## How does a batch size influence a model?

*Increasing batch size drops the learners' ability to generalize.* The idea is that smaller batches are more likely to push out local minima and find the Global Minima.

large batch size means the model makes very large gradient updates and very small gradient updates. The size of the update depends heavily on which particular samples are drawn from the dataset. On the other hand using small batch size means the model makes updates that are all about the same size. The size of the update only weakly depends on which particular samples are drawn from the dataset

* Large Batch Training methods tend to overfit compared to the same network trained with smaller batch size.
* Large Batch Training methods tend to get trapped or even attracted to potential saddle points in the loss landscape.
* Large Batch Training methods tend to zoom in on the closest relative minima that it finds, whereas networks trained with a smaller batch size tend to “explore” the loss landscape before settling on a promising minimum.
* Large Batch Training methods tend to converge to completely “different” minima points than networks trained with smaller batch sizes.

Furthermore, the authors tackled the Generalization Gap from the perspective of how Neural Networks navigate the loss landscape during training. Training with a relatively large batch size tends to converge to sharp minimizers, while reducing the batch size usually leads to falling into flat minimizers. A sharp minimizer can be thought of as a narrow and steep ravine, whereas a flat minimizer is analogous to a valley in a vast landscape of low and mild hill terrains. To phrase it in more rigorous terms:

***Sharp minimizers****are characterized by a significant number of large positive eigenvalues of the Hessian Matrix of f(x),* *while****flat minimizers****are characterized by a considerable number of smaller positive eigenvalues of the Hessian Matrix of* *f(x)*.

“Falling” into a sharp minimizer may produce a seemingly better loss than a flat minimizer, but it’s more prone to generalizing poorly to unseen datasets. The diagram below illustrates a simple 2-dimensional loss landscape from Keskar et al.

Изображение выглядит как линия, текст, График, диаграмма

Автоматически созданное описание

A sharp minimum compared to a flat minimum. From Keskar et al.

We assume that the relationship between features and labels of unseen data points is similar to that of the data points that we used for training but not exactly the same. As the example shown above, the “difference” between train and test can be a slight horizontal shift. The parameter values that result in a sharp minimum become a relative maximum when applied to unseen data points due to its narrow accommodation of minimum values. With a flat minimum, though, as shown in the diagram above, a slight shift in the “Testing Function” would still put the model at a relatively minimum point in the loss landscape.

Typically, adopting a small batch size adds noise to training compared to using a bigger batch size. Since the gradients were estimated with a smaller number of samples, the estimation at each batch update will be rather “noisy” relative to the “loss landscape” of the entire dataset. Noisy training in the early stages is helpful to the model as it encourages exploration of the loss landscape. Keskar et al. also stated that…

*“We have observed that the loss function landscape of deep Neural Networks is such that large-batch methods are attracted to regions with sharp minimizers and that, unlike small-batch methods, are****unable to escape basins of attraction of these minimizers.****”*

Although larger batch sizes are considered to bring more stability to training, the noisiness that small batch training provides is actually beneficial to explore and avoiding sharp minimizers. We can effectively utilize this fact to design a “batch size scheduler” where we start with a small batch size to allow for exploration of the loss landscape. Once a general direction is decided, we hone in on the (hopefully) flat minimum and increase the batch size to stabilize training. The details of how one can increase the batch size during training to obtain faster and better results are described in the following article.

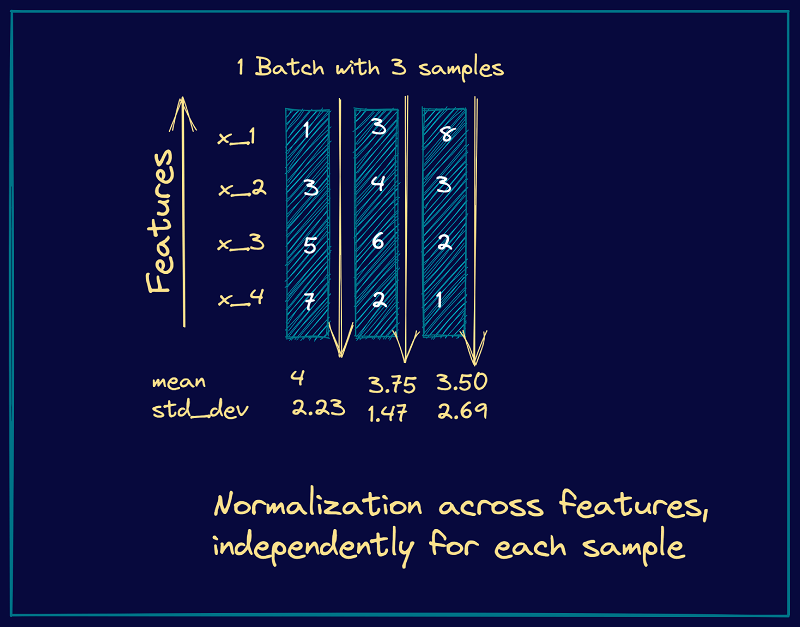
## What is Layer Normalization?

[Layer Normalization](https://arxiv.org/abs/1607.06450) was proposed by researchers Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. In layer normalization, all neurons in a particular layer effectively have the same distribution across all features for a given input.

For example, if each input has **d** features, it’s a d-dimensional vector. If there are **B** elements in a batch, the normalization is done along the length of the d-dimensional vector and not across the batch of size **B**.

Normalizing *across all features* but for each of the inputs to a specific layer removes the dependence on batches. This makes layer normalization well suited for sequence models such as [transformers](https://www.pinecone.io/learn/sentence-embeddings/) and [recurrent neural networks (RNNs)](https://www.ibm.com/cloud/learn/recurrent-neural-networks) that were popular in the pre-transformer era.

Here’s an example showing the computation of the mean and variance for layer normalization. We consider the example of a mini-batch containing three input samples, each with four features.



Изображение выглядит как текст, рукописный текст, Шрифт, каллиграфия

Автоматически созданное описание

From these steps, we see that they’re similar to the steps we had in batch normalization. However, instead of the batch statistics, we use the mean and variance corresponding to specific input to the neurons in a particular layer, say **k**. This is equivalent to normalizing the output vector from the layer **k-1**.

## Batch Normalization vs Layer Normalization

Batch normalization normalizes each feature independently across the mini-batch. Layer normalization normalizes each of the inputs in the batch independently across all features.

As batch normalization is dependent on batch size, it’s not effective for small batch sizes. Layer normalization is independent of the batch size, so it can be applied to batches with smaller sizes as well.

Batch normalization requires different processing at training and inference times. As layer normalization is done along the length of input to a specific layer, the same set of operations can be used at both training and inference times.

Изображение выглядит как куб

Автоматически созданное описание

Batch Normalization in Convolutional Neural Networks

Batch Norm works in a very similar way in Convolutional Neural Networks. Although we could do it in the same way as before, we have to follow the convolutional property.

In convolutions, we have shared filters that go along the feature maps of the input (in images, the feature map is generally the height and width). These filters are the same on every feature map. It is then reasonable to normalize the output, in the same way, sharing it over the feature maps.

In other words, this means that the parameters used to normalize are calculated along with each entire feature map. In a regular Batch Norm, each feature would have a different mean and standard deviation. Here,**each feature map will have a single mean and standard deviation**, used on all the features it contains.

## Why can’t I use Softmax on the hidden layer?

The following steps explain why using the softmax function on the hidden layer is not a good idea:

1. **Variables independence:** A lot of regularization and effort is required to keep your variables independent, uncorrelated and quite sparse. If you use the softmax layer as a hidden layer, then you will keep all your nodes linearly dependent which may result in many problems and poor generalization.

2.**Training issues**:  if your network is working better, you have to make a part of activations from your hidden layer a little bit lower. Here automatically you are making the rest of them have mean activation on a higher level which might, in fact, increase the error and harm your training phase.

3. **Mathematical issues:** If you create constraints on activations of your model you decrease the expressive power of your model without any logical explanation.

4. **Batch normalization does it better:** You may consider the fact that mean output from a network may be useful for training. But on the other hand, a technique called Batch Normalization has been already proven to work better, but it was reported that setting softmax as the activation function in a hidden layer may decrease the accuracy and speed of learning.

## Global Average Pooling

* The feature maps of the last convolutional layer are vectorized and fed into fully connected layers followed by a softmax logistic regression layer. This structure bridges the convolutional structure with traditional neural networks. It treats the convolutional layers as feature extractors, and the resulting feature is classified in a traditional way.
* The fully connected layers are prone to overfitting. You can use Dropout as a regularizer which randomly sets half of the activations to the fully connected layers to zero during training. It has improved the generalization ability and largely prevents overfitting.
* You can use another strategy called global average pooling to replace the Flatten layers in CNN. It generates one feature map for each corresponding category of the classification task in the last Conv layer.

## What are some of the uses of Autoencoders in Deep Learning?

An autoencoder is a type of [artificial neural network](https://www.v7labs.com/blog/neural-network-architectures-guide) used to learn data encodings in an unsupervised manner.

The aim of an autoencoder is to learn a lower-dimensional representation (encoding) for a higher-dimensional data, typically for dimensionality reduction, by [training the network](https://www.v7labs.com/training) to capture the most important parts of the input image.

* Autoencoders are used to convert black and white images into colored images.
* Autoencoder helps to extract features and hidden patterns in the data.
* It is also used to reduce the dimensionality of data.
* It can also be used to remove noises from images.

## Why is a convolutional neural network preferred over a dense neural network for an image classification task?

* The number of parameters in a convolutional neural network is much more diminutive than that of a Dense Neural Network. Hence, a CNN is less likely to overfit.
* CNN allows you to look at the weights of a filter and visualize what the network learned. So, this gives a better understanding of the model.
* CNN trains models in a hierarchical way, i.e., it learns the patterns by explaining complex patterns using simpler ones.

## Xavier (Glorot) initialization

Xavier initialization sets a layer’s weights to values chosen from a random uniform distribution that’s bounded between +-sqrt(6/(ni+ni+1))

Изображение выглядит как Шрифт, белый, символ, линия

Автоматически созданное описание

where *nᵢ* is the number of incoming network connections, or “fan-in,” to the layer, and *nᵢ₊₁* is the number of outgoing network connections from that layer, also known as the “fan-out.”

## What is an optimizer?

**Optimizers** are algorithms or methods used to minimize an error function(*loss function*)or to maximize the efficiency of production. Optimizers are mathematical functions which are dependent on model’s learnable parameters i.e Weights & Biases. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

## List Optimizer Types

Gradient Descent, Stochastic Gradient Descent, Mini-Batch Gradient Descent, SGD with Momentum, Nesterov Accelerated Gradient, RMS-Prop, AdaGrad(Adaptive Gradient Descent), AdaDelta, Adam

## Gradient Descent

We now consider the problem of solving for the minimum of a real-valued function min x f(x), where f : Rd → R is an objective function that captures the machine learning problem at hand. We assume that our function f is differentiable, and we are unable to analytically find a solution in closed form. Gradient descent is a first-order optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point. Recall that the gradient points in the direction of steepest ascent.

Let us consider multivariate functions. Imagine a surface (described by the function f(x)) with a ball starting at a particular location x0. When the ball is released, it will move downhill in the direction of steepest descent. Gradient descent exploits the fact that f(x0) decreases fastest if one moves from x0 in the direction of the negative gradient −((∇f)(x0))⊤ of f at x0. We assume in this book that the functions are differentiable, and refer the reader to more general settings in Section 7.4. Then, if x1 = x0 − γ((∇f)(x0))⊤ (7.5) for a small step-size γ ⩾ 0, then f(x1) ⩽ f(x0). Note that we use the transpose for the gradient since otherwise the dimensions will not work out. This observation allows us to define a simple gradient descent algorithm: If we want to find a local optimum f(x∗) of a function f : Rn → R, x 7→ f(x), we start with an initial guess x0 of the parameters we wish to optimize and then iterate according to xi+1 = xi − γi((∇f)(xi))⊤ . (7.6) For suitable step-size γi , the sequence f(x0) ⩾ f(x1) ⩾ . . . converges to a local minimum.

In summary, Gradient Descent method’s steps are:

1. choose a starting point (initialisation)
2. calculate gradient at this point
3. make a scaled step in the opposite direction to the gradient (objective: minimise)
4. repeat points 2 and 3 until one of the criteria is met:

* maximum number of iterations reached
* step size is smaller than the tolerance.

Below there’s an exemplary implementation of the Gradient Descent algorithm (with steps tracking):

This function takes 5 parameters:

1.**starting poin**t - in our case, we define it manually but in practice, it is often a random initialisation

2. **gradient functio**n - has to be specified before-hand

3. **learning rate**- scaling factor for step sizes

4. maximum number of iterations

5. tolerance to conditionally stop the algorithm (in this case a default value is 0.01)

Изображение выглядит как черный, темнота

Автоматически созданное описание

**Advantages of Gradient Descent**

1. Easy to understand
2. Easy to implement

**Disadvantages of Gradient Descent**

1. Because this method calculates the gradient for the entire data set in one update, the calculation is very slow.
2. It requires large memory and it is computationally expensive.

## Stochastic Gradient Descent

It is a variant of Gradient Descent. If the model has 10K dataset SGD will update the model parameters 10k times.

Изображение выглядит как круг, Шрифт, рукописный текст, зарисовка

Автоматически созданное описание

Stochastic Gradient Descent

**Advantages of Stochastic Gradient Descent**

1. Frequent updates of model parameter
2. Requires less Memory.
3. Allows the use of large data sets as it has to update only one example at a time.

**Disadvantages of Stochastic Gradient Descent**

1. The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
2. High Variance.
3. Frequent updates are computationally expensive.

## Mini-Batch Gradient Descent

It is a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. it can reduce the variance when the parameters are updated, and the convergence is more stable. It splits the data set in batches in between 50 to 256 examples, chosen at random.

Изображение выглядит как круг, линия, диаграмма, дизайн

Автоматически созданное описание

Mini Batch Gradient Descent

**Advantages of Mini Batch Gradient Descent:**

1. It leads to more stable convergence.
2. more efficient gradient calculations.
3. Requires less amount of memory.

**Disadvantages of Mini Batch Gradient Descent**

1. Mini-batch gradient descent does not guarantee good convergence,
2. If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value.

## SGD with Momentum

**SGD with Momentum** is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent. Momentum simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.

Изображение выглядит как круг, зарисовка, дизайн

Автоматически созданное описание

Изображение выглядит как Шрифт, текст, рукописный текст, белый

Автоматически созданное описание

**Advantages of SGD with momentum**

1. Momentum helps to reduce the noise.
2. Exponential Weighted Average is used to smoothen the curve.

**Disadvantage of SGD with momentum**

1. Extra hyperparameter is added.

## Nesterov Accelerated Gradient

Изображение выглядит как текст, Шрифт, рукописный текст, белый

Автоматически созданное описание

in Nesterov Accelerated Gradient, we apply the velocity *vt* to the parameters *θ* to compute interim parameters *θ*. We then compute the gradient using the interim parameters

We can view Nesterov Accelerated Gradients as the correction factor for Momentum method. Consider the case when the velocity added to the parameters gives you immediate unwanted high loss, e.g., exploding gradient case. In this case, the Momentum method can be very slow since the optimization path taken exhibits large oscillations. In Nesterov Accelerated Gradient case, you can view it like *peeking* through the interim parameters where the added velocity will lead the parameters. If the velocity update leads to bad loss, then the gradients will direct the update back towards *θ*𝑡. This help Nesterov Accelerated Gradient to avoid the oscillations.

## AdaGrad(Adaptive Gradient Descent)

In all the algorithms that we discussed previously the learning rate remains constant. The intuition behind AdaGrad is can we use different Learning Rates for each and every neuron for each and every hidden layer based on different iterations.

Изображение выглядит как рукописный текст, текст, Шрифт, каллиграфия

Автоматически созданное описание

**Advantages of AdaGrad**

1. Learning Rate changes adaptively with iterations.
2. It is able to train sparse data as well.

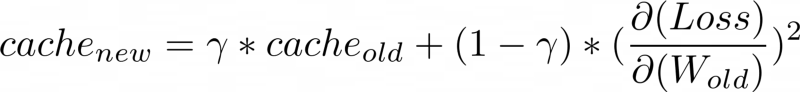
**Disadvantage of AdaGrad**

1. If the neural network is deep the learning rate becomes very small number which will cause dead neuron problem.

## RMS-Prop (Root Mean Square Propagation)

RMS-Prop is a special version of Adagrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients. RMS-Prop basically combines momentum with AdaGrad.

orИзображение выглядит как Шрифт, текст, диаграмма, линия

Автоматически созданное описание

**Advantages of RMS-Prop**

1. In RMS-Prop learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

**Disadvantages of RMS-Prop**

1. Slow Learning

## AdaDelta

Adadelta is an extension of Adagrad and it also tries to reduce Adagrad’s aggressive, monotonically reducing the learning rate and remove decaying learning rate problem. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous parameters to the current gradient.

Изображение выглядит как текст, Шрифт, рукописный текст, линия

Автоматически созданное описание

E – running average

**Advantages of Adadelta**

1. The main advantage of AdaDelta is that we do not need to set a default learning rate.

**Disadvantages of Adadelta**

1. Computationally expensive

## Adam (Adaptive Moment Estimation)

Adam optimizer is one of the most popular and famous gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients, similar to momentum, and also the decaying average of the past squared gradients, similar to RMS-Prop and Adadelta. Thus, it combines the advantages of both the methods.

We compute the decaying averages of past and past squared gradients mt and vt respectively as follows:

Изображение выглядит как Шрифт, текст, рукописный текст, белый

Автоматически созданное описание

They counteract these biases by computing bias-corrected first and second moment estimates:

Изображение выглядит как рукописный текст, Шрифт, диаграмма, линия

Автоматически созданное описание

Изображение выглядит как Шрифт, линия, рукописный текст, диаграмма

Автоматически созданное описание

**Advantages of Adam**

1. Easy to implement
2. Computationally efficient.
3. Little memory requirements.

## How to choose optimizers?

* If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
* RMSprop, Adadelta, Adam have similar effects in many cases.
* Adam just added bias-correction and momentum on the basis of RMSprop,
* As the gradient becomes sparse, Adam will perform better than RMSprop.

## Step size in Gradient Descent

Choosing a good step-size, or learning rate, is important in gradient descent. If the step-size is too small, gradient descent can be slow. If the step-size is chosen too large, gradient descent can overshoot, fail to converge, or even diverge

There are two simple heuristics:

* When the function value increases after a gradient step, the step-size was too large. Undo the step and decrease the step-size.
* When the function value decreases the step could have been larger. Try to increase the step-size.

## Solving a Linear Equation System with Gradient Descent

When we solve linear equations of the form Ax = b, in practice we solve Ax−b = 0 approximately by finding x∗ that minimizes the squared error

∥Ax − b∥^2 = (Ax − b) ⊤(Ax − b)

if we use the Euclidean norm. The gradient of (7.9) with respect to x is

∇x = 2(Ax − b) ⊤A

We can use this gradient directly in a gradient descent algorithm. However, for this particular special case, it turns out that there is an analytic solution, which can be found by setting the gradient to zero.

When applied to the solution of linear systems of equations Ax = b, gradient descent may converge slowly. The speed of convergence of gradient descent is dependent on the condition number κ = σ(A)/max σ(A)min, which is the ratio of the maximum to the minimum singular value of A. The condition number essentially measures the ratio of the most curved direction versus the least curved direction, which corresponds to our imagery that poorly conditioned problems are long, thin valleys: They are very curved in one direction, but very flat in the other. Instead of directly solving Ax = b, one could instead solve P −1 (Ax − b) = 0, where P is called the preconditioner. The goal is to design P −1 such that P −1A has a better condition number, but at the same time P −1 is easy to compute.

## Gradient Descent With Momentum

Gradient descent with momentum (Rumelhart et al., 1986) is a method that introduces an additional term to remember what happened in the previous iteration. This memory dampens oscillations and smoothes out the gradient updates. Continuing the ball analogy, the momentum term emulates the phenomenon of a heavy ball that is reluctant to change directions. The idea is to have a gradient update with memory to implement a moving average. The momentum-based method remembers the update ∆xi at each iteration i and determines the next update as a linear combination of the current and previous gradients

xi+1 = xi − γi((∇f)(xi))⊤ + α∆xi

∆xi = xi − xi−1 = α∆xi−1 − γi−1((∇f)(xi−1))⊤ ,

where α ∈ [0, 1]. Sometimes we will only know the gradient approximately. In such cases, the momentum term is useful since it averages out different noisy estimates of the gradient. One particularly useful way to obtain an approximate gradient is by using a stochastic approximation, which we discuss next

## Stochastic Gradient Descent

Stochastic gradient descent descent (often shortened as SGD) is a stochastic approximation of the gradient descent method for minimizing an objective function that is written as a sum of differentiable functions. The word stochastic here refers to the fact that we acknowledge that we do not know the gradient precisely, but instead only know a noisy approximation to it. By constraining the probability distribution of the approximate gradients, we can still theoretically guarantee that SGD will converge.

Why should one consider using an approximate gradient? A major reason is practical implementation constraints, such as the size of central processing unit (CPU)/graphics processing unit (GPU) memory or limits on computational time. We can think of the size of the subset used to estimate the gradient in the same way that we thought of the size of a sample when estimating empirical means. Large mini-batch sizes will provide accurate estimates of the gradient, reducing the variance in the parameter update. Furthermore, large mini-batches take advantage of highly optimized matrix operations in vectorized implementations of the cost and gradient. The reduction in variance leads to more stable convergence, but each gradient calculation will be more expensive.

In contrast, small mini-batches are quick to estimate. If we keep the mini-batch size small, the noise in our gradient estimate will allow us to get out of some bad local optima, which we may otherwise get stuck in. In machine learning, optimization methods are used for training by minimizing an objective function on the training data, but the overall goal is to improve generalization performance. Since the goal in machine learning does not necessarily need a precise estimate of the minimum of the objective function, approximate gradients using mini-batch approaches have been widely used. Stochastic gradient descent is very effective in large-scale machine learning problems.

## Constrained Optimization and Lagrange Multipliers

we consider the constrained optimization problem

min x f(x)

subject to gi(x) ⩽ 0 for all i = 1, . . . , m .

We associate to problem the Lagrangian by introducing the Lagrange multipliers λi ⩾ 0 corresponding to each inequality constraint respectively so that

L(x,λ) = f(x) +sum\_i=1..m(λi \* gi(x)) = f(x) + λ ⊤ g(x),

where in the last line we have concatenated all constraints gi(x) into a vector g(x), and all the Lagrange multipliers into a vector λ ∈ Rm.

The associated Lagrangian dual problem is given by problem

max λ∈Rm D(λ)

subject to λ ⩾ 0 ,

where λ are the dual variables and D(λ) = minx∈Rd L(x,λ).

In contrast to the original optimization problem, which has constraints, minx∈Rd L(x,λ) is an unconstrained optimization problem for a given value of λ. If solving minx∈Rd L(x,λ) is easy, then the overall problem is easy to solve. We can see this by observing from that L(x,λ) is affine with respect to λ. Therefore minx∈Rd L(x,λ) is a pointwise minimum of affine functions of λ, and hence D(λ) is concave even though f(·) and gi(·) may be nonconvex. The outer problem, maximization over λ, is the maximum of a concave function and can be efficiently computed.

Assuming f(·) and gi(·) are differentiable, we find the Lagrange dual problem by differentiating the Lagrangian with respect to x, setting the differential to zero, and solving for the optimal value.

## What is an activation function?

Simply put, an activation function is a function that is added into an artificial neural network in order to help the network learn complex patterns in the data. When comparing with a neuron-based model that is in our brains, the activation function is at the end deciding what is to be fired to the next neuron. That is exactly what an activation function does in an ANN as well. It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell.

## Why do we need activation functions?

1. They help in keeping the value of the output from the neuron restricted to a certain limit as per our requirement. This is important because input into the activation function is **W\*x + b**where **W** is the weights of the cell and the **x** is the inputs and then there is the bias **b** added to that. This value if not restricted to a certain limit can go very high in magnitude especially in case of very deep neural networks that have millions of parameters. This will lead to computational issues. For example, there are some activation functions (like softmax) that out specific values for different values of input (0 or 1).
2. The most important feature in an activation function is its ability to add non-linearity into a neural network.

What if we use an ANN with a single cell but without an activation function. So our output is basically **W\*x + b.**But this is no good because **W\*x also has a degree of 1**, hence linear and **this is basically identical to a linear classifier.**

What if we stack multiple layers. Let’s represent nᵗʰ layer as a function fₙ(x). So we have:

**o(x) = fₙ(fₙ₋₁(….f₁(x))**

However, this is also not complex enough especially for problems with very high patterns such as that faced in computer vision or natural language processing.

In order to make the model get the power (aka the higher degree complexity) to learn the non-linear patterns, specific non-linear layers (activation functions) are added in between.

## Desirable features of an activation function

1. **Vanishing Gradient problem:**Neural Networks are trained using the process gradient descent. The gradient descent consists of the backward propagation step which is basically chain rule to get the change in weights in order to reduce the loss after every epoch. Consider a two-layer network and the first layer is represented as f₁(x) and the second layer is represented as f₂(x). The overall network is o(x) = f₂(f₁(x)). If we calculate weights during the backward pass, we get o`(x) = f₂(x)\*f₁`(x). Here f₁(x) is itself a compound function consisting of *Act*(W₁\*x₁ + b₁) where *Act* is the activation function after layer 1. Applying chain rule again, we clearly see that f₁`(x) = *Act*(W₁\*x₁ + b₁)\*x₁ which means it also depends directly on the activation value. Now imagine such a chain rule going through multiple layers while backpropagation. If the value of *Act*() is between 0 and 1, then several such values will get multiplied to calculate the gradient of the initial layers. This reduces the value of the gradient for the initial layers and those layers are not able to learn properly. In other words, their gradients tend to vanish because of the depth of the network and the activation shifting the value to zero. This is called the **vanishing gradient problem**. So we want our activation function to not shift the gradient towards zero.
2. **Zero-Centered:**Output of the activation function should be symmetrical at zero so that the gradients do not shift to a particular direction.
3. **Computational Expense**: Activation functions are applied after every layer and need to be calculated millions of times in deep networks. Hence, they should be computationally inexpensive to calculate.
4. **Differentiable:**As mentioned, neural networks are trained using the gradient descent process, hence the layers in the model need to differentiable or at least differentiable in parts. **This is a necessary requirement for a function to work as activation function layer.**

## List Activation functions

* Sigmoid

Изображение выглядит как текст, диаграмма, линия, График

Автоматически созданное описание

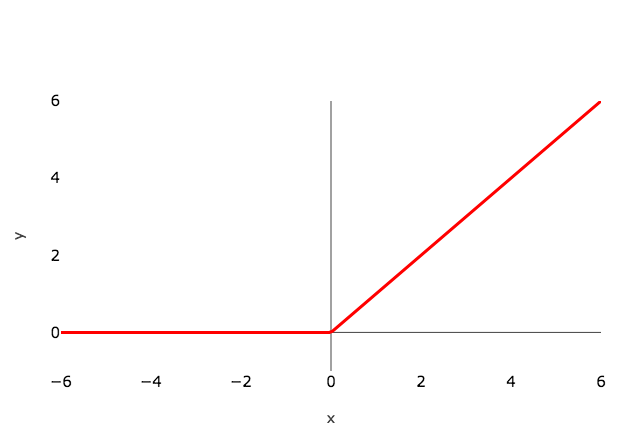
**σ(x) = 1 / (1 + e^-x)**

**0 <= σ(x) <= 1**

* Softmax - The softmax is a more generalised form of the sigmoid. It is used in **multi-class classification problems**. Similar to sigmoid, it produces values in the range of 0–1 therefore it is used as the final layer in classification models.

**Softmax(xi) = e^xi / sum\_j(e^xj)**

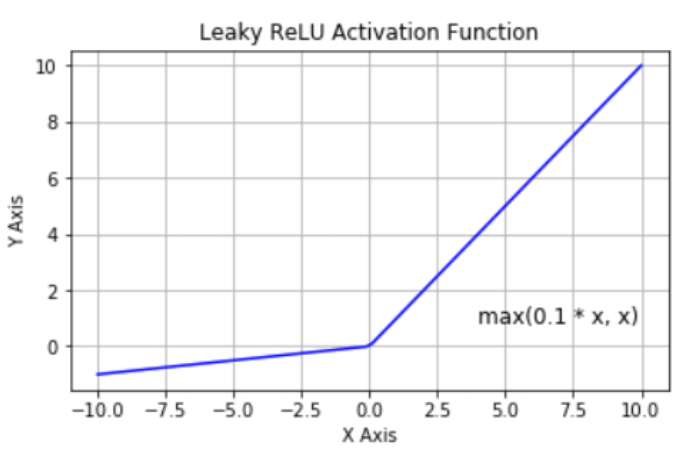
* **Tanh -** If you compare it to sigmoid, it solves just one problem of being zero-centred.
* **ReLU**: ReLU **(Rectified Linear Unit)** is defined as **f(x) = max(0,x):**



This is a widely used activation function, especially with Convolutional Neural networks. It is easy to compute and does not saturate and does not cause the Vanishing Gradient Problem. It has just one issue of not being zero centred. It suffers from **“dying ReLU”** problem. Since the output is zero for all negative inputs. It causes some nodes to completely die and not learn anything.

Another problem with ReLU is of exploding the activations since it higher limit is, well, inf. This sometimes leads to unusable nodes.

* **Leaky ReLU and Parametric ReLU**: It is defined as **f(x) = max(αx, x)**



the figure is for α = 0.1

Here α is a hyperparameter generally set to **0.01**. Clearly, Leaky ReLU solves the **“dying ReLU”**problem to some extent. Note that, if we set α as 1 then Leaky ReLU will become a linear function f(x) = x and will be of no use. Hence, the value of **α is never set close to 1.**If we set **α**as a hyperparameter for each neuron separately, we get **parametric ReLU** or **PReLU**.

* **ReLU6**: It is basically ReLU restricted on the positive side and it is defined as **f(x) = min(max(0,x),6)**

Изображение выглядит как линия, диаграмма, График, Параллельный

Автоматически созданное описание

This helps to stop blowing up the activation thereby stopping the gradients to explode(going to inf) as well another of the small issues that occur with normal ReLUs.

**Notable non-linear activations coming out of latest research**

* **Swish**: This was proposed in 2017 by Ramachandran et.al. It is defined as **f(x) = x\*sigmoid(x)**.

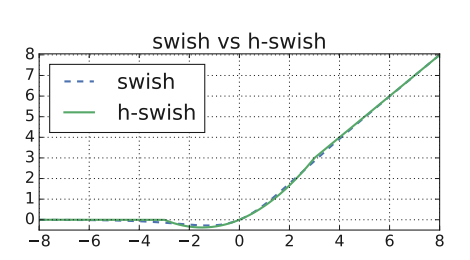
Изображение выглядит как текст, линия, диаграмма, снимок экрана

Автоматически созданное описание

It is slightly better in performance as compared to ReLU since its graph is quite similar to ReLU. However, because it does not change abruptly at a point as ReLU does at x = 0, this makes it easier to converge while training.

But, the drawback of Swish is that it is computationally expensive. To solve that we come to the next version of Swish.

* **Hard-Swish or H-Swish**: This is defined as:



Изображение выглядит как Шрифт, текст, белый, линия

Автоматически созданное описание

The best part is that it is almost similar to swish but it is less expensive computationally since it replaces sigmoid (exponential function) with a ReLU (linear type).

## Applications of Sigmoid

The sigmoid can be used simply as an activation function throughout a neural network, applying it to the outputs of each network layer. It isn’t used as much nowadays, however, because it has a couple of inefficiencies.

The first is the problem of saturating gradients. Looking at its graph, we can see that the sigmoid has a strong slope in the middle, but at the ends, its slope is very shallow. This is a problem for learning. At a high level, when we run gradient descent, many of the neurons in our network will be outputting values in the shallow regions of the sigmoid. Changing the network weights will then have little effect on its overall output, and learning comes to a halt.

In a little more detail, to run backpropagation and learn, we must take the gradient of the loss function with respect to each parameter in our network. At first, some neurons may be outputting values in the middle of the sigmoid range, where the slope is strong. But as we make updates, we move up or down this slope and quickly end up in a shallow region. The magnitude of our gradient then becomes smaller and smaller, meaning we take smaller and smaller learning steps. Learning is not very efficient this way.

The other problem with the sigmoid is that it’s not symmetric about the origin. In the brain, neurons either fire or don’t, so we may have the intuition that neuron activations should be zero or one. Despite this, researchers have actually found that neural networks learn better when activations are centered around zero. This is one of the reasons it’s a good idea to [standardize](https://builtin.com/data-science/when-and-why-standardize-your-data) your data (i.e., shift it to have mean zero) before feeding it into a neural network. It’s also one of the reasons for [batch normalization](https://builtin.com/data-science/when-and-why-standardize-your-data), a similar process where we standardize our network activations at intermediate layers rather than just at the start.

If you look at the beginning of the previous section, you’ll see that the tanh function ranges from -1 to one and is centered around zero. For this reason, it’s often preferable to the sigmoid. It also has the problem of saturating gradients, though. The most common activation function nowadays is the [rectified linear unit](https://builtin.com/machine-learning/relu-activation-function) (ReLU):

This function has a strong slope everywhere to the right of zero, although it’s obviously not symmetric around zero. So, tanh has saturating gradients, and ReLU is non-symmetric. In practice, the former is a bigger problem than the latter. The moral here, though, is that the sigmoid is the worst of both worlds on these fronts.

Despite all this, the sigmoid still has a place in modern machine learning: binary classification. In binary classification, we categorize inputs as one of two classes. If we’re using neural networks, the output of our network must be a number between zero and one, representing the probability that the input belongs to class one (with the probability for class two being immediately inferable).

The output layer of such a network consists of a single neuron. Consider the output value of this neuron. Before applying any activation function, it can be any real number, which is no good. If we apply a ReLU, it will be positive (or zero). If we use tanh, it will be between -1 and one. None of these work. We must apply a sigmoid to this last neuron. We need a number between zero and one, and we still need the activation function to be smooth for the purposes of training. The sigmoid is the right choice.