If an inadequate number of neurons are used, the network will be unable to model complex data, and the resulting fit will be poor. If too many neurons are used, the training time may become excessively long, and, worse, the network may over fit the data. When overfitting occurs, the network will begin to model random noise in the data. The result is that the model fits the training data extremely well, but it generalizes poorly to new, unseen data. Validation must be used to test for this. For choosing number of neurons the following was considered:

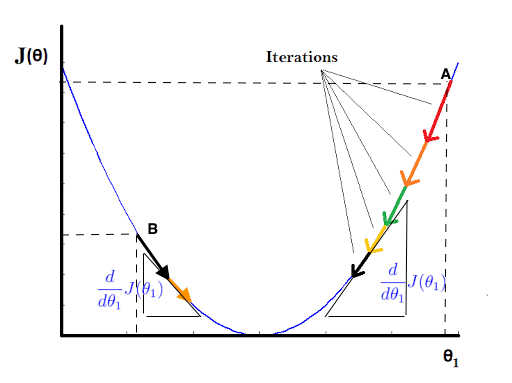
* number of hidden neurons the size of the input layer, plus the size of the output layer;
* number of hidden neurons < twice the size of the input layer.

Deep learning provides evidences to get the most reliability findings in pattern recognition. The universal function approximation theorem points out that one hidden layer is enough for approximation purposes. While most “useful” functions can be approximated with a neural network that has a single hidden layer, the number of neurons in that layer can grow exponentially. On the extreme case, we can use a very large single hidden layer as a lookup table for all different valid inputs of size *n* bits - there are at most such inputs (Ravfogel 2022). A deeper network can use different layers to abstract increasingly complex features of the input, for example, a first layer to detect edges, a second layer to detect simple shapes, a third layer to detect objects composed of these shapes, etc. This often leads to better generalization ability. Neural networks need multiple layers in order to learn more detailed and more abstractions relationships within the data and how the features interact with each other on a non-linear level. If we look carefully at how a single layer is sufficient, we'll also see that it's very easy to over fit by just adding locally correct segments. There are plenty of different ways NN can converge, but lots of them are going to be prone to overfitting, at least in some regions. For particularly complex problems, we're more likely to run into that kind of issue. Using multiple layers introduce more activation functions, meaning the features that it produces mid-network are “less linear” with respect to the inputs and, ideally, “more linear” with respect to the desired output. Those terms give a general intuition when remembering that each layer is essentially a linear regression. With more layers, we can, but won't necessarily, get a smoother transition from the domain to the codomain, improving generalization and reducing overfitting. It is also often the case that we can get similar performance with fewer parameters by using more layers. Adding more layers allows for more easy representation of the interactions within the input data, as well as allows for more abstract features to be learned: features of one layer may be used as input into the next hidden layer. As we get deeper into the network, the features it will have learned will be much more complex and much more abstract. The first layer might learn simple features that signify the object we are trying to classify. The deeper layers will learn much more abstract, detailed features about the objects, and then eventually sets of the sets from the earlier layers that make up the object we are trying to classify. Deep is essentially features’ learning. We need to have many layers of abstraction since we want the neural network to learn as well as possible what type of non-linear manifold in the high dimensions the input data lies on. Shallower layers are the layers closer to input layer, while deeper layers are those more distant from input layer (Quora 2022, Stats 2022). The idea is compositionality - we want each lower level layer to feed a layer above such that the upper layer builds features based on the composition of features from the lower layers. The hierarchy of grouping is infinite, we can go on, like a group of libraries, a group of group of libraries and so on. A single high-level feature can be abstract and more meaningful than a low level feature. For example, a word is more meaningful than a letter, a sentence more meaningful than a word and a paragraph even more meaningful. High-level features composed of simpler features also have a good signal-to-noise ratio compared to, say, a set of pixels with a lot of noise. More hidden layers mean more representational power for the model. Mathematically, a single layer gives one linear representation. But it is usually not enough to approximate the output. So more than one layers are being used with non-linearity in-between. This prevents the consecutive layers from condensing into one. One can think of approximating a complex curve using straight lines. Since it cannot be done by a one line, so multiple lines are being used, and we have to be careful that no line equation get added directly with the other (Quora 2022).

Training of the NN model is done by repeatedly updating the weights and biases of the NN and by carrying out forward and backward propagation in order to minimize the loss function. NN tries to separate the input space to make it possible for generalization in the current feature space. Although each layer seems similar to other layers, because of the cost function, it's corresponding weights are set to decrease the cost function. In neural network model, we seek to minimize the error, i. e. we try to find the “best” values of the model parameters that minimize the loss function. In theory, there are various types of loss functions: Regression Loss Function; Binary Classification Loss Functions; Multi-class Classification Loss Functions. A loss function/error function is for a single training example/input. Loss function represents the price paid for inaccuracy of predictions in classification problems (problems of identifying which category a particular observation belongs to). The value calculated by the loss function is referred to as simply “loss”. A cost function, on the other hand, is the average loss over the entire training dataset. The optimization strategies aim at “minimizing the cost function”.

As such, the objective function is referred to as a cost function. In order to minimize the cost function, neural network uses Gradient descent method. *Gradient descent* enables a model to learn the gradient or direction that the model should take in order to reduce errors (differences between actual label and predicted label ). The main idea is to compute the value of the loss function at every value of the model parameters and iteratively update the model parameters, to take steps in the local direction of steepest (negative) slope, i.e., the negative gradient. ***Gradient descent*** is a [first-order](https://en.wikipedia.org/wiki/Category:First_order_methods) [iterative](https://en.wikipedia.org/wiki/Iterative_algorithm) [optimization](https://en.wikipedia.org/wiki/Mathematical_optimization) [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding a [local minimum](https://en.wikipedia.org/wiki/Local_minimum) of a differentiable function. To find a local minimum of a function using gradient descent, we take steps proportional to the negative of the [gradient](https://en.wikipedia.org/wiki/Gradient) (or approximate gradient) of the function at the current point ([Singh](https://medium.com/@garingh128?source=post_page-----3d3ba3823fd4--------------------------------) 2020). In other words, *Gradient descent* is a mathematical tool for finding the direction of steepest descent. In multivariable calculus, the *gradient* of a point on a surface represents the direction that has the steepest increase. The *gradient* could be represented mathematically as a partial derivative of height over change in that direction. It is therefore also true that the negative gradient represents the direction of steepest descent. In other words, g*radient* of the function is a vector field and means change in the value of a quantity with change in a given variable and especially per unit on a linear scale. A gradient measure how much the output of a function changes if you change the inputs a little bit.

*Gradient descent* builds on the ideas described above by finding the direction of steepest decrease, “moving” in that direction, and then finding the direction of steepest decrease again by taking the negative of the gradient. This is kind of like a ball rolling down a mountain. When the hill is steep, we go down fast, and when the hill becomes less steep, we go slower (this is how Adam optimizer varies the learning rate). *Gradient descent* is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. It finds the global minimum of the objective function - this is feasible if the objective function is convex, i.e. any local minimum is a global minimum. Thus, *Gradient descent* (also called Batch gradient descent) computes the gradient of the cost function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent. Gradient descent is used to find the values of a function's parameters (coefficients) that minimize a cost function as far as possible. Consider a cost function and the objective, (minimize a cost function according to all . Start off by **random initialization** say at point A as shown in the plot below. Then according to the update definition of the *Gradient descent* algorithm, the **derivative term is represented by the slope at any point** also shown in the plot. In this case it would be a **positive slope** and, since the **learning rate** that basically defines how big the steps are during the descent, is a positive real number, the overall term – would be negative. This means the value of will be decreased in magnitude as shown by the arrows (Fig. 1.6.) ([Shams](https://machinelearningmedium.com/about/) 2022).



**Fig. 1.6.** Gradient descent steps ([Shams S](https://machinelearningmedium.com/about/)**.** 2022)

In the context of neural networks, negative gradient descent derivatives represent how the output will change with relation to each weight value by calculating the negative partial derivative of the output with relation to each weight () value in the whole network. These weight values change according to the principles of gradient descent for each batch of training data that the neural network is trained on. Iteratively repeating the process of running the neural network on a set of training data points, applying a *gradient descent* protocol, and adjusting the weights of the network, allows the network to activate very specific neural patterns in response to specific inputs thereby allowing it to learn, minimizing its cost function. In the example in Fig. 1.8., no matter where the value is initialized, the algorithm ensures that parameter is updated in the right direction towards the minima, given proper value for learning rate which basically defines how big the steps are during the descent is chosen. will determine the magnitude of the update term, i.e. if the value is higher the steps of update would be proportionally larger. In case we have higher dimensionality problem, the parameters (weights) should be updated simultaneously. The notion of simultaneous update is introduced because that is how the gradient descent would work naturally, i.e. in nature the path taken at a point would be defined by the gradient along components at a point. But if the update is not simultaneous then the gradient is not computed at the same point because the updated value of one parameter is used in calculating the update of another ([Shams S](https://machinelearningmedium.com/about/)**.** 2022).

A machine learning model always wants low error with maximum accuracy. Generally, steps to implement *Gradient Descent* are the following (Mazur 2022):

* randomly initialize values;
* update values: *=* where is loss function (derivative of error with respect to weight); here is a learning rate;
* repeat until .

Standard *back-propagation* is just *gradient desc*ent with chained functions where the network *weights are moved along the negative of the gradient of the loss function.* Through the *back propagation* loss function is reduced. It is a way of propagating the total loss back into the neural network to know how much of the loss every node is responsible for, and subsequently updating the weights in such a way that minimizes the loss by giving the nodes with higher error rates lower weights and vice versa. For a single training example, *back-propagation* algorithm calculates the gradient of the error function. Generally, the steps involved are: forward propagation -> backward propagation -> putting all the values together and calculating the updated weight value (Mazur 2022).

The *back-propagation* algorithm is used to compute the gradient and is used in conjunction with *gradient descent* to update the weights in a direction that moves it closer to optimizing the objective function. The gradient is defined as the vector of partial derivatives with respect to the weights. Normally how a gradient in multivariable calculus is computed is that a function in terms of each of the weight variables is used and each partial derivative is directly computed. And such a function does exist between each layer of the neural network, but in real world neural networks are with billions of features and billions of values – trying directly compute this gradient from a function is very impractical. For backpropagation, the intuition follows directly from the chain rule. The values of the penultimate layer are computed at first so that they act as inputs to the final outputs. Then the two-layer problem must be solved. Once the penultimate weights are fixed, they are treated as outputs of the layer before that and so on until weights of the first layer will be found. Doing this and iterating as a part of the *gradient descent* process, a converged set of weights is obtained. The set is such that the ultimate outputs given the bottom layer inputs are as close to the real thing as possible for all of the examples. In other words, the calculation of partial derivatives is split up into calculation of intermediate derivatives and these derivatives are combined through the chain rule. The chain rule lets to turn a set of conditional probability distributions into a joint probability distribution (Hmkode 2019). This process is similar to linear regression in that it involves iteratively optimizing an initial guess for the weights with many gradient descent steps. Each training instance is sent through the network in a “forward pass” and a prediction is made using the current guess for the weights. The total error is then calculated and propagated back through the network in a “reverse pass.” This way, we can figure out which neurons are at most to “blame” for the error and we can update our weights accordingly in order to minimize error. A forward pass and a backward pass together are counted as one pass.

*Back-propagation* performs one gradient descent step for each “batch” of training instances. The number of batches it takes to go through all of the training instances once is equivalent to one “epoch”. It can take many epochs of training to arrive at a good set of weights. But in case there it will be too many epochs we have a risk of overfitting. Unlike gradient descent, *back-propagation* does not make its initial guess for the weights with a simple random initialization - this actually doesn’t work very well when training deep NN. Different initialization parameters are used depending on the choice of activation function. By default, *Keras* (an open-source software library that provides a Python interface for artificial neural networks) uses *Xavier(glorot*) initializer (*GlorotUniform* class). Initializers define the way to set the initial random weights of *Keras* layers. Each layer has its own default value for initializing the weights. The default Glorot initializer draws samples from a uniform distribution within where limit is , where is the number of input units in the weight tensor and is the number of output units in the weight tensor. The bias initializer by default is Each neuron has a *bias*. Biases are learnable as weights are. Bias' role is similar to that of a threshold. Bia*s* determines whether or not or by how much neuron will fire. Bias increases the flexibility of the model. Bias gets added to the weighted sum before being passed to the activation function. After being initialized, during training, these biases will be updated as the model learns the optimized values for them. If we were to train the model and then call the function which returns weights and/or biases, then the values for the weights and biases would likely be very different.

For most cases, the neural network will generalize better if it is trained on a larger data set. This is because of that a neural network is fundamentally trying to approximate the data in the form of the function. The more complex the data, the more difficult it would be to approximate it. Larger data sets help to generalize better as long as the data set is unbiased and the function which we are trying to approximate is complex enough. The size of the data set may manifest issues relating to generalization, data imbalance, and difﬁculty in reaching the global optimum. A large dataset helps to avoid overﬁtting and generalizes better as it captures the inherent data distribution more effectively. Small size of the data set may be one of the reasons why the NN prediction accuracy obtained is not high, however, this does not fundamentally change the suitability of the particular approach and/or method for the learner’s learning-style prediction.

It is important for NN that the data is clean. In NN, noise is estimated by a reasonable target error, such as human performance. There is nothing we can do about noise as this is an error inherent to the structure of the problem and it cannot be avoided.

Bias is estimated as the training set error compared to noise. Increasing the size of the training set will not help reduce bias. We can reduce bias by increasing the complexity of the model and training a larger number of parameters, changing the model to, for example, apply different activation functions, or reduce regularization. Regularization (*L1 Regularization or Lasso Regularization; L2 Regularization or Ridge Regularization*) refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or under fitting.

Variance is estimated as the generalization error compared to the training error. We may fight variance with regularization or a reduced complexity, but it is best to increase the size of the training set if this is possible. The famous “rule of ten”, which is an empirical rule of thumb, not a mathematical truth, recommends a training set of size ten times the number of free parameters. Of the three terms that constitute the prediction error, to increase the size of the training set only helps with variance. This is called the *bias-variance decomposition* and it constitutes the scientific basis for diagnosing and fixing problems with machine learning models in general, and deep learning in particular.

It is extremely important to shuffle the training data, so that entire mini batches of highly correlated examples wouldn’t be obtained. Different random orderings will perform slightly differently from each other but this will be a small factor that does not matter much.

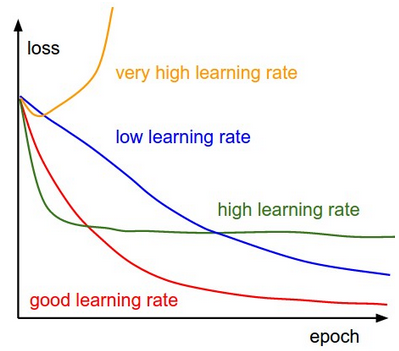
Following [Holzbauer](https://medium.com/@lauren.holzbauer?source=post_page-----b0998c6216a1--------------------------------) (2019), once the weights are initialized, the algorithm begins training. For one batch of training instances*, back-propagation* makes initial test predictions by feeding all of the training instances in the current batch through the network in a “forward pass”, keeping track of the outputs of each neuron within each layer along the way. The output error is then computed by comparing initial predictions with ground truth values. After the total output error is obtained, the gradient of error must be computed in order to plug this into the update equation and tweak the weights. The error gradient is determined by making a “reverse pass” through the network, starting with the output and ending with the input. Since the output error is known and the track of the input/output values is kept for each neuron during the forward pass, it is possible to propagate the error backwards through the network and figure out how much each neuron contributed to the total error. This allows to directly compute the error gradient across each neuron and tells how to tweak that neuron’s associated weight. In general, there are many different variations of the update equation. Each variant is called an optimizer. There is a plethora to choose from, including *Gradient Descent (GD)*, *Stochastic Gradient Descent (SGD)*, and *Mini Batch* Gradient Descent, etc. *Stochastic gradient descent* is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or sub differentiable). It can be regarded as a stochastic approximation of *gradient descent optimization*, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate.

The purpose of *back-propagation* is to define the partial derivative of the cost function with respect to a particular weight. *Gradient descent* may be used to optimize the weights and thus optimize the network. If we do this with even one example, it is called s*tochastic gradient descent*. One can continuously take one example at a time and apply *stochastic gradient descent* until the weights aren’t changing much. This comes in handy when storing all data in memory isn’t feasible because one can just read a single data point into memory and still optimize the network (Nanda 2016). For comparison, *Vanilla gradient descent* (also called *Batch gradient descent*) calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated (it uses cumulative error). This whole process is like a cycle and it's called a training *epoch*. *Batch gradient descent*, at all steps, takes the steepest route to reach the true input distribution. SGD, on the other hand, chooses a random point within the shaded area, and takes the steepest route towards this point. At each iteration, though, it chooses a new point. SGD picks up a  
“random” instance of training data at each step and then computes the gradient making it much faster as there is much fewer data to manipulate at a single time, unlike *Batch GD*. Some advantages of *batch gradient descent* are its computational efficient, it produces a stable error gradient and a stable convergence. Some disadvantages are the stable error gradient can sometimes result in a state of convergence that isn’t the best the model can achieve. It also requires the entire training dataset be in memory and available to the algorithm.

*Optimizers* within the *GD* family are usually challenged in that the learning rate must be defined a priori and they can get stuck in local minima or around saddle points if the error function is not convex (as it is for linear regression). Learning rate (also called the step size) is the amount that the weights are updated during training. Learning rate (also called the step size) is the amount that the weights are updated during training. Learning rate must be chosen wisely as ([Trehan](https://dakshtrehan.medium.com/?source=post_page-----9b953fc0d2c--------------------------------) 2020):

* if it is too small, then the model will take some time to learn (will lead to slow training);
* if it is too large, model will converge as the pointer will shoot and we’ll not be able to get to minima (will lead to overshooting of slope).

In simple words, every step taken towards minima tends to decrease the slope; in steep region of curve derivative is going to be large therefore steps taken by the model would be large, too, but as we will enter gentle region of slope the derivative will decrease and so will the time to reach  
minima ([Trehan](https://dakshtrehan.medium.com/?source=post_page-----9b953fc0d2c--------------------------------) 2020).



**Fig. 1.7.** Gradient descent with different learning rates (Stanford CS class 2020)

The plot above tries to summarize the effect of value on the convergence of the gradient descent algorithm (Fig. 1.7.):

* the yellow plot shows the divergence of the algorithm when the learning rate is really high wherein the learning steps overshoot;
* the green plot shows the case where learning rate is not as large as the previous case but is high enough that the steps keep oscillating at a point which is not the minima;
* the red plot would be the optimum curve for the cost drop as it drops steeply initially and then saturates very close to the optimum value;
* the blue plot is the least value of and converges very slowly as the steps taken by the algorithm during update steps are very small.

Generally, learning rate is selected by hyperparameter optimization. In selecting a learning rate, trade-off between the rate of convergence and overshooting should be found. In general, the optimal learning rate is about half of the maximum learning rate (i.e. the learning rate above which the training algorithm diverges). Following Geron (2019), one way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g., ) and gradually increasing it up to a very large value (e.g., ). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by to go from to 10 in 500 iterations). If we plot the loss as a function of the learning rate (using log scale for a learning rate), we should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the optimal learning rate will be a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point). We can then reinitialize the model and train it normally using this good learning rate (Geron 2019).

*Stochastic gradient descent* maintains a single learning rate (often termed *alpha*) for all weight updates and the learning rate does not change during training. In order to train network with different learning rates, learning rate scheduler may be prepared. Adaptive optimizers embed learning rate optimization directly into the optimizer itself. *Adam* optimizer, for example, goes one step further, managing the learning rates on a per-weight basis and using learning rate with decay. *Simple (Vanilla) Gradient Descent* completely relies only on calculation, i.e. if there are steps, then the NN model will try to implement *Simple Gradient Descent* for times that would be obviously too much time consuming and computationally expensive. In order to avoid drawbacks of *Vanilla* *Gradient Descent*, *Momentum based Gradient Descent* may be introduced, where the goal is to lower the computation time, and that can be achieved introducing the concept of experience i.e. the confidence using previous steps. Rather than computing new steps again and again we are averaging the decay, and as decay increases, its effect in decision making decreases and thus the older the step less effect on decision making. More the history (experience) more bigger steps will be taken. This is the case of *Momentum based GD* where due to high experience the model is taking larger steps that is leading to overshooting and hence missing the goal but to achieve minima model have to trace back its steps as ([Trehan](https://dakshtrehan.medium.com/?source=post_page-----9b953fc0d2c--------------------------------) 2020). A better alternative is to choose an optimizer that utilizes an “adaptive learning rate“, such as *Adaptive Gradient Algorithm (AdaGrad)*, *AdaDelta*, *Root Mean Square Propagation (RMSProp)*, or *Adam*. For example, *Adam* serves for first-order gradient-based optimization of stochastic objective functions. This algorithm is used to accelerate *the Gradient descent* algorithm by taking into consideration the „exponentially weighted average“ of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace. *Adam* is based on first- and second-order statistical moments, i.e. mean and variance. In the *Adam optimization* method, the learning rate is a hyperparameter and needs to be tuned. *Adam* optimizer updates any parameter (weight, bias) with an individual learning rate. This means that every parameter in the network has a specific learning rate associated. But the single learning rate for each parameter is computed using the - initial learning rate as an upper limit. This means that every single learning rate can vary from 0 (no update) to (maximum update). The learning rates adapt themselves during training steps, but if we want to be sure that every update step doesn't exceed we can then lower using exponential decay or whatever. It can help to reduce loss during the latest step of training, when the computed loss with the previously associated parameter has stopped to decrease. As *Adam* is an algorithm for gradient-based optimization of stochastic objective functions, it is a replacement optimization algorithm for *stochastic gradient descent*. In the *Adam* algorithm, the training sample changes with time, and the objective function changes also because it is defined on the given train sample. Therefore, there are series of objective functions that form a stochastic process (from Cross Validated 2020). Adam combines the advantages of two extensions of *stochastic gradient descent*, specifically ([Brownlee](https://machinelearningmastery.com/author/jasonb/" \o "Posts by Jason Brownlee) 2017):

* *Adaptive Gradient Algorithm* (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients;
* *Root Mean Square Propagation* (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Adaptive Moment Estimation (Adam) is a method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum (a product of the mass of a particle and its velocity). Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. Adam uses estimates of the first and second moments of the gradient to perform updates, which can be seen as incorporating gradient descent with momentum (the first-order moment) and RMSProp algorithm (the second-order moment) (Ruder 2016). Instead of adapting the parameter learning rates based on the average first momentum (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters and control the decay rates of these moving averages. The initial value of the moving averages and values close to (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates (Damarla 2022). *Adam* optimizer is well suited for large data sets and is computationally efficient. The disadvantage is that although the *Adam* optimizer tends to converge faster, but other algorithms like the *Stochastic gradient descent* focus on the data points (in general, momentu m is defined as a small subset or random selection of data examples) and generalize in a better manner.