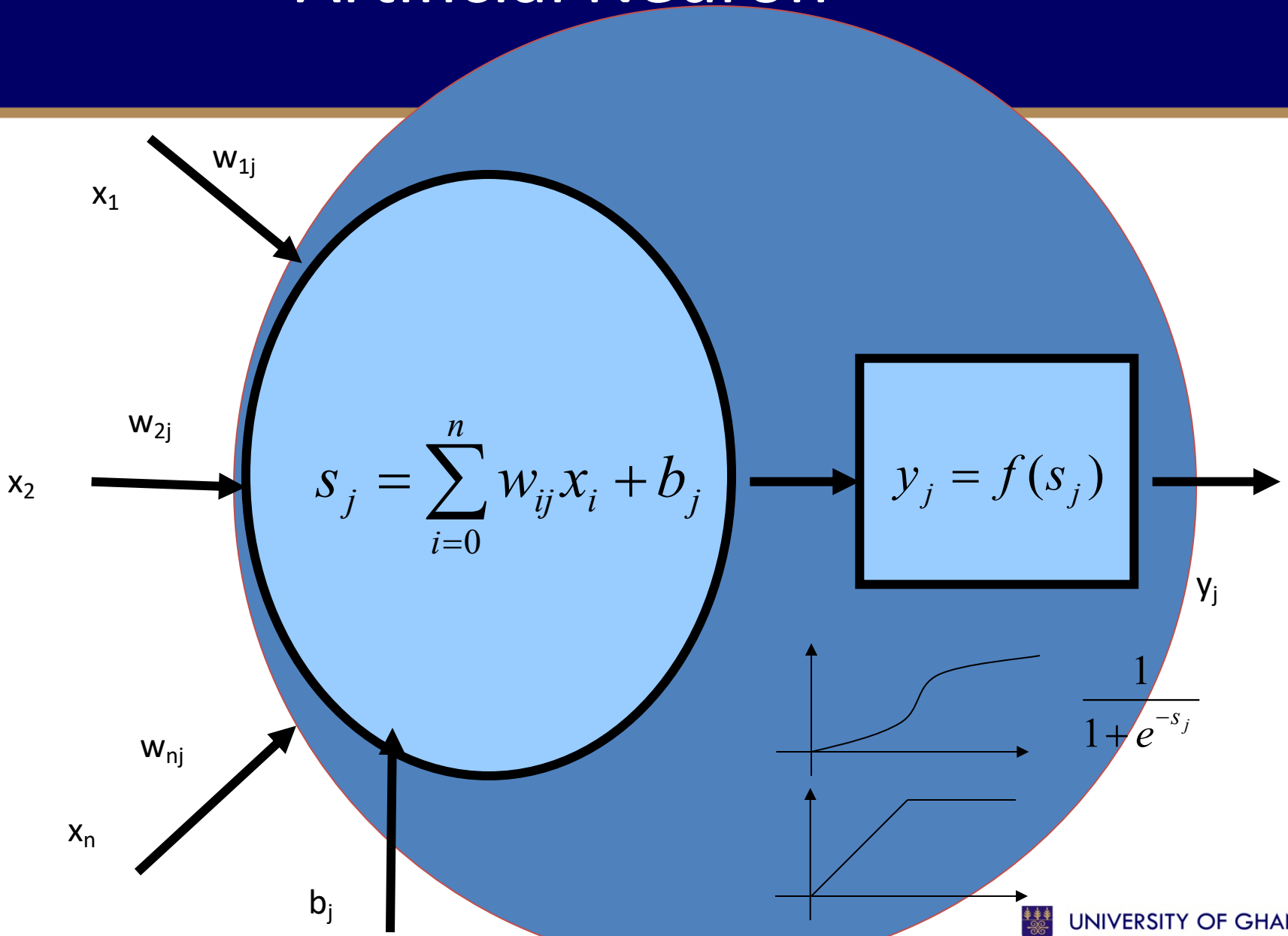
Activation Functions

ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		f(x) = x	f'(x) = 1	(-∞, ∞)
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	{0, 1}
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$	(0, 1)
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$f'(x) = 1 - f(x)^2$	(-1, 1)
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x<0 \\ x & \text{if } x\geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	[0, ∞)
Softplus		$f(x) = \ln(1+e^{x})$	$f'(x) = \frac{1}{1+e^{-x}}$	(0, 1)
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	(-1, 1)
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^{x}-1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^{x} & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	[0, ∞)

Artificial Neural Networks

- An artificial network consists of a pool of simple processing units which communicate by sending signals to each other over a large number of weighted connections.

Artificial Neuron

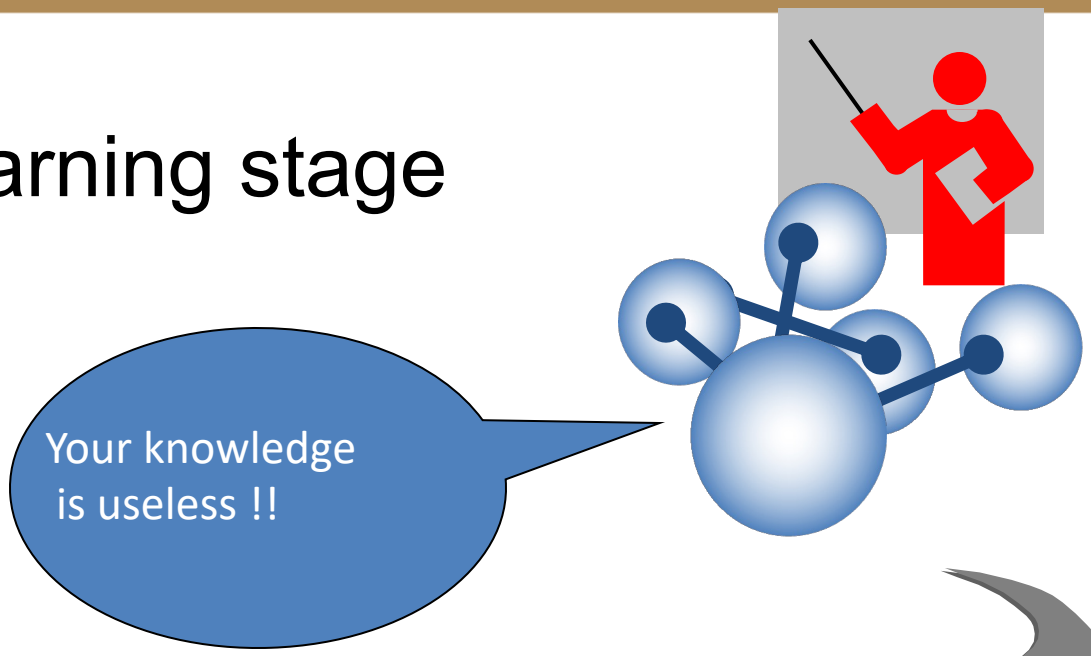


Artificial Neural Networks

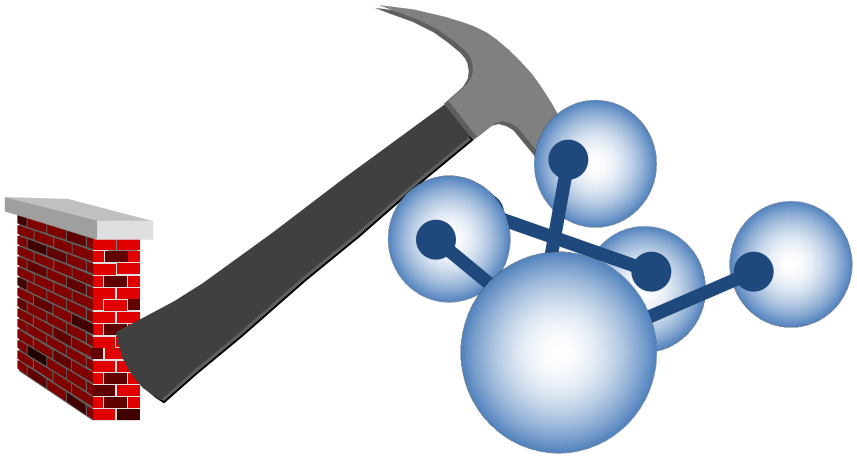
- Each unit performs a relatively simple job: receive input from neighbors or external sources and use this to compute an output signal which is propagated to other units (**Test stage**).
- Apart from this processing, there is the task of the adjustment of the weights (**Learning stage**).
- The system is inherently parallel in the sense that many units can carry out their computations at the same time.

Artificial Neural Networks

1. Learning stage



2. Test stage
(working stage)



Classification (connections)

As for this pattern of connections, the main distinction we can make is between:

- **Feed-forward networks**, where the data flow from input to output units is strictly feed-forward. The data processing can extend over multiple layers of units, but no feedback connections or connections between units of the same layer are present.

Classification (connections)

- **Recurrent networks** that do contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which these activations do not change anymore.

Recurrent Networks

- In other applications, the change of the activation values of the output neurons are significant, such that the dynamical behavior constitutes the output of the network.

Classification (Learning)

We can categorise the learning situations in two distinct sorts. These are:

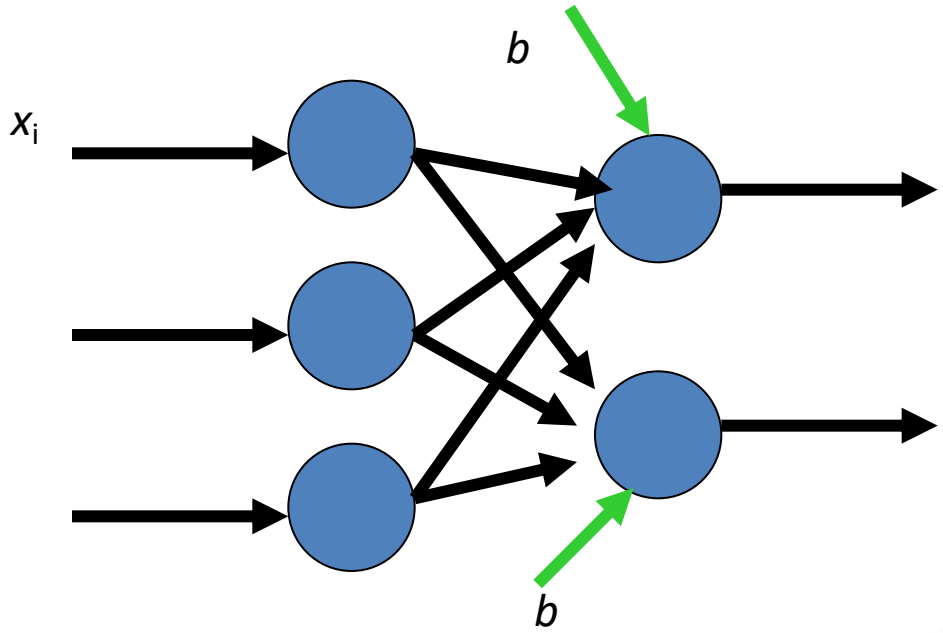
- **Supervised learning** in which the network is trained by providing it with input and matching output patterns. These input-output pairs are usually provided by an external teacher.

Classification (Learning)

- **Unsupervised learning** in which an (output) unit is trained to respond to clusters of pattern within the input. In this paradigm the system is supposed to discover statistically salient features of the input population. Unlike the supervised learning paradigm, there is no a priori set of categories into which the patterns are to be classified; rather the system must develop its own representation of the input stimuli.

Perceptron

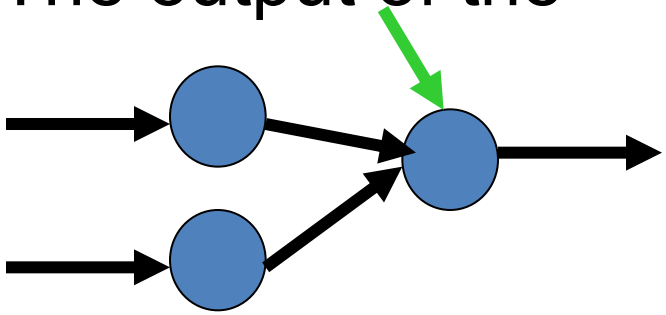
- A single layer feed-forward network consists of one or more output neurons, each of which is connected with a weighting factor w_{ij} to all of the inputs x_i.



Perceptron

- In the simplest case the network has only two inputs and a single output. The output of the neuron is:

$$y = f\left(\sum_{i=1}^{2} w_i x_i + b\right)$$



- suppose that the activation function is a threshold

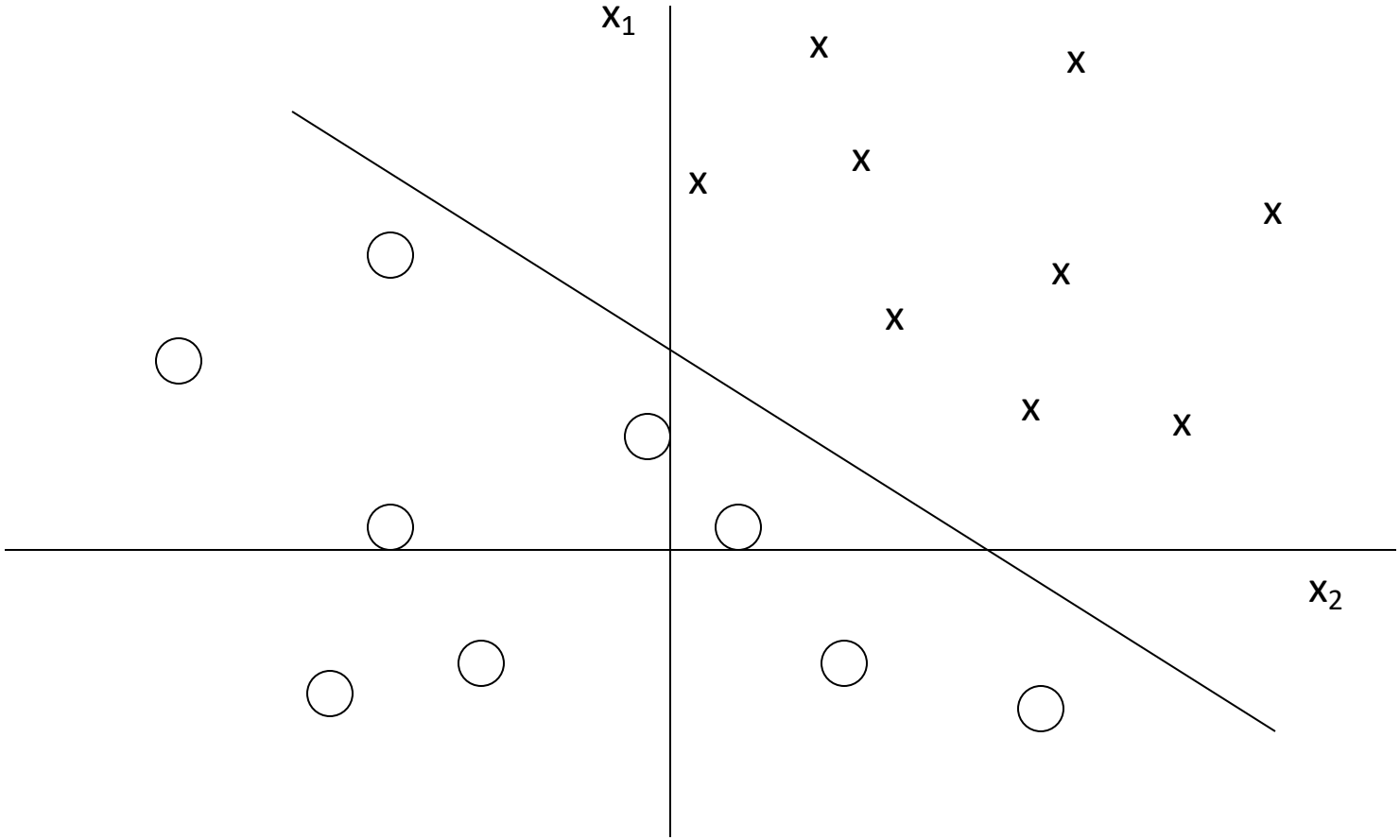
$$f = \begin{cases} 1 & if \quad s > 0 \\ -1 & if \quad s \leq 0 \end{cases}$$

Perceptron

- In this example the simple network (the neuron) can be used to separate the inputs in two classes.
- The separation between the two classes is given by

$$w_1 x_1 + w_2 x_2 + b = 0$$

Perceptron



Learning in Perceptrons

- The weights of the neural networks are modified during the learning phase

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$$

$$b_{ij}(t+1) = b_{ij}(t) + \Delta b_{ij}$$

Learning in Perceptrons

- Start with random weights
- Select an input couple $(\boldsymbol{x}, d(\boldsymbol{x}))$
- if $y \neq d(x)$ then modify the weight according with

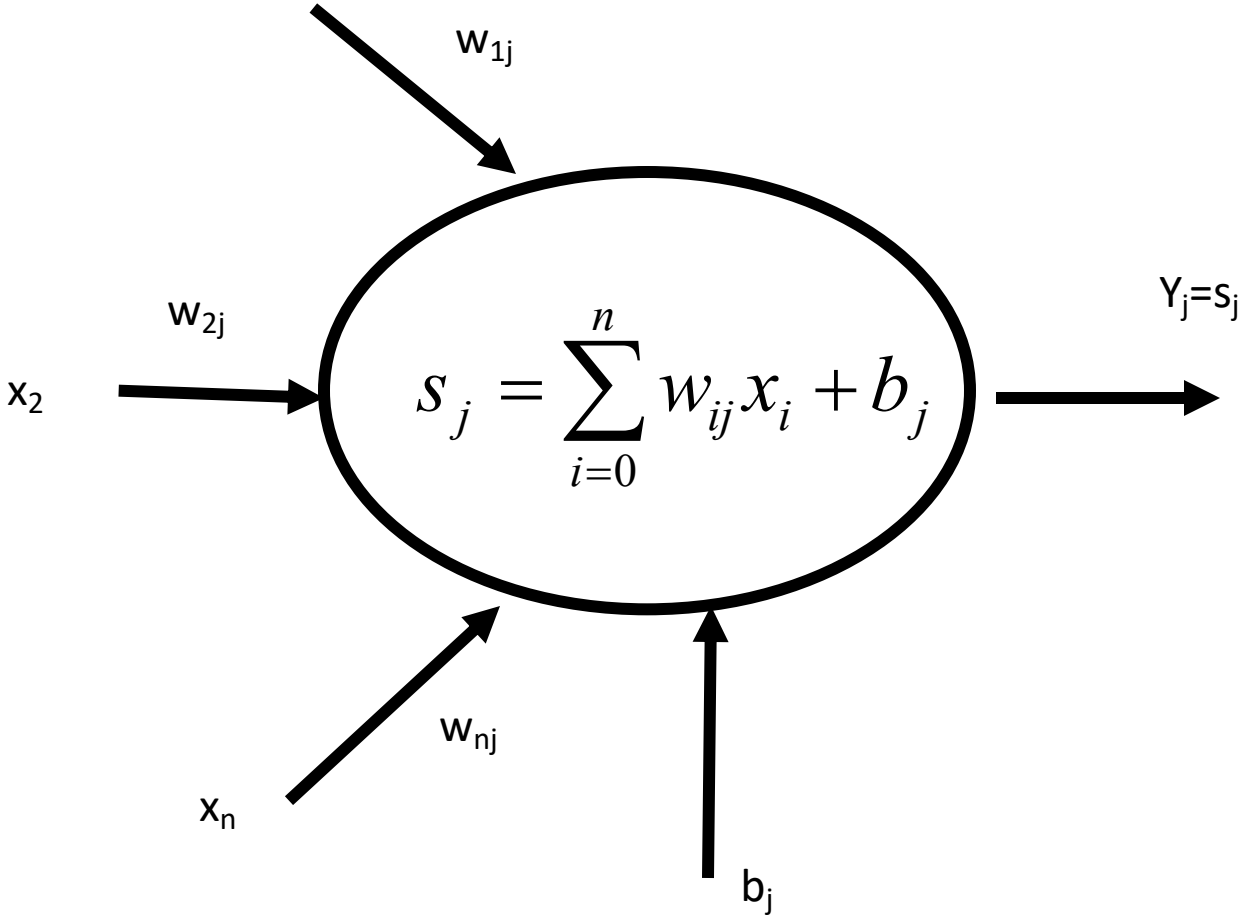
$$\Delta w_{ij} = d(x)x_i$$

Note that the weights are not modified if the network gives the correct answer

Convergence theorem

- If there exists a set of connection weights w^* which is able to perform the transformation $y = d(x)$, the perceptron learning rule will converge to some solution (which may or may not be the same as w^*) in a finite number of steps for any initial choice of the weights.

Linear Units



The Delta Rule 1

- The idea is to make the change of the weight proportional to the negative derivative of the error

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

The Delta Rule 2

$$\frac{\partial y_i}{\partial w_{ij}} = x_j$$

$$\frac{\partial E}{\partial y_i} = -\left(d_i - y_i\right) = \delta_i$$

$$\Delta w_{ij} = \gamma \delta_i x_j \quad (1)$$

Classification by Back propagation

- *Back Propagation learns by iteratively processing a set of training data (samples).*

- For each sample, weights are modified to minimize the error between network's classification and actual classification.

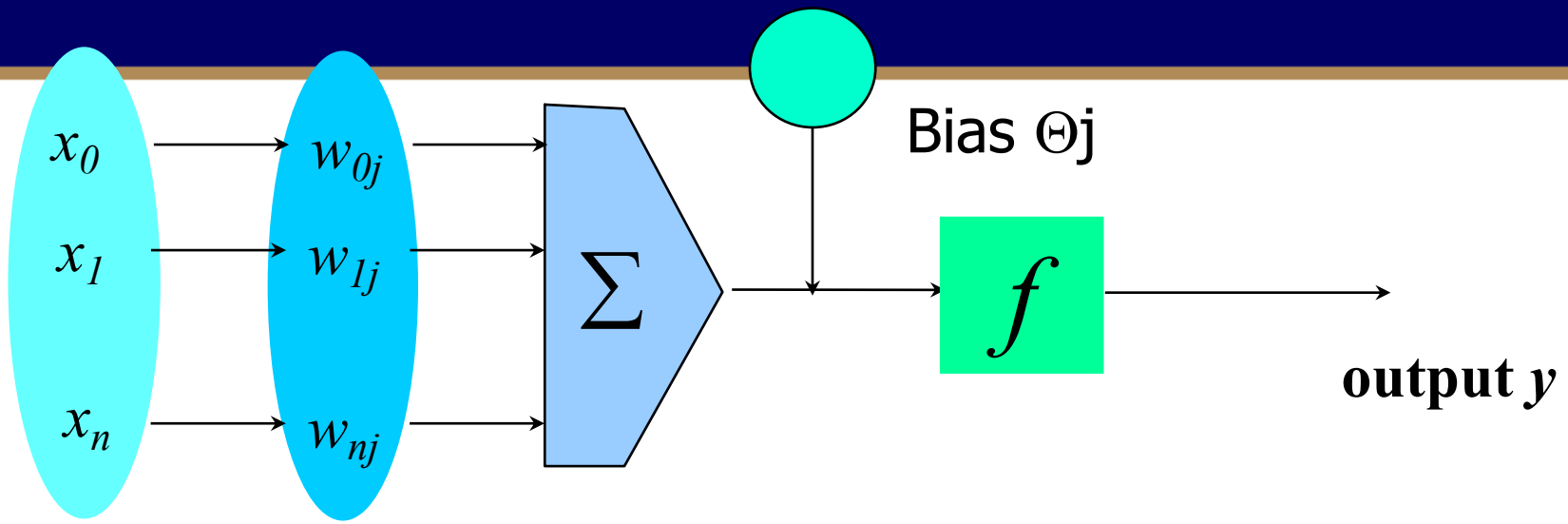
Steps in Back propagation Algorithm

- STEP ONE: initialize the weights and biases.
- The weights in the network are initialized to random numbers from the interval [-1,1].
- Each unit has a BIAS associated with it
- The biases are similarly initialized to random numbers from the interval [-1,1].
- STEP TWO: feed the training sample.

Steps in Back propagation Algorithm (cont..)

- STEP THREE: Propagate the inputs forward; we compute the net input and output of each unit in the hidden and output layers.
- STEP FOUR: back propagate the error.
- STEP FIVE: update weights and biases to reflect the propagated errors.
- STEP SIX: terminating conditions.

Propagation through Hidden Layer (One Node)



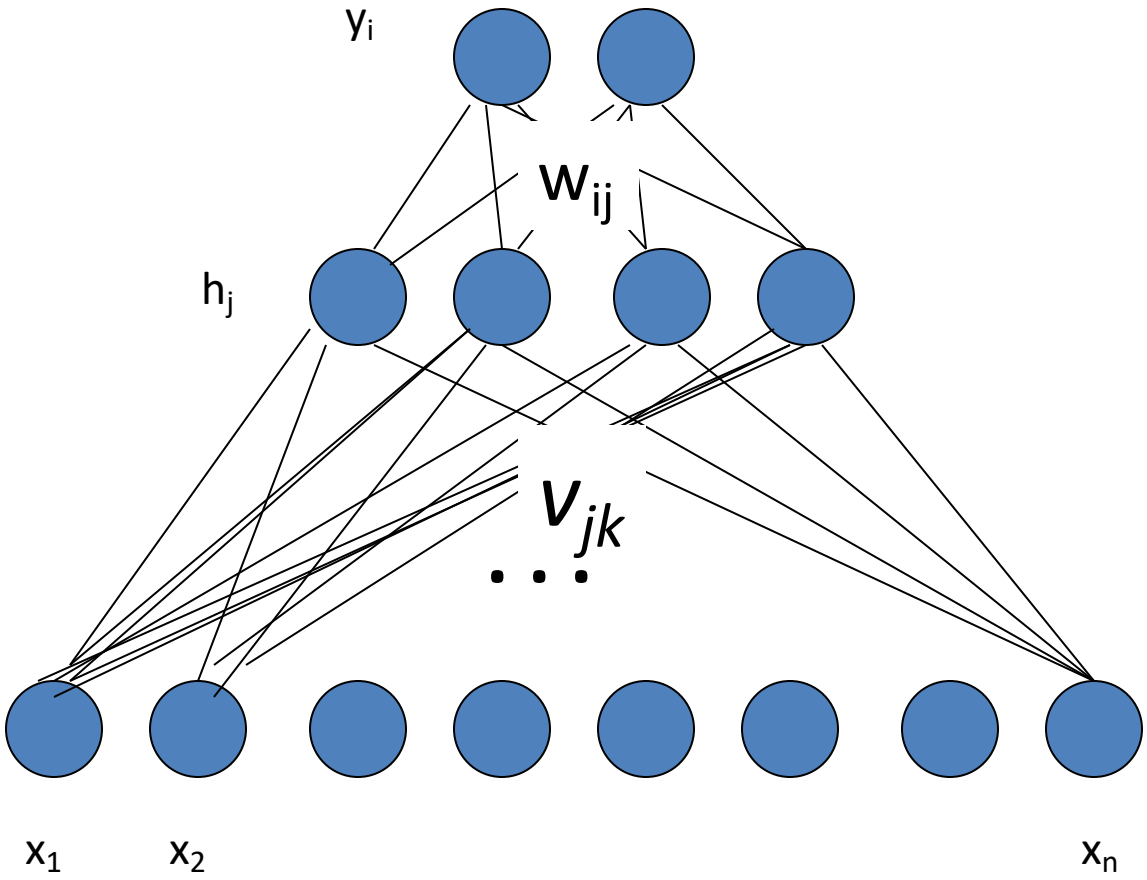
Input vector x **weight vector w** **weighted sum** **Activation function**

- The inputs to unit j are outputs from the previous layer. These are multiplied by their corresponding weights in order to form a weighted sum, which is added to the bias associated with unit j.
- A nonlinear activation function f is applied to the net input.

Backpropagation

- The multi-layer networks with a linear activation can classify only linear separable inputs or, in case of function approximation, only linear functions can be represented.

Backpropagation



Backpropagation

- When a learning pattern is clamped, the activation values are propagated to the output units, and the actual network output is compared with the desired output values, we usually end up with an error in each of the output units. Let's call this error e_o for a particular output unit o. We have to bring e_o to zero.

Backpropagation

- The simplest method to do this is the greedy method: we strive to change the connections in the neural network in such a way that, next time around, the error e_o will be zero for this particular pattern. We know from the delta rule that, in order to reduce an error, we have to adapt its incoming weights according to the last equation (1)

Backpropagation

- In order to adapt the weights from input to hidden units, we again want to apply the delta rule. In this case, however, we do not have a value for for the hidden units.

Backpropagation

- Calculate the activation of the hidden units

$$h_j = f\left(\sum_{k=0}^{n} v_{jk} x_k\right)$$

Backpropagation

- And the activation of the output units

$$y_i = f\left(\sum_{j=0} w_{ij} h_j\right)$$

Backpropagation

- If we have μ pattern to learn the error is

$$E = \frac{1}{2}\sum_{\mu}\sum_{i}\left(t_i^{\mu} - y_i^{\mu}\right)^2 =$$

$$= \frac{1}{2}\sum_{\mu}\sum_{i}\left[t_i^{\mu} - f\left(\sum_{j} w_{ij} h_j^{\mu}\right)\right]^2$$

$$= \frac{1}{2}\sum_{\mu}\sum_{i}\left[t_i^{\mu} - f\left(\sum_{j} w_{ij} f\left(\sum_{k=0}^{n} v_{jk} x_k^{\mu}\right)\right)\right]^2$$

Backpropagation

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

$$= \eta \sum_{\mu} \left(t_i^{\mu} - y_i^{\mu} \right) \dot{f}\left(A_i^{\mu} \right) h_j^{\mu} =$$

$$= \eta \sum_{\mu} \delta_i^{\mu} h_j^{\mu}$$

$$\delta_i^{\mu} = \left(t_i^{\mu} - y_i^{\mu} \right) \dot{f}\left(A_i^{\mu} \right)$$

Backpropagation

$$\Delta v_{jk} = -\eta \frac{\partial E}{\partial v_{jk}} = -\eta \sum_{\mu} \frac{\partial E}{\partial h_j^{\mu}} \frac{\partial h_j^{\mu}}{\partial v_{jk}} =$$

$$= \eta \sum_{\mu} \sum_{i} \left(t_i^{\mu} - y_i^{\mu} \right) \dot{f}\left(A_i^{\mu} \right) w_{ij} \; \dot{f}\left(A_j^{\mu} \right) x_k^{\mu} =$$

$$= \eta \sum_{\mu} \sum_{i} \delta_i^{\mu} w_{ij} \; \dot{f}\left(A_j^{\mu} \right) x_k^{\mu}$$

Backpropagation

- The weight correction is given by :

$$\Delta w_{mn} = \eta \sum_{\nu} \delta_m^{\mu} x_n^{\mu}$$

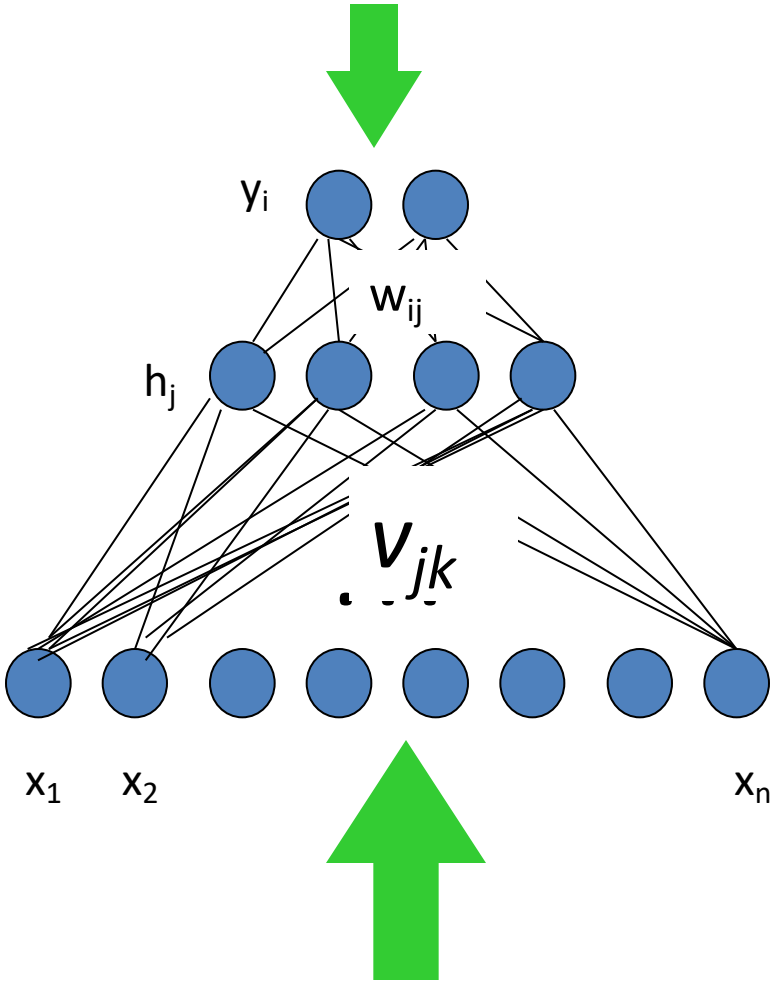
Where

$$\delta_m^{\mu} = \left(t_m^{\mu} - y_m^{\mu}\right) f'\left(A_m^{\mu}\right)$$ If *m* is the output layer

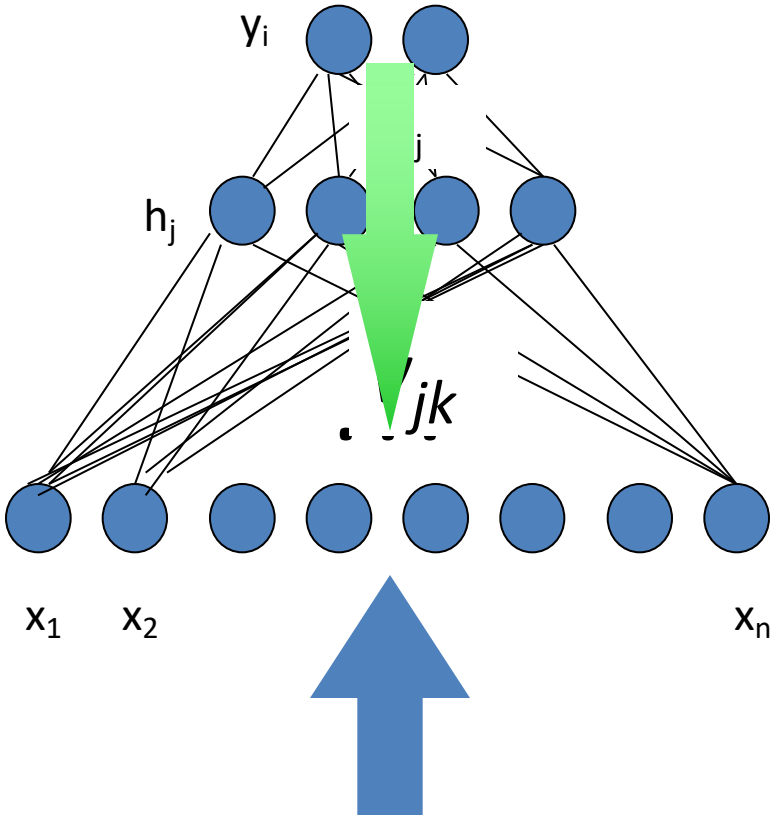
or

$$\delta_m^{\mu} = f'\left(A_m^{\mu}\right) \sum_{s} w_{sm} \delta_s^{\mu}$$ If *m* is an hidden layer

Backpropagation



Backpropagation



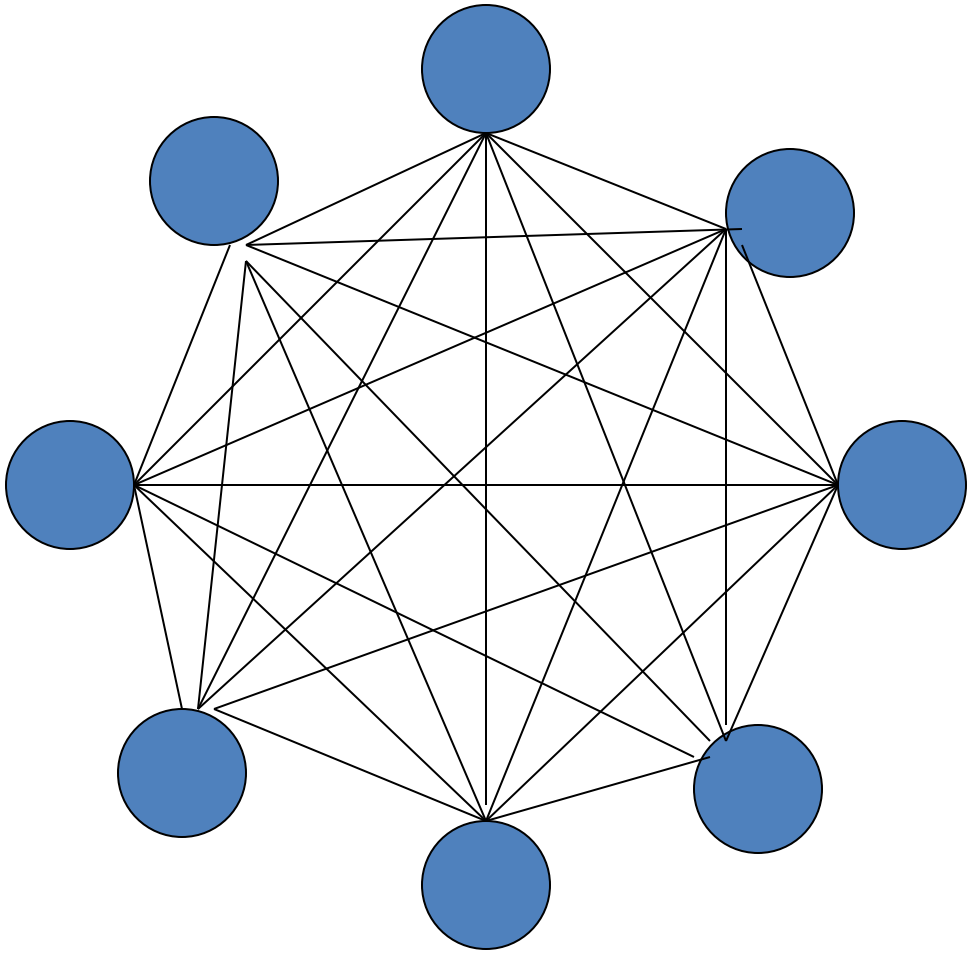
Recurrent Networks

- What happens when we introduce a cycle? For instance, we can connect a hidden unit with itself over a weighted connection, connect hidden units to input units, or even connect all units with each other ?

Hopfield Network

- The Hopfield network consists of a set of N interconnected neurons which update their activation values asynchronously and independently of other neurons.
- All neurons are both input and output neurons. The activation values are binary (+1, -1)

Hopfield Network



Summary

- Basic building block of Artificial Neural Network.
- Construction , working and limitation of single layer neural network (Single Layer Neural Network).
- Back propagation algorithm for multi layer feed forward NN.
- Some Advanced Features like training with subsets, Quicker convergence, Modular Neural Network, Evolution of NN.
- Application of Neural Network.

Remember.....

- ANNs perform well, generally better with larger number of hidden units
- More hidden units generally produce lower error
- Determining network topology is difficult
- Choosing single learning rate impossible
- Difficult to reduce training time by altering the network topology or learning parameters
- NN(Subset) often produce better results

Thank you

CSCD618: Deep Learning & Neural Networks

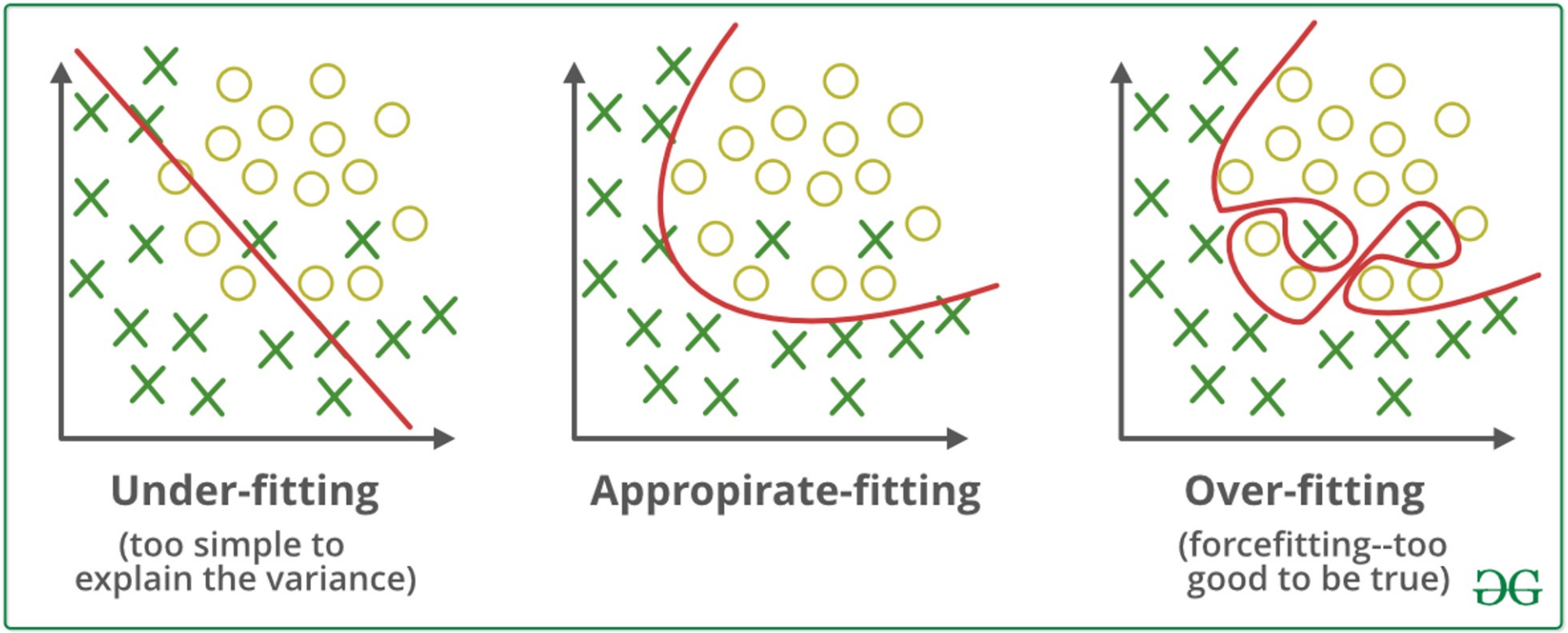
Session 3 – Regularization for Deep learning

By
Solomon Mensah (PhD)

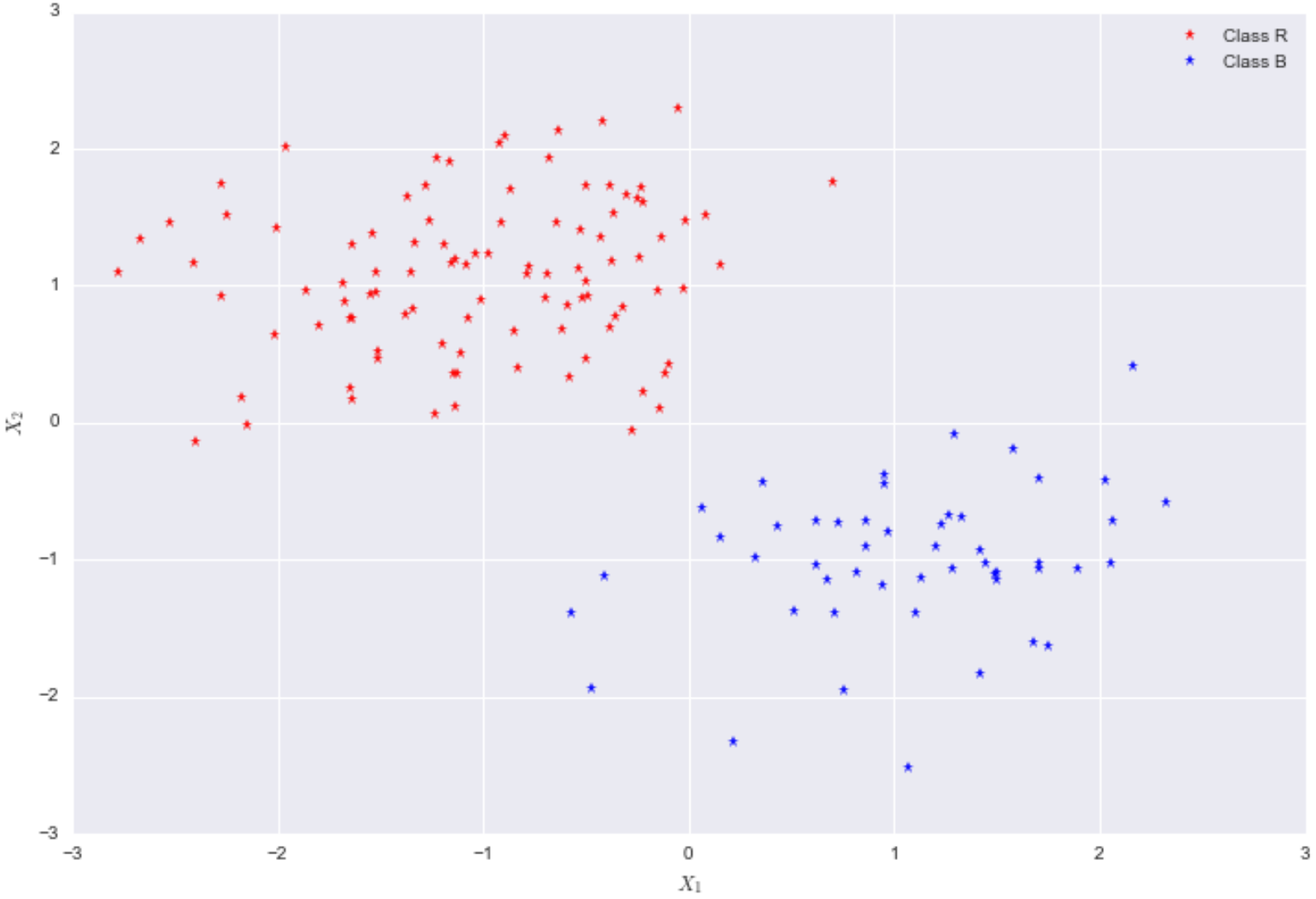
UNIVERSITY OF GHANA

Introduction

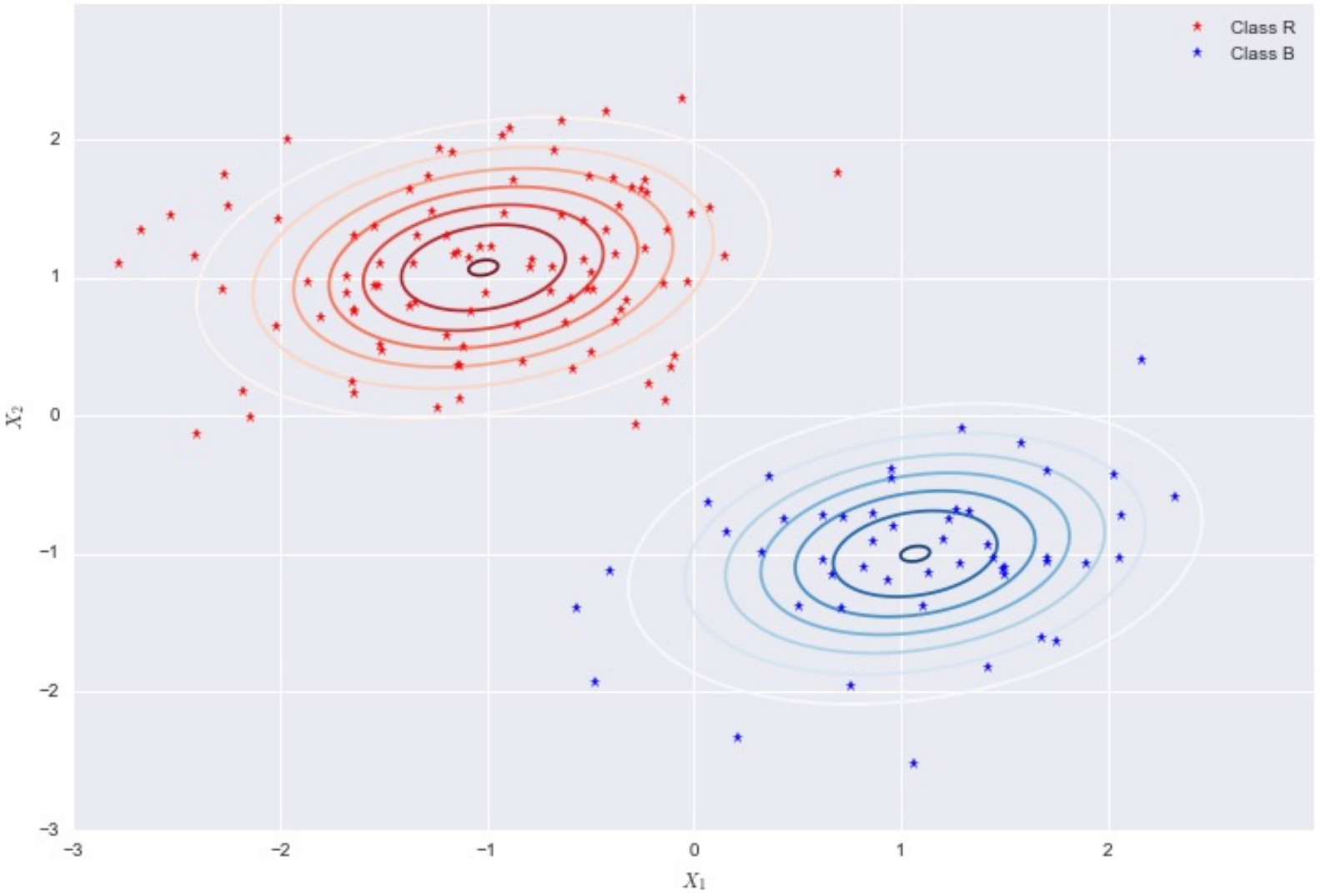
- Regularization in machine learning acts as a safeguard against overfitting, ensuring models excel in real-world scenarios.
- In the ever-evolving realm of Deep learning, the pursuit of building accurate and robust models often comes with a perilous pitfall—overfitting.
- It's a challenge that every data scientist encounters at some point.
- Solution is regularization

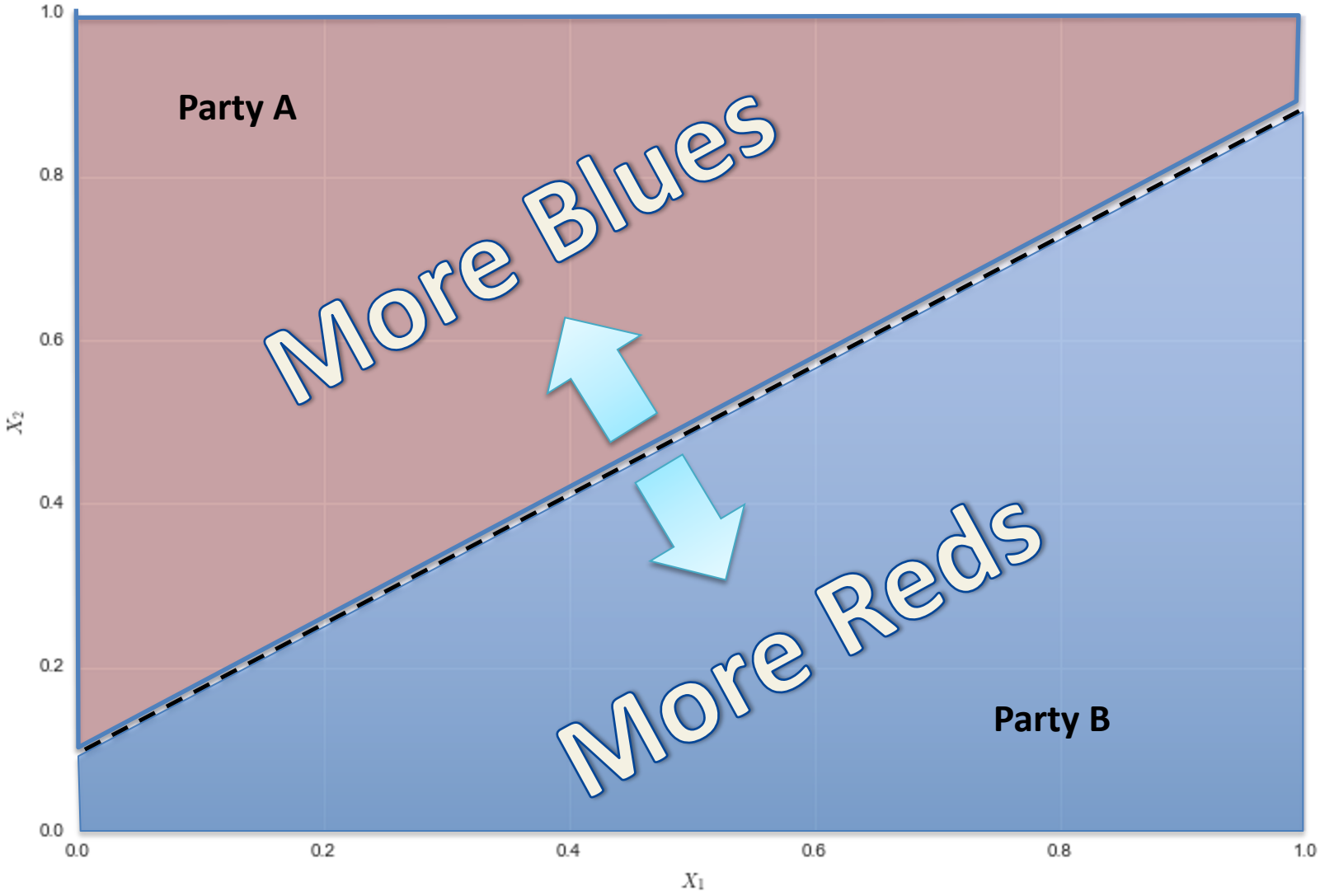

Under-fitting
(too simple to explain the variance)
Appropirate-fitting
Over-fitting
(forcefitting--too good to be true)

Start with some data



Calculate mean and covariances for each class assuming Normal distributions.




Party A
More Blues
More Reds
Party B
1.0
0.8
0.6
0.4
0.2
0.0
X_2
0.0
0.2
0.4
0.6
0.8
1.0
X_1

When do we need to regularize?

- Overfitting
- Colinearity
- Lasso/Ridge/Random Sampling/Dropout
- Model Selection

when, what, how?

Overfitting

How do we know we overfit?

- Test error is rising as a function of some tuning parameter that captures the flexibility of the model
- A model has overfit the training data when its test error is larger than its training error
- Model is too complex
  - How many parameters vs how many training points
- Variance of the training or testing error is too large
- Trade-off bias and variance

What is Regularization?

- Regularization is a set of techniques employed in machine learning to prevent models from fitting the training data too closely, which can lead to overfitting.
- Regularization methods introduce constraints or penalties to the model during training, discouraging it from becoming overly complex and encouraging a better balance between fitting the training data and maintaining generalization.

Why Regularization is Essential?

- Preventing Overfitting
- Feature Selection
- Enhancing Model Stability

Why Regularization is Essential?

- **Preventing Overfitting:** As mentioned earlier, the primary role of regularization is to prevent overfitting, ensuring that a model generalizes well to unseen data.
- **Feature Selection:** Regularization techniques like L1 can automatically perform feature selection by driving some feature coefficients to zero. This simplifies the model and reduces the risk of multicollinearity.
- **Enhancing Model Stability:** Regularization can make models more stable by reducing the variance in their predictions, leading to more reliable and consistent results.

Basic Types of Regularization

1. **L1 Regularization (Lasso):** L1 regularization adds the absolute values of the model's coefficients as a penalty term to the loss function. This encourages sparsity in the model, effectively selecting a subset of the most important features while setting others to zero.
2. **L2 Regularization (Ridge):** L2 regularization adds the square of the model's coefficients as a penalty term. It discourages extreme values in the coefficients and tends to distribute the importance more evenly across all features.

Basic Types of Regularization

- L1 tends to shrink coefficients to zero whereas L2 tends to shrink coefficients evenly.
- L1 is therefore useful for feature selection, as we can drop any variables associated with coefficients that go to zero.
- L2, on the other hand, is useful when you have collinear/co-dependent features.

Ridge Regression (L2 Regularization)

Slope has been reduced with ridge regression penalty and therefore the model becomes less sensitive to changes in the independent variable (#Years of experience)

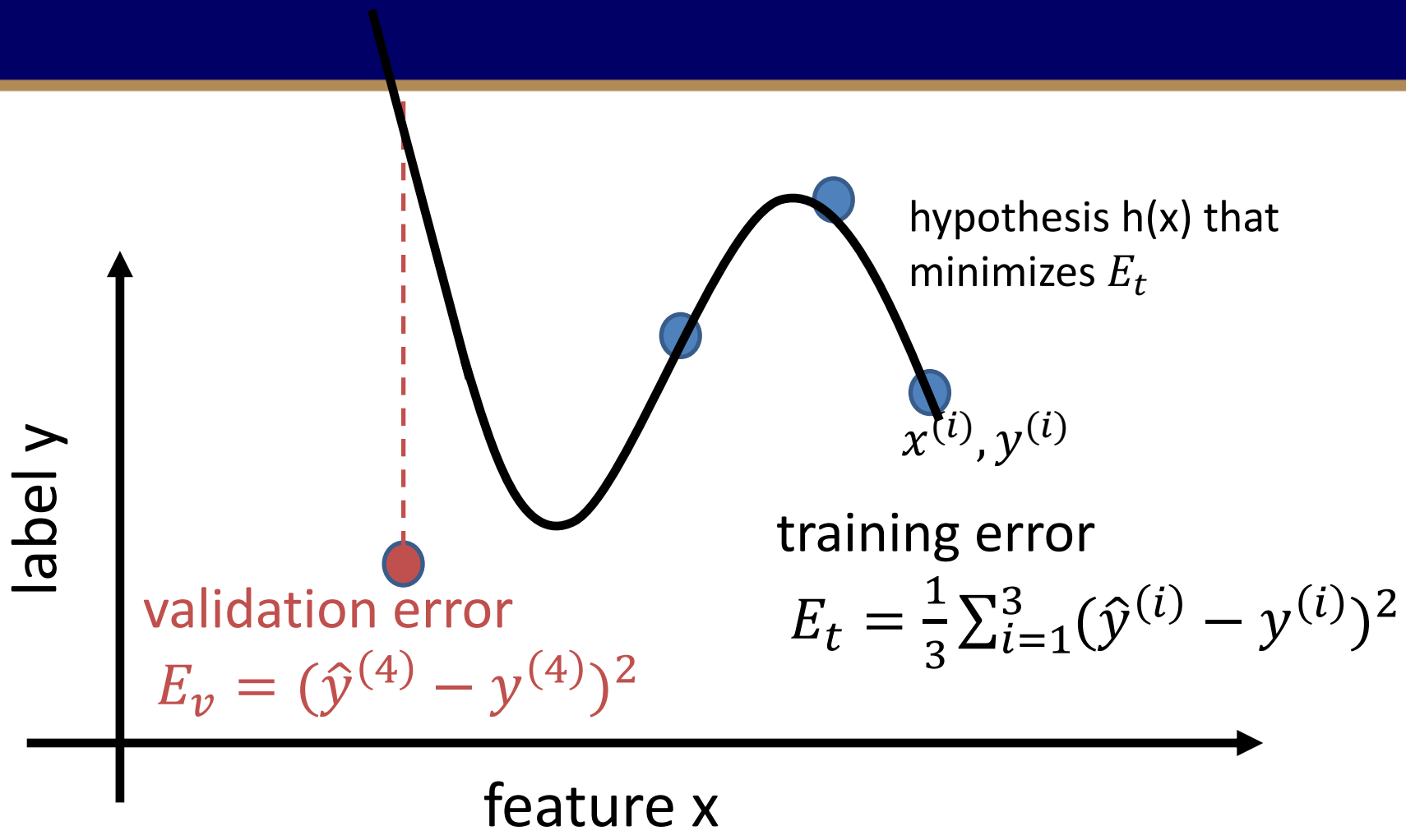
PENALTY TERM

Least Squares Regression:

$$Min(sum\ of\ the\ squared\ residuals)$$

Ridge Regression:

$$Min(sum\ of\ squared\ residuals + \alpha * slope^2)$$


hypothesis h(x) that minimizes E_t
$x^{(i)}, y^{(i)}$
training error
$E_t = \frac{1}{3}\sum_{i=1}^{3}(\hat{y}^{(i)} - y^{(i)})^2$
validation error
$E_v = (\hat{y}^{(4)} - y^{(4)})^2$
label y
feature x

How not to overfit?

Method	Why	How
Linear Regression	Too many predictors	Shrinkage or subset selection
Linear Regression	Degree of polynomial	Shrinkage or subset selection
KNN	Small K	Control K
Logistic Regression	Same as linear regression	Same as linear regression just in the odds space
QDA	Different covariance matrices	Control of how much common cov and individual cov
Decision Trees	Depth of the tree	Control max_depth, max_num_cells via pruning
Boosting	# iteration (B)	Control B or learning rate.

Model selection using AIC/BIC VS Regularization

We know that the more predictors, or more terms in the polynomial we have, the more likely is to overfit.

Subset selection using AIC/BIC just removes terms with the hope that overfitting goes away.

Shrinkage methods on the other hand hope that by restricting the coefficients we can avoid overfitting.

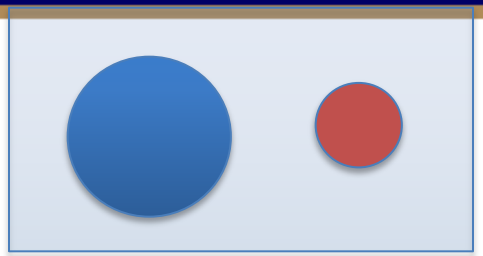
At the end: We ignore overfitting and just find the best model on the test set.

Imbalanced Data

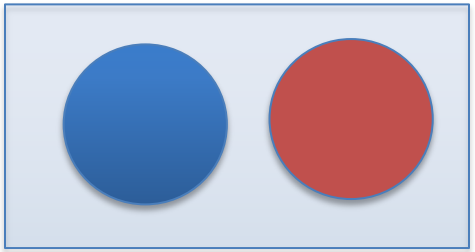
- Subsample
- Oversample
- Re-weight sample points
- Use clustering to reduce majority class
- Re-calibrate classifier output

Imbalanced Data

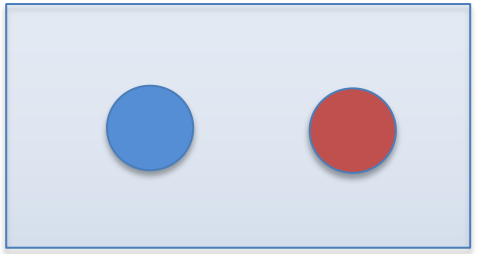
- The problem:



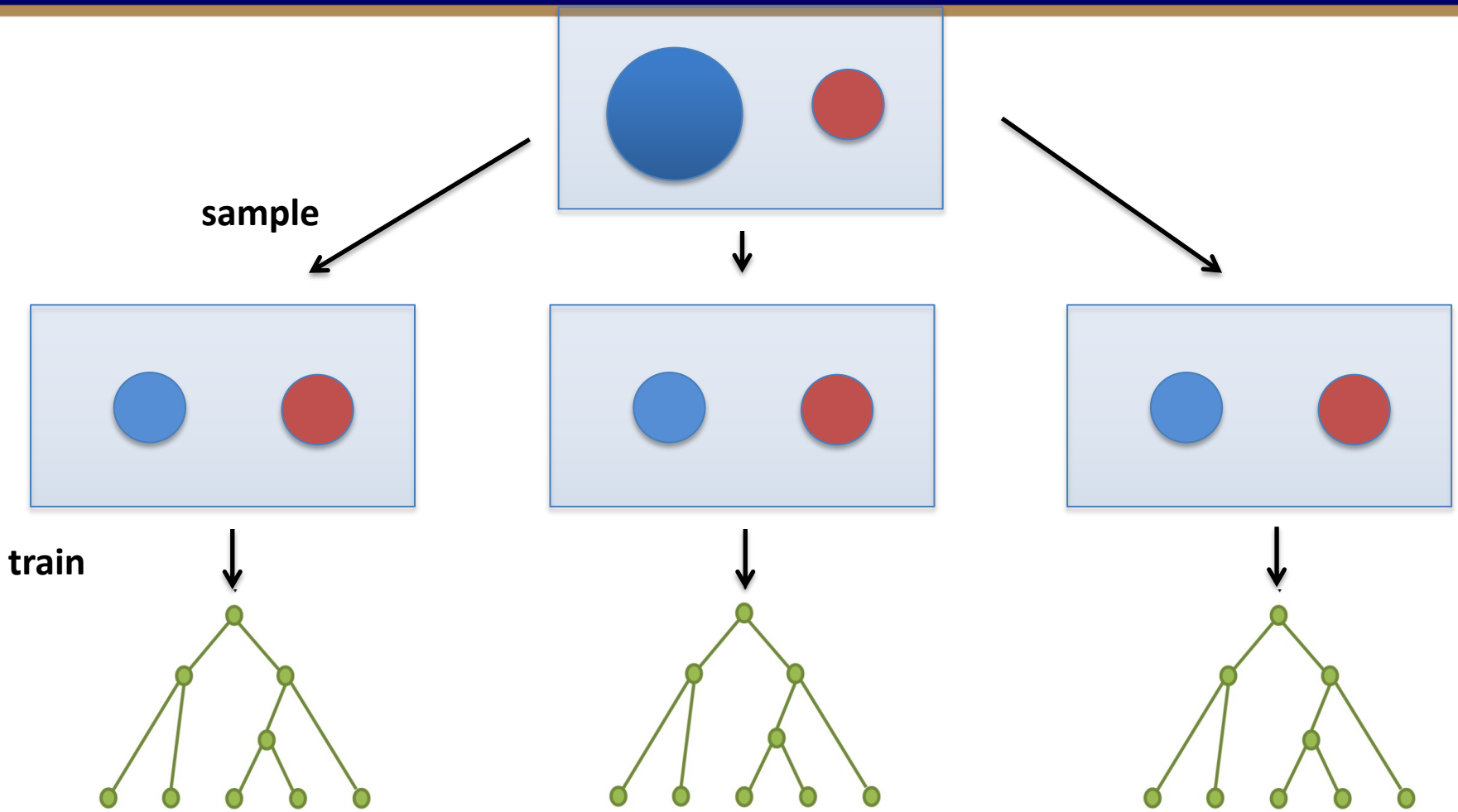
- Oversample:



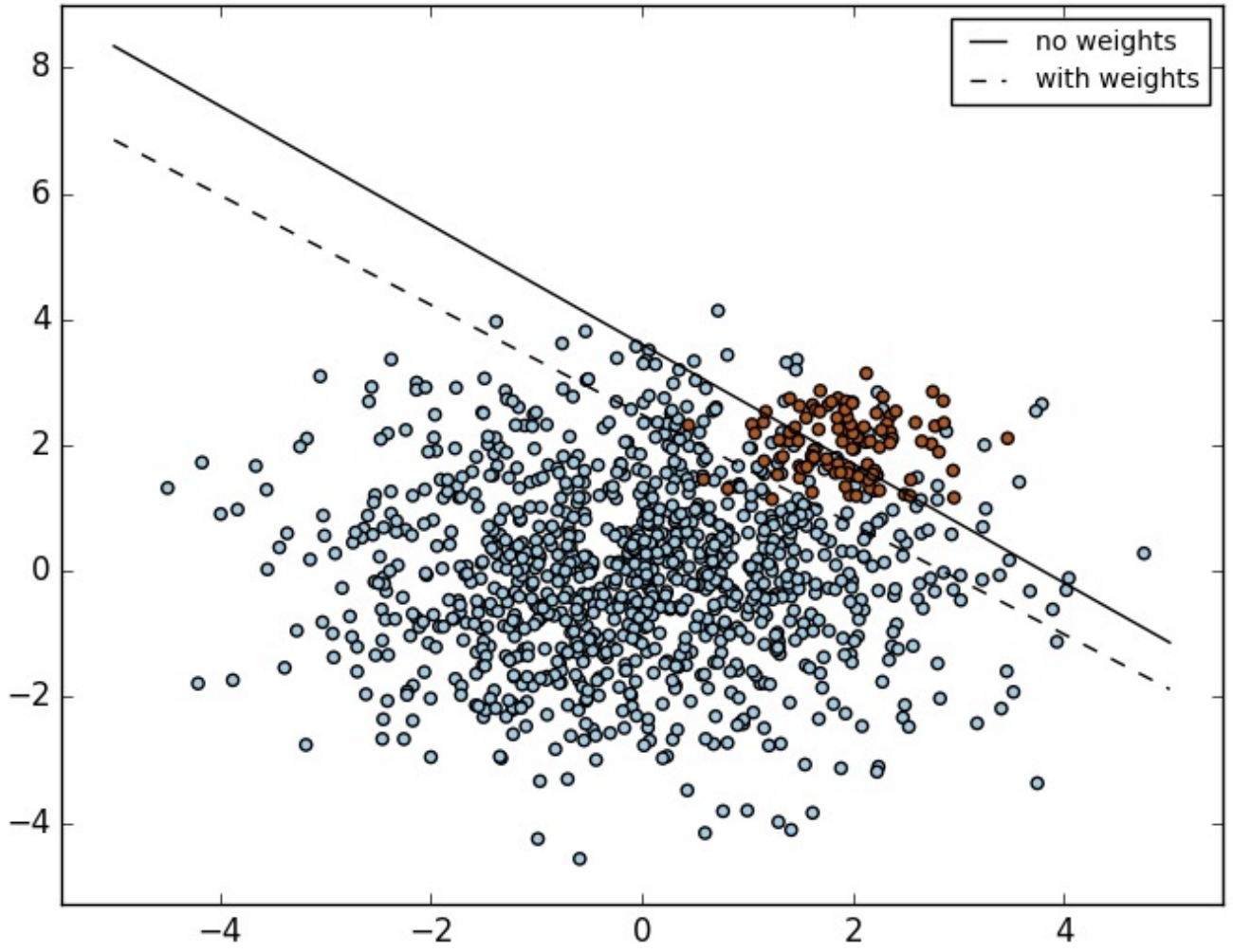
- Subsample:



Example: Random Forest Subsampling



Class weights



Problem of fitting

- Too many parameters = overfitting
- Not enough parameters = underfitting

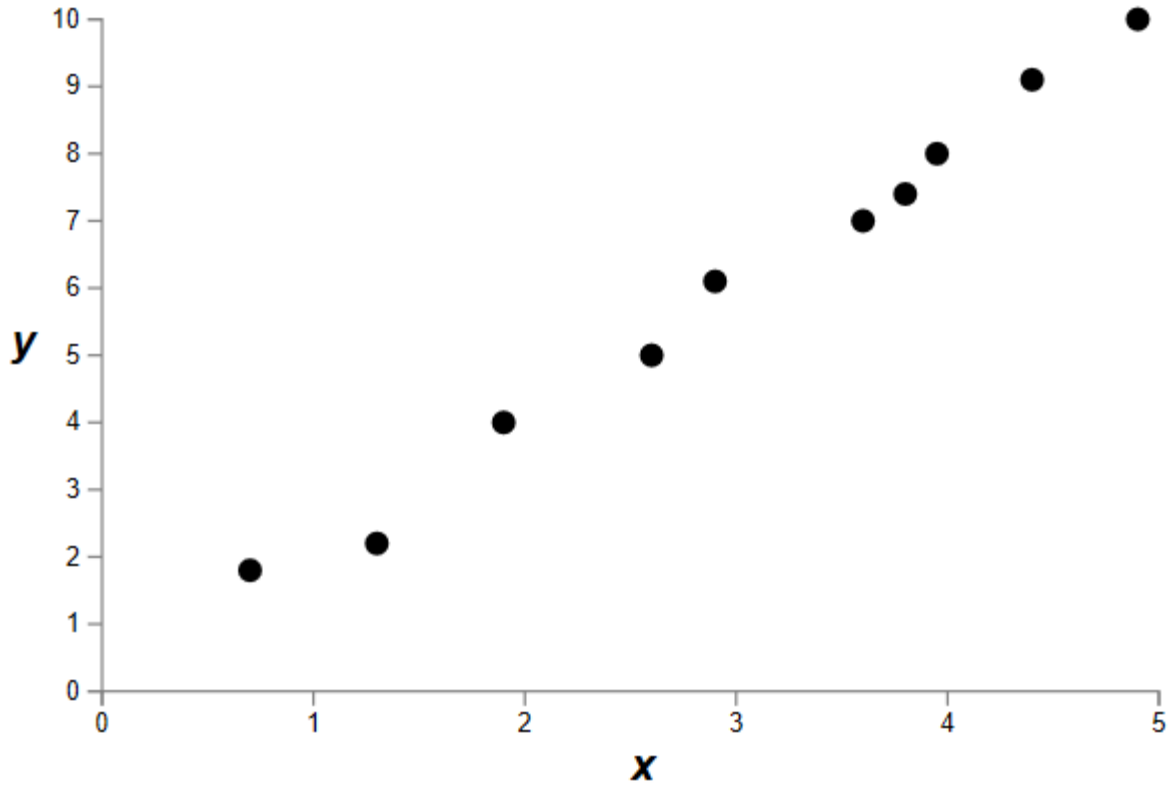
- More data = less chance to overfit

- How do we know what is required?

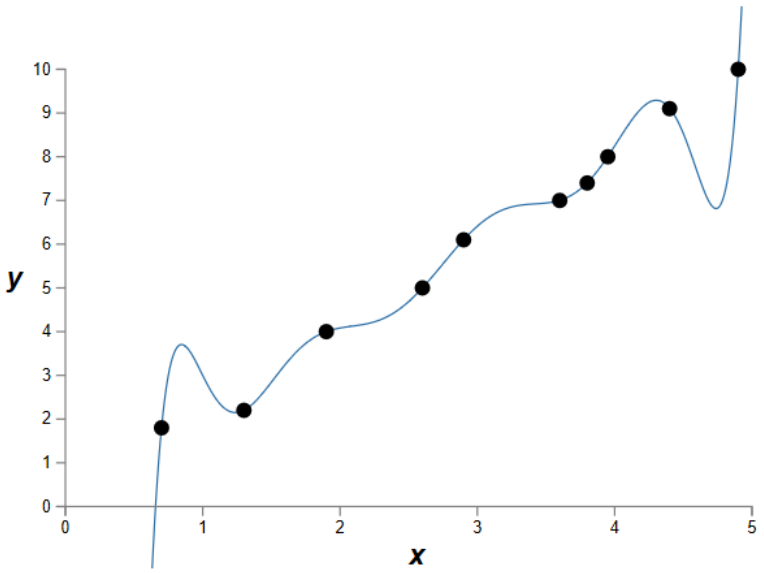
Regularization

- Attempt to guide solution *not to overfit*
- But still give freedom with many parameters

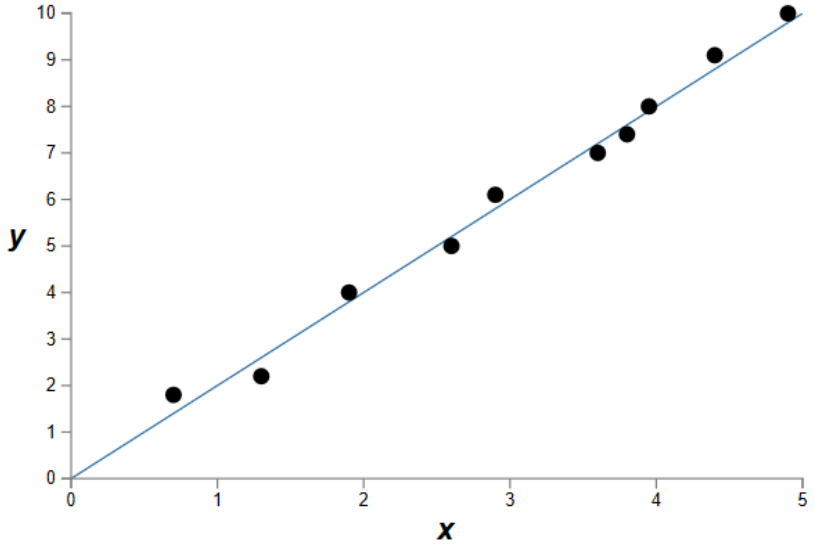
Data fitting problem



Which is better?



9th order polynomial



1st order polynomial

Regularization

- Attempt to guide solution *not to overfit*
- But still give freedom with many parameters

- Idea:
  Penalize the use of parameters to prefer small weights.

Regularization:

- Idea: add a cost to having high weights
- λ = regularization parameter

$$C = C_0 + \lambda \sum_w w^2$$

Both can describe the data…

- …but one is simpler

- Occam's razor:

  "Among competing hypotheses, the one with the fewest assumptions should be selected"

For us:

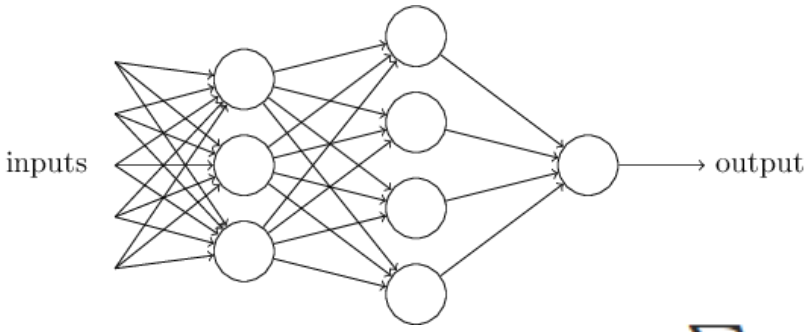
Large weights cause large changes in behaviour in response to small changes in the input.

Simpler models (or smaller changes) are more robust to noise.

Regularization: Dropout

- Our networks typically start with random weights.
- Every time we train = slightly different outcome.

- Why random weights?
- If weights are all equal, response across filters will be equivalent.
  - Network doesn't train.



$$w \cdot x \equiv \sum_j w_j x_j$$

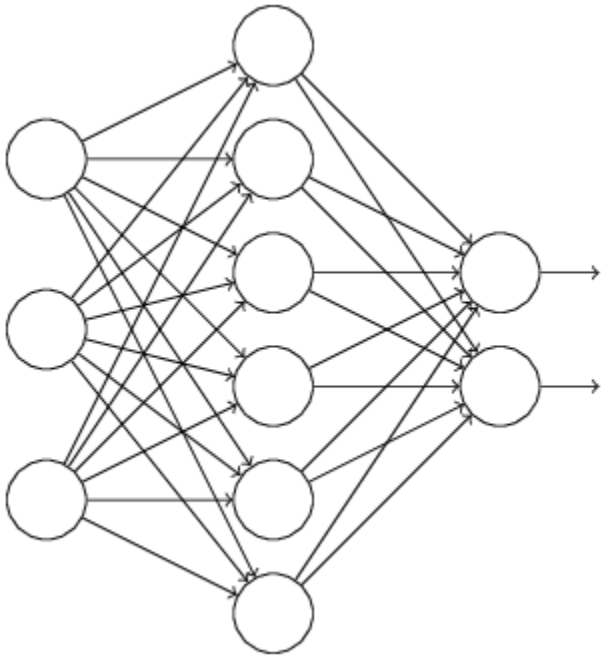
[Nielson]

Regularization

- Our networks typically start with random weights.
- Every time we train = slightly different outcome.

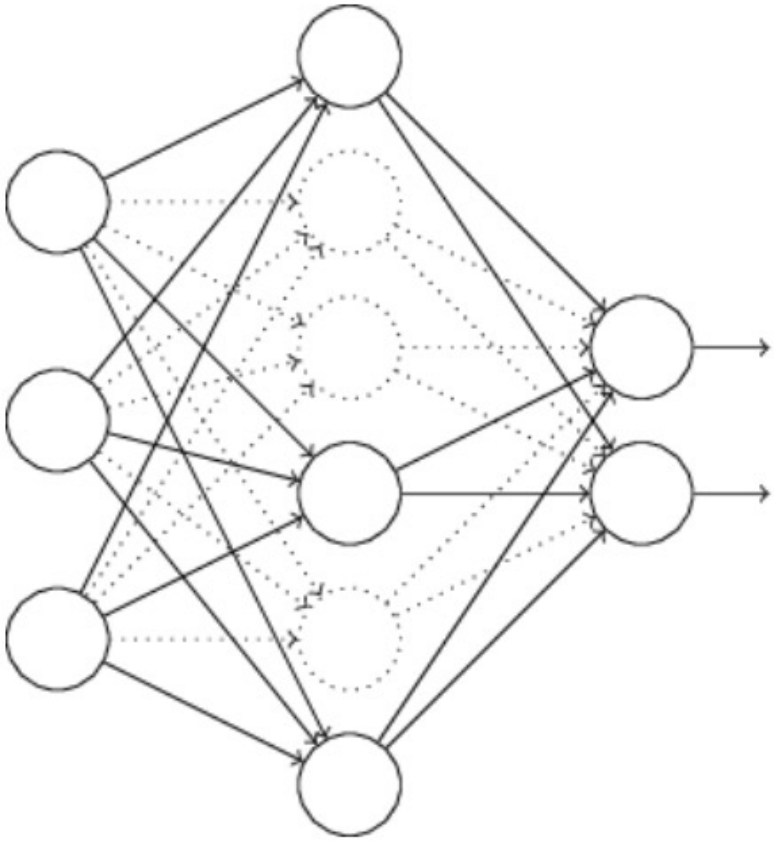
- Why not train 5 different networks with random starts and vote on their outcome?
  - Works fine!
  - Helps generalization because error is averaged.

Regularization: Dropout



[Nielson]

Regularization: Dropout



At each mini-batch:

- Randomly select a subset of neurons.
- Ignore them.

On test: half weights outgoing to compensate for training on half neurons.

Effect:

- Neurons become less dependent on output of connected neurons.
- Forces network to learn more robust features that are useful to more subsets of neurons.
- Like averaging over many different trained networks with different random initializations.
- Except cheaper to train.

[Nielson]

Many forms of ‘regularization’

- Adding more data is a kind of regularization
- Pooling is a kind of regularization
- Data augmentation is a kind of regularization

Thank you

CSCD618: Deep Learning & Neural Networks

Session 4 – Recurrent Neural Networks

By
Solomon Mensah (PhD)

UNIVERSITY OF GHANA

Outline

- RNN Definition
- Why use RNN?
- RNN Trade-offs
- RNN Properties
- Training RNN
- LSTM

Introduction

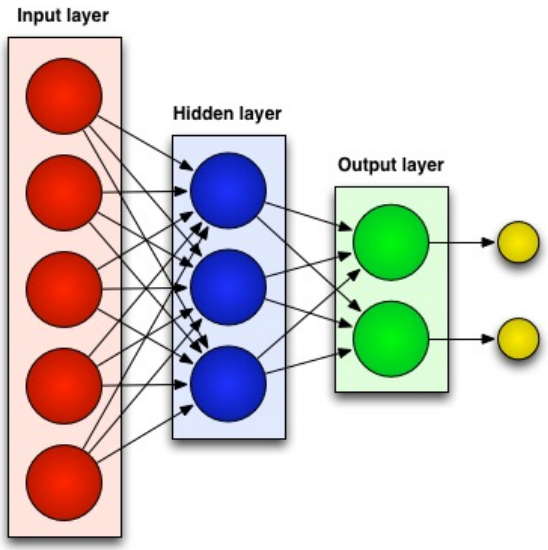
- Recurrent neural networks (RNNs) are models with bi-directional data flow
  - While a feed-forward network propagates data linearly from input to output, RNNs also propagate data from later processing stages to earlier stages.

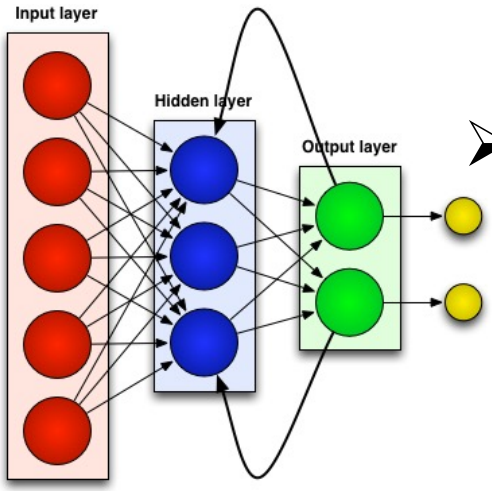
- In a *fully RNN*, every neuron receives inputs from every other neuron in the network.
- Usually only a subset of the neurons receive external inputs in addition to the inputs from all the other neurons, and another disjunct subset of neurons report their output externally as well as sending it to all the neurons.
- These distinctive inputs and outputs perform the function of the input and output layers of a feed-forward or simple RNN, and also join all the other neurons in the recurrent processing.

Introduction

- **Generally there are two kinds of neural networks:**

  - **Feedforward Neural Networks:**
    - ✓ **connections between the units do not form a cycle**

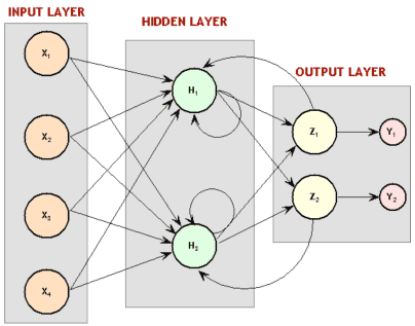


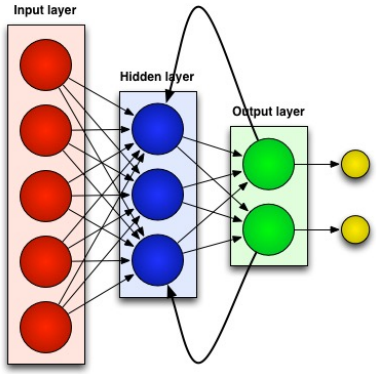


  - **Recurrent Neural Network:**
    - ✓ **connections between units form cyclic paths**

Introduction

- Recurrent since they receive inputs, update the hidden states depending on the previous computations, and make predictions for every element of a sequence.
- RNNs are a neural network with memory.
- RNNs are very powerful dynamic system for sequential tasks such as speech recognition or handwritten recognition since they maintain a state vector that implicitly contains information about the history of all the past elements of a sequence.





Why use RNN?

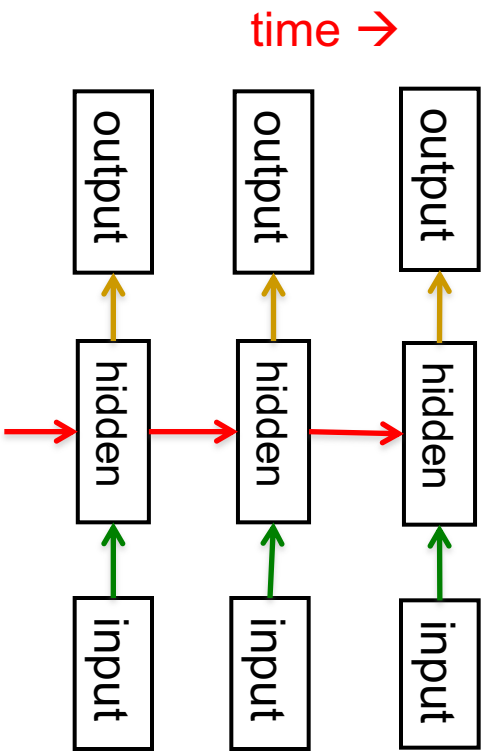
- RNN captures long-term contextual effect over time
- Therefore, can use temporal context to compensate for missing data.
- Also allows a single net to perform both imputation and classification.
- RNNs are better suited to analyzing temporal and sequential data, such as text or videos.
- RNN is ideal for tasks like speech recognition, machine translation, natural language processing (NLP) and text generation.

RNN Trade-offs

- RNN Advantages:
- - Can process any length input
- - Computation for step *t* can (in theory) use information from many steps back
- - Model size doesn't increase for longer input
- RNN Disadvantages:
- - Recurrent computation is slow
- - In practice, difficult to access information from many steps back

RNN Properties

- RNNs are very powerful, because they combine two properties:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.

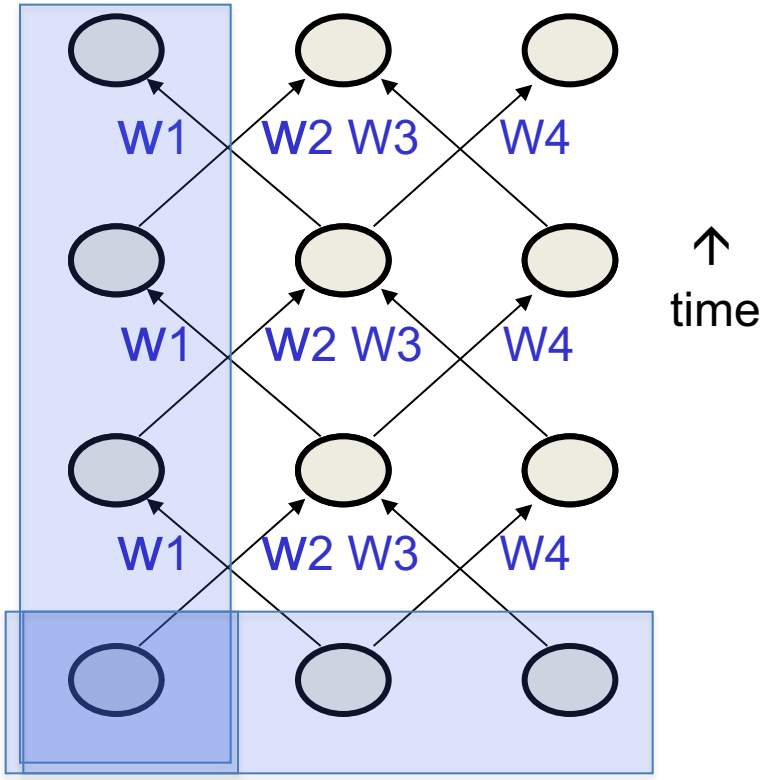


Backpropagation through time

- We can think of the recurrent net as a layered, feed-forward net with shared weights and then train the feed-forward net with weight constraints.
- We can also think of this training algorithm in the time domain:
  - The forward pass builds up a stack of the activities of all the units at each time step.
  - The backward pass peels activities off the stack to compute the error derivatives at each time step.
  - After the backward pass we add together the derivatives at all the different times for each weight.

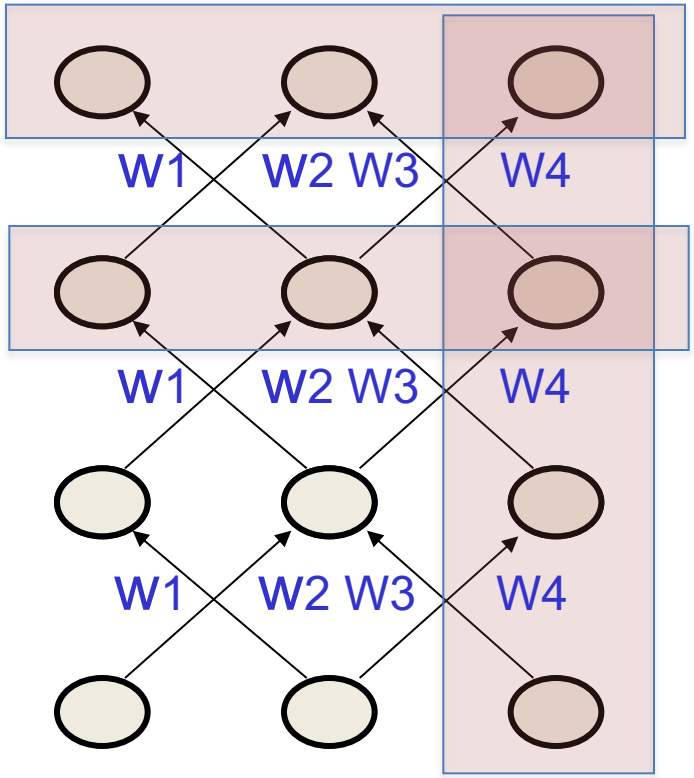
Providing input to recurrent networks

- We can specify inputs in several ways:
  - Specify the initial states of all the units.
  - Specify the initial states of a subset of the units.
  - Specify the states of the same subset of the units at every time step.
    - This is the natural way to model most sequential data.



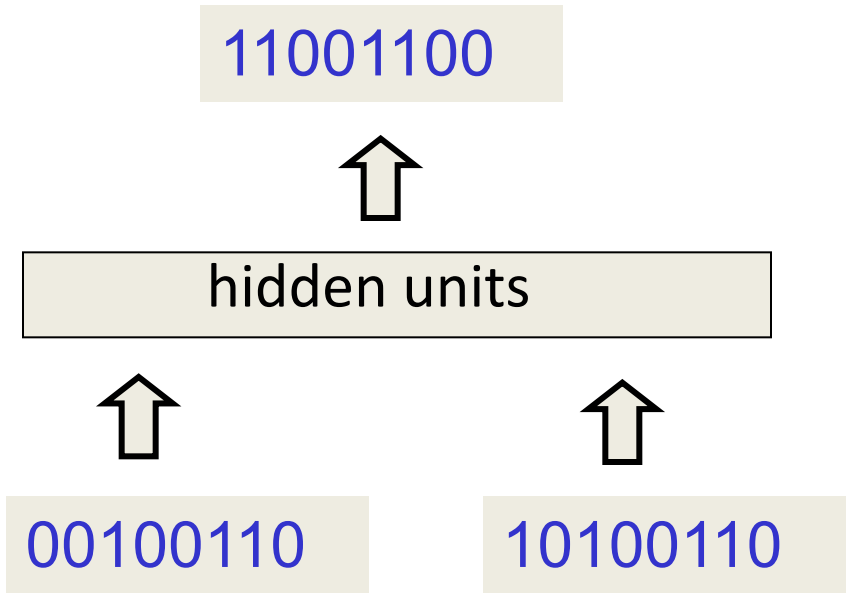
Teaching signals for recurrent networks

- We can specify targets in several ways:
  - Specify desired final activities of all the units
  - Specify desired activities of all units for the last few steps
    - Good for learning attractors
    - It is easy to add in extra error derivatives as we backpropagate.
  - Specify the desired activity of a subset of the units.
    - The other units are input or hidden units.



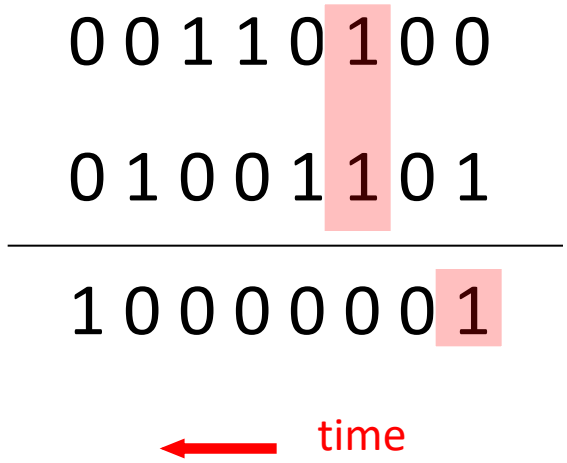
A good toy problem for a recurrent

- We can train a feedforward net to do binary addition, but there are obvious regularities that it cannot capture efficiently.
  - We must decide in advance the maximum number of digits in each number.
  - The processing applied to the beginning of a long number does not generalize to the end of the long number because it uses different weights.
- As a result, feedforward nets do not generalize well on the binary addition task.



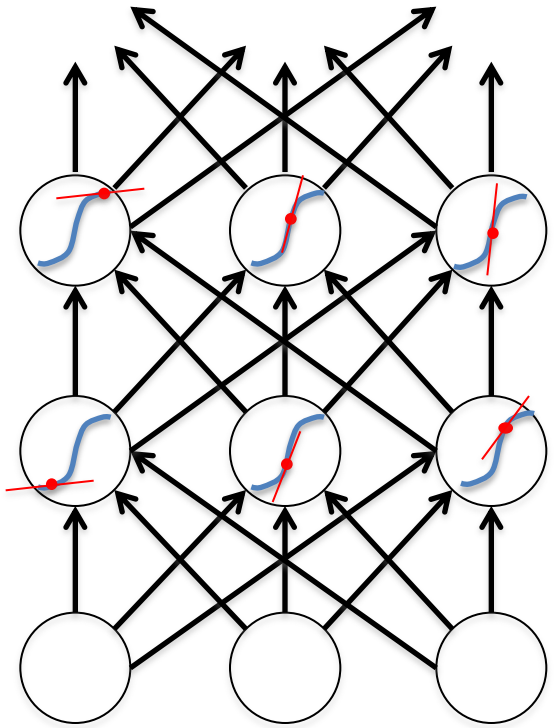
A recurrent net for binary addition

- The network has two input units and one output unit.
- It is given two input digits at each time step.
- The desired output at each time step is the output for the column that was provided as input two time steps ago.
  - It takes one time step to update the hidden units based on the two input digits.
  - It takes another time step for the hidden units to cause the output.



The backward pass is linear

- There is a big difference between the forward and backward passes.
- In the forward pass we use squashing functions (like the logistic) to prevent the activity vectors from exploding.
- The backward pass, is completely linear. If you double the error derivatives at the final layer, all the error derivatives will double.
  - The forward pass determines the slope of the linear function used for backpropagating through each neuron.



The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.

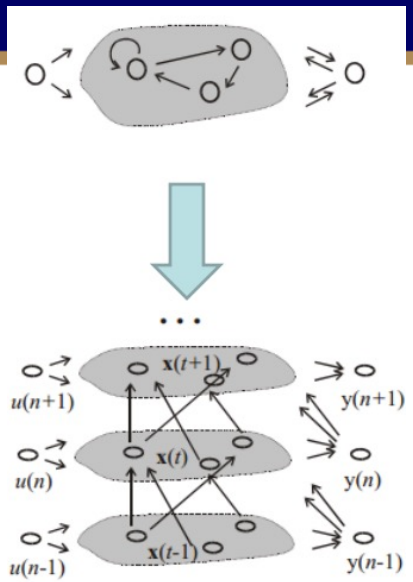
Four effective ways to learn an RNN

- Long Short-Term Memory: Make the RNN out of little modules that are designed to remember values for a long time.
- Hessian Free Optimization: Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
  - The HF optimizer (Martens & Sutskever, 2011) is good at this.
- Echo State Networks: Initialize the input→hidden and hidden→hidden and output→hidden connections very carefully so that the hidden state has a huge reservoir of weakly coupled oscillators which can be selectively driven by the input.
  - ESNs only need to learn the hidden→output connections.
- Good initialization with momentum Initialize like in ESNs, but then learn all of the connections using momentum.

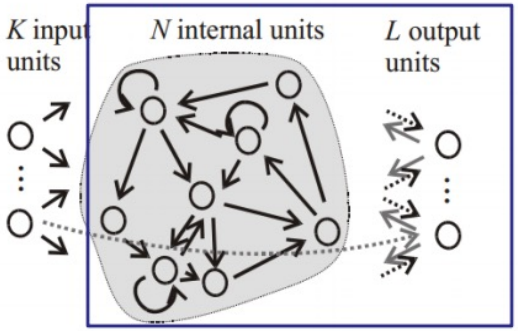
RNN Training

There are different approaches for training RNNs:

- Back-Propagation Through Time (BPTT): Unfolding an RNN in time and using the extended version of backpropagation.
- Extended Kalman Filtering (EKF):
  a set of mathematical equations that provides an efficient computational means to estimate the state of a process, in a way that minimizes the mean of the squared error on linear system.
- Real-Time Recurrent Learning (RTRL):
  Computing the error gradient and update weights for each time step.



BPTT



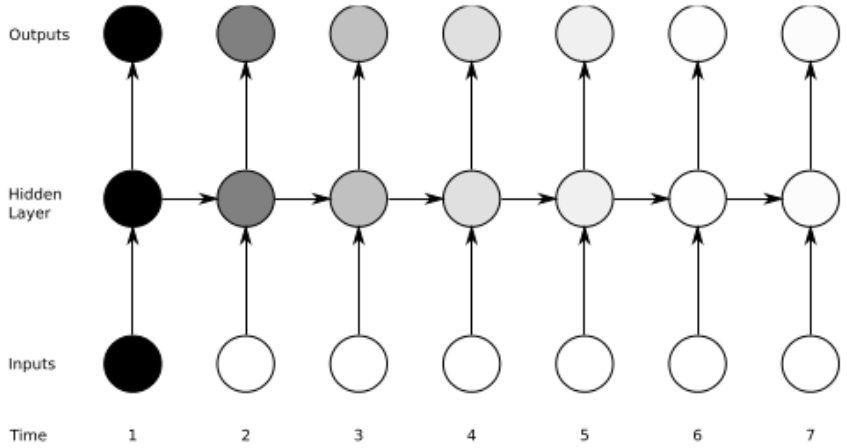
RTRL

Backpropagation Through Time

- The backpropagation algorithm can be extended to BPTT by unfolding RNN in time and stacking identical copies of the RNN.
- As the parameters that are supposed to be learned (*U*, *V* and *W*) are shared by all time steps in the network, the gradient at each output depends, not only on the calculations of the current time step, but also the previous time steps.
- In RNNs, a common choice for the loss function is the cross-entropy loss which is given by:
  - $L(y_l, y) = -\frac{1}{N}\sum_{n \in N} y_{ln}\, log\, y_n$
  - N: the number of training examples
  - y: the prediction of the network
  - y_l: the true label

The Vanishing Gradient Problem

- Definition: The influence of a given input on the hidden layer, and therefore on the network output, either decays or grows exponentially as it propagates through an RNN.
- In practice, the range of contextual information that standard RNNs can access are limited to approximately 10 time steps between the relevant input and target events.
- Solution: LSTM networks.

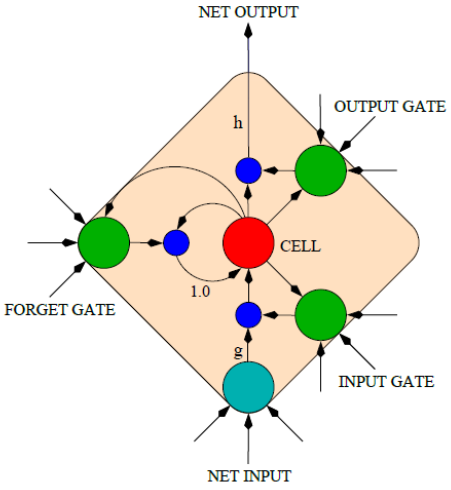


Long Short-Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.

LONG SHORT-TERM MEMORY (LSTM)

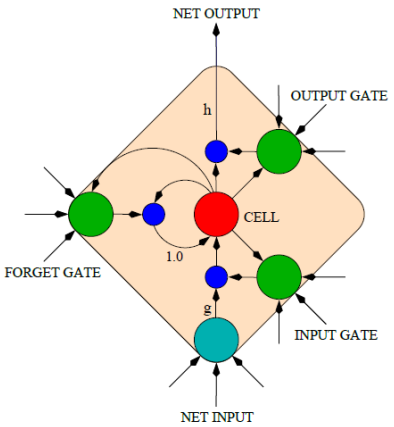
- An LSTM is a special kind of RNN architecture, capable of learning long-term dependencies.
- An LSTM can learn to bridge time intervals in excess of 1000 steps.
- LSTM networks outperform RNNs and Hidden Markov Models (HMMs):
  - Speech Recognition: 17.7% phoneme error rate on TIMIT acoustic phonetic corpus.
  - Winner of the ICDAR handwriting competition for the best known results in handwriting recognition.
- This is achieved by multiplicative gate units that learn to open and close access to the constant error flow.



LSTM Memory Cell

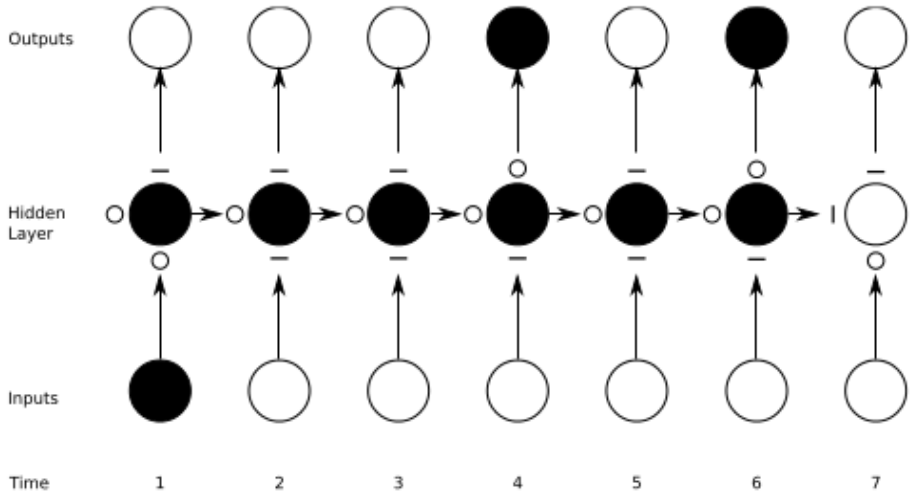
Memory Cell in LSTM

- LSTM networks introduce a new structure called a memory cell.
- Each memory cell contains four main elements:
  - Input gate
  - Forget gate
  - Output gate
  - Neuron with a self-recurrent
- These gates allow the cells to keep and access information over long periods of time.



LSTM Memory Cell

Preserving the information in LSTM



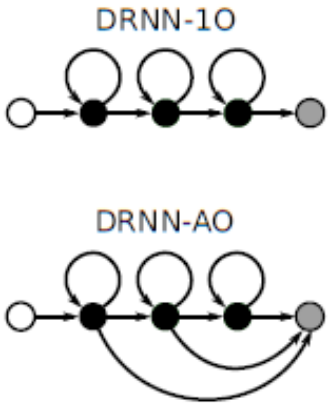
- A demonstration of how an unrolled LSTM preserves sequential information.
- The input, forget, and output gate activations are displayed to the left and above the memory block (respectively). The gates are either entirely open ('O') or entirely closed ('—').
- Traditional RNNs are a special case of LSTMs: Set the input gate to all ones (passing all new information), the forget gate to all zeros (forgetting all of the previous memory) and the output gate to all ones (exposing the entire memory).

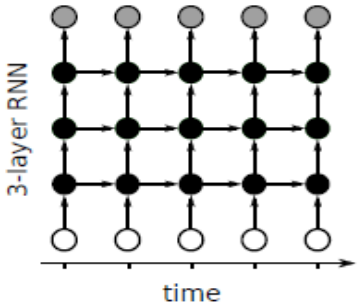
Deep Bidirectionsal LSTMs (DBLSTM)

- By stacking multiple LSTM layers on top of each other, DBLSTMs could also benefit from depth in space.
- Bidirectional LSTM: process information in both directions with two separate hidden layers, which are then fed forwards to the same output layer, providing it with access to the past and future context of every point in the sequence.
- BLSTM outperform unidirectional LSTMs and standard RNNs and it is also much faster and more accurate.
- DBLSTMs equations: repeat the equations of LSTM for every layer, and replace each hidden state, c_t^l, in every layer with the forward and backward states, $\vec{c}_t^l$ and $\overleftarrow{c}_t^l$, in a way that every hidden layer receives input from both the forward and backward layers at the level below.

DEEP RECURRENT NEURAL NETWORKS (DRNN)

- Standard RNN: the information only passes through one layer of processing before going to the output.
- In sequence tasks we usually process information at several time scales.
- DRNN: a combination of the concepts of deep neural networks (DNN) with RNNs.
- By stacking RNNs, every layer is an RNN in the hierarchy that receives the hidden state of the previous layer as input.
- Processing of different time scales at different levels, and therefore a temporal hierarchy will be created.





DRNN Training

- Use stochastic gradient decent for optimization:
  - $\theta(j+1) = \theta(j) - \eta_0 \left(1 - \frac{j}{T}\right) \frac{\nabla_\theta(j)}{\|\nabla_\theta(j)\|}$
    - $\theta(j)$: the set of all trainable parameters after j updates
    - $\nabla_\theta(j)$: the gradient of a cost function with respect to this parameter set, as computed on a randomly sampled part of the training set.
    - T: the number of batches
    - η_0: the learning rate is set at an initial value which decreases linearly with each subsequent parameter update.
- Incremental layer-wise method: train the full network with BPTT and linearly reduce the learning rate to zero before a new layer is added. After adding a new layer the previous output weights will be discarded, and new output weights are initialized connecting from the new top layer.
- For DRNN-AO, we test the influence of each layer by setting it to zero, assuring that model is efficiently trained.

Thank you

What is Convolutional Neural Network?

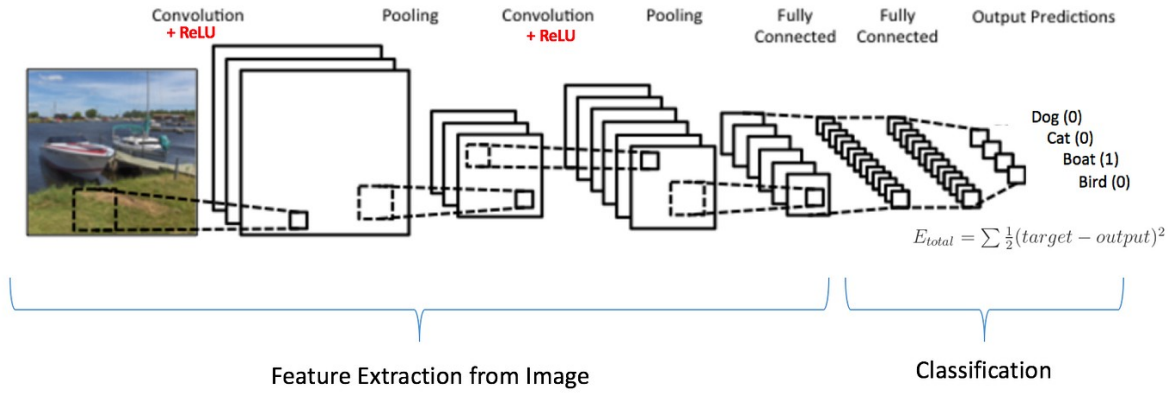
Mankind is an awesome natural machine and is capable of looking at multiple images every second and processing them without realizing how the processing is done. But the same is not with machines.

The first step in image processing is to understand, how to represent an image so that the machine can read it.

Every image is a cumulative arrangement of dots (a pixel) arranged in a special order. If you change the order or color of a pixel, the image would change as well.

Definition 1: A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

Definition 2: A convolutional neural network is a specific kind of neural network with multiple layers. It processes data that has a grid-like arrangement then extracts important features. One huge advantage of using CNNs is that you don't need to do a lot of pre-processing on images.



Three basic components to define a basic convolutional neural network.

- ***The Convolutional Layer***
- ***The Pooling layer***
- ***The Output layer***

The Convolutional Layer :

In this layer if we have an image of size 66. *We define a weight matrix that extracts certain features from the images*

INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429

We have initialized the weight as a 3x3 matrix. This weight shall now run across the image such that all the pixels are

covered at least once, to give a convolved output. The value 429 above, is obtained by adding the values obtained by element-wise multiplication of the weight matrix and the highlighted 3x3 part of the input image.

INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429

The 6x6 image is now converted into a 4x4 image. Think of a weight matrix like a paintbrush painting a wall. The brush first paints the wall horizontally and then comes down and paints the next row horizontally. Pixel values are used again when the weight matrix moves along the image. This basically enables parameter sharing in a convolutional neural network.

Let’s see what this looks like in a real image.

Original Image

Convoluted image

- The weight matrix behaves like a filter in an image, extracting particular information from the original image matrix.
- A weight combination might be extracting edges, while another one might a particular color, while another one might just blur the unwanted noise.
- The weights are learned such that the loss function is minimized and extracts feature from the original image which helps the network incorrect prediction.
- When we use multiple convolutional layers, the initial layer extracts more generic features, and as the network gets deeper the features get complex.

Let us understand some concepts here before we go further deep

What is Stride?

As shown above, the filter or the weight matrix we moved across the entire image moving one pixel at a time. If this is a hyperparameter to move the weight matrix 1 pixel at a time across an image it is called the stride of 1. Let us see for stride of 2 how it looks.

INPUT IMAGE

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

WEIGHT

1	0	1
0	1	0
1	0	1

429

Stride of 2

As you can see the size of the image keeps on reducing as we increase the stride value.

Padding the input image with zeros across it solves this problem for us. We can also add more than one layer of zeros around the image in case of higher stride values.

0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

Padding in CNN

We can see how the initial shape of the image is retained after we padded the image with a zero. This is known as the same padding since the output image has the same size as the input.

0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139

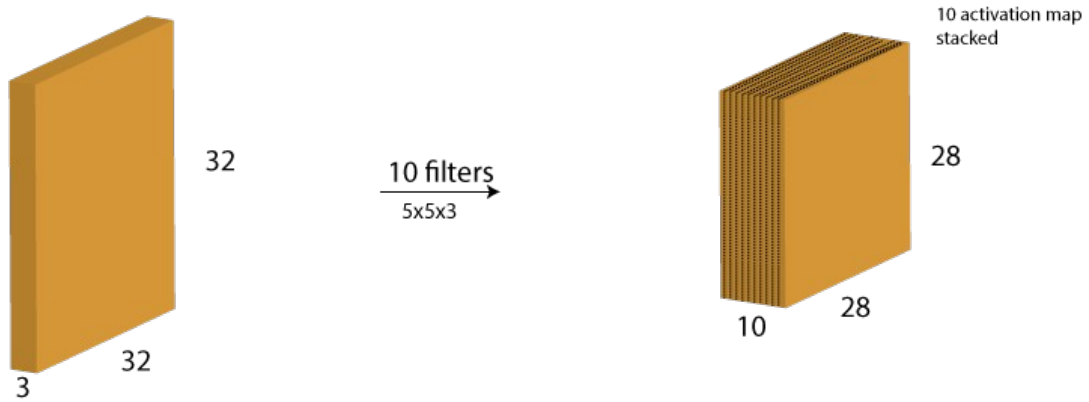
This is known as the same padding (which means that we considered only the valid pixels of the input image). The middle 4x4 pixels would be the same. Here we have retained more information from the borders and have also preserved the size of the image.

Having Multiple filters & the Activation Map

- The depth dimension of the weight would be the same as the depth dimension of the input image.
- The weight extends to the entire depth of the input image.
- Convolution with a single weight matrix would result into a convolved output with a single depth dimension. In the case of multiple filters, all have the same dimensions applied together.
- The output from each filter is stacked together forming the depth dimension of the convolved image.

Suppose we have an input image of size 32x32x3. And we apply 10 filters of size 5x5x3 with valid padding. The output would have the dimensions as 28x28x10.

You can visualize it as –



Activation map

This activation map is the output of the convolution layer.

The Pooling Layer

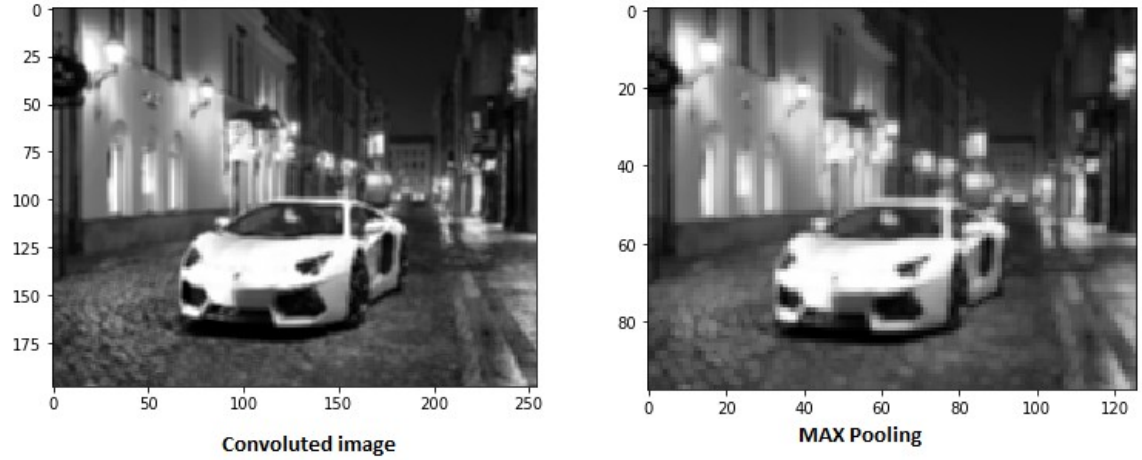
If images are big in size, we would need to reduce the number of trainable parameters. For this, we need to use pooling layers between convolution layers. Pooling is used for reducing the spatial size of the image and is implemented independently on each depth dimension resulting in no change in image depth. Max pooling is the most popular form of pooling layer.

429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

792	856
913	851

Here we have taken stride as 2, while pooling size also as 2. The max-pooling operation is applied to each depth dimension of the convolved output. As you can see, the 4x4 convolved output has become 2x2 after the max-pooling operation.

Let’s see how max-pooling looks on a real image.



In the above image, we have taken a convoluted image and applied max pooling on it which resulted in still retaining the image information that is a car but if we closely observe the dimensions of the image are reduced to half which basically means we can reduce the parameters to a great number.

There are other forms of pooling like average pooling, and L2 norm pooling.

Output dimensions

It is tricky at times to understand the input and output dimensions at the end of each convolution layer. For this, we will use three hyperparameters that would control the size of the output volume.

1. No. of Filter: The depth of the output volume will be equal to the number of filters applied. The depth of the activation map will be equal to the number of filters.

2. Stride: When we have a stride of one we move across and down a single pixel. With higher stride values, we move a large number of pixels at a time and hence produce smaller output volumes.
3. Zero padding: This helps us to preserve the size of the input image. If a single zero padding is added, a single stride filter movement would retain the size of the original image.

We can apply a simple formula to calculate the output dimensions.

The spatial size of the output image can be calculated as:

([W-F+2P]/S)+1

Where: W: Input volume size
F: Size of the filter
P: Number of padding applied
S: Number of strides.

where W is the input volume size, F is the size of the filter, P is the number of padding applied S is the number of strides.

Let us take an example of an input image of size 64x64x3, we apply 10 filters of size 3x3, with a single stride and no zero padding.

Here W=64, F=3, P=0 and S=1. The output depth will be equal to the number of filters applied i.e. 10.

The size of the output volume will be ([64–3+0]/1)+1 = 62. Therefore the output volume will be 62x62x10.

The Output layer

- With a number of layers of convolution and padding, we need the output in the form of a class.
- To generate the final output we need to apply a fully connected layer to generate an output equal to the number of classes we need.
- Convolution layers generate 3D activation maps while we just need the output as to whether or not an image belongs to a particular class.
- The Output layer has a loss function like categorical cross-entropy, to compute the error in prediction. Once the forward pass is complete the backpropagation begins to update the weight and biases for error and loss reduction.

Summary:

- Pass an input image to the first convolutional layer. The convoluted output is obtained as an activation map. The filters applied in the convolution layer extract relevant features from the input image to pass further.
- Each filter shall give a different feature to aid the correct class prediction. In case we need to retain the size of the image, we use the same padding(zero

padding), otherwise valid padding is used since it helps to reduce the number of features.

- Pooling layers are then added to further reduce the number of parameters
- Several convolution and pooling layers are added before the prediction is made. Convolutional layer help in extracting features. As we go deeper into the network more specific features are extracted as compared to a shallow network where the features extracted are more generic.
- The output layer in a CNN as mentioned previously is a fully connected layer, where the input from the other layers is flattened and sent so as the transform the output into the number of classes as desired by the network.
- The output is then generated through the output layer and is compared to the output layer for error generation. A loss function is defined in the fully connected output layer to compute the mean square loss. The gradient of error is then calculated.
- The error is then backpropagated to update the filter(weights) and bias values.
- One training cycle is completed in a single forward and backward pass.

What Is Deep Learning?

Deep learning is a type of machine learning that uses artificial neural networks to enable digital systems to learn and make decisions based on unstructured, unlabeled data.

In general, machine learning trains AI systems to learn from acquired experiences with data, recognize patterns, make recommendations, and adapt. With deep learning in particular, instead of just responding to sets of rules, digital systems build knowledge from examples and then use that knowledge to react, behave, and perform like humans.

Why deep learning matters

Data scientists and developers use deep learning software to train computers to analyze big and complex data sets, complete complicated and nonlinear tasks, and respond to text, voice, or images, often faster and more accurately than humans. These capabilities have many practical applications and have made many modern innovations possible. For example, deep learning is what driverless cars use to process images and distinguish pedestrians from other objects on the road or what your smart home devices use to understand your voice commands.

Deep learning matters because as data volumes increase and computing capacity becomes more powerful and affordable, companies across retail, healthcare, transportation, manufacturing, technology, and other sectors are investing in deep learning to drive innovation, unlock opportunities, and stay relevant.

How deep learning works

Deep learning works by relying on neural network architectures in multiple layers, high-performance graphics processing units deployed in the cloud or on clusters, and large volumes of labeled data to achieve very high levels of text, speech, and image recognition accuracy. All that power can help your developers create digital systems with something like human intelligence and streamline time to value by accelerating model training from weeks to hours.

For example, a driverless car model might require thousands of video hours and millions of images to train. Without deep learning, this level of training couldn’t be done at scale.

What is a deep learning framework?

To make complex machine learning models easier to implement, developers turn to deep learning frameworks like TensorFlow or PyTorch. These frameworks help streamline the process of collecting data which can then be used to train neural networks. In addition, accelerators like ONNX Runtime can be used with these frameworks to accelerate training and inferencing models.

Training deep learning models

There are different strategies and methods for training deep learning models. Let's take a closer look at a few of them.

Supervised learning

With supervised learning, an algorithm is trained on datasets that are labeled. This means that when the algorithm makes a determination about a piece of information, it can use the labels included with the data to check if that determination is correct. With supervised learning, the data that models are trained on must be provided by humans, who label the data before using it to train the algorithm.

Unsupervised learning

With unsupervised learning, algorithms are trained on data that does not contain labels or information that the algorithm can use to check its determinations against. Instead, the system sorts and classifies the data based on the patterns that it recognizes on its own.

Reinforcement learning

With reinforcement learning, a system solves tasks using trial and error to make series of decisions in sequence and achieve an intended outcome even in an environment that is not straightforward. With reinforcement learning, the algorithm doesn't use datasets to make determinations, but rather information that it gathers from an environment.

Deep reinforcement learning

When deep learning and reinforcement learning techniques are combined, they create a type of machine learning called deep reinforcement learning. Deep reinforcement learning uses the same trial-and-error decision making and complex goal achievement as reinforcement learning, but also relies on deep learning capabilities to process and make sense of large amounts of unstructured data.

What is deep learning used for?

Deep learning is used within businesses in a variety of industries for a wide range of use cases. Here are some examples of how deep learning is commonly used:

Image, speech, and emotion recognition

Deep learning software is used to increase image, speech, and emotion recognition accuracy and to enable photo searches, personal digital assistants, driverless vehicles, public safety, digital security, and other intelligent technologies.

Tailored experiences
Streaming services, e-commerce retailers, and other businesses use deep learning models to drive automated recommendations for products, movies, music, or other services and to perfect customer experiences based on purchase histories, past behavior, and other data.

Chatbots
Savvy businesses use deep learning to power text- or voice-activated online chatbots for frequently asked questions, routine transactions, and especially for customer support. They replace teams of service agents and queues of waiting customers with automated, contextually appropriate, and useful responses.

Personal digital assistants
Voice-activated personal digital assistants use deep learning to understand speech, respond appropriately to queries and commands in natural language, and even crack wise occasionally.

Driverless vehicles
The unofficial representative for AI and deep learning, self-driving cars use deep learning algorithms to process multiple dynamic data feeds in split seconds, never have to ask for directions, and react to the unexpected—faster than a human driver.

What is Recurrent Neural Network?

If you want to get into machine learning, recurrent neural networks are a powerful technique that is important to understand. If you use a smartphone or frequently surf the internet, odd's are you've used applications that leverages RNN's. Recurrent neural networks are used in speech recognition, language translation, stock predictions; It's even used in image recognition to describe the content in pictures.

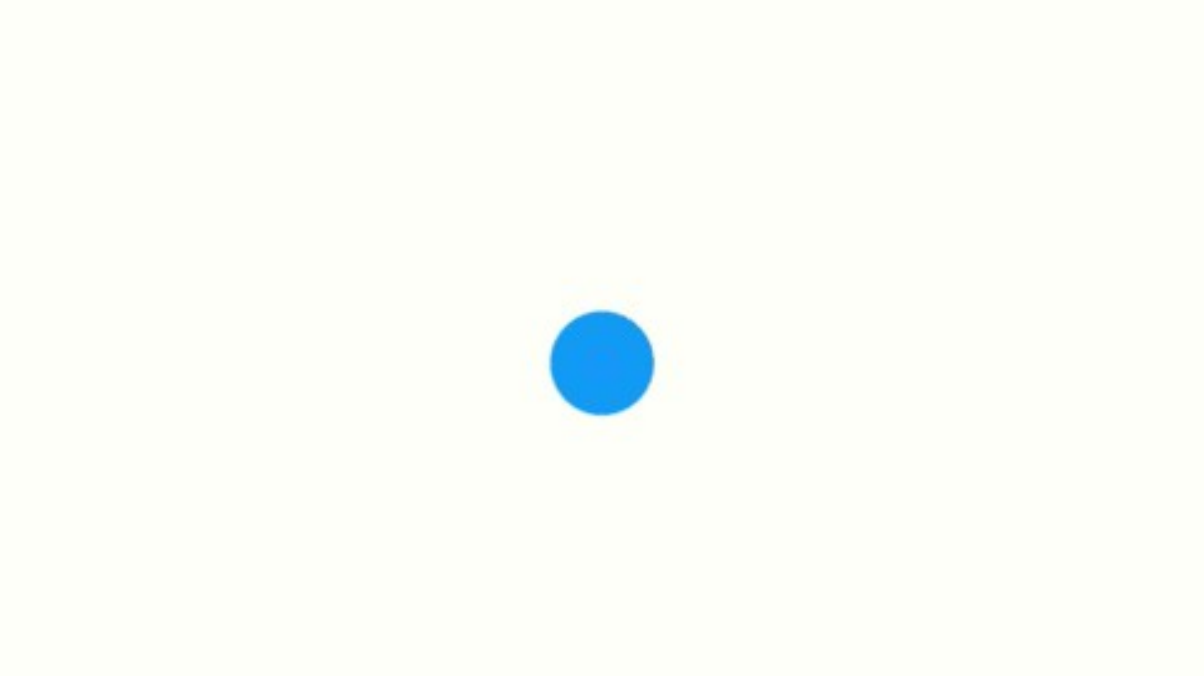
Sequence Data

RNN's are neural networks that are good at modeling sequence data. To understand what that means let's do a thought

experiment. Say you take a still snapshot of a ball moving in time.

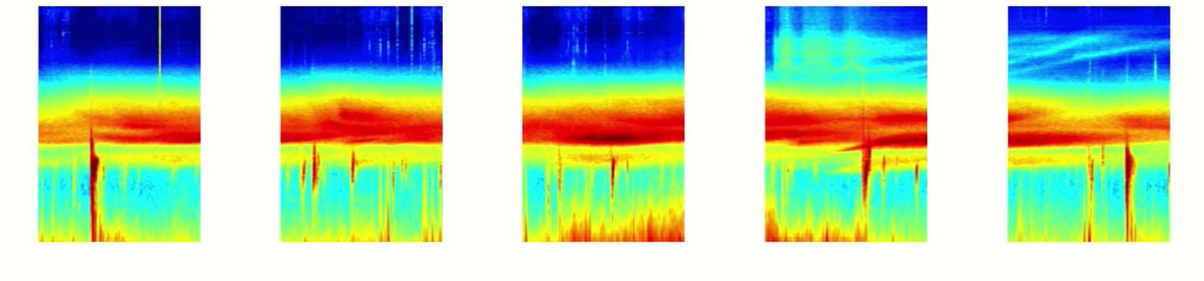
Let's also say you want to predict the direction that the ball was moving. So with only the information that you see above, how would you do this? Well, you can go ahead and take a guess, but any answer you'd come up with would be that, a random guess. Without knowledge of where the ball has been, you wouldn't have enough data to predict where it's going.

If you record many snapshots of the ball's position in succession, you will have enough information to make a better prediction.



So this is a sequence, a particular order in which one thing follows another. With this information, you can now see that the ball is moving to the right.

Sequence data comes in many forms. Audio is a natural sequence. You can chop up an audio spectrogram into chunks and feed that into RNN's.



Audio spectrogram chopped into chunks

Text is another form of sequences. You can break Text up into a sequence of characters or a sequence of words.

Sequential Memory

RNN's are good at processing sequence data for predictions. But how??

Well, they do that by having a concept called sequential memory. To get a good intuition behind what sequential memory means…

Say the alphabet in your head.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

If you were taught this specific sequence, it should come quickly to you.

Now try saying the alphabet backward.

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

This is much harder. Unless you've practiced this specific sequence before, you'll likely have a hard time.

Here's a fun one, start at the letter F.

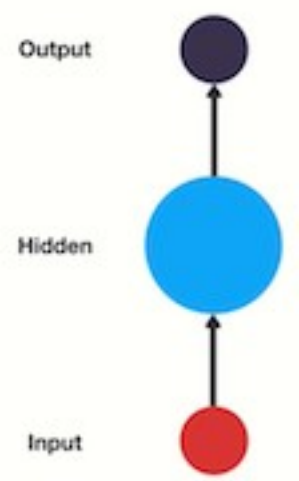
F

At first, you'll struggle with the first few letters, but then after your brain picks up the pattern, the rest will come naturally.

So there is a very logical reason why this can be difficult. You learn the alphabet as a sequence. Sequential memory is a mechanism that makes it easier for your brain to recognize sequence patterns.

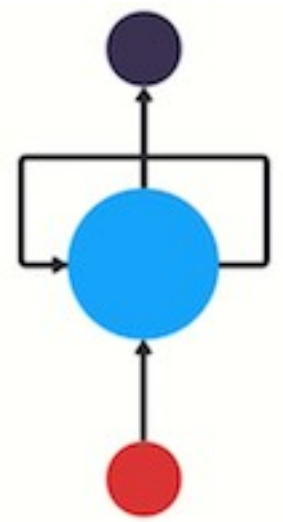
Recurrent Neural Networks

RNN's have this abstract concept of sequential memory, but how the heck does an RNN replicate this concept? Well, let's look at a traditional neural network also known as a feed-forward neural network. It has its input layer, hidden layer, and the output layer.



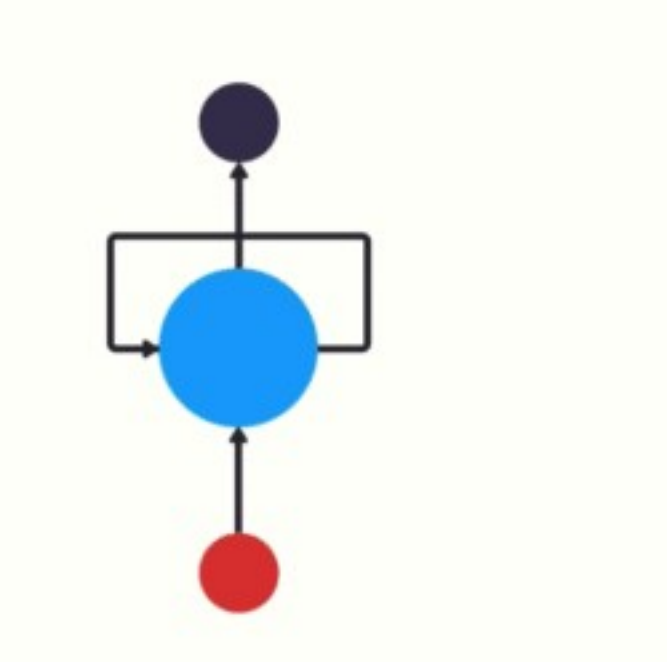
Feed Forward Neural Network

How do we get a feed-forward neural network to be able to use previous information to effect later ones? What if we add a loop in the neural network that can pass prior information forward?



Recurrent Neural Network

And that’s essentially what a recurrent neural network does. An RNN has a looping mechanism that acts as a highway to allow information to flow from one step to the next.



Passing Hidden State to next time step

This information is the hidden state, which is a representation of previous inputs. Let’s run through an RNN use case to have a better understanding of how this works.

Let’s say we want to build a chatbot. They’re pretty popular nowadays. Let’s say the chatbot can classify intentions from the users inputted text.

Classifying intents from users inputs

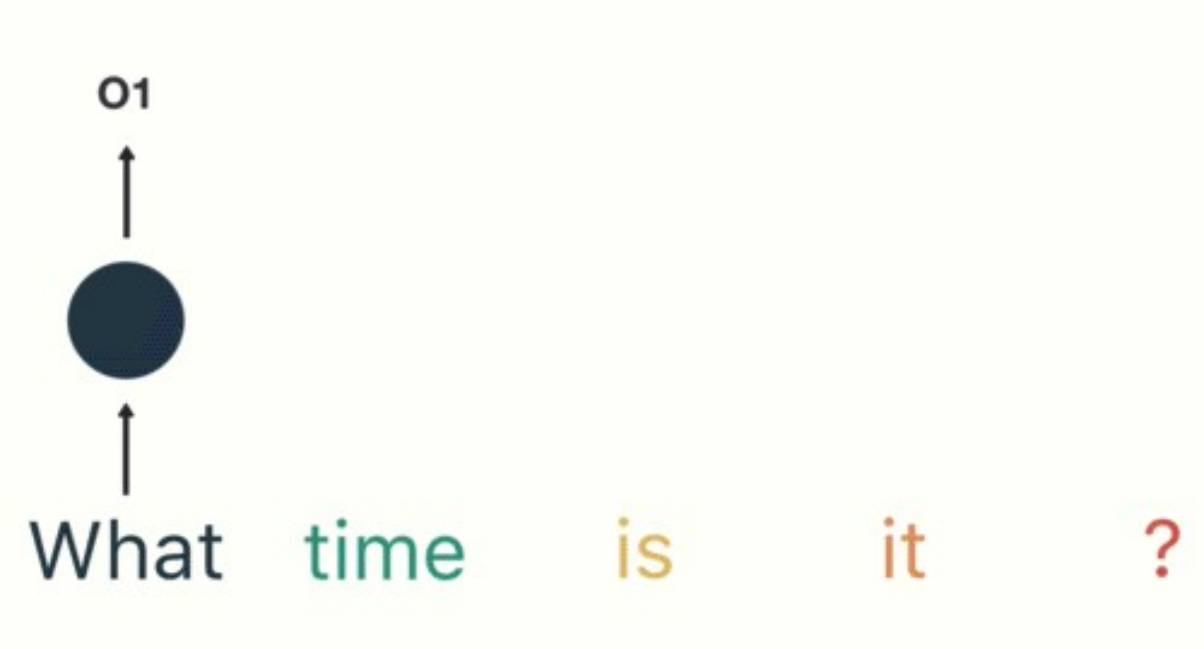
To tackle this problem. First, we are going to encode the sequence of text using an RNN. Then, we are going to feed the RNN output into a feed-forward neural network which will classify the intents.

So a user types in… ***what time is it?***. To start, we break up the sentence into individual words. RNN’s work sequentially so we feed it one word at a time.

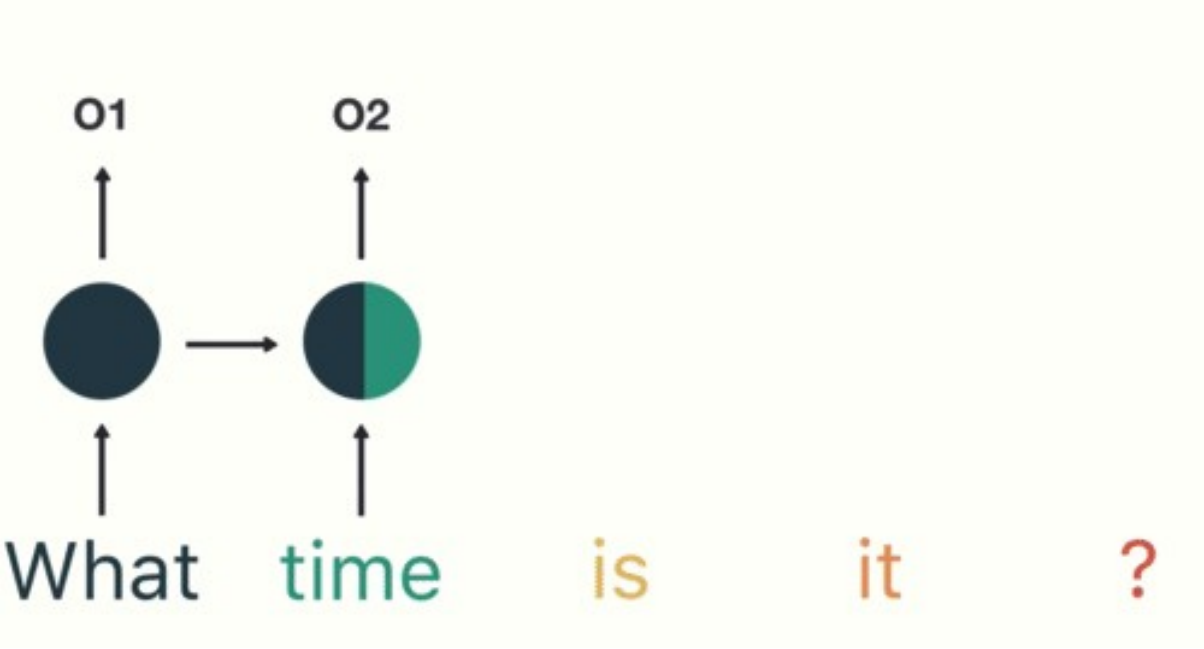
Breaking up a sentence into word sequences

The first step is to feed “What” into the RNN. The RNN encodes “What” and produces an output.

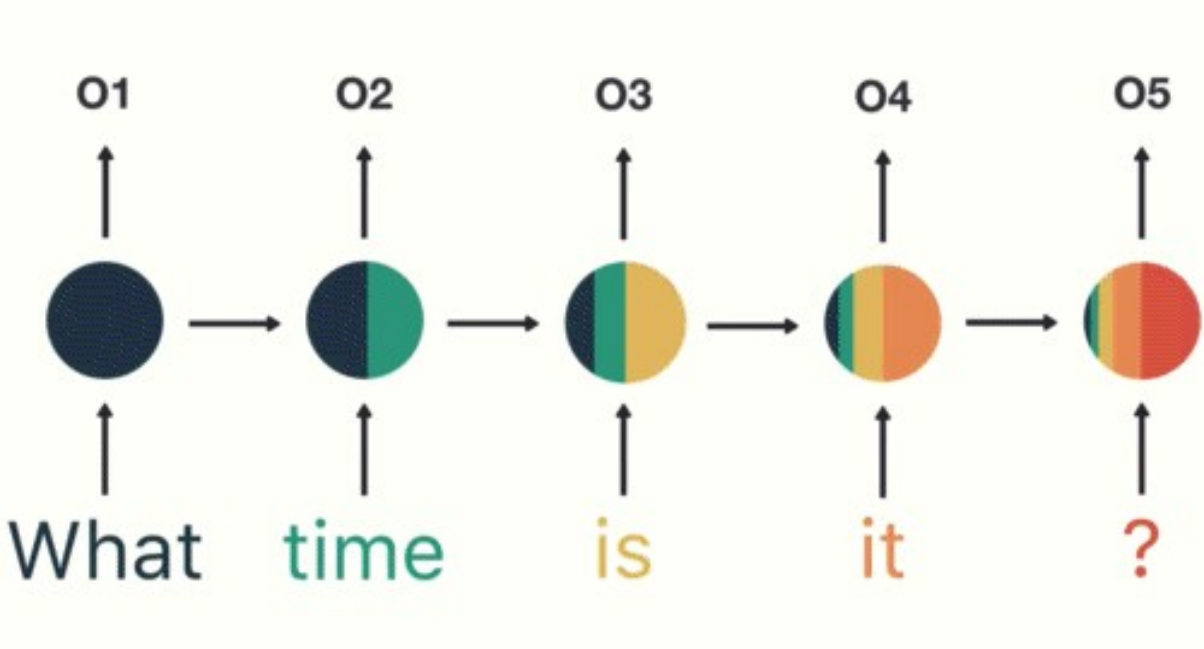
For the next step, we feed the word “time” and the hidden state from the previous step. The RNN now has information on both the word “What” and “time.”



We repeat this process, until the final step. You can see by the final step the RNN has encoded information from all the words in previous steps.



Since the final output was created from the rest of the sequence, we should be able to take the final output and pass it to the feed-forward layer to classify an intent.



For those of you who like looking at code here is some python showcasing the control flow.

```
rnn = RNN()
ff = FeedForwardNN()
hidden_state =[0.0, 0.0, 0.0, 0.0]

for word in input:
    output, hidden_state = rnn(word, hidden_state)

prediction = ff(output)
```

Pseudo code for RNN control flow

First, you initialize your network layers and the initial hidden state. The shape and dimension of the hidden state will be dependent on the shape and dimension of your recurrent neural network. Then you loop through your inputs, pass the

word and hidden state into the RNN. The RNN returns the output and a modified hidden state. You continue to loop until you're out of words. Last you pass the output to the feedforward layer, and it returns a prediction. And that's it! The control flow of doing a forward pass of a recurrent neural network is a for loop.

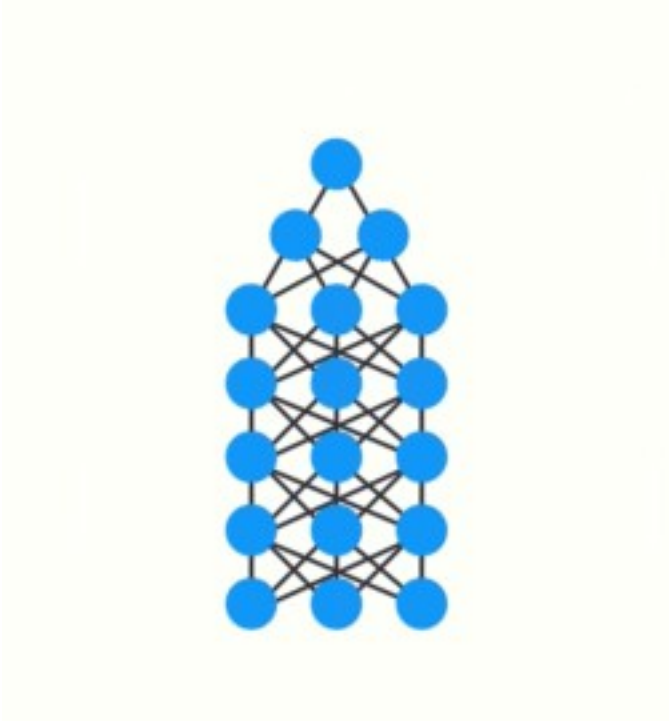
Vanishing Gradient

You may have noticed the odd distribution of colors in the hidden states. That is to illustrate an issue with RNN's known as short-term memory.

Final hidden state of the RNN

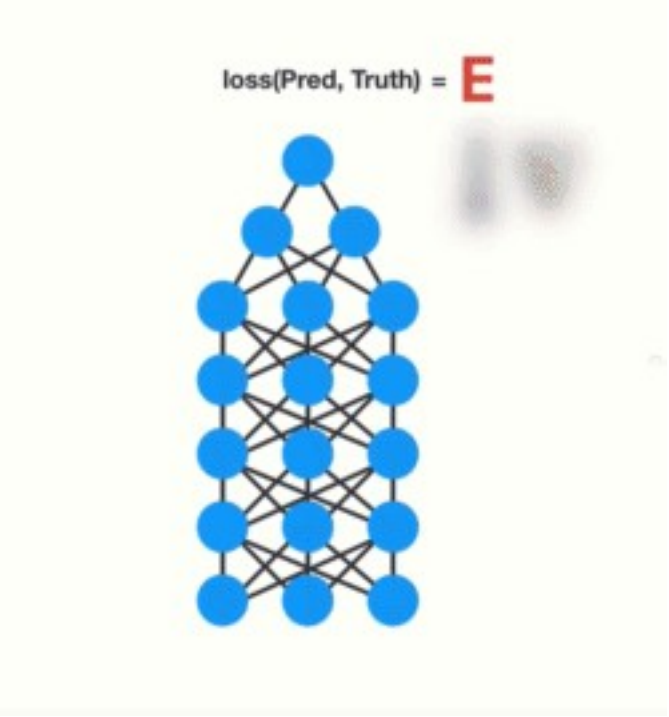
Short-term memory is caused by the infamous vanishing gradient problem, which is also prevalent in other neural network architectures. As the RNN processes more steps, it has troubles retaining information from previous steps. As you can see, the information from the word "what" and "time" is almost non-existent at the final time step. Short-Term memory and the vanishing gradient is due to the nature of back-propagation; an algorithm used to train and optimize neural networks. To understand why this is, let's take a look at the effects of back propagation on a deep feed-forward neural network.

Training a neural network has three major steps. First, it does a forward pass and makes a prediction. Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing. Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.



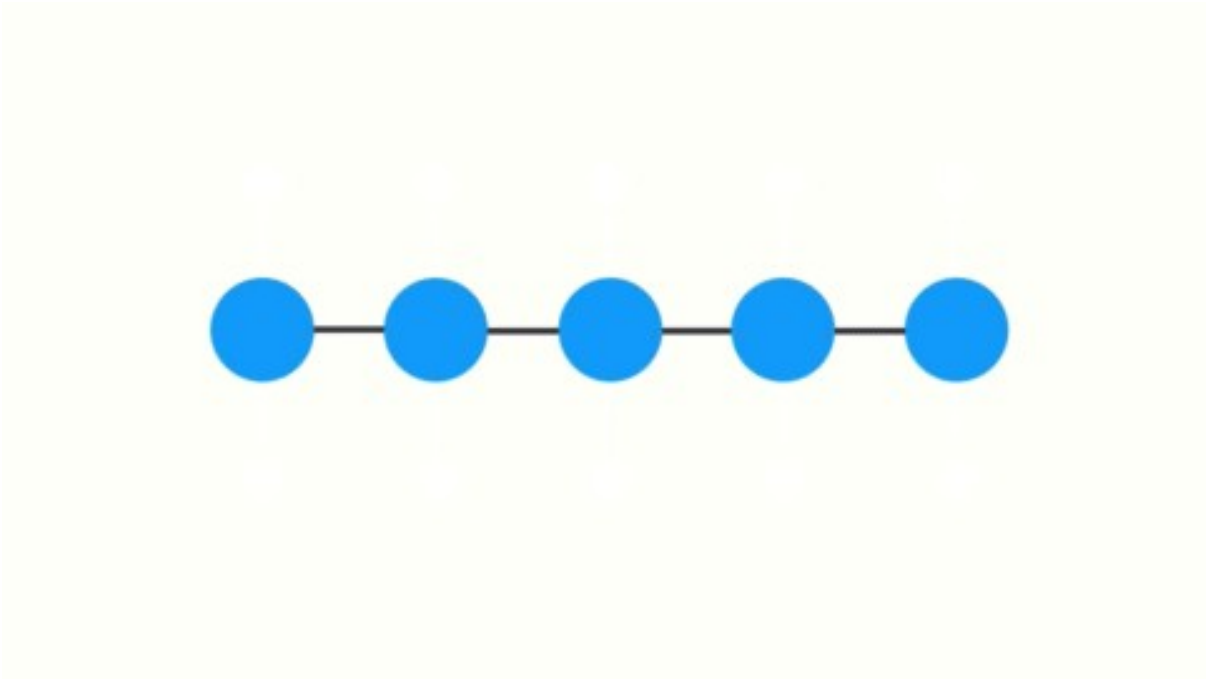
The gradient is the value used to adjust the networks internal weights, allowing the network to learn. The bigger the gradient, the bigger the adjustments and vice versa. Here is where the problem lies. When doing back propagation, each node in a layer calculates it's gradient with respect to the effects of the gradients, in the layer before it. So if the adjustments to the layers before it is small, then adjustments to the current layer will be even smaller.

That causes gradients to exponentially shrink as it back propagates down. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that's the vanishing gradient problem.



Gradients shrink as it back-propagates down

Let's see how this applies to recurrent neural networks. You can think of each time step in a recurrent neural network as a layer. To train a recurrent neural network, you use an application of back-propagation called back-propagation through time. The gradient values will exponentially shrink as it propagates through each time step.



Gradients shrink as it back-propagates through time

Again, the gradient is used to make adjustments in the neural networks weights thus allowing it to learn. Small gradients mean small adjustments. That causes the early layers not to learn.

Because of vanishing gradients, the RNN doesn't learn the long-range dependencies across time steps. That means that there is a possibility that the word "what" and "time" are not considered when trying to predict the user's intention. The network then has to make the best guess with "is it?". That's pretty ambiguous and would be difficult even for a human. So not being able to learn on earlier time steps causes the network to have a short-term memory.

LSTM's and GRU's

Ok so RNN's suffer from short-term memory, so how do we combat that? To mitigate short-term memory, two specialized recurrent neural networks were created. One called Long Short-Term Memory or LSTM's for short. The other is Gated Recurrent Units or GRU's. LSTM's and GRU's essentially function just like RNN's, but they're capable of learning long-term dependencies using mechanisms called "gates." These gates are different tensor operations that can learn what information to add or remove to the hidden state. Because of this ability, short-term memory is less of an issue for them. If you'd like to learn more about LSTM's and GRU's you can view my post on them.

That wasn't too bad

To sum this up, RNN's are good for processing sequence data for predictions but suffers from short-term memory. The short-term memory issue for vanilla RNN's doesn't mean to skip them entirely and use the more evolved versions like LSTM's or GRU's. RNN's have the benefit of training faster and uses less computational resources. That's because there are fewer tensor operations to compute. You should use LSTM's or GRU's when you expect to model longer sequences with long-term dependencies.