Theoretical exercises

Q2-

1- MSE loss function: *Mean Squared Error* is a loss function which is widely used in regression tasks, it calculates the mean difference between predicted and the actual value, and the domain of numbers is continuous in real number space. The formula of MSE is:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

It calculates the square of the difference between actual value(Yi) and predicted value(Y-hat) and takes the mean of the sum of squares. It tells us how wrong our model is predicting our data. So in the task of linear regression we are trying to minimize the MSE using an optimizer algorithm which is gradient-descent in this case most of the time.

2- Cross-Entropy: Cross-entropy loss measures the divergence of two distributions. It is widely used in classification tasks, in which the model output is probability of an input belonging to a class, in case of binary classification there are two classes(0 or 1). So the output of the model is the probability of each input, and also a threshold like 0.5 which if P(x) > 0.5 belongs to class 1 and otherwise belongs to 0. The formula of binary cross entropy is:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

When the output is 0: yi is 0 so we have (1-yi)*log(1-p(yi))
When the output is 1: (1-yi) is 0 so we have yi * log(p(yi))
The task is minimizing cross entropy loss, which means that we want our prediction distribution to get close enough to the actual distribution.

“For both binary and multi-class classification problems, the main goal is to minimize the cross entropy loss, which in turn maximizes the *likelihood* of assigning the correct class labels to the input data points”

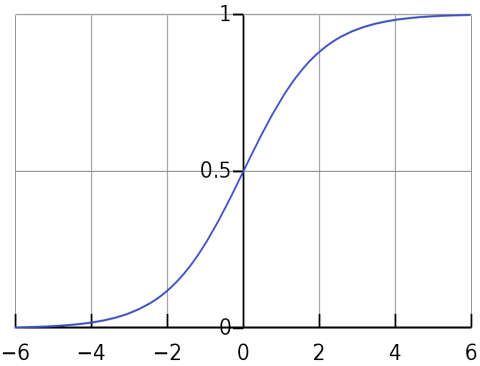
Conclusion:
According to the above explanation of MSE and Cross-Entropy loss, the main difference between them is MSE is for the models whose outputs are in real number space, like linear regression, and it calculates the *mean difference* between actual value and model prediction. Cross-entropy is for classification tasks in which the output of a model is probability of an instance xi belonging to one class, so it calculates the *divergence* of two distributions.
Then we use Cross-entropy loss for classification problems.

Q4-

The activation functions are used in neural networks for determining which neuron should fire(being active) and which should not, and another purpose of activation functions is to add *non-linearity* to models for dealing with non-linear relations in data for more advanced tasks like images and NLP etc. *Sigmoid* and *ReLU* are two such functions.

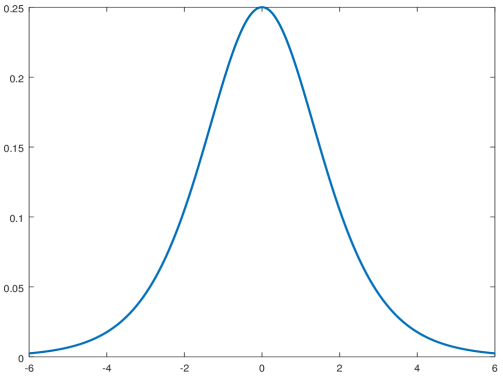
Sigmoid:
The sigmoid activation function maps weighted sum to a bounded space between 0 and 1. The formula of sigmoid is:

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



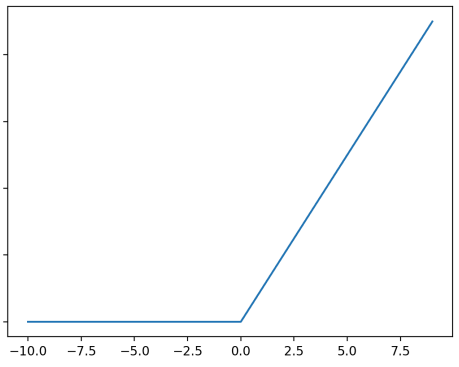
Limit of f(x) to +inf is 1 and to -inf is 0. The plot of sigmoid is:
It's usually used in classification problem because it maps
Continuous input to a probability output between 0 and 1.

Advantages of Sigmoid:
1- The output of sigmoid functions is bounded and prevents large or unstable outputs during training.
Disadvantages of Sigmoid:
1- Gradient vanishing: “The gradient vanishing problem occurs during training when the gradients of the loss functions become really small as they backpropagate through different layers. This causes slow learning due to less updation of the weights of early layers.”
2- computationally expensive since we have exponential term.
If we plot the gradient of sigmoid function, we can visualize
The gradient of very negative and very positive numbers
Is almost 0 and it cause the slow learning(Gradient vanishing)



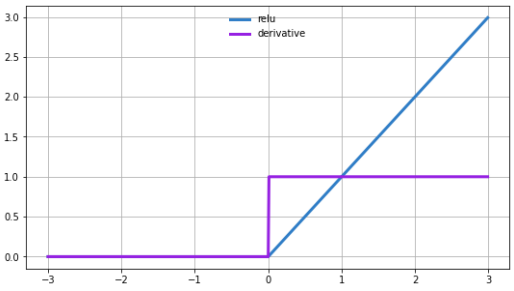
ReLU: this activation function treats the weighted sum of neurons as an identity function if it is positive and 0 otherwise.

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



Advantages of ReLU: ReLU could overcome both disadvantages of sigmoid function, first we don't have gradient vanishing problem since the gradient of ReLU is 1 or positive numbers and 0 otherwise, and second its more computationally efficient than sigmoid, because it's simply discard negative values and doesn't have exponential term. The ReLU is mostly used in deep learning architecture in hidden layers and it makes faster convergence in networks because the backpropagation becomes very efficient and fast and we don't have a learning slowdown problem.

Disadvantages of ReLU: As we said the gradient of ReLU maps all negative numbers to 0, the ReLU prones to causing "*dying*". Dying of a neuron means the output is always 0 and it happens when the weighted sum of the neuron becomes negative. In these cases we can use variants of ReLU like "Leaky ReLU".



Q3-

In the process of training a neural network, we must have a point to start, so we need to manually initialize the weights. We have several options like setting all weights to 0, *Random initialization* and other techniques which we discuss further.

1- One of the issues of neural networks is "symmetry" of weights, which means that all the weights become equal and all neurons learn the same thing about data and it may be stuck in a local minima! so we can't start with setting all weights to 0!!

2- Avoiding vanishing and exploding of gradient: if we initialize weights with very small numbers, it may cause gradient vanishing and also if we set them with very large numbers, it causes the gradient to explode!

3- Avoiding saturation of activation functions: Some activation functions like sigmoid saturates with very large positive and large negative numbers.

4- Better generalization and Faster convergence: Helps the model to start with a reasonable set of parameters close to optimal value and also the optimization algorithm can more quickly lead to optimal solution.

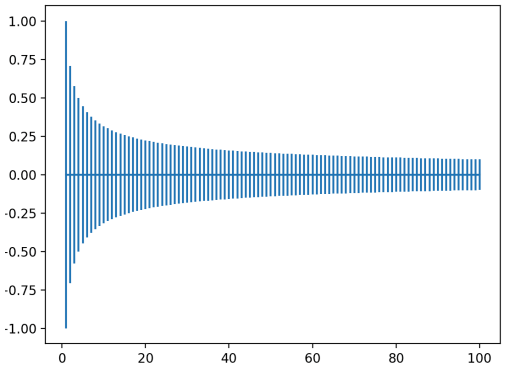
Weight initialization techniques:

1- Random initialization:

In this technique we initialize weight randomly from a specific distribution, this technique first introduced to solve the problem of 0 initialization or "Symmetry".

2- Xavier:

In this method we calculate weights with random numbers from a Uniform distribution with mean = 0 and range = [-1 / sqrt(n), 1/ sqrt(n)] and std = sqrt(2 / n_input + n_output). which n_input is the number of input neurons and n_output for output neurons. It is mainly used with sigmoid and Tanh activation functions. So the spread of weights is dependent on the number of input neurons, which we can see in the plot. So with a greater number of input neurons, We have smaller weights. "The rationale behind this scaling is to prevent the activations and gradients from becoming too large or too small as they propagate through the network. If the weights are too large, it can lead to exploding gradients, causing training instability. Conversely, if the weights are too small, it can lead to vanishing gradients, making it difficult for the network to learn effectively."



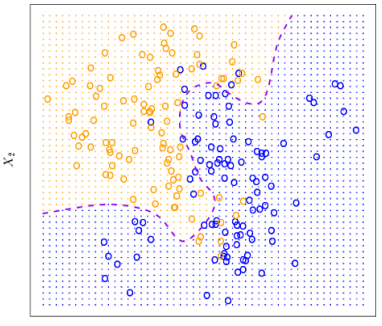
He method:
This method is used with the ReLU activation function. We initialize weights by calculating random numbers from a Guassian distribution with mean = 0 and std =sqrt(2 / n_input). When using ReLU activation functions, it's important to initialize the weights properly to prevent the issue of dead neurons, where neurons in the network can become stuck in a state where they always output zero due to negative weights. If weights are initialized too small, it's possible for a large portion of neurons to never activate during training, effectively rendering them useless. Compared with the Xavier method, which considers both input and output neurons, the He method only considers inputs, because ReLU ignores the negative values and almost halves the gradient, it must be initialized with weights with higher variance to counteract this effect.
Potential issues with symmetric weights:
Redundancy and lack of diversity, as the output of all neurons in the layer is the same! It may cause Convergence issues, since we can get stuck in suboptimal points. And the solution for these issues is using a proper weight initialization technique which we discussed above.

Q1-

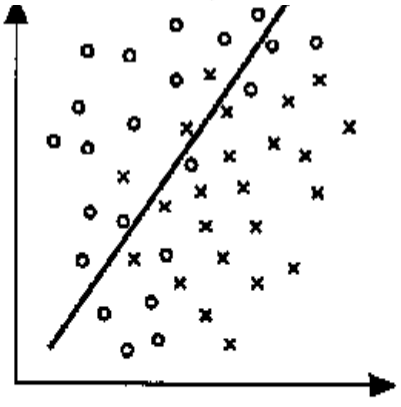
Let us first start with explaining the *Logistic regression* and *Perceptron* individually.

Note: We assume a Perceptron is a single neuron with sign(step) function as activation function.

1- Logistic regression:
Logistic regression is a machine learning algorithm which is used mostly in Binary classification tasks, it takes the net input and calculates the probability of an instance belonging to a class which means the output is a continuous number between 0 and 1. The activation function of the model is sigmoid,(introduced in Q4). The loss function which is used in this model is Binary cross entropy(introduced in Q2), which calculates the divergence of two distributions, which is quite suitable for the task. So by minimizing the BCE or maximizing likelihood, we can train the model on the data, which is done by an optimization algorithm(like SGD or Adam etc). Since the sigmoid function is a non-linear activation function, the decision boundary of Logistic regression is non-linear, so it could solve non-linear separable data as well(but not very complex!). We can see in the figure below. Also the logistic regression could solve the multi-class classification with appropriate modification(one-vs-all, softmax). And due to the probabilistic nature of the output and loss module, it's less sensitive to outliers(if the skewness of the data isn't very high).



Perceptron:
It's a single layer neural network which takes the weighted sum of input and feeds it to a sign(step) function to make a binary output 0 or 1(with a threshold). So at the first step, the



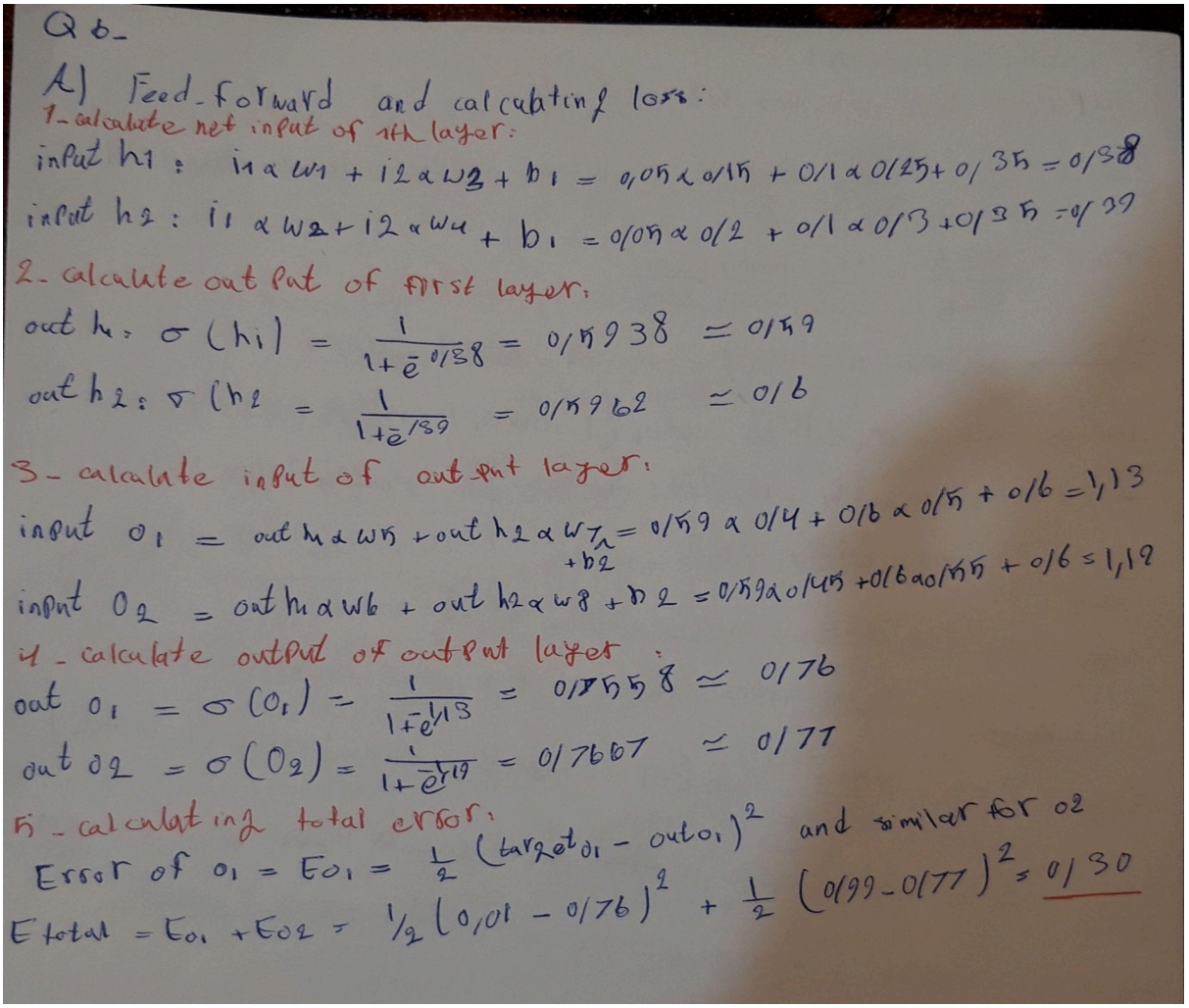
output space is linear! So it can not solve the nonlinear problems. The loss function used in this algorithm is typically MSE(Mean Squared Error introduced in Q2), which measures the square error of the model output with actual value, which is sensitive to outliers! And it learns with an optimization technique like SGD or Adam.

Differences and use cases:

Aspect	Perceptron	Logistic Regression
Model Type	Single-layer neural network with a binary activation function.	Linear regression model with a logistic (sigmoid) activation function.
Activation Function	Uses a step function (binary threshold) as the activation function.	Employs the logistic (sigmoid) function as the activation function.
Output	Produces binary output (0 or 1) directly based on the weighted sum of inputs.	Outputs continuous values between 0 and 1, representing probabilities.
Learning Algorithm	Uses a simple update rule to adjust weights based on misclassified examples (Perceptron learning rule).	Utilizes gradient descent or other optimization algorithms to find the best weights by minimizing a cost function (typically cross-entropy).
Decision Boundary	Forms a linear decision boundary, suitable for linearly separable data.	Can create non-linear decision boundaries by using higher-order features or polynomial terms.
Classification Task	Typically used for binary classification tasks.	Can handle both binary and multi-class classification problems with appropriate modifications (One-vs-All, Softmax).
Convergence	Guaranteed to converge if the data is linearly separable.	Converges for most datasets, even when they aren't perfectly linearly separable.

Robustness	Sensitive to outliers and noisy data due to its update rule.	More robust to outliers and noise due to probabilistic output.
Probabilistic Output	Does not provide probabilistic outputs; it's a deterministic model.	Provides probabilistic outputs, which can be interpreted as class probabilities.
Use Cases	Simple and suitable for basic classification tasks with linearly separable data.	Widely used in various classification problems, including those with non-linear decision boundaries.

Q6-



B) - Backpropagation:

1- updating w_5, w_6, w_7, w_8: $\alpha = 0/3$

$$\frac{\partial E_t}{\partial w_i} = \frac{\partial E_t}{\partial out_{o_j}} \frac{\partial out_{o_j}}{\partial in_{o_j}} \cdot \frac{\partial in_{o_j}}{\partial w_i}$$ which o_j is the neoron w_i connected to.

for w_5: $\frac{\partial E_t}{\partial w_5} = \frac{\partial E_t}{\partial out_{o_1}} \frac{\partial out_{o_1}}{\partial in_{o_1}} \frac{\partial in_{o_1}}{\partial w_5}$

$\frac{\partial E_t}{\partial out_{o_1}} \rightarrow -(target_{o_1} - out_{o_1}) = -(0/01 - 0/76) = 0/75$

$\frac{\partial out_{o_1}}{\partial net_{o_1}} \rightarrow out_{o_1} = \sigma(t) \Rightarrow \sigma' \; out_{o_1}(1 - out_{o_1}) = 0/76(0/24) = 0/18$

$\frac{\partial net_{o_1}}{\partial w_5} \rightarrow in_{o_1} = w_5 out_{h_1} + w_7 out_{h_2} + b_2 \Rightarrow \frac{\partial in_{o_1}}{\partial w_5} = out_{h_1} = 0/59$

$$\frac{\partial E_t}{\partial w_5} = 0/75 \times 0/18 \times 0/59 = 0/07$$

$$w'_5 = w_5 - \alpha \frac{\partial E_t}{\partial w_5} = 0/4 - 0/02 = 0/38$$

+ for avoiding all this extra calculation for other weights we define a δ_{o_j}:

$$\delta_{o_j} = \frac{\partial E_t}{\partial out_{o_j}} \frac{\partial out_{o_j}}{\partial net_{o_j}} \Rightarrow \delta_{o_j} = -(target_{o_j} - out_{o_j}) \; out_{o_j}(1 - out_{o_j})$$

$$\frac{\partial E_t}{\partial w_i} = \delta_{o_j} out_{h_k}$$ (o_j is neuron in forward layer h_k in α = back (opar))

$w'_6 := \delta_{o_2} out_{h_1} = -(0/99 - 077) \times 0,77(0/23) \times 0/59 = -0/02$

$w_6 - \alpha$

$w'_6 = 0,45 - (0,8 \times (-0/02)) = 0/44$

$w'_7 := \delta_{o_1} out_{h_2} = -(0/01 - 0,76) \times 0/76(0/24) \times 0,6 = 0/08$

$w'_7 = 0/50 - (0/8 \times 0/08) = 0,48$

$w'_8 = \delta_{o_2} out_{h_2} = 0,55 + 0/006 = 0/556$

2- updating w_1, w_2, w_3, w_4:

using delta rule to make it easier:

$$\frac{\partial E_t}{\partial w_i} = \left(\sum_o \frac{\partial E_t}{\partial out_o} \times \frac{\partial out_o}{\partial in_o} \times \frac{\partial in_o}{\partial out_{hk}}\right) \times \frac{\partial out_{hk}}{\partial in_{hk}} \times \frac{\partial in_{hk}}{\partial w_i}$$

$$= \left(\sum_o \delta_o \times w_{ho}\right) \times out_{hk}(1 - out_{hk}) \times i_i$$

$\delta h_1 = 0{,}009$

$$w_1' := w_1 - \alpha\left(\left(\sum_o \delta_o w_{ho}\right) \times out_{h_1}(1 - out_{h_1}) \times i_i\right)$$

$$= w_1 - 0{,}3\left(\left(\underbrace{\delta_{o_1} \times w_5 + \delta_{o_2} \times w_6}_{0{,}038}\right) \times 0{,}59\,(0{,}41) \times 0{,}05\right) =$$

$\delta_{o_1} = 0{,}13 \quad \delta_{o_2} = -0{,}03$

$$\rightarrow w_1 - (0{,}3 \times 0{,}0008) = 0{,}15 - 0{,}0002 = 0{,}1498$$

$\delta h_2 = 0{,}009$

$$w_2' := w_2 - \alpha\left(\left(\underbrace{\delta_{o_1} \times w_7 + \delta_{o_2} \times w_8}_{0{,}04}\right) \times out_{h_2}(1 - out_{h_2}) \times i_i\right) =$$

$$w_2' = w_2 - (0{,}3 \times 0{,}0004) = 0{,}2 - 0{,}0001 = 0{,}1999$$

$$w_3' := w_3 - \alpha\left(\left(\delta_{o_1} w_5 + \delta_{o_2} w_6\right) \times out_{h_1}(1 - out_{h_1}) \times i_2\right)$$

$$= w_3 - (0{,}3 \times 0{,}0009) = 0{,}25 - 0{,}0002 = 0{,}2498$$

$$w_4' = w_4 - \alpha\left(\left(\delta_{o_1} w_7 + \delta_{o_2} w_8\right) \times out_{h_2}(1 - out_{h_2}) \times i_2\right) =$$

$$w_4' = 0{,}3 - (0{,}3 \times 0{,}0009) = 0{,}3 - 0{,}0002 = 0{,}2998$$

3- update biases:

by using delta rule: $b' = b - \alpha \delta_k'$ layer l neuron k.

$$b_{2o_1}' := b_{2o_1} - \alpha \delta_{o_1} = 0{,}6 - (0{,}1 \times 0{,}3) = 0{,}57$$

$$b_{2o_2}' = b_{2o_2} - \alpha \delta_{o_2} = 0{,}6 - ((-0{,}03) \times 0{,}3)) = 0{,}59$$

$$b_1' h_1 = b_1 h_1 - \alpha \delta h_1 = 0{,}35 - (0{,}3 \times 0{,}009) = 0{,}34$$

$$b_1' h_2 = b_1 h_2 - \alpha \delta h_2 = 0{,}35 - (0{,}3 \times (0{,}009) = 0{,}34$$

Q5-

In the realm of neuron connectivity, two contrasting concepts prevail: Sparse and Dense connectivity.
Within the domain of neural networks, a dense layer manifests as a network layer wherein every neuron establishes connections with every other neuron in the preceding layer. Conversely, in a sparse layer, each neuron selectively connects only to a subset of neurons from the previous layer.
The primary allure of sparse layers lies in their computational efficiency, coupled with their adeptness at curbing overfitting and reducing overall complexity. These layers tend to excel in domains such as Natural Language Processing (NLP), where input data, often text, is inherently noisy.
Pruning stands as a viable option to transition a densely connected neural network into a sparser form.
Optimal Brain Damage (OBD): OBD aims to minimize the network's complexity by identifying and eliminating redundant weights. It involves evaluating the second-order derivatives of the loss function, typically represented as the Hessian matrix, with respect to each weight. Mathematically, this can be expressed as:

$$[\frac{\partial^2 \mathcal{L}}{\partial w_i \partial w_j}]$$

where $(\mathcal{L})$ denotes the loss function and (w_i) and (w_j) represent individual weights. Weights with second derivatives close to zero are deemed insignificant and are pruned from the network.
Sensitivity-Based Pruning: Sensitivity-based pruning involves quantifying the sensitivity of the network's output to changes in individual weights. This is typically accomplished through gradient calculations using techniques like backpropagation. Mathematically, the sensitivity of the output y with respect to a weight :

$$[\frac{\partial y}{\partial w_i}]$$

Magnitude-Based Pruning: Magnitude-based pruning ranks weights based on their magnitudes and selectively removes those below a predefined threshold. Mathematically, this process involves comparing the absolute values of individual weights:

$$[|w_i| < \text{threshold}]$$

where weights below the threshold are pruned, reducing the network's overall complexity**.**