Artificial neural networks (ANNs)

SUMMARY

1. Artificial neural networks (ANNs)

- Biological inspirations
  - Properties of the brain
    - it can **learn**, reorganize itself from experience
    - it **adapts** to changing conditions
    - it is **robust** and fault tolerant
- **Robustness** in ML?
  - the degree to which a model's performance changes when using new data
    - noise
  - ideally, performance should not deviate significantly
  - how to test it?
    - CV, CIs

- **Two types of learning in NNs**
  - **supervised**
    - FFNNs, RBFNs, RNNs, DNNs, CNNs, …
  - **unsupervised**
    - self-organizing maps (SOM)
    - Hebbian learning
    - **autoencoders (**AEs**)**
      - self-supervised learning
      - **encoder-decoder** architecture

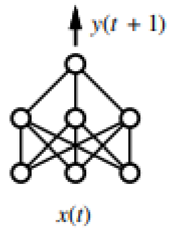
- **Characteristics** of supervised NN learning models
  - represent (complex) non-linear functions
  - **eager** inductive learning models
  - appropriate for **offline** and **online** learning
  - used for **classification** and **regression**

- black-box models
  - achieving **interpretability** – explainable neural models
- are **robust** to noisy data
- need for multiple and representative examples
- enable to model complex **static** phenomena (Feed-forward neural networks – FFNNs) as well as **dynamic** ones (e.g., Recurrent neural networks – RNNs)
  - **static** phenomena
    - time has no role
  - **dynamic** phenomena
    - temporal events
    - image and video recognition, time series, handwritten recognition, motion detection, signal processing, stock market prediction, speech recognition, aso
- **NNs require**
  - a good representation of the data
  - training vectors must be statistically representative of the entire input space
  - the use of NNs needs a good comprehension of the problem
  - NNs require good *data preprocessing* (e.g., data normalization for numerical data)
    - the range of all features should be normalized
      - comparable range for the features
      - transpose the input variables into the range of the activation function codomain (i.e. for *logistic* [0, 1], for *tanh* [-1, 1])
    - speeds up learning, faster convergence
- Research domain: **NAS** (*Neural Architecture Search*)
  - subfield of **automated machine learning (AutoML)**
    - process of automating the tasks of applying ML to real-world problems
  - technique for automating the design of ANNs (both classical and deep)
    - **search space** defines the type(s) of ANN that can be designed and optimized
    - **search strategy** defines the approach used to explore the search space.
    - **performance estimation strategy** evaluates the performance of a possible ANN from its design (without constructing and training it).
  - RL, Hill climbing, Evolutionary algorithms, PSO, Multi-objective optimization,…

2. Types of NNs for supervised learning

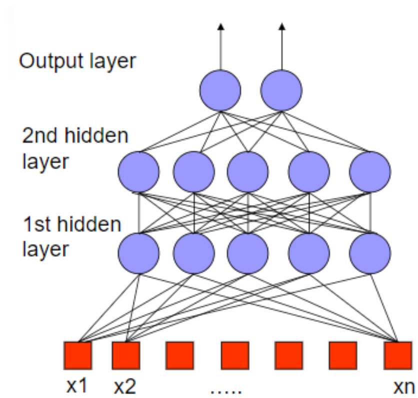
1. Feed-forward neural networks (FFNNs)

- an ANN where connections between the units do not form a directed cycle
- the first and simplest form of ANN
- the information moves in only one direction, forward, from the input nodes through the hidden nodes and the output nodes
- there are **no cycles** or **loops** in the network (time has no role)



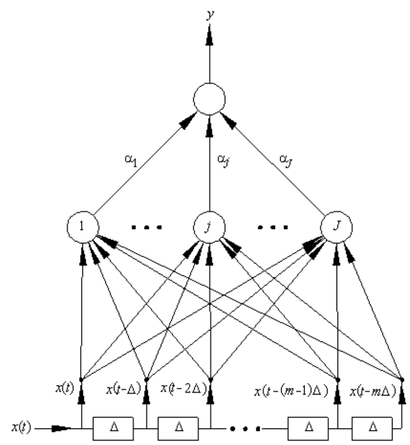
(a) Feedforward network

- types of FFNNs
  - **<u>Multilayer perceptron (MLP)</u>**
    - consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one



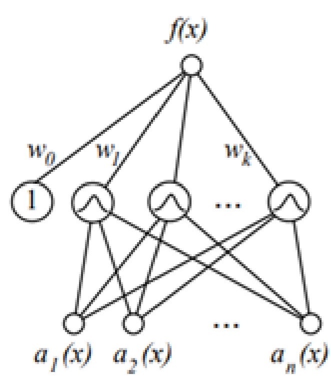
    - except for the input nodes, each node is a *neuron* (processing element) with a *nonlinear* activation function
    - utilizes a supervised learning technique called *backpropagation* for training the network
    - MLP is a modification of the standard linear perceptron and can distinguish data that are not linearly separable

  - **<u>Time delay neural networks (TDNNs)</u>**
    - theory
    - an alternative to an NN architecture whose purpose is to work on continuous data
    - learning a **temporal** sequence of events
    - maps a finite time sequence $\{X(t), X(t-\Delta), X(t-2\cdot\Delta) \dots X(t-m\cdot\Delta)\}$ into a single output *y* (this can be generalized for the case when *x* and/or *y* are vectors)



- Pytorch
- helpful in many applications like:
    - time series predictions
    - online spell check
    - speech recognition (generation)
    - image analysis
    - aso
- Deep TDNN

- **<u>Radial basis function networks (RBFNs)</u>**
    - specific feed-forward architecture
    - 1 hidden layer
    - Gaussian activation function at the hidden layer



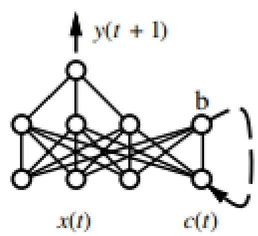
[1]

- connected to the **Instance based learning** (IBL) literature, but **eager** instead of **lazy**
    - computes a global approximation to the target function *f*, in terms of a linear combination of local approximations ("**kernel**" functions)
    - is a different kind of two-layer neural network
        i. the hidden units compute the values of kernel functions (local approximations)
        ii. the output unit computes *f* as a linear combination of kernel functions

- applications
  - fault diagnosis
  - forecasting
  - image classification
  - image reconstruction
  - aso

2. Recurrent neural networks (RNNs)

- **sequential** or time series data
  - RNNs are a variant of the conventional FFNNs that can deal with sequential data and can be trained to hold knowledge about the past.
    - a mechanism is required to retain past or historical information to forecast future values.
- connections between units form a **directed cycle**
  - this creates an internal state of the network which allows to exhibit **dynamic temporal behavior**
  - can model systems with internal state (dynamic ones)



(b) Recurrent network [1]

- unlike FFNNs, RNNs can use their **internal memory** to process arbitrary sequences of inputs
- appropriate for time series data
  - learning is **sequential**
- applications:
  - handwritten recognition
  - motion detection
  - signal processing
  - text generation
  - time series prediction
  - stock market forecasting
  - aso
- the vanishing gradient problem of RNNs cause the network not to learn much → specialised versions of RNN
  - **LSTM**
  - **GRU (Gated Recurrent Unit)**

Long-Short Term Memory networks (LSTMs)

- a type of RNN
- this model is an attempt to allow the unit activations to retain important information over a much longer period of time
- handling the *vanishing* and *exploding* gradient problem
  - includes specialized "gates" within its structure to manage long-term and short-term memory
- applications:
  - language learning
  - robot control
  - music composition
  - speech and handwriting recognition
  - video processing
  - …

- other architectures: DeepLSTM, ConvLSTM, BiLSTM (Bidirectional LSTM), ensemble of LSTMs

3. Deep Neural Networks (DNNs)

- multiple hidden layers
- can express easier complex functions
- a layer may be viewed as a "***feature hierarchy***"
- Classes of DNNs [4]
  - DNNs for supervised learning
  - DNNs for unsupervised or generative learning
    - *generative models* – can learn and mimic any distribution of data
      - Bolzmann Machines, Restricted Boltzmann Machines, Deep Belief Networks, Deep Bolzmann Machines
        - Diffusion models
          - probabilistic models
          - parameterized Markov chain trained using variational inference (VI) to produce samples matching the data after finite time
          - VI – approximating of probability densities through optimization
          - Image Synthesis, Video Generation, 3D Generation, Medical Analysis, Text Generation, Speech Synthesis, Time Series Generation, Molecule Design, Graph Generation, etc
          - Diffusion classifiers
      - *Generative adversarial networks* (GANs) [3]
        - two nets competing one against the other (generator/discriminator)
          - learn to generate new data
          - generating images, faces, photographies
        - bidirectional GAN
        - various types of GAN models

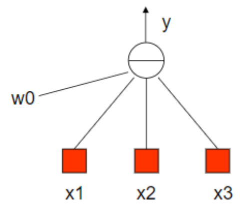
                - Conditional GAN, Cycle GAN, Convolutional GAN, etc
            - generative adversarial exploration for [reinforcement learning]
        - [Variational AEs] (VAEs)
            - probabilistic latent space
            - provides a statistical manner for describing the samples of the dataset in latent space
            - latent code generated by the encoder is a probabilistic encoding
                - VAEs express not just a single point in the latent space but a distribution of potential representations
        - Generative models in [reinforcement learning]
    - **research**: solving unsupervised learning problems with DNNs (e.g. ICA – independent component analysis, feature analysis, aso)
- Hybrid DNNs, ensemble of DNNs, fuzzy DNNs
- [Graph Neural Networks] (GNNs)
    - a type of DL model that can be used to learn from graph data

4. Convolutional Neural Networks (CNNs)

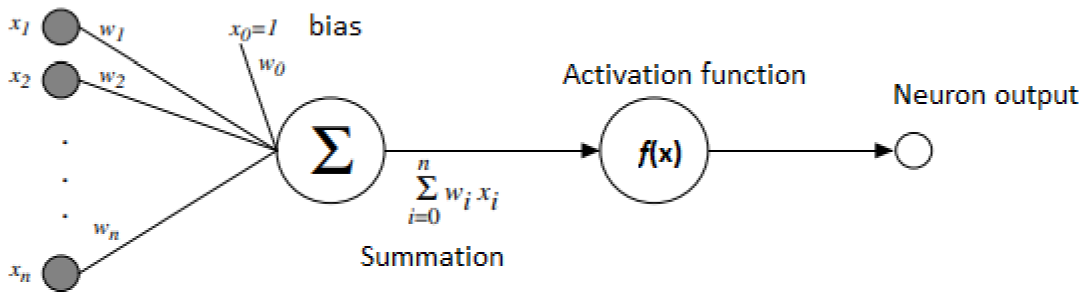
- inspired by the organization of the visual cortex (biological inspiration)
- applications:
    - [computer vision] [3]
    - natural language processing
        - e.g., [sentence classification]
    - video processing
    - object detection and recognition
- are deep FFNNs
    - **convolution**
        - from a dimension of an input, a filter is applied to it to take some of the interesting features from that dimension
- **GCN** - graph convolutional networks
    - graph-structured data
    - handle higher dimensional (non-grid) data
    - **applications**
        - [semi-supervised learning]
        - supervised learning ([text classification])
        - [unsupervised learning]
- **MobileNets** – efficient CNN architecture for mobile devices
- **low resource devices**
    - ***distillation***
        - compressing the knowledge from a large network into a smaller one
    - distilling the knowledge in a [NN] (Hinton, 2015), distilling knowledge from [GCN]
- Ensemble of CNNs

5. Classical NN architectures

- **Artificial neuron**
  - non-linear, parameterized function with restricted output range



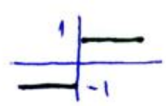
$$y = f\left(w_0 + \sum_{i=1}^{n-1} w_i x_i \right)$$



  - the output of the neuron is obtained by applying an (non-linear) **activation function *f*** on the linear combination of the neuron inputs

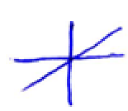
  - **Activation functions**
    - *Signum* – output range: -1, +1
      - does not have a derivative, undifferentiable in 0
      - perceptron

Signum $\quad f(x) = \begin{cases} +1 & x > 0 \\ -1 & \text{otherwise} \end{cases}$



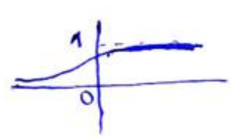
    - *Identity* – output range: (-∞, +∞)

Identity $\quad f(x) = x$



    - *Sigmoid* (*logistic*) – output range: (0, +1)
      - e.g. predict a probability

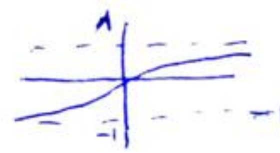
Sigmoid $\quad f(x) = \frac{1}{1+e^{-x}}$ $\quad (\sigma(x))$



- *Hyperbolic* – output range: (-1, +1)
  - smooth approximation for the perceptron function
  - learning smoother than the perceptron

Hyperbolic

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



- *Gaussian* – output range: (0, 1)
  - Radial basis function networks (RBFN)

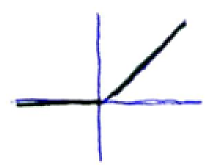
Gaussian

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- *ReLU* (Rectified Linear Unit) – output range: (0, ∞)
  - undifferentiable in 0
  - main advantages
    - it does not activate all the neurons at the same time
    - converge very fast
  - problem
    - “dying”/”dead” ReLU
    - neurons becoming inactive during training and consistently outputting zero => loss of learning capability
  - deep architectures

ReLU

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



- *Leaky ReLU*
  - introducing a small slope for negative values in the activation function => small gradient for negative inputs, preventing complete inactivity
  - allow a small, non-zero gradient when the unit is not active

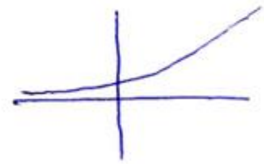
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \mathbf{0.01}x & \text{otherwise} \end{cases}$$

- *PReLU* (Parametric Rectified Linear Unit) – output range: (-∞, ∞)
  - improves ReLU and Leaky RELU
    - **idea**: making the coefficient of leakage (0.01) into a parameter that is learned along with the other neural network parameters:

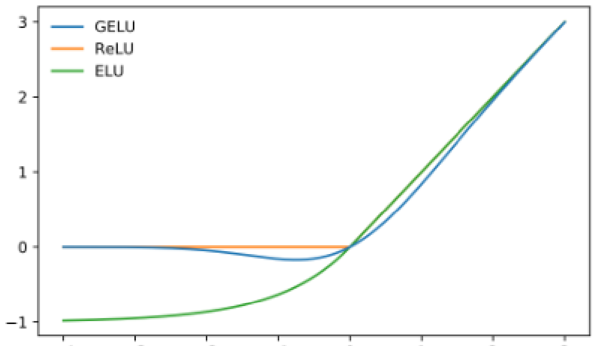
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

- *Soft plus* – output range: (0, ∞)

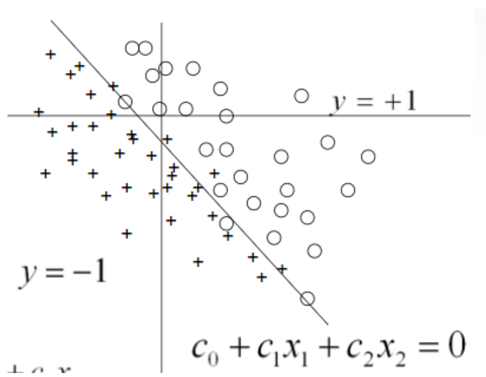
<u>Soft plus</u> $f(x) = \ln(1 + e^x)$



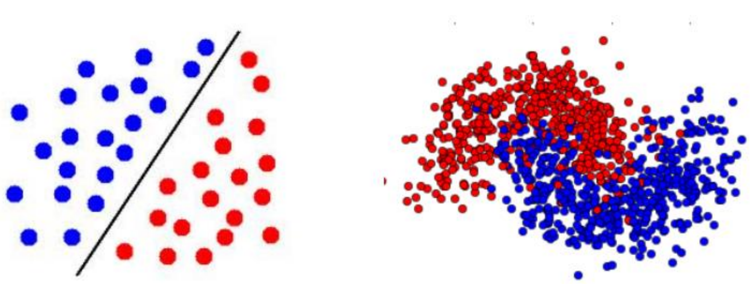
- *eLU* (Exponential Linear Unit)
- *GeLU* (Gaussian Error Linear Unit)



- **Perceptron**
  - represents a hyperplane decision surface in the high dimensional space of instances



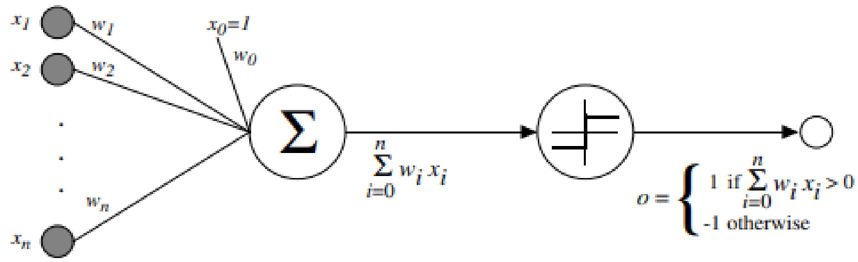
  - binary classification (outputs: -1, +1)



(a) (b)

(a) **linearly separable** data set (i..e, data set can be separated by a straight line)
(b) the classes are **not** linearly separable

- linear classifier
- appropriate for online learning



[1]

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

__**Learning task**__

- **training examples** $D = \{(x_d, t_d)\}_{d=1,s}$ $x_d = (x_{1d}, x_{2d}, \ldots x_{nd}) \in \Re^n$, $t_d \in \{-1,+1\}$
- **goal**
  - learn the separating hyperplane
  - hypothesis: $w=(w_0, w_1, \ldots w_n) \in \Re^{n+1}$
- **error function**
  - **online learning**
    $E_d(\vec{w}) = t_d - o_d$
  - **offline learning**
    $E(\vec{w}) = \frac{\sum_{d=1}^{s} |t_d - o_d|}{s}$

- weights initialization
  - small random values (or 0)

- **Training rule**

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o) x_i$$

Where:

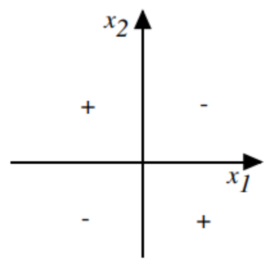
- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate* [1]

- **Linear classifier**

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

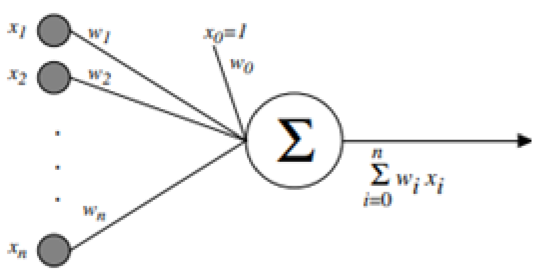
- Example - perceptron

- The perceptron is able represent some useful boolean function: AND, OR, ¬AND, ¬OR
- converges only if training data is linearly separable and the learning rate is sufficiently small (e.g., 0.1)
    - Perceptron convergence theorem Rosenblatt
        - for a finite set of linearly separable labeled examples, after a finite number of iterations, the algorithm yields a vector *w* that classifies perfectly all the examples.
    - XOR function is not representable using a perceptron ⇒ we need multilayered networks



XOR function

- **Linear unit**
    - consider a *linear unit*, whose output *o* is $o = w_0 + w_1 x_1 + \cdots + w_n x_n$



<u>Learning task</u>

- **training examples** $D = \{(x_d, t_d)\}$ $x_d = (x_{1d}, x_{2d}, \ldots x_{nd}) \in \Re^n$, $t_d \in \Re$
    - t_d represents the output of the neuron for the input instance d
- **goal**
    - learn the weights that minimize the squared error

      (1) over the entire training samples *D*

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- e.g., using *gradient descent optimization* for minimizing the error => **batch mode gradient descent**

(2) for each training sample d ∈ *D*

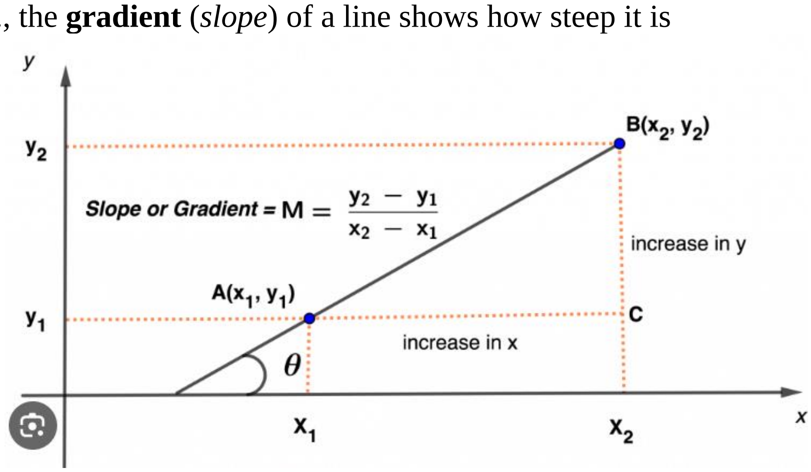
$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$$

- e.g., using *gradient descent optimization* for minimizing the error => **incremental (stochastic) gradient descent**
- online learning

- **Gradient descent**
  - the gradient of a function is a vector of *first derivatives* taken with respect to its constituent variables

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

  - the gradient specifies the direction that produces the steepest increase in E
    - - $\nabla E[\vec{w}]$ - the direction of steepest descent
  - e.g., the **gradient** (*slope*) of a line shows how steep it is



Gradient

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Training rule:

$$\Delta\vec{w} = -\eta\nabla E[\vec{w}]$$

i.e.,

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i}$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\ &= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\ &= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\ \frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \end{aligned}$$

[1]

- Training rule for
  - **batch mode gradient descent**

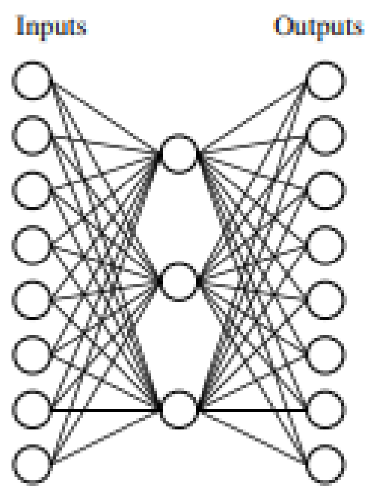
$$\Delta w_i = \eta \cdot \sum_{d \in D} (t_d - o_d) \cdot x_{i,d}$$

  - **stochastic gradient descent** (Delta rule)

$$\Delta w_i = \eta \cdot (t_d - o_d) \cdot x_{i,d}$$

- linear unit training rule using gradient descent is guaranteed to converge
  - to hypothesis H with minimum squared error
  - given a small learning rate
  - even when training data contains noise
  - even when training data not separable by H

- **Multilayer network**
  - also known as Multilayer perceptron (MLP [3])
  - can express a rich variety of non-linear decision surfaces



[1]

  - NN learning
    - eager
      - learning model – ***network***
    - ***hypothesis*** – the vector of weights
  - Example of two 2-layer perceptron network for representing the XOR function
  - **Gradient descent** (GD) training rule over a multilayer network is the ***backpropagation*** training algorithm
    - **idea**
      - For each training example
        - propagate the input forward through the network
        - propagate the errors backward through the network
      - derive gradient descent rules for training
        - one network unit (e.g. *sigmoid*)

- *multilayer network* of units → ***backpropagation*** [1]
  - D – training data
  - **batch mode GD**
    - The learning rule is applied after all training instances are provided

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

  - **stochastic GD**
    - The learning rule is applied incrementally, after each training instance

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

    - E.g., for each output neuron j, the weight w_{ji} (from input i to unit j) is updated with

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

      - E.g., if sigmoid output units

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1 - o_j)$$

=>

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j) x_{ji}$$

- x_{ji} – the i-th input to unit j

- **Loss (error) functions**
  - **regression**
    - Mean Squared Error (MSE) – L2 loss
      - Sensitive to outliers
    - Mean of Absolute Errors (MAE) – L1 loss
      - L1 – more robust and generally not affected by outliers
  - **classification**
    - cross-entropy loss
    - measures the error between two probability distributions
      - e.g., for binary classification (two classes – 0,1)
      - binary cross entropy loss
        - the average cross-entropy across all data samples

$$L = -\frac{1}{N}\left[\sum_{j=1}^{N}[t_j \log(p_j) + (1-t_j)\log(1-p_j)]\right]$$

for N data points where t_i is the truth value taking a value 0 or 1 and p_i is the Softmax probability for the i^{th} data point.

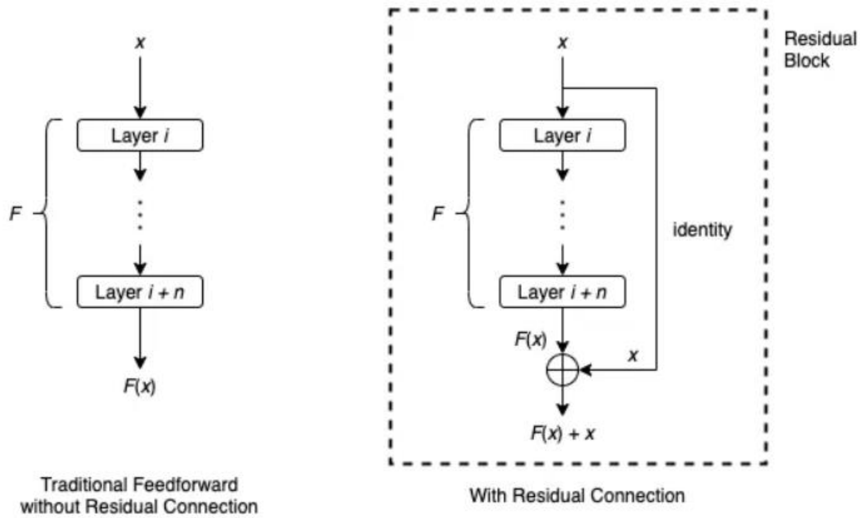
- **Kullback-Leibler (KL) Divergence**
  - for comparing two data distributions
  - measures the difference between two probability distributions
    - used in unsupervised learning tasks where the objective is to uncover structure in data by minimizing the divergence between the true and learned data distributions
  - ML/Data sciences
    - VAEs optimization
    - GANs
    - **data model drifting**
      - model drift refers to the degradation of ML model performance due to changes in **data** or in the **relationships** between input and output variables (the underlying distributions of the features have changed over time)
      -

- **characteristics of backpropagation**
  - GD over the entire network weight vector
  - training is slow (eager model)
    - using network after training is fast
  - will find a **local** error minimum (the error surface may contain many different local minima)
    - practice: run multiple times
    - weights initialization
      - Xavier initialization (using a Gaussian distribution)
  - **momentum**

$$\Delta w_{ji}(n) = \eta\,\delta_j\,x_{ji} + \alpha\,\Delta w_{ji}(n-1)$$

      - speed up the convergence of the network
      - avoid convergence to a local minimum
- Optimization algorithms [3] used in NN learning
  - GD
    - first order optimization algorithms
      - gradient
    - **stochastic GD**
      - extensions
        - **Adagrad** (Adaptive gradient algorithm)

                - adaptive learning rate in optimization algorithm to reduce the needs of extensive hyperparameter fine-tuning
            - **Adam** (Adaptive moment estimation)
                - a learning rate is maintained for each network weight (parameter) and separately adapted
                - adaptive learning rates
                - *momentum* adaptation
                - deep learning
            - **RMSProp**
                - **Hinton, 2012**
                - speed up mini-batch learning
                - divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight
        - **minibatch GD**
            - performs an update for every batch with *n* training examples
    - **second order** optimization algorithms
        - use the second derivative (the **Hessian**)
        - Hessian computations expensive
            - Lazy Hessians
- **optimize** the NN architecture
    - number of hidden layers, number of hidden neurons per layer, learning rate, momentum, etc
    - **genetic algorithms**
- **Problems** with gradient-based learning methods and backpropagation (the weights receive an update proportional to the partial derivative of the error function with respect to the weight)
    - **vanishing gradient**
        - in some cases, the gradient will be vanishingly small $\Rightarrow$ preventing the weights in changing their values
            - classical activation functions such as **sigmoid** or **hyperbolic tangent** have gradients in (0,1)
        - **solutions** to prevent the vanishing gradient problem
            - use other **activations functions** (whose derivative has a larger domain): ReLU, eLU, PReLU
            - use **residual networks** (ResNet)
                - residual connections/learning
                    - a residual connection provides another path for data to reach latter parts of the neural network by skipping some layers.



- use **batch normalization** layers, normalize the input

- <u>**exploding gradient**</u>
    - the gradient is too large
    - the model became **unstable** and unable to learn from the training data
    - solutions to prevent the exploding gradient problem
        - fewer layers in the network
        - *clipping*
            - thresholding the value of the gradient
            - before performing the GD, assign a clip value if the gradient exceeds a threshold
            - **PyTorch, Tensorflow**
        - weight regularization (L1, L2)

- **Overfitting** in ANNs
    - may be due to
        - too many neurons (complex networks)
        - insufficient training data
        - not appropriate network architecture
            - it is not close enough to the problem context
    - reducing overfitting
        - use a **validation** set during training
        - **weight decay**
            - decrease weights with a small factor during each iteration
        - **regularization** techniques (L1, L2)
            - penalize large weights
            - add to the error function a regularization term

$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss function with L1 regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Loss function with L2 regularisation

- the hyperparameter has a value between 0 (no penalty) and 1 (full penalty)
    - *high* penalty => the model will underestimate the weights and underfit the problem
    - *low* penalty => the model will be allowed to overfit the training data
- **dropout**
    - randomly drop up neurons (with their connections) during training
    - deep networks

- **Underfitting** in ANNs
    - the model is too simple, it cannot capture the essence of the data
    - insufficient training, simplicity of the model, insufficient neurons

- Expressive capabilities of classical ANNs

Boolean functions:

- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

[1]

- **shallow** neural network (1 or 2 hidden layers)

6. Other ANNs related research topics

- Boosted ANNs
  - using a boosting algorithm for improving the performance of ANNs
- Ensemble of ANNs (LSTMs, Deep LSTMs)
- Fuzzy ANNs, Fuzzy Deep Neural Networks
- Lazy ANNs
- Hybrid models
  - ANN + DT
  - ANN + SVM (Support Vector Machines)
  - ANN for function approximation in Reinforcement Learning (RL)
- Parallel/Distributed ANNs
- Deep Residual Networks (ResNets), Progressive Neural Networks, Attention mechanism [3]
- **Transformers**
  - Language models
  - Multi-head Attention
  - *Sequential data*
    - handling long dependencies between input sequence elements and enable parallel processing
  - Applications
    - **NLP**
      - QA (VQA), text summarization, language translation, etc
    - **Computer vision**
      - Image processing (classification, segmentation, translation, etc)
      - Vision Transformers (ViT)
    - **Multi-modal learning**
      - Visual captioning, Text/Image/Video/Speech Generation
    - Classification tasks
      - Time series classification
    - Semi-supervised approaches
    - Contrastive learning
      - an approach that focuses on extracting meaningful representations by contrasting positive and negative pairs of instances

- ….

[SLIDES]

– Artificial neural networks (T. Mitchell) [1]

[READING]

– Artificial neural networks (T. Mitchell) [1]
– Modern practical Deep networks (Goodfellow et al.) [2]
– CNNs and CNN architectures (Zhang et al.) [3]
– RNNs and Moderns RNNs (Zhang et al.) [3]

Bibliography

[1] Mitchell, T., *Machine Learning*, McGraw Hill, 1997 (available at www.cs.ubbcluj.ro/~gabis/ml-books)

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT Press, 2016 (online edition at http://www.deeplearningbook.org/)

[3] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola, *Dive into Deep Learning*, 2020 (http://d2l.ai/)

[4] Li Deng and Dong Yu, *Deep Learning*. *Methods and Applications*, Foundations and Trends® in Signal Processing, Volume 7 Issues 3-4, 2014 (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/DeepLearning-NowPublishing-Vol7-SIG-039.pdf)