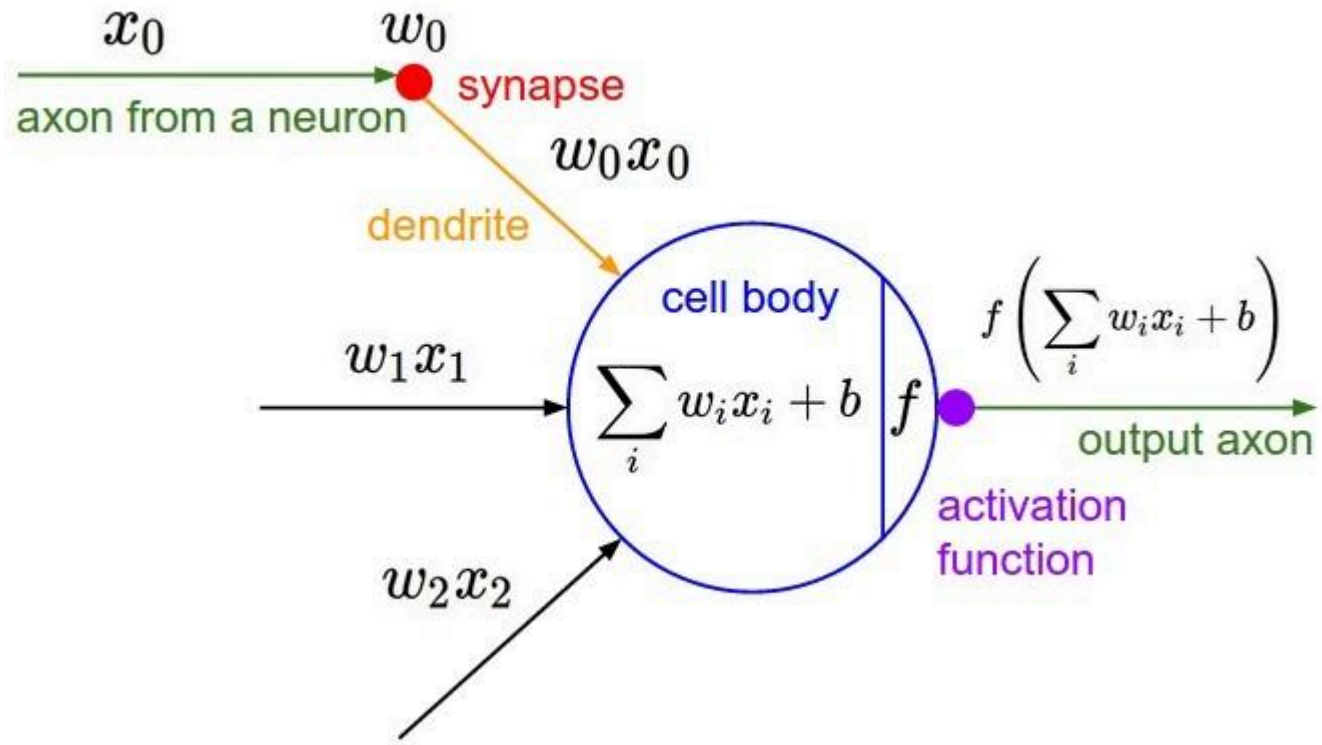
Activation Function

Activation functions helps to determine the output of a neural network. These type of functions are attached to each neuron in the network, and determines whether it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.

> Activation function also helps to normalize the output of each neuron to a range between 1 and 0 or between -1 and 1.



In a neural network, inputs are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight gives the output of the neuron, which is transferred to the next layer.

The activation function is a mathematical "gate" in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold.

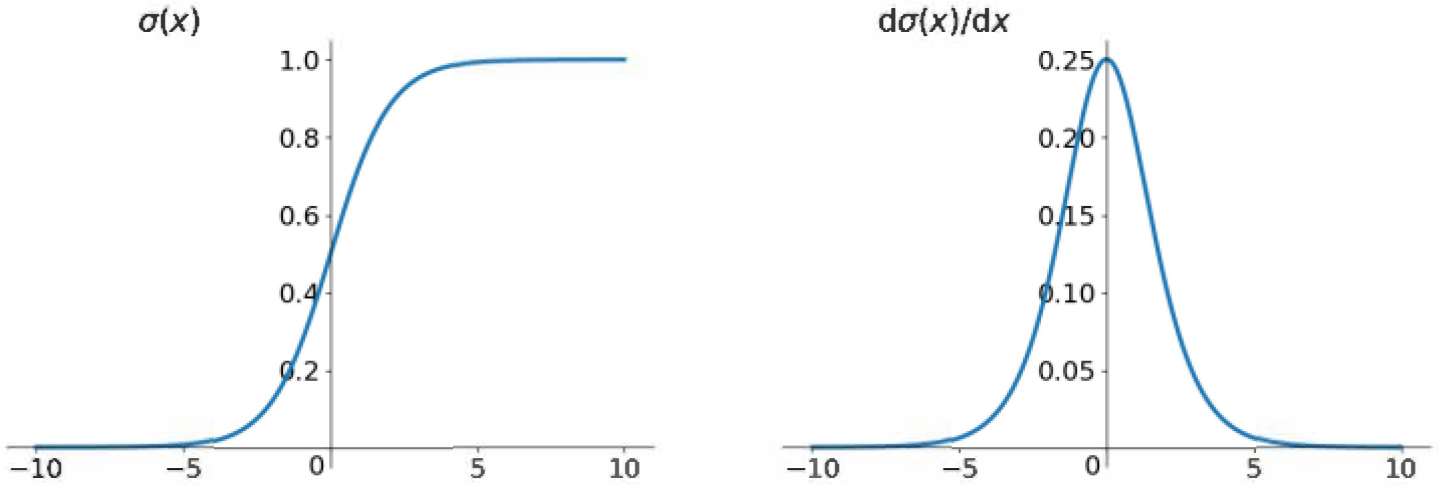
> Neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

Commonly used activation functions

1. Sigmod function

The function formula and chart are as follows

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



The Sigmoid function is the most frequently used activation function in the beginning of deep learning. It is a smoothing function that is easy to derive.

In the sigmoid function, we can see that its output is in the open interval (0,1). We can think of probability, but in the strict sense, don't treat it as probability. The sigmoid function was once more popular. It can be thought of as the firing rate of a neuron. In the middle where the slope is relatively large, it is the sensitive area of the neuron. On the sides where the slope is very gentle, it is the neuron's inhibitory area.

The function itself has certain defects.

1. When the input is slightly away from the coordinate origin, the gradient of the function becomes very small, almost zero. In the process of neural network backpropagation, we all use the chain rule of differential to calculate the differential of each weight w. When the backpropagation passes through the sigmod function, the differential on this chain is very small. Moreover, it may pass through many sigmod functions, which will eventually cause the weight w to have little effect on the loss function, which is not conducive to the optimization of the weight. This The problem is called gradient saturation or gradient dispersion.
2. The function output is not centered on 0, which will reduce the efficiency of weight update.
3. The sigmod function performs exponential operations, which is slower for computers.

Advantages of Sigmoid Function : -

1. Smooth gradient, preventing “jumps” in output values.
2. Output values bound between 0 and 1, normalizing the output of each neuron.
3. Clear predictions, i.e very close to 1 or 0.

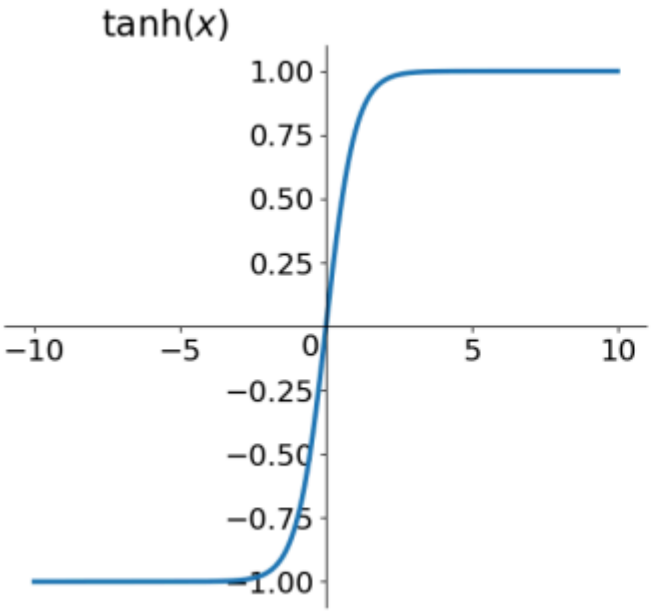
Sigmoid has three major disadvantages:

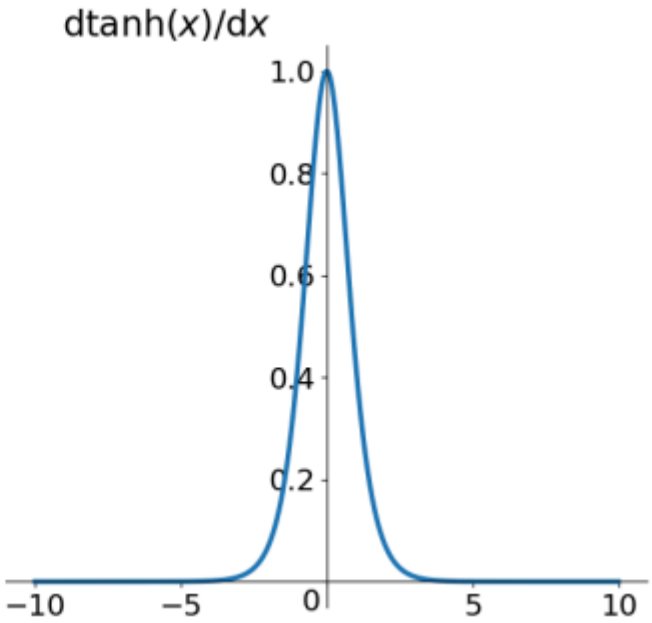
- Prone to gradient vanishing
- Function output is not zero-centered
- Power operations are relatively time consuming

2. tanh function

The tanh function formula and curve are as follows

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$





Tanh is a hyperbolic tangent function. The curves of tanh function and sigmod function are relatively similar. Let 's compare them. First of all, when the input is large or small, the output is almost smooth and the gradient is small, which is not conducive to weight update. The difference is the output interval.

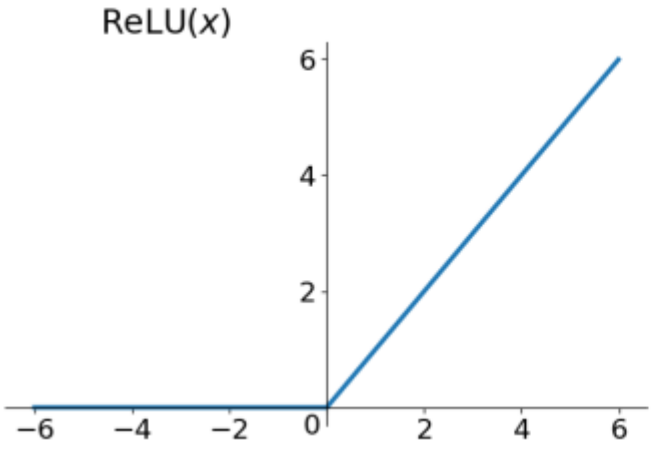
The output interval of tanh is 1), and the whole function is 0-centric, which is better than sigmod.

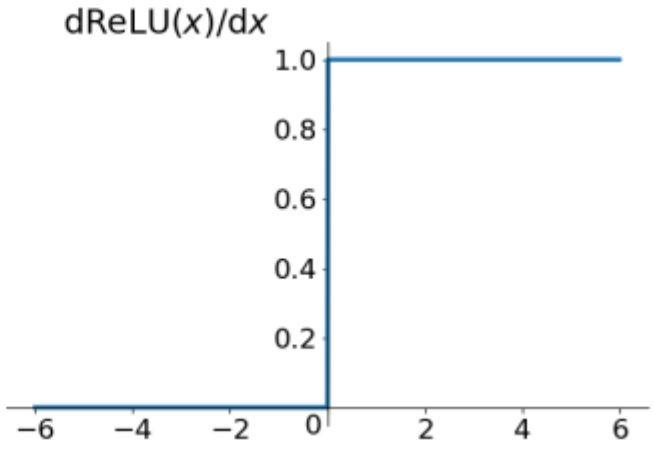
In general binary classification problems, the tanh function is used for the hidden layer and the sigmod function is used for the output layer. However, these are not static, and the specific activation function to be used must be analyzed according to the specific problem, or it depends on debugging.

3. ReLU function

ReLU function formula and curve are as follows

$\mathbf{ReLU} = \mathbf{max}(0, x)$





The ReLU function is actually a function that takes the maximum value. Note that this is not fully interval-derivable, but we can take sub-gradient, as shown in the figure above. Although ReLU is simple, it is an important achievement in recent years.

The ReLU (Rectified Linear Unit) function is an activation function that is currently more popular. Compared with the sigmod function and the tanh function, it has the following advantages:

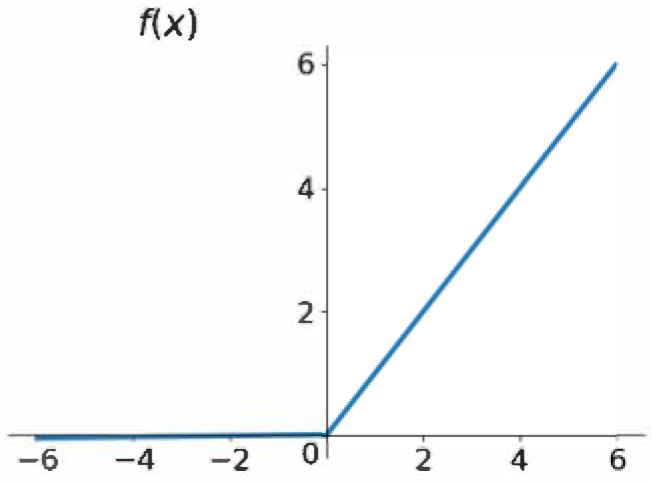
1. When the input is positive, there is no gradient saturation problem.
2. The calculation speed is much faster. The ReLU function has only a linear relationship. Whether it is forward or backward, it is much faster than sigmod and tanh. (Sigmod and tanh need to calculate the exponent, which will be slower.)

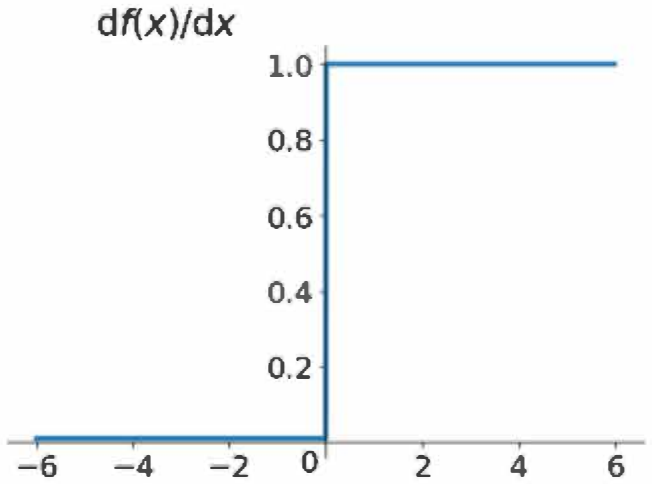
Ofcourse, there are disadvantages:

1. When the input is negative, ReLU is completely inactive, which means that once a negative number is entered, ReLU will die. In this way, in the forward propagation process, it is not a problem. Some areas are sensitive and some are insensitive. But in the backpropagation process, if you enter a negative number, the gradient will be completely zero, which has the same problem as the sigmod function and tanh function.
2. We find that the output of the ReLU function is either 0 or a positive number, which means that the ReLU function is not a 0-centric function.

4. Leaky ReLU function

$f(x) = \max(0.01x, x)$

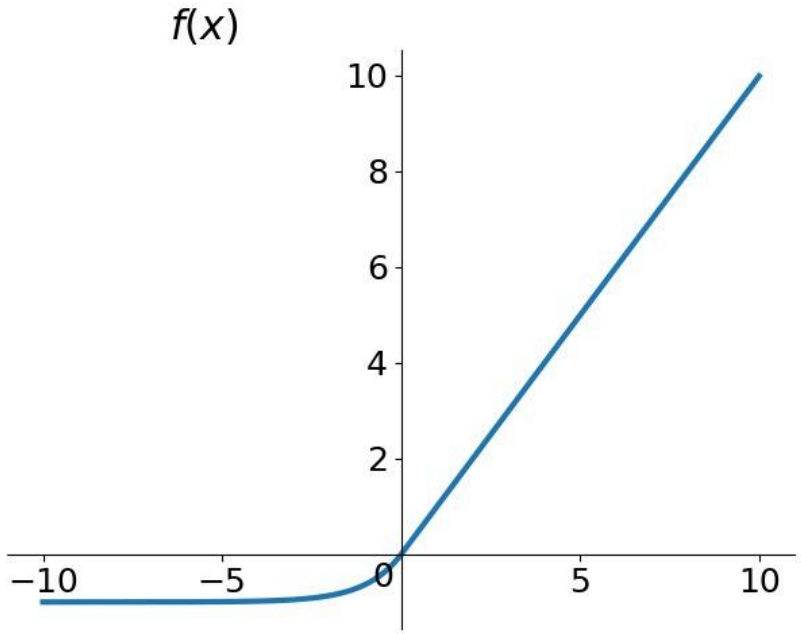


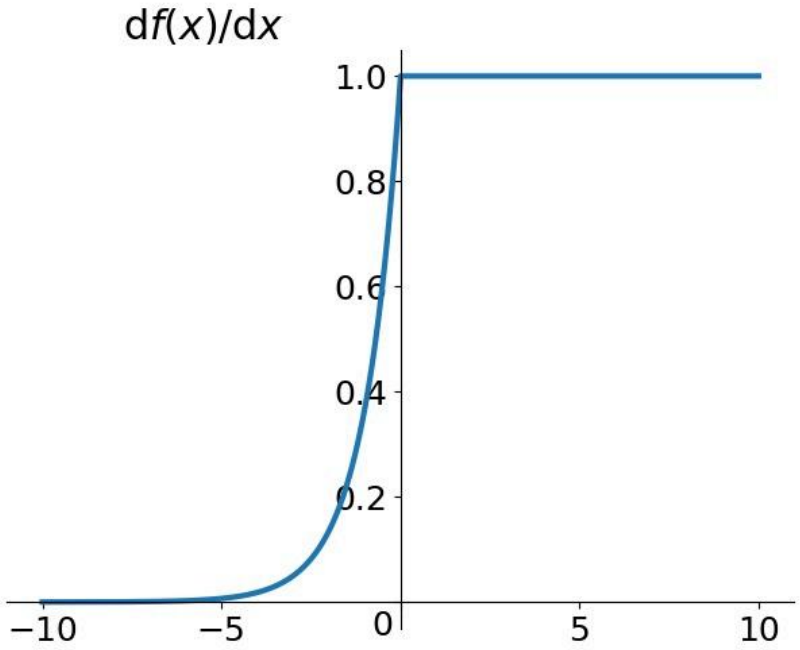


In order to solve the Dead ReLU Problem, people proposed to set the first half of ReLU 0.01x instead of 0. Another intuitive idea is a parameter-based method, Parametric ReLU : f(x)= max(alpha x,x), which alpha can be learned from back propagation. In theory, Leaky ReLU has all the advantages of ReLU, plus there will be no problems with Dead ReLU, but in actual operation, it has not been fully proved that Leaky ReLU is always better than R[illegible].

5- ELU (Exponential Linear Units) function

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$





ELU is also proposed to solve the problems of ReLU. Obviously, ELU has all the advantages of ReLU, and:

- No Dead ReLU issues
- The mean of the output is close to 0, zero-centered

One small problem is that it is slightly more computationally intensive. Similar to Leaky ReLU, although theoretically better than ReLU, there is currently no good evidence in practice that ELU is always better than ReLU.

6. Softmax

$$S(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, j = 1, 2, \ldots, K$$

for an arbitrary real vector of length K, Softmax can compress it into a real vector of length K with a value in the range (0, 1), and the sum of the elements in the vector is 1.

It also has many applications in Multiclass Classification and neural networks. Softmax is different from the normal max function: the max function only outputs the largest value, and Softmax ensures that smaller values have a smaller probability and will not be discarded directly. It is a "max" that is "soft".

The denominator of the Softmax function combines all factors of the original output value, which means that the different probabilities obtained by the Softmax function are related to each other. In the case of binary classification, for Sigmoid, there are:

$$p(y=1|x) = \frac{1}{1+e^{-\theta^T x}}$$

$$p(y=0|x) = 1 - p(y=1|x) = \frac{e^{-\theta^T x}}{1+e^{-\theta^T x}}$$

For Softmax with K = 2, there are:

$$p(y=1|x) = \frac{e^{\theta_1^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{1}{1+e^{(\theta_0^T - \theta_1^T)x}} = \frac{1}{1+e^{-\beta x}}$$

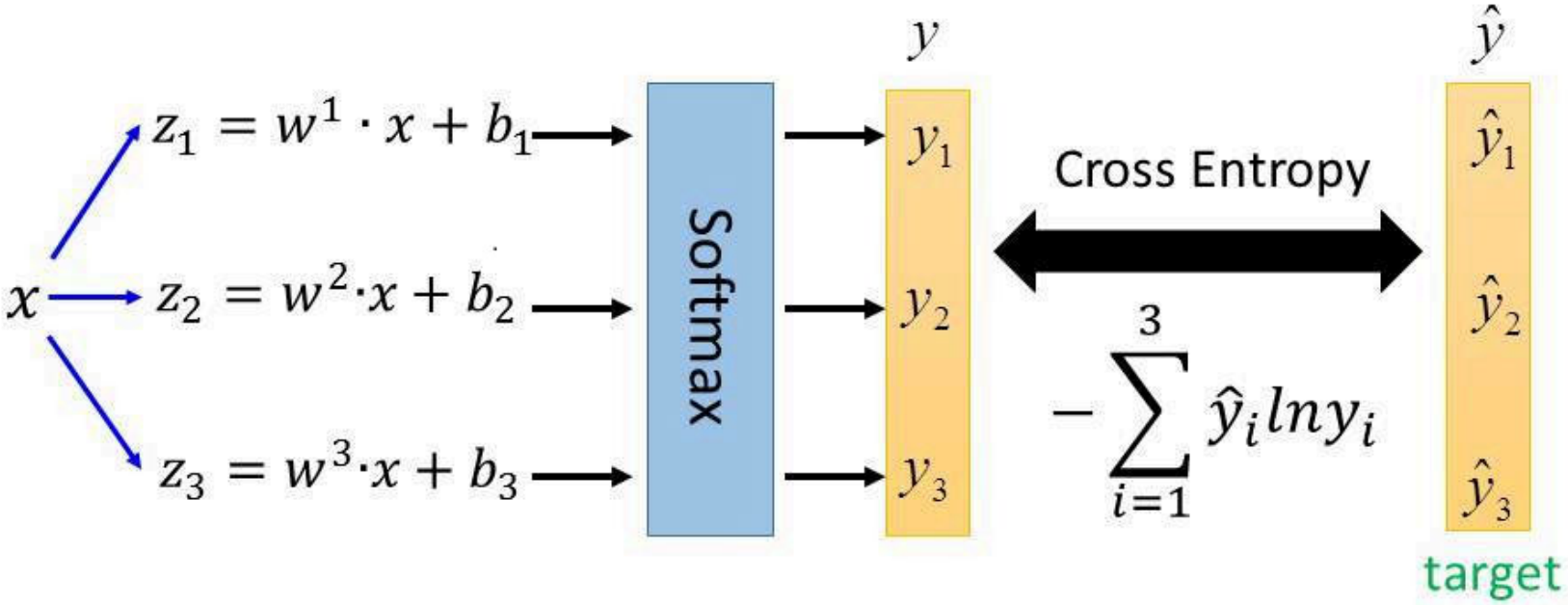
$$p(y=0|x) = \frac{e^{\theta_0^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{e^{(\theta_0^T - \theta_1^T)x}}{1+e^{(\theta_0^T - \theta_1^T)x}} = \frac{e^{-\beta x}}{1+e^{-\beta x}}$$

Among them: It

$$\beta = -(\theta_0^T - \theta_1^T)$$

can be seen that in the case of binary classification, Softmax is degraded to Sigmoid.

Multi-class Classification (3 classes as example)



If x ∈ class 1

$$\hat{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$-ln y_1$$

If x ∈ class 2

$$\hat{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

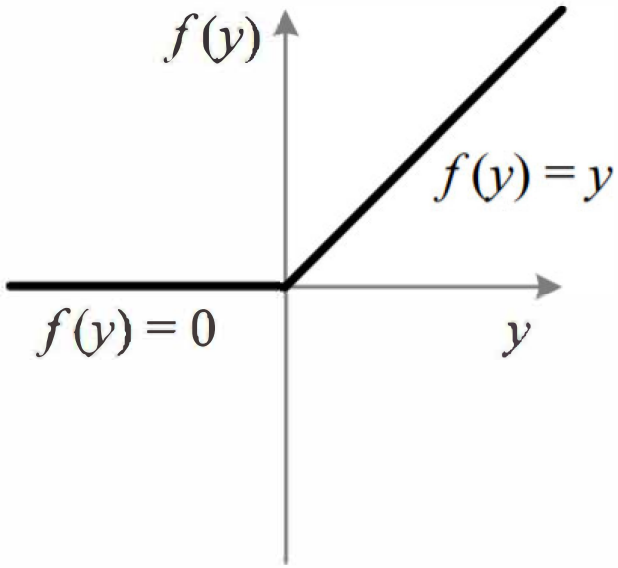
$$-ln y_2$$

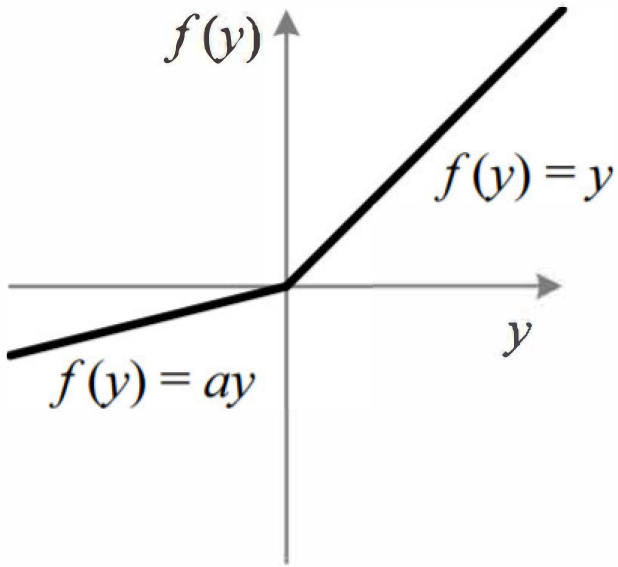
If x ∈ class 3

$$\hat{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$-ln y_3$$

7. PRelu (Parametric ReLU)





ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

PReLU is also an improved version of ReLU. In the negative region, PReLU has a small slope, which can also avoid the problem of ReLU death. Compared to ELU, PReLU is a linear operation in the negative region. Although the slope is small, it does not tend to 0, which is a certain advantage.

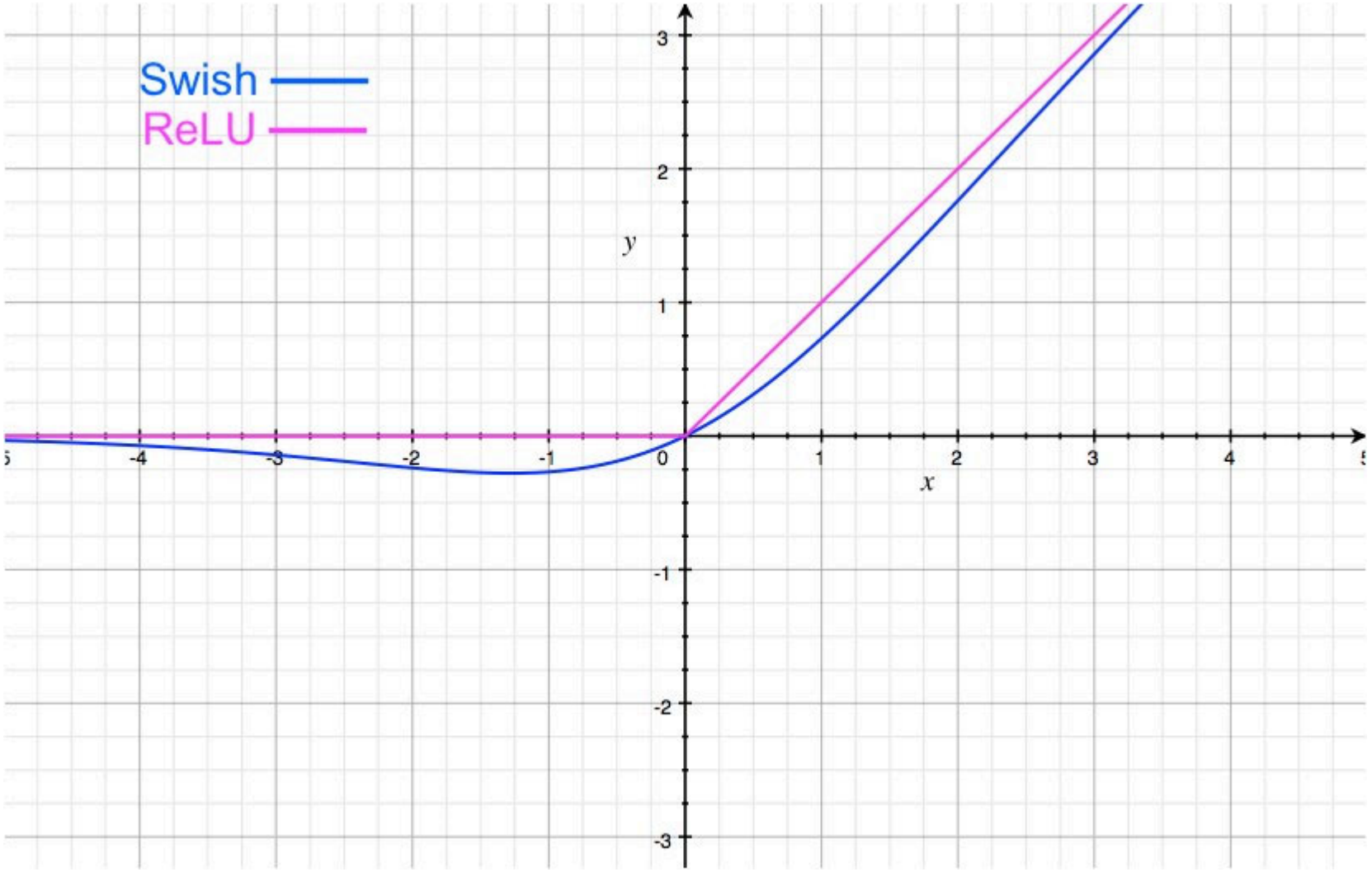
$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

We look at the formula of PReLU. The parameter α is generally a number between 0 and 1, and it is generally relatively small, such as a few zeros. When α = 0.01, we call PReLU as Leaky Relu , it is regarded as a special case PReLU it.

Above, y_i is any input on the ith channel and a_i is the negative slope which is a learnable parameter.

- if a_i=0, f becomes ReLU
- if a_i>0, f becomes leaky ReLU
- if a_i is a learnable parameter, f becomes PR[illegible]

8. Swish (A Self-Gated) Function



The formula is: **y = x * sigmoid (x)**

Swish's design was inspired by the use of sigmoid functions for gating in LSTMs and highway networks. We use the same value for gating to simplify the gating mechanism, which is called **self-gating.**

The advantage of self-gating is that it only requires a simple scalar input, while normal gating requires multiple scalar inputs. This feature enables self-gated activation functions such as Swish to easily replace activation functions that take a single scalar as input (such as ReLU)

without changing the hidden capacity or number of parameters.

1. Unboundedness (unboundedness) is helpful to prevent gradient from gradually approaching 0 during slow training, causing saturation. At the same time, being bounded has advantages, because bounded active functions can have strong reguairzation, and larger negative inputs will be resolved.

2. At the same time, smoothness also plays an important role in optimization and generalization.

9. Maxout

The Maxout activation function is defined as follows

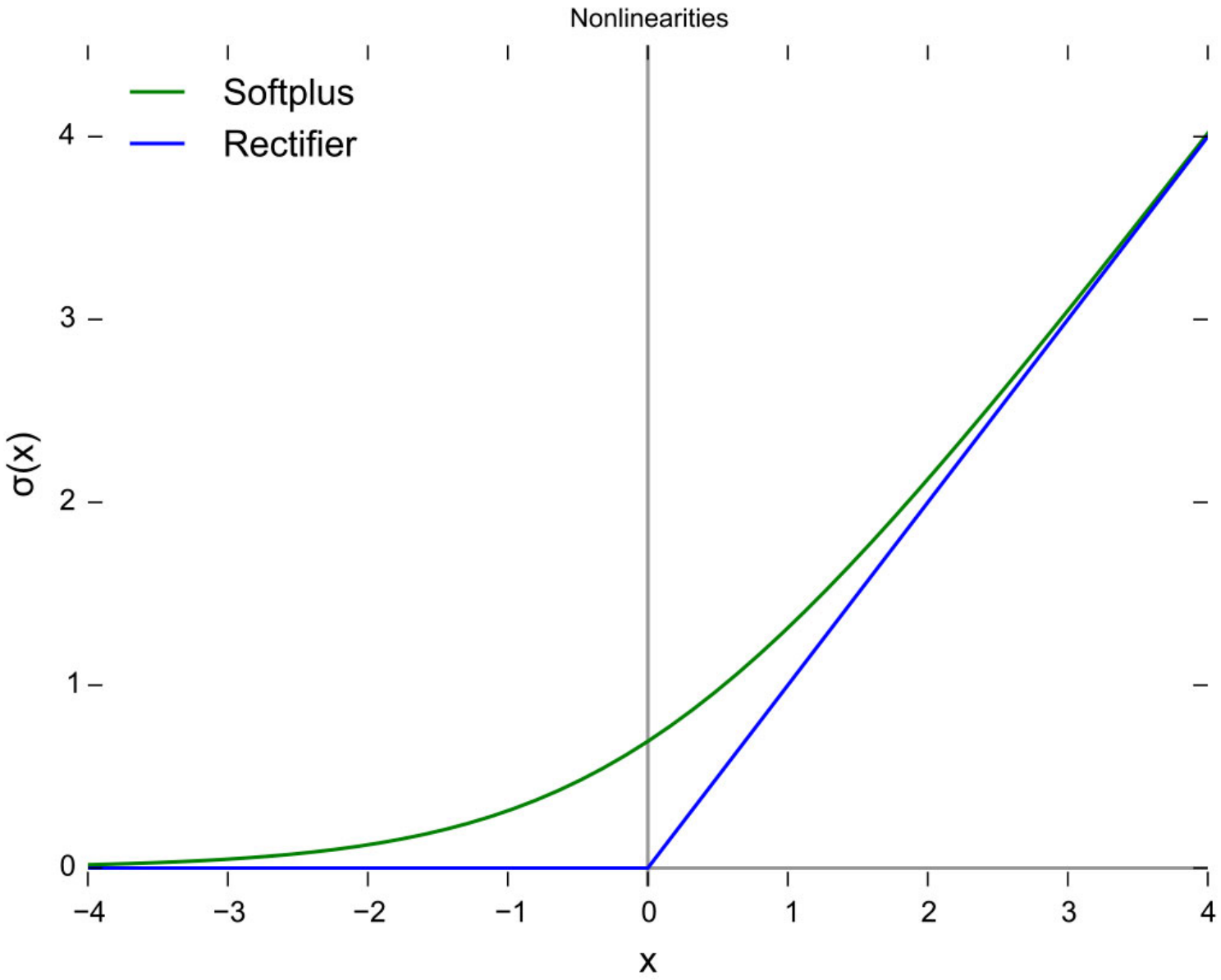
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

One relatively popular choice is the Maxout neuron (introduced recently by Goodfellow et al.) that generalizes the ReLU and its leaky version. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have w1,b1 =0).The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks

The Maxout activation is a generalization of the ReLU and the leaky ReLU functions. It is a learnable activation function.

Maxout can be seen as adding a layer of activation function to the deep learning network, which contains a parameter k. Compared with ReLU, sigmoid, etc., this layer is special in that it adds k neurons and then outputs the largest activation value. value.

10. Softplus


Nonlinearities
Softplus
Rectifier
σ(x)
4
3
2
1
0
−4
−3
−2
−1
0
1
2
3
4
x

The softplus function is similar to the ReLU function, but it is relatively smooth.It is unilateral suppression like ReLU.It has a wide acceptance range (0, + inf).

Softplus function: **f(x) = ln(1+exp x)**

Generally speaking, these

activation functions have their own advantages and disadvantages. There is no statement that indicates which ones are not working, and which activation functions are good. All the good and bad must be obtained by experiments.

In [ ]:

Loss Functions

1. L1 and L2 loss

L1 and *L2* are two common loss functions in machine learning which are mainly used to minimize the error.

L1 loss function are also known as **Least Absolute Deviations** in short **LAD**. **L2 loss function** are also known as **Least square errors** in short **LS**.

Let's get brief of these two

L1 Loss function

It is used to minimize the error which is the sum of all the absolute differences in between the true value and the predicted value.

$$L1LossFunction = \sum_{i=1}^{n} |y_{true} - y_{predicted}|$$

L2 Loss Function

It is also used to minimize the error which is the sum of all the squared differences in between the true value and the pedicted value.

$$L2LossFunction = \sum_{i=1}^{n} (y_{true} - y_{predicted})^2$$

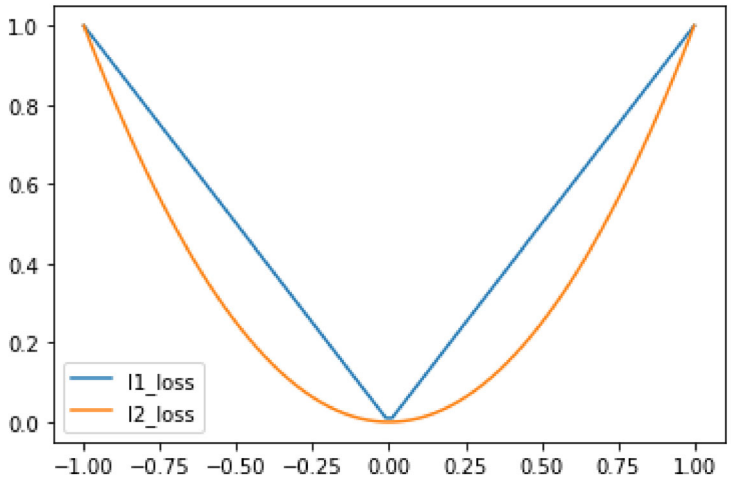
The disadvantage of the **L2 norm** is that when there are outliers, these points will account for the main component of the loss. For example, the true value is 1, the prediction is 10 times, the prediction value is 1000 once, and the prediction value of the other times is about 1, obviously the loss value is mainly dominated by 1000.

In [1]:
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

In [2]:
```
x_guess = tf.lin_space(-1., 1., 100)
x_actual = tf.constant(0,dtype=tf.float32)
```

In [3]:
```
l1_loss = tf.abs((x_guess-x_actual))
l2_loss = tf.square((x_guess-x_actual))
```

In [4]:
```
with tf.Session() as sess:
    x_,l1_,l2_ = sess.run([x_guess, l1_loss, l2_loss])
    plt.plot(x_,l1_,label='l1_loss')
    plt.plot(x_,l2_,label='l2_loss')
    plt.legend()
    plt.show()
```



2. Huber Loss

Huber Loss is often used in regression problems. Compared with L2 loss, Huber Loss is less sensitive to outliers(because if the residual is too large, it is a piecewise function, loss is a linear function of the residual).

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta |y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Among them, δ is a set parameter, y represents the real value, and $f(x)$ represents the predicted value.

The advantage of this is that when the residual is small, the loss function is L2 norm, and when the residual is large, it is a linear function of L1 norm

Pseudo-Huber loss function

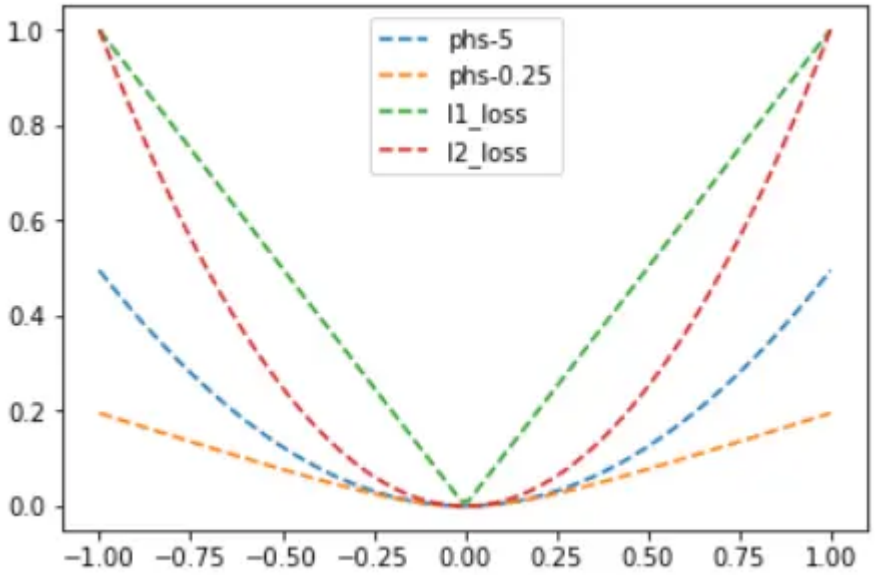
A smooth approximation of Huber loss to ensure that each order is differentiable.

$$L_\delta(a) = \delta^2(\sqrt{1 + (a/\delta)^2} - 1).$$

As such, this function approximates $a^2/2$ for small values of a, and approximates a straight line with slope δ for large values of a.

While the above is the most common form, other smooth approximations of the Huber loss function also exist.[5]

Where δ is the set parameter, the larger the value, the steeper the linear part on both sides.



3.Hinge Loss

Hinge loss is often used for binary classification problems, such as ground true: t = 1 or -1, predicted value y = wx + b

In the svm classifier, the definition of hinge loss is

Hinge loss

From Wikipedia, the free encyclopedia

In machine learning, the **hinge loss** is a loss function used for training classifiers. The hinge loss is used for "maximum-margin" classification, most notably for support vector machines (SVMs).[1] For an intended output $t = \pm 1$ and a classifier score y, the hinge loss of the prediction y is defined as

$$\ell(y) = \max(0, 1 - t \cdot y)$$

Note that y should be the "raw" output of the classifier's decision function, not the predicted class label. For instance, in linear SVMs, $y = \mathbf{w} \cdot \mathbf{x} + b$, where $(\mathbf{w}, b)$ are the parameters of the hyperplane and $\mathbf{x}$ is the point to classify.

It can be seen that when t and y have the same sign (meaning y predicts the right class) and $|y| \geq 1$, the hinge loss $\ell(y) = 0$, but when they have opposite sign, $\ell(y)$ increases linearly with y (one-sided error).

In other words, the closer the y is to t, the smaller the loss will be.

In [ ]:
```
x_guess2 = tf.linspace(-3.,5.,500)
x_actual2 = tf.convert_to_tensor([1.]*500)

#Hinge loss
#hinge_loss = tf.losses.hinge_loss(labels=x_actual2, logits=x_guess2)
hinge_loss = tf.maximum(0.,1.-(x_guess2*x_actual2))
with tf.Session() as sess:
    x_,hin_ = sess.run([x_guess2, hinge_loss])
    plt.plot(x_,hin_,'--', label='hinge_loss')
    plt.legend()
    plt.show()
```

```
  File "<ipython-input-5-2caf33f96afd>", line 7
    0with tf.Session() as sess:
     ^
SyntaxError: invalid syntax
```

4.Cross-entropy loss

Cross-entropy loss function and logistic regression [edit]

Cross entropy can be used to define a loss function in machine learning and optimization. The true probability p_i is the true label, and the given distribution q_i is the predicted value of the current model.

More specifically, consider logistic regression, which (among other things) can be used to classify observations into two possible classes (often simply labelled 0 and 1). The output of the model for a given observation, given a vector of input features x, can be interpreted as a probability, which serves as the basis for classifying the observation. The probability is modeled using the logistic function $g(z) = 1/(1+e^{-z})$ where z is some function of the input vector x, commonly just a linear function. The probability of the output $y = 1$ is given by

$$q_{y=1} = \hat{y} \equiv g(\mathbf{w} \cdot \mathbf{x}) = 1/(1+e^{-\mathbf{w} \cdot \mathbf{x}}),$$

where the vector of weights $\mathbf{w}$ is optimized through some appropriate algorithm such as gradient descent. Similarly, the complementary probability of finding the output $y = 0$ is simply given by

$$q_{y=0} = 1 - \hat{y}$$

Having set up our notation, $p \in \{y, 1-y\}$ and $q \in \{\hat{y}, 1-\hat{y}\}$, we can use cross entropy to get a measure of dissimilarity between p and q:

$$H(p,q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1-y) \log(1-\hat{y})$$

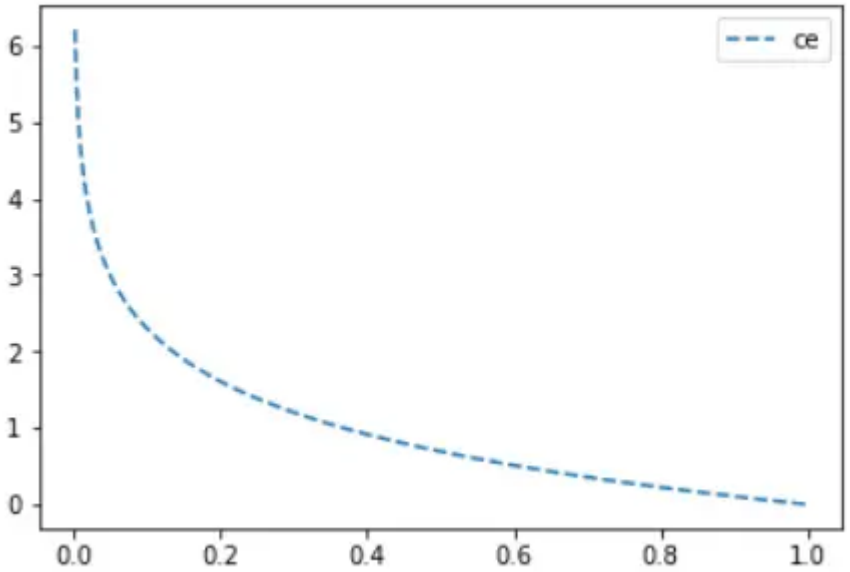
Logistic regression typically optimizes the log loss for all the observations on which it is trained, which is the same as optimizing the average cross-entropy in the sample. For example, suppose we have N samples with each sample indexed by $n = 1, \ldots, N$. The average of the loss function is then given by:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N} \sum_{n=1}^{N} \left[y_n \log \hat{y}_n + (1-y_n) \log(1-\hat{y}_n) \right],$$

where $\hat{y}_n \equiv g(\mathbf{w} \cdot \mathbf{x}_n) = 1/(1+e^{-\mathbf{w} \cdot \mathbf{x}_n})$, with $g(z)$ the logistic function as before.

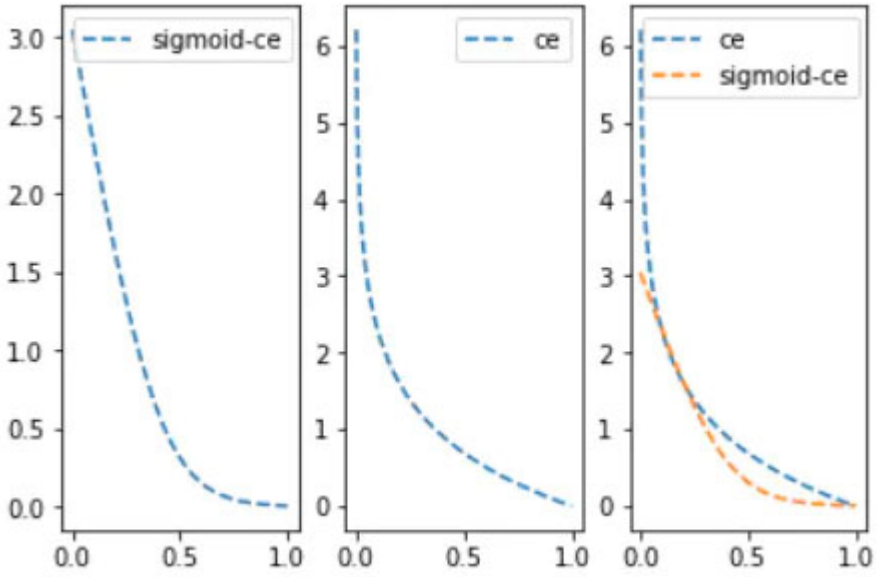
The logistic loss is sometimes called cross-entropy loss. It is also known as log loss (In this case, the binary label is often denoted by {-1,+1}).[2]

The above is mainly to say that cross-entropy loss is mainly applied to binary classification problems. The predicted value is a probability value and the loss is defined according to the cross entropy. Note the value range of the above value: the predicted value of y should be a probability and the value range is [0,1]



5.Sigmoid-Cross-entropy loss

The above cross-entropy loss requires that the predicted value is a probability. Generally, we calculate $scores = x * w + b$. Entering this value into the sigmoid function can compress the value range to (0,1).



It can be seen that the sigmoid function smoothes the predicted value(such as directly inputting 0.1 and 0.01 and inputting 0.1, 0.01 sigmoid and then entering, the latter will obviously have a much smaller change value), which makes the predicted value of sigmoid-ce far from the label loss growth is not so steep.

6.Softmax cross-entropy loss

First, the softmax function can convert a set of fraction vectors into corresponding probability vectors. Here is the definition of softmax function

---

In mathematics, the **softmax function**, or **normalized exponential function**,[1]:198 is a generalization of the logistic function that "squashes" a K-dimensional vector $\mathbf{z}$ of arbitrary real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values in the range (0, 1] that add up to 1. The function is given by

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

As above, softmax also implements a vector of 'squashes' k-dimensional real value to the [0,1] range of k-dimensional, while ensuring that the cumulative sum is 1.

According to the definition of cross entropy, probability is required as input.Sigmoid-cross-entropy-loss uses sigmoid to convert the score vector into a probability vector, and softmax-cross-entropy-loss uses a softmax function to convert the score vector into a probability vector.

According to the definition of cross entropy loss.

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

where $p(x)$ represents the probability that classification x is a correct classification, and the value of p can only be 0 or 1. This is the prior value

$q(x)$ is the prediction probability that the x category is a correct classification, and the value range is (0,1)

So specific to a classification problem with a total of C types, then $p(x_j)$, $0 \leq j \leq C$ must be only 1 and C-1 is 0(because there can be only one correct classification, correct the probability of classification as correct classification is 1, and the probability of the remaining classification as correct classification is 0)

Then the definition of softmax-cross-entropy-loss can be derived naturally.

Here is the definition of softmax-cross-entropy-loss.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

Where f_j is the score of all possible categories, and f_{y_i} is the score of ground true class

In [ ]:

Optimizers

Batch gradient descent

Gradient update rule: BGD uses the data of the entire training set to calculate the gradient of the cost function to the parameters:

Disadvantages:

Because this method calculates **the gradient for the entire data set in one update, the calculation is very slow**, it will be very tricky to encounter a large number of data sets, and you cannot invest in new data to update the model in real time.

We will define an iteration number epoch in advance, first calculate the gradient vector params_grad, and then update the parameter params along the direction of the gradient. The learning rate determines how big we take each step.

Batch gradient descent can converge to a global minimum for convex functions and to a local minimum for non-convex functions.

SGD (Stochastic gradient descent)

Gradient update rule: Compared with BGD's calculation of gradients with all data at one time, SGD updates the gradient of each sample with each update.

```
x += - learning_rate * dx
```

where x is a parameter, dx is the gradient and learning rate is constant

For large data sets, there may be similar samples, so BGD calculates the gradient. **There will be redundancy, and SGD is updated only once, there is no redundancy, it is faster, and new samples can be added.**

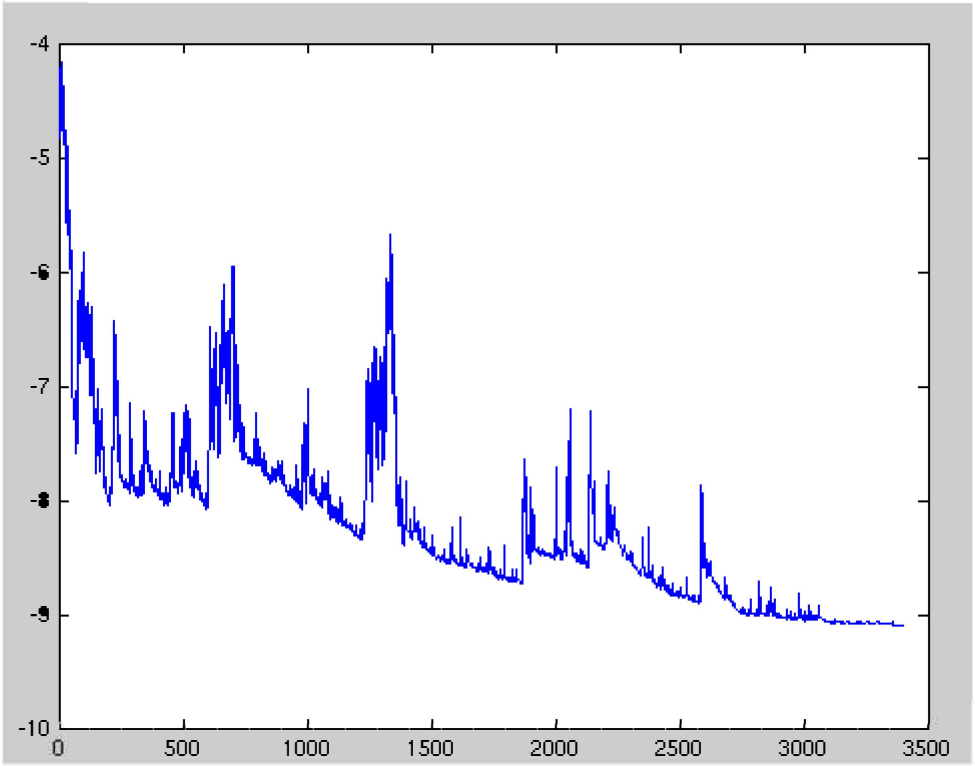


Figure :- Fluctuations in SGD

Disadvantages: However, because SGD is updated more frequently, the cost function will have severe oscillations. BGD can converge to a local minimum, of course, the oscillation of SGD may jump to a better local minimum.

When we decrease the learning rate slightly, the convergence of SGD and BGD is the same.

Mini-batch gradient descent

Gradient update rule:

MBGD uses a small batch of samples, that is, n samples to calculate each time. In this way, it can reduce the variance when the parameters are updated, and the convergence is more stable. It can make full use of the highly optimized matrix operations in the deep learning library for more efficient gradient calculations.

The difference from SGD is that each cycle does not act on each sample, but a batch with n samples.

> Setting value of hyper-parameters: n Generally value is 50 ~ 256

Cons:

- Mini-batch gradient descent does not guarantee good convergence,
- 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. One measure is to set a **larger learning rate**. When the change between two iterations is lower than a certain threshold, the learning rate is reduced.

However, the setting of this threshold needs to be written in advance adapt to the characteristics of the data set.

In addition, this method is to apply the **same learning rate** to all parameter updates. If our data is sparse, we would prefer to update the features with lower frequency.

In addition, for **non-convex functions**, it is also necessary to avoid trapping at the local minimum or saddle point, because the error around the saddle point is the same, the gradients of all dimensions are close to 0, and SGD is easily trapped here.

Saddle points are the curves, surfaces, or hypersurfaces of a saddle point neighborhood of a smooth function are located on different sides of a tangent to this point. For example, this two-dimensional figure looks like a saddle: it curves up in the x-axis direction and down in the y-axis direction, and the saddle point is (0,0).

Momentum

One disadvantage of the SGD method is that its update direction depends entirely on the current batch, so its update is very unstable. A simple way to solve this problem is to introduce momentum.

Momentum is momentum, which 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.

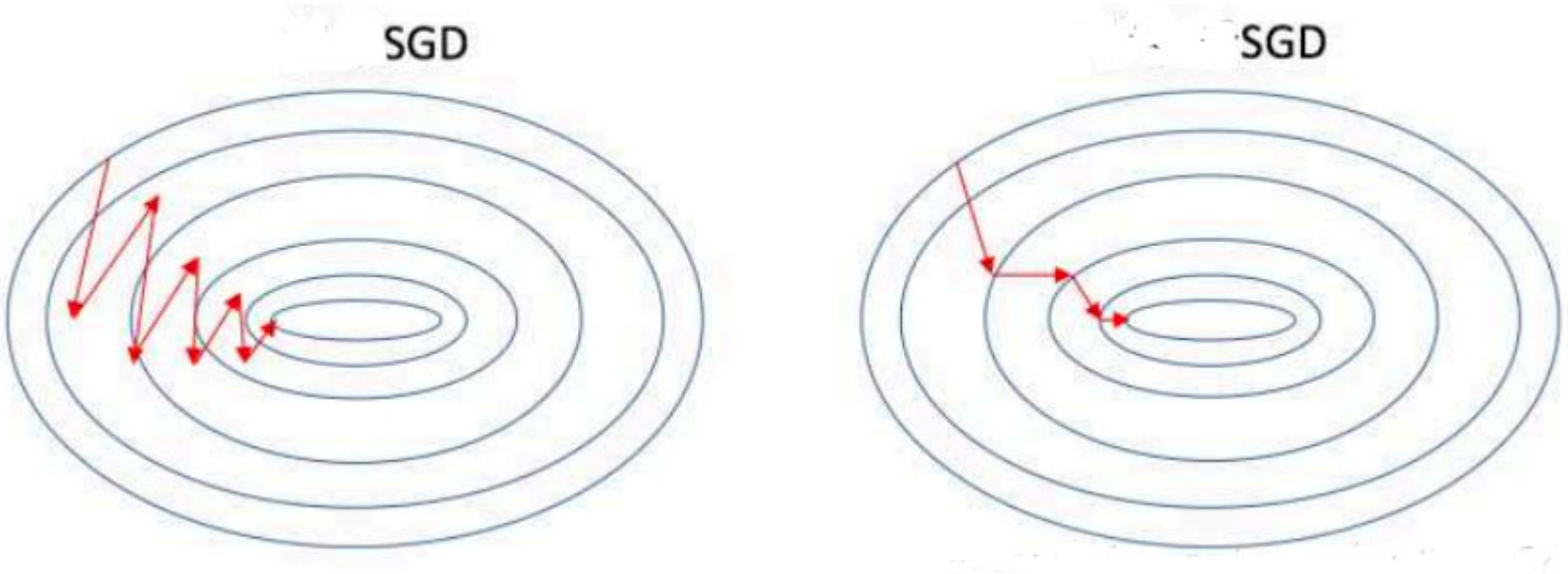


Figure :- SGD without Momentum &&& SGD without Momentum

Adagrad

Adagrad is an algorithm for gradient-based optimization which adapts the learning rate to the parameters, using low learning rates for parameters associated with frequently occurring features, and using high learning rates for parameters associated with infrequent features.

So, it is well-suited for dealing with sparse data.

But the same update rate may not be suitable for all parameters. For example, some parameters may have reached the stage where only fine-tuning is needed, but some parameters need to be adjusted a lot due to the small number of corresponding samples.

Adagrad proposed this problem, an algorithm that adaptively assigns different learning rates to various parameters among them. The implication is that for each parameter, as its total distance updated increases, its learning rate also slows.

> **GloVe word embedding uses adagrad where infrequent words required a greater update and frequent words require smaller updates.**

> **Adagrad eliminates the need to manually tune the learning rate.**

Adadelta

There are three problems with the Adagrad algorithm

- The learning rate is monotonically decreasing.
- The learning rate in the late training period is very small.
- It requires manually setting a global initial learning rate.

> **Adadelta is an extension of Adagrad and it also tries to reduce Adagrad's aggressive, monotonically reducing the learning rate.**

> It does this by restricting the window of the past accumulated gradient to some fixed size of w. Running average at time t then depends on the previous average and the current gradient.

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

RMSProp

The full name of RMSProp algorithm is called **Root Mean Square Prop**, which is an adaptive learning rate optimization algorithm proposed by Geoff Hinton.

> RMSProp tries to resolve Adagrad's radically diminishing learning rates by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.

Adagrad will accumulate all previous gradient squares, and RMSprop just calculates the corresponding average value, so it can alleviate the problem that the learning rate of the Adagrad algorithm drops quickly.

The difference is that RMSProp calculates the **differential squared weighted average of the gradient** . This method is beneficial to eliminate the direction of large swing amplitude, and is used to correct the swing amplitude, so that the swing amplitude in each dimension is smaller. On the other hand, it also makes the network function converge faster.

> In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

> RMSProp divides the learning rate by the average of the exponential decay of squared gradients

Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop.

> Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

> Adam can be viewed as a combination of Adagrad and RMSprop,(Adagrad) which works well on sparse gradients and (RMSProp) which works well in online and nonstationary settings repectively.

> Adam implements the **exponential moving average of the gradients** to scale the learning rate instead of a simple average as in Adagrad. It keeps an exponentially decaying average of past gradients.

> Adam is computationally efficient and has very less memory requirement.

> Adam optimizer is one of the most popular and famous gradient descent optimization algorithms.

Comparisions

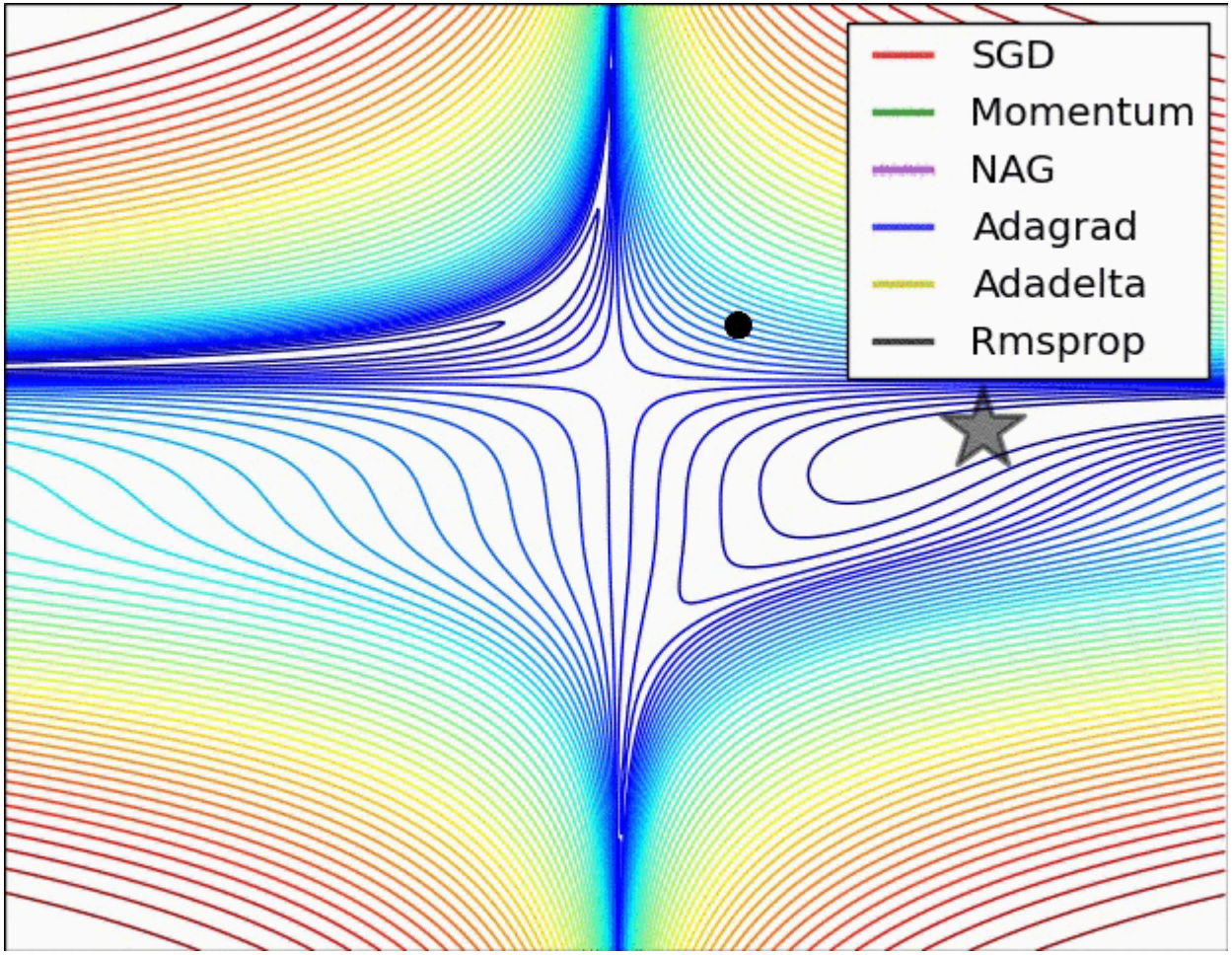


Figure :- SGD optimization on loss surface contours

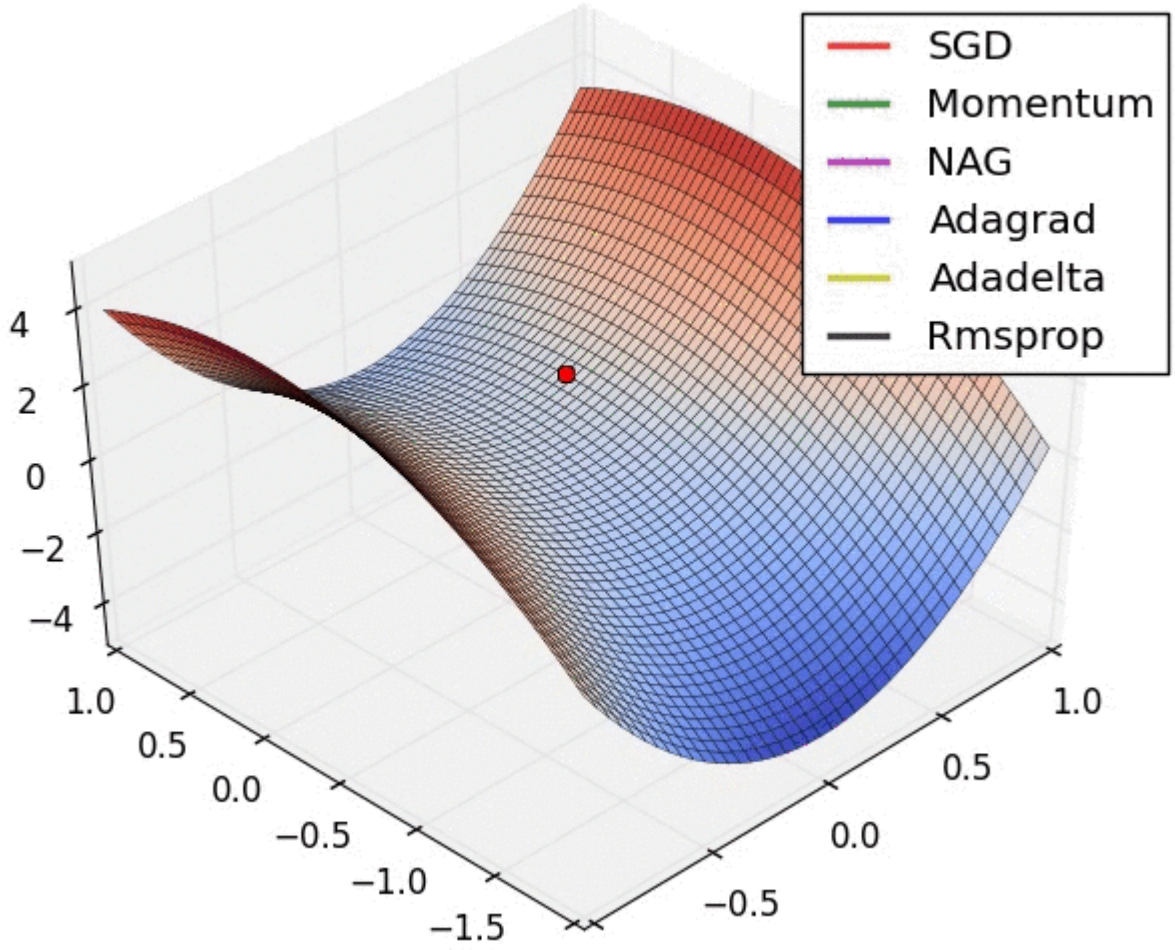


Figure :- SGD optimization on saddle point

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.

Overall, Adam is the best choice.

> SGD is used in many papers, without momentum, etc. Although SGD can reach a minimum value, it takes longer than other algorithms and may be trapped in the saddle point.

- If faster convergence is needed, or deeper and more complex neural networks are trained, an adaptive algorithm is needed.