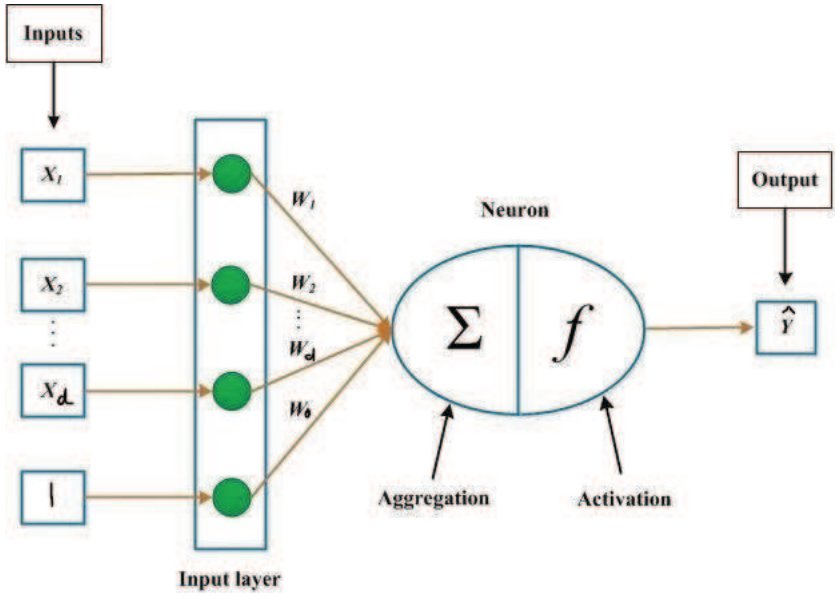
8. ARTIFICIAL NEURAL NETWORKS

- Recall that percentron, linear regression, and logistic regression can be represented using a single neuron model as a composition of preactivation

$$z = \sum_{i=1}^{d} w_i x_i + w_0$$

where $w_1, \ldots, w_d$ are weights and w_0 is a bias, and activation

$$\hat{y} = f(z)$$



https://www.researchgate.net/figure/A-single-artificial-neuron-perceptron_fig3_319857496

In those three models, the activation functions used are the sign function, the identity function, and the sigmoid function.

$$\text{sign}(z) = \begin{cases} 1, & z > 0 \\ 0, & z = 0 \\ -1, & z < 0 \end{cases} \qquad f(z) = z \qquad \sigma(z) = \frac{1}{1 + e^{-z}}$$

In other words, the output is the following function of inputs

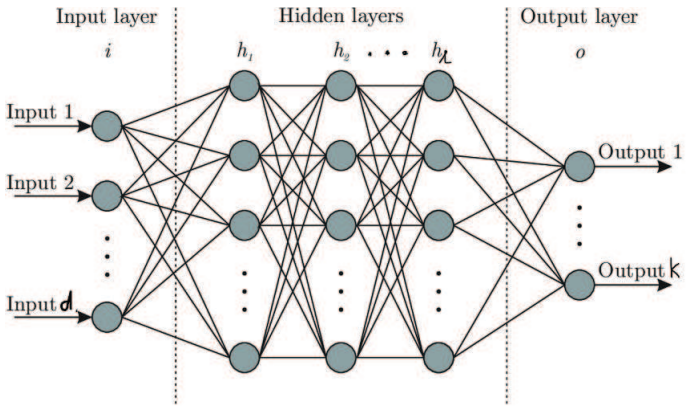
$$\hat{y} = f(z) = f\left(\sum_{i=1}^{d} w_i + w_0\right) = f(w^T x + w_0).$$

Once we choose the learning algorithm we want to use, to find parameters (weights and bias) we need to define the loss function that measures how off our predicted outputs are compared

to the actual target values across the training dataset $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$. The optimal parameters are those that minimize the loss function

$$L(w, w_0; D) = \sum_{i=1}^{n} L(\hat{y}^{(i)}, y^{(i)}).$$

- We will study *feed-forward* artificial neural networks with *back-propagation* training.

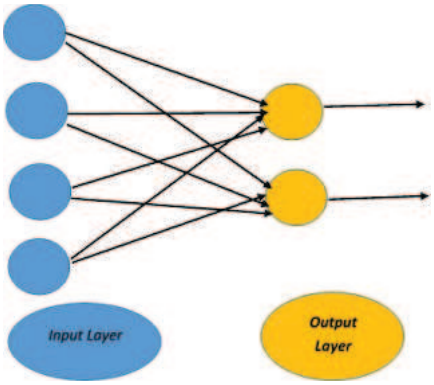


https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051

A neural network takes in an input $x \in \mathbb{R}^d$ and generates the output $\hat{y} \in \mathbb{R}^k$. It consists of multiple layers on neurons. In a feed-forward network, the data flows one way, from the input layer to the output layer. Each node in the hidden and output layers is a neuron with preactivation (weighted sum of inputs plus bias) and activation. The output of one layer becomes the input for the next layer.

The term "hidden" is because we do not have the ground truth for the number of neurons in such a layer in contrast to the input and output layers whose sizes are determined from the dimensions of data instances $(x^{(i)}, y^{(i)})$.

- Consider a single layer.



https://www.i2tutorials.com/what-is-single-layer-perceptron-and-difference-between-single-layer-vs-multilayer-perceptron/

Assume we have d inputs and k outputs

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix} \in \mathbb{R}^{d \times 1} \qquad A = \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_k \end{bmatrix} \in \mathbb{R}^{k \times 1}$$

We can organize weights into a matrix $W \in \mathbb{R}^{d \times k}$ and bias values in a vector $W_0 \in \mathbb{R}^{k \times 1}$. Then the preactivation

$$Z = W^T X + W_0$$

is in $\mathbb{R}^{k \times 1}$ and the output vector

$$A = f(Z) = f(W^T X + W_0)$$

is in $\mathbb{R}^{k \times 1}$, where the activation function f is applied element-wise to elements of Z.

Note that if our algorithm is just a single layer, even with sign or sigmoid activation functions, we can only make a linear hypothesis, i.e., the decision boundary is linear. To obtain non-linear hypothesis, we need more than one layer.

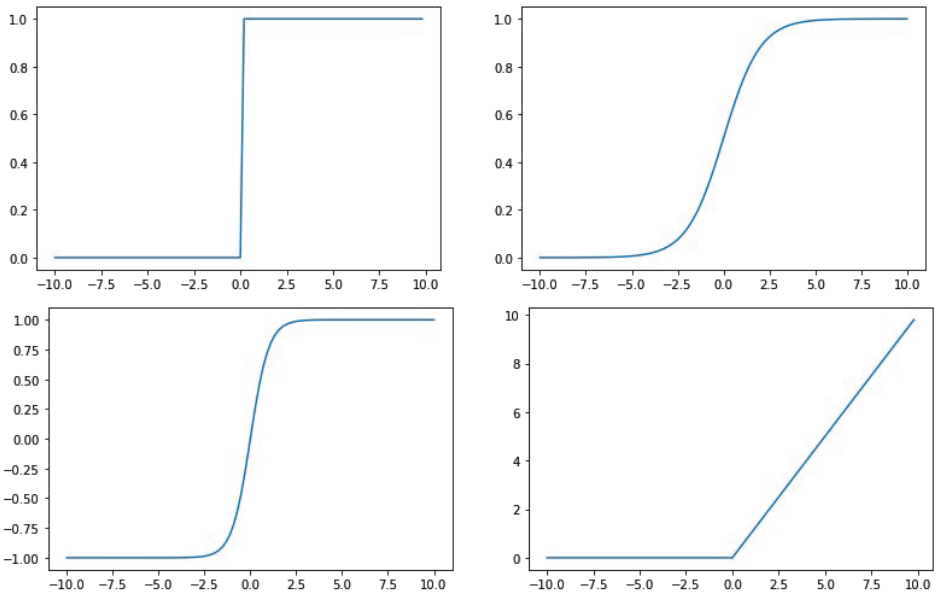
If there are L layers, then in the lth layer, $l \in \{1, \dots, L\}$, we compute the preactivation

$$Z^l = (W^l)^T A^{l-1} + W_0^l$$

and the activation output

$$A^l = f(Z^l).$$

- **Activation functions**



  - *step function*

$$\text{step}(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

– *sigmoid function*

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

– *hyperbolic tangent*

$$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

– *rectified linear unit*

$$\mathrm{ReLU}(z) = \max\{0, z\} = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

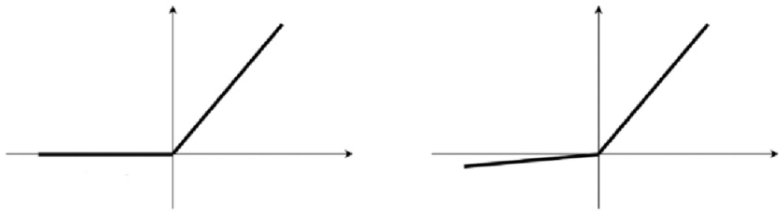
– *softmax*

$$\mathrm{softmax}(Z) = \begin{bmatrix} \frac{e^{Z_1}}{e^{Z_1}+e^{Z_2}+\ldots+e^{Z_d}} \\ \frac{e^{Z_2}}{e^{Z_1}+e^{Z_2}+\ldots+e^{Z_d}} \\ \cdots \\ \frac{e^{Z_d}}{e^{Z_1}+e^{Z_2}+\ldots+e^{Z_d}} \end{bmatrix}$$

Softmax takes as an input a vector $Z \in \mathbb{R}^d$ and generates the output $A \in (0,1)^d$ such that $\sum_{i=1}^{d} A_i = 1$ so it can be interpreted as a probability distribution over d items.

Remarks:

* The original idea of neural network used the step function, but since it is discontinuous and its derivative is 0 on both sides, we cannot use it in gradient descent methods to tune the weights and biases.
* ReLU is the most common in the internal (hidden layers).
* Some of these functions have problems with the vanishing gradient meaning their derivative is 0 and LeakyReLU, ELU and SELU are often used to avoid that problem.



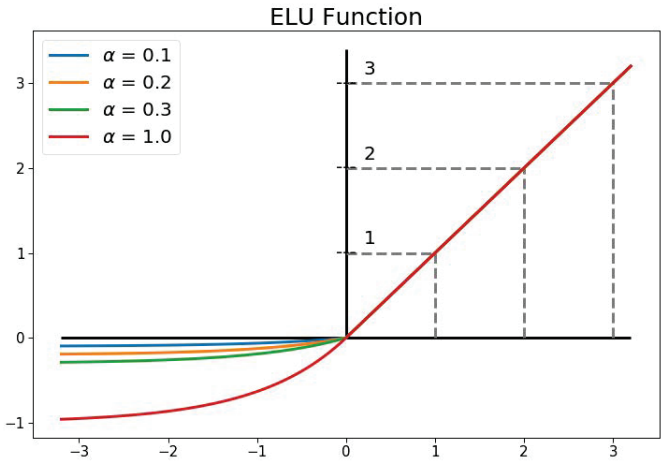
ReLU activation function *LeakyReLU activation function*

https://www.researchgate.net/figure/ReLU-activation-function-vs-LeakyReLU-activation-function_fig2_358306930

$$\mathrm{LeakyReLU}(z) = \max\{0.01z, z\}$$

ELU (Exponential Linear Unit)
https://tungmphung.com/elu-activation-a-comprehensive-analysis/

$$\mathrm{ELU}_\alpha(z) = \begin{cases} z, & z \geq 0 \\ \alpha(e^z - 1), & z < 0 \end{cases}$$



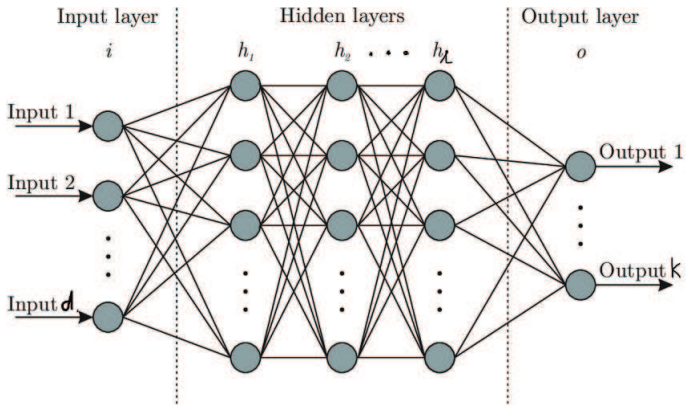
https://tungmphung.com/elu-activation-a-comprehensive-analysis/

SELU (Scaled Exponential Linear Unit)
https://iq.opengenus.org/scaled-exponential-linear-unit/

∗ Sigmoid is common in the output layer for binary classification since its range is in $(0, 1)$ and it can be interpreted as probability, while softmax is used in the output layer for multi-class classification.

- **Loss (or cost) functions**

  A neural network is a function between the input variables and the output variables.



https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051

These variables are related by a series of linear functions (denoted by Σ) and activation functions (sigmoid, tanh, ReLU, or some other). The linear functions in the nodes are defined using parameters weights and biases. To make our algorithm effective and accurate, we need to find the optimal values of these parameters. How can we do this and what does it mean "optimal values of the parameters"?

1. First, we initialize these parameters (weights and biases) randomly.
2. We do a *forward pass* and calculate all preactivations and outputs in all layers. The outputs of the last layer are the predictions.
3. We compare the predictions to the target values and measure the loss for each data instance. There are different ways to measure this loss depending on whether we are doing classification (the target variable contains classes such as classifying people as "will buy insurance" or "will not buy insurance") or regression (the target variable is a real number such as predicting house price).
4. The overall loss function L is usually the sum or the average of all those individual losses. We find a better choice of parameters that makes this loss function lower and go back to step 2. One of the ways to mimimize the loss is to utilize gradient-descent.
5. Once we find the parameters that give the satisfactory small loss (in theory, the minimum of the loss function), then those parameters are called "optimal". We are done and we have the model.

There are many loss functions available in tensorflow that can be used. https://keras.io/api/losses/

The choice of the loss function depends on the problem (classification vs. regression) as well as on the choice of the activation function in the last layer. Some common loss functions are

- classification: binary cross entropy, categorical cross entropy, sparse categorical cross entropy
- regression: mean absolute error, mean squared error
  - *binary cross entropy* (when the activation in the last layer is sigmoid)

$$\text{Binary Cross Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)}\cdot\log\hat{y}^{(i)} + (1-y^{(i)})\cdot\log(1-\hat{y}^{(i)})\right)$$

$y^{(i)} \in \{0,1\}$ is the true label for $x^{(i)}$

$\hat{y}^{(i)} \in (0,1)$ is the predicted probability

  - *categorical cross entropy* (when the activation in the last layer is softmax)

$$\text{Categorical Cross Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{k} y^{(i),c}\cdot\log\hat{y}^{(i),c}$$

$y^{(i),c}$ = true label of the ith data instance belonging to the class c

$\hat{y}^{(i),c}$ = predicted probability of the ith data instance belonging to the class c

* *mean absolute error* and *mean square error* (when the activation in the last layer is linear)

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}^{(i)} - y^{(i)}| \qquad \text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$$

$$y^{(i)} = \text{true target value}$$
$$\hat{y}^{(i)} = \text{predicted output value}$$

Example: Some data scientists also use MSE when working on binary classification problems. Assume we have a data instance that was misclassified and

$$y^{(5)} = 0 \text{ and } \hat{y}^{(5)} = 0.82.$$

$$\begin{aligned}\text{Binary cross entropy loss for the 5th data instance} &= -\left(y^{(5)} \cdot \log \hat{y}^{(5)} + (1 - y^{(5)}) \cdot \log(1 - \hat{y}^{(5)})\right) \\ &= -\left(0 \cdot \log 0.82 + 1 \cdot \log 0.18\right) \\ &= 1.71\end{aligned}$$

$$\begin{aligned}\text{Squared loss for the 5th data observation} &= (\hat{y}^{(5)} - y^{(5)})^2 \\ &= (0.82 - 0)^2 \\ &= 0.6724\end{aligned}$$

Note that the cross entropy loss penalizes misclassified data more than the squared error loss. As a result, the backpropagation algorithm will produce the optimal parameters faster if the cross entropy loss is used.

Remark:

- In simple words, the value of the loss function tells us how good our model is. If the loss is smaller, the model is better.
- Categorical cross entropy is used for labels that are one-hot encoded.
- Sparse categorical cross entropy is used for labels that are integers.

- **Optimizing neural network parameters**

  Once we decide how many layers and neurons our neural network has, we need to determine the optimal weights and biases which make our model the most effective. To measure this effectiveness we need to see how good our predictions are. In other words, we measure the errors between the predictions and the actual true target values for each data instance and then combine them into a loss function. The loss function depends on weights and biases, while the data training set is fixed

$$D = \{(x^{(i)}, y^{(i)}), \ldots, (x^{(n)}, y^{(n)})\}.$$

We denote the output of the neural network by $\hat{y}^{(i)} = h(x^{(i)}, \overline{W})$, where $\overline{W}$ is a vector containing all parameters (weights $w_1, \ldots, w_p$ and biases $b_1, \ldots, b_q$). We can write that the loss function is a function of all weights and biases by

$$L(\overline{W}; D) = \sum_{i=1}^{n} L(\hat{y}^{(i)}, y^{(i)})$$

$$\boxed{L(\overline{W}; D) = \sum_{i=1}^{n} L\left(h(x^{(i)}, \overline{W}), y^{(i)}\right)}$$

The optimal parameters are those for which L has a minimum and they are found using the gradient descent.

First, we need to find derivative of L with respect to each of the variables $w_1, \ldots, w_p, b_1, \ldots, b_q$. These derivatives are combined in the vector form, called the gradient of L:

$$\nabla L = \left(\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_p}, \frac{\partial L}{\partial b_1}, \ldots, \frac{\partial L}{\partial b_q}\right)$$

The method of gradient descent is as follows:

1. We start with the random choice of initial values for all the weights and biases.
2. We feed ALL the data instances into a model, compute all predicted outputs (this is called a *forward pass*), compute individual errors, and compute the overall value of the loss function L.
3. We make the updates for weights and biases using the following formulas for each parameter

$$w_1 \leftarrow w_1 - \alpha \frac{\partial L}{\partial w_1}(\overline{W}; D)$$
$$\ldots$$
$$w_p \leftarrow w_p - \alpha \frac{\partial L}{\partial w_p}(\overline{W}; D)$$
$$b_1 \leftarrow b_1 - \alpha \frac{\partial L}{\partial b_1}(\overline{W}; D)$$
$$\ldots$$
$$b_q \leftarrow b_q - \alpha \frac{\partial L}{\partial b_q}(\overline{W}; D)$$

   These derivatives are computed using the chain rule, starting from the output and going backwards until the particular weight/bias is reached (this is called *backpropagation*). If we use vector notation $\overline{W} = (w_1, \ldots, w_p, b_1, \ldots, b_q)$, the formula for the updates can be written as

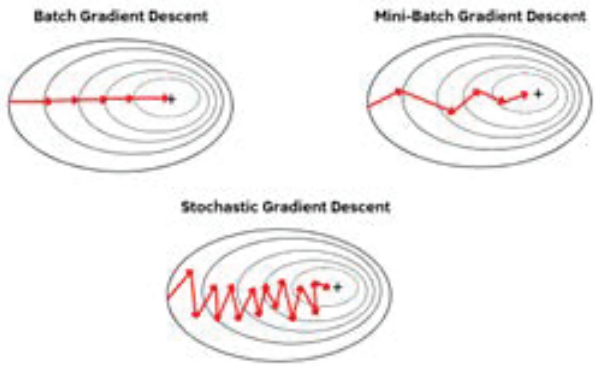
$$\overline{W}_{new} = \overline{W}_{old} - \alpha \nabla L(\overline{W}_{old}; D)$$

$$\overline{W}_{new} = \overline{W}_{old} - \alpha \sum_{i=1}^{n} \nabla L\left(h(x^{(i)}, \overline{W}_{old}), y^{(i)}\right)$$

4. We repeat steps 2 and 3 until the gradient of L becomes close to zero.

Remarks:

– Instead of feeding ALL the data points in step 2 (*batch gradient descent*), which can become very costly and inefficient, sometimes *stochastic gradient descent* and *mini-batch gradient descent* are used.



https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/

∗ stochastic gradient descent
Each update is computed at a randomly chosen data instance.

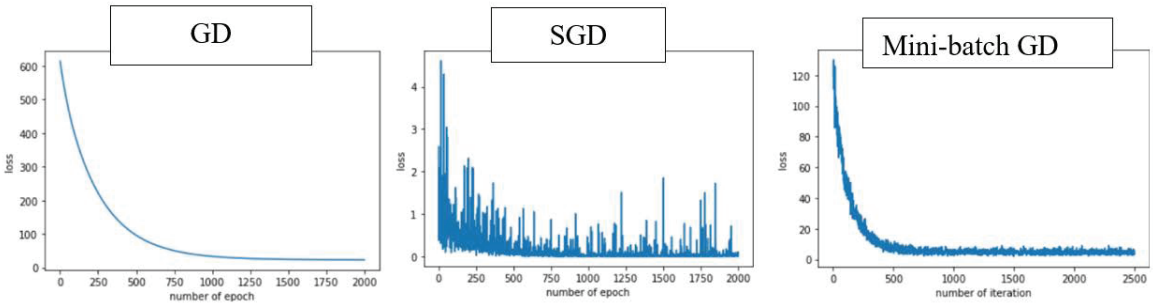
$$\overline{W}_{new} = \overline{W}_{old} - \alpha \nabla L \left(h(x^{(i)}, \overline{W}_{old}), y^{(i)} \right)$$

By convention, we iterate by rounds of n iterations and each round is called an epoch.

∗ mini-batch gradient descent
Each update is computed at a randomly chosen, fixed size k, subset of data instances.

$$\overline{W}_{new} = \overline{W}_{old} - \alpha \sum_{i=1}^{k} \nabla L \left(h(x^{(i)}, \overline{W}_{old}), y^{(i)} \right)$$

By convention, we divide the data set into distinct mini-batches of k data points and we iterate by rounds of $\approx n/k$ iterations. Each round is called an epoch.



https://towardsdatascience.com/deep-learning-optimizers-436171c9e23f

– *Initializing weights and biases*

The choice of initial weights and biases is very important and they should be initialized at random values. We would like different parts of neural network to learn different aspects of the problem and if all weights start at the same values, the symmetry will often keep the values from moving in the useful directions. Also, many of the activation functions have very small slopes when the pre-activation values have large magnitude so we want to keep the initial weights small so we will be in the domain where the activation functions have non-zero derivatives, so that the gradient descent would have some useful signal which way to go. We do not want the signal to die out, but we do not want it to explode either.

One of the problems with deep neural networks is that the partial derivatives of the loss function with respect to weights and biases, using the chain rule, are long products of derivatives throughout the layers. If we multiply many numbers less than 0, we get much smaller number, and if we multiply many large numbers, we can end of up with a very large number. This is known as *vanishing* or *exploding* gradients.

We need signal to flow properly in both directions: in forward pass when making predictions and in reverse direction when backpropagating gradients. Glorot and Bengio ("Understanding the difficulty of training deep feedforward neural networks", 2010) claim that we need variance of the outputs of each layer to be equal to the variance of its inputs.

One common strategy is to initialize each weight at random from the Gaussian (normal) distribution with mean 0 and variance of $\frac{1}{fan_{avg}}$, where fan_{avg} is the the average of the number of inputs to the node and number of outputs from the node; and to initialize each bias randomly from the standard normal distribution (mean 0 and variance 1).

https://mmuratarat.github.io/2019-02-25/xavier-glorot-he-weight-init

initialization	activation functions	normal (mean 0, variance σ^2)	uniform $(-r, r)$
Glorot	tanh, sigmoid, softmax	$\sigma^2 = \frac{1}{fan_{avg}}$	$r = \sqrt{3\sigma^2}$
He	ReLU and variants	$\sigma^2 = \frac{2}{fan_{in}}$	$r = \sqrt{3\sigma^2}$
LeCun	SELU	$\sigma^2 = \frac{1}{fan_{in}}$	$r = \sqrt{3\sigma^2}$

– *Adaptive learning rate α and scheduling*

Choosing a value for α is difficult. If α is too small, then convergence of the gradient descent method will be very slow and if it is too large, then we might have very slow convergence due to oscillations or we might have that the method diverges. This problem is more evident in stochastic and mini-batch versions.

Another problem is with vanishing or exploding gradients in which the back-propagated gradient is either too small or too big to be used as an update with the fixed learning rate.

* Power scheduling

$$\alpha(t) = \frac{\alpha_0}{(1 + t/s)^c}$$

The initial learning rate α_0, the power c (typically set to 1) and the steps s are hyperparameters. Note that after s steps the learning rate drops to $\alpha_0/2$, and after

some more steps it drops to $\alpha_0/3$, etc. The learning rate drops quickly at first and then more and more slowly.

* Exponential scheduling

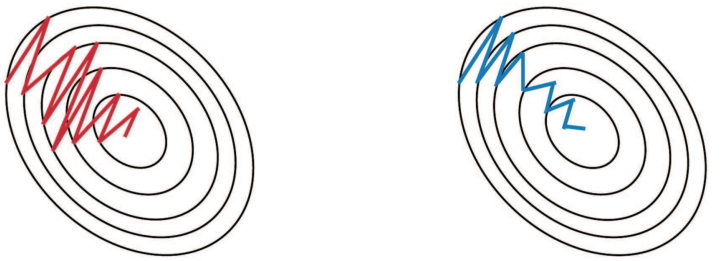
$$\alpha(t) = \alpha_0 \cdot 0.1^{t/s}$$

The learning rate drops by factor of 10 every s steps.

* Piecewise constant scheduling
* Performance scheduling

– *Faster Optimizers*

* The idea of *momentum* is used in stochastic and mini-batch gradient descent to "average" recent gradient updates and if they have been bouncing back and forth in the same direction, we take that component of the motion. Momentum essentially smooths the path towards the minimum and leads to faster convergence.



https://eloquentarduino.github.io/2020/04/stochastic-gradient-descent-on-your-microcontroller/

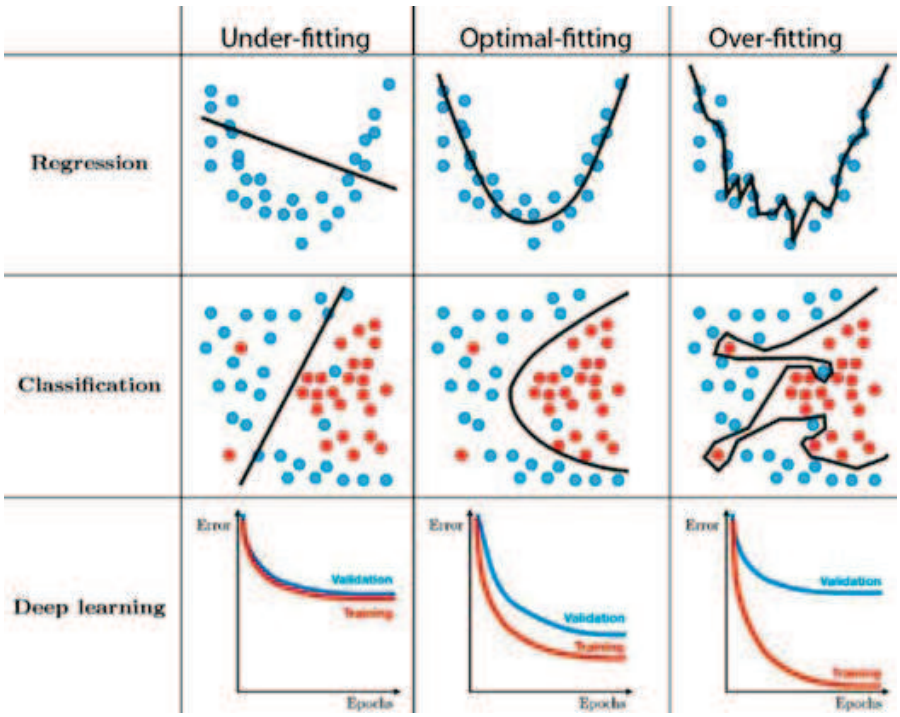
* The idea of *adadelta* is to take larger steps in parts of the region where the gradient is nearly 0 and smaller steps where it is large. *Adam* combines both the ideas of momentum and adadelta and has become to default method of managing learning rate for neural networks.

- **Regularization**:

  Deep neural networks can often learn the training data very well and result in *overfitting*.

  Universal Approximation Theorems imply that neural networks can represent a wide variety of functions when given appropriate weights.

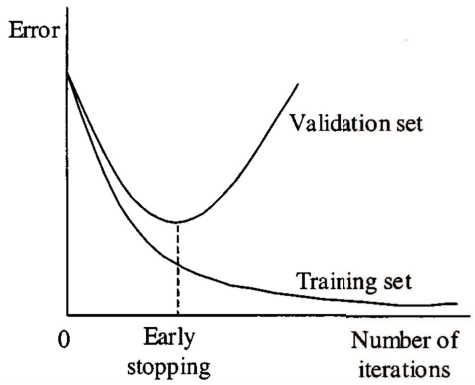
  https://en.wikipedia.org/wiki/Universal_approximation_theorem



https://www.kaggle.com/getting-started/166897

To avoid overfitting, we can use the following techniques.

– *early stopping*



https://paperswithcode.com/method/early-stopping

The main idea is to to train the neural network on the training data set, but evaluate the loss after each epoch on a validation set as well. Typically, the loss on the training data will decrease after each iteration, while the loss on the validation set will also decrease initially, but then it will begin to increase. Once we see that the loss on the validation set is systematically increasing, we stop the training and return the weights and biases when the validation loss was the lowest.

– *weight decay*

The main idea is to penalize the norm of all weights (denoted by W) as in ridge regression by considering the loss function

$$Loss(\overline{W}; D) = \sum_{i=1}^{n} L(\hat{y}^{(i)}, y^{(i)}) + \lambda \|W\|_2^2 \qquad \text{(L2 regularization)}$$

or as in lasso regression by considering the loss function

$$Loss(\overline{W}; D) = \sum_{i=1}^{n} L(\hat{y}^{(i)}, y^{(i)}) + \lambda \|W\|_1 \qquad \text{(L1 regularization)}$$

– *adding noise to the data*

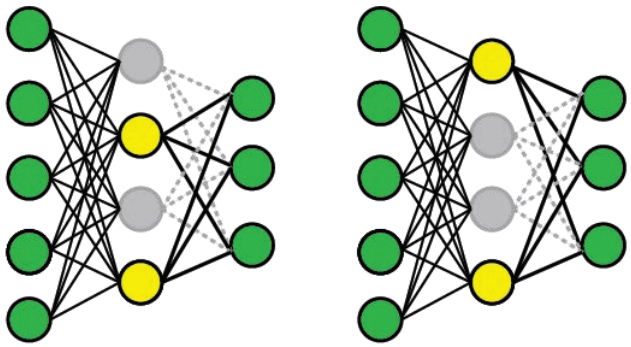
In this technique, we perturb the values of $x^{(i)}$ in the training data set by adding a small amount of zero-mean normally distributed noise before each gradient computation. This will make it more difficult for the neural network to overfit to particular training data since the data will be changed slightly at each iteration.

– *dropout*

Instead of perturbing data at each iteration during training as in the previous technique, we can perturb the network. More precisely, each time a data instance is fed in the input layer, we can drop randomly some neurons in all layers except the output layer. This ensures that neural network is not learning noisy or redundant patterns in the data.

The dropout rate (a hyperparameter) is typically set between 10% and 50% (in recurrent neural networks between 20% and 30% and in convolutional neural networks closer to 40% to 50%).

For example, we can drop randomly 50% of the neurons in the hidden layer.

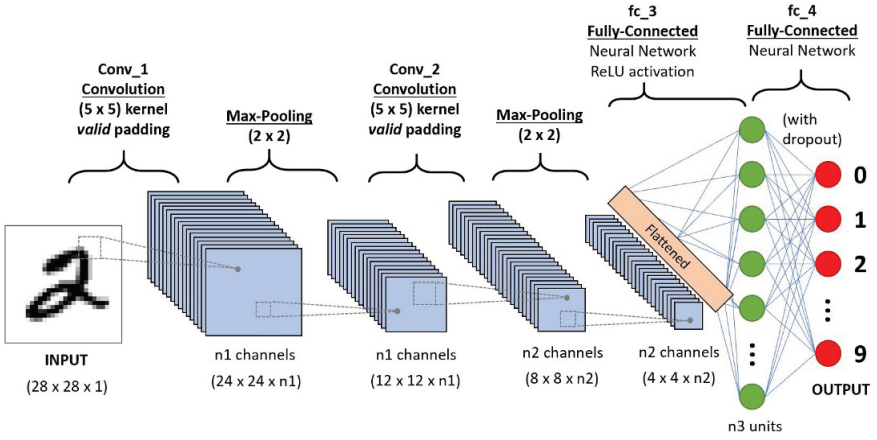


https://blog.jovian.ai/dropout-68913941f569

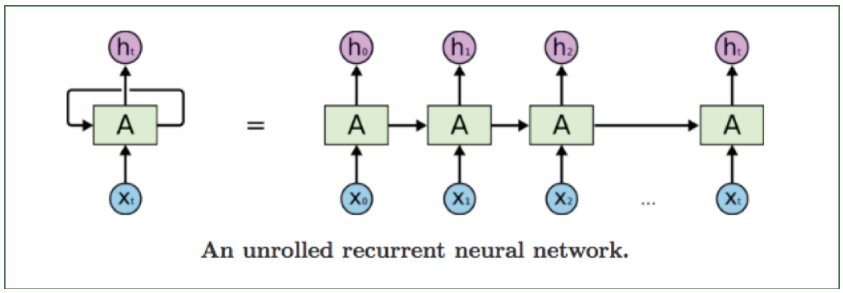
– *batch normalization*

Read [1, Chapter 8, pages 58-60], [2]-[4], [5, pages 338-345].

- **Remarks**:
  - Watch the following video which shows nice animations of neural networks and gradient descent.
    https://www.youtube.com/watch?v=IHZwWFHWa-w&t=1009s
  - Tensorflow playground
    https://playground.tensorflow.org/
  - Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs)



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53



https://datascience.eu/machine-learning/an-introduction-to-recurrent-neural-networks/

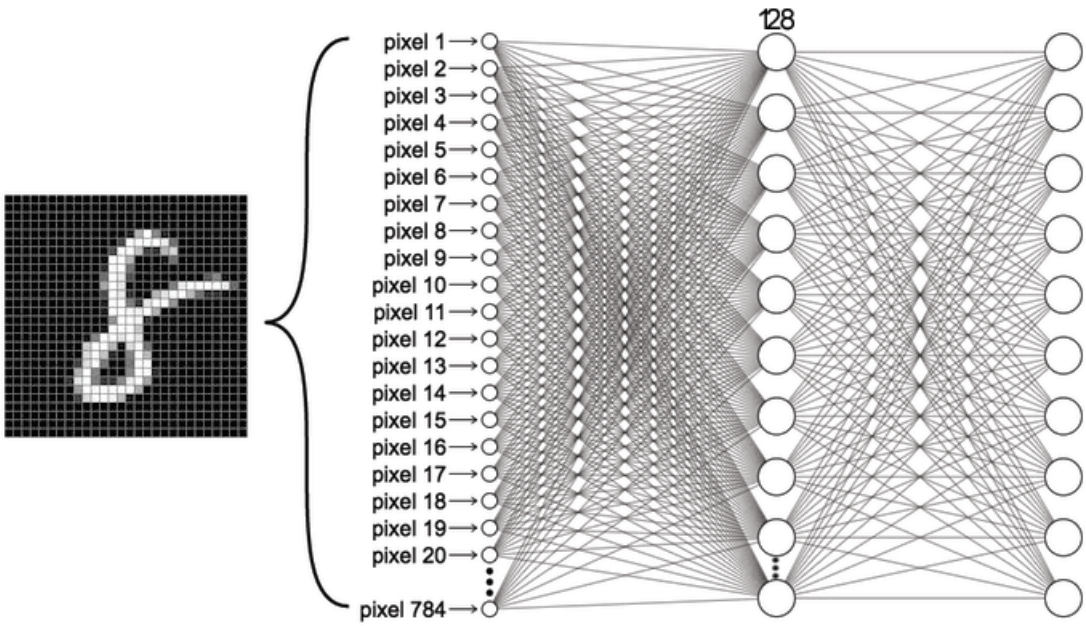
- **Python code:** Lecture_8_ANNs.ipynb

Classifying handwritten digits using ANNs

Our goal is to consider images of handwritten digits and build an artificial neural network to classify these images as "0", "1", ..., "9". In other words, we want to create a model with some inputs $x_1, x_2, \ldots, x_d$ (obtained from the image) and output labels "0", "1", ..., "9".

How do we get the inputs from the image?

The image is a matrix of pixels of certain size (28 by 28) with gray color scale represented by numbers ranging from 0 to 255, where 0 stands for black and 255 stands for white.



https://colab.research.google.com/github/aamini/introtodeeplearning/

We flatten that matrix and obtain an input vector that we feed to our neural network. In this case the vector has 784 entries implying that the network will have 784 inputs. The model here also shows one hidden layer of 128 nodes as well the output layer of 10 nodes. Note that the total number of connections between the input and hidden layers is 784*128=100,352 and between the hidden and the output layers is 128*10=1280. Therefore the total number of weights is 101,632 and we also have 128+10=138 biases. Hence, this neural network has 101,770 parameters that we need to find.

For a given input vector, the output of the model will consist of 10 numbers between 0 and 1 (we'll use softmax activation function in the output layer). These numbers are used to predict the label associated with the image. For example, maybe the highest of the output values is 0.92 at the output node "8", so we classify this image as "8".

- **Homework 4:**
  - *Part 1:* Describe Batch Normalization
  - *Part 2:* Load MNIST Fashion Data Set from keras
    https://keras.io/api/datasets/fashion_mnist/
    Create several neural network models investigating the effect of hyperparameters and techniques we studied on the model performance (number of layers, number of neurons in hidden layers, optimizers, batch size and learning rate in the gradient descent optimizers, L1 and L2 regularization, dropout, batch normalization, weight and bias initialization, etc.) using the information provided at
    https://keras.io/api/

- **References and Reading Material:**

[1] MIT course (Chapter on Neural Networks)

[2] https://arxiv.org/pdf/1502.03167.pdf (original paper)

[3] https://en.wikipedia.org/wiki/Batch_normalization

[4] https://towardsdatascience.com/implementing-batch-normalization-in-python-a044b0369567

[5] A. Geron, *Hands–on Machine Learning with Scikit-Learn, Keras & TensorFlow*, O'Reilly, 2nd edition, 2019 (Chapters 10 and 11)

[6] J. Krohn, *Deep Learning Illustrated*, Addison Wesley Data & Analytics Series, 2020

[7] Youtube videos by Prof. Patrick Winston (MIT course 6.034 Artificial Intelligence)
Video 12a: neurons, backpropagation
Video 12b: convolutional neural networks, softmax, dropout

[8] *Neural Networks and Deep Learning* e-book by M. Nielsen
http://neuralnetworksanddeeplearning.com/