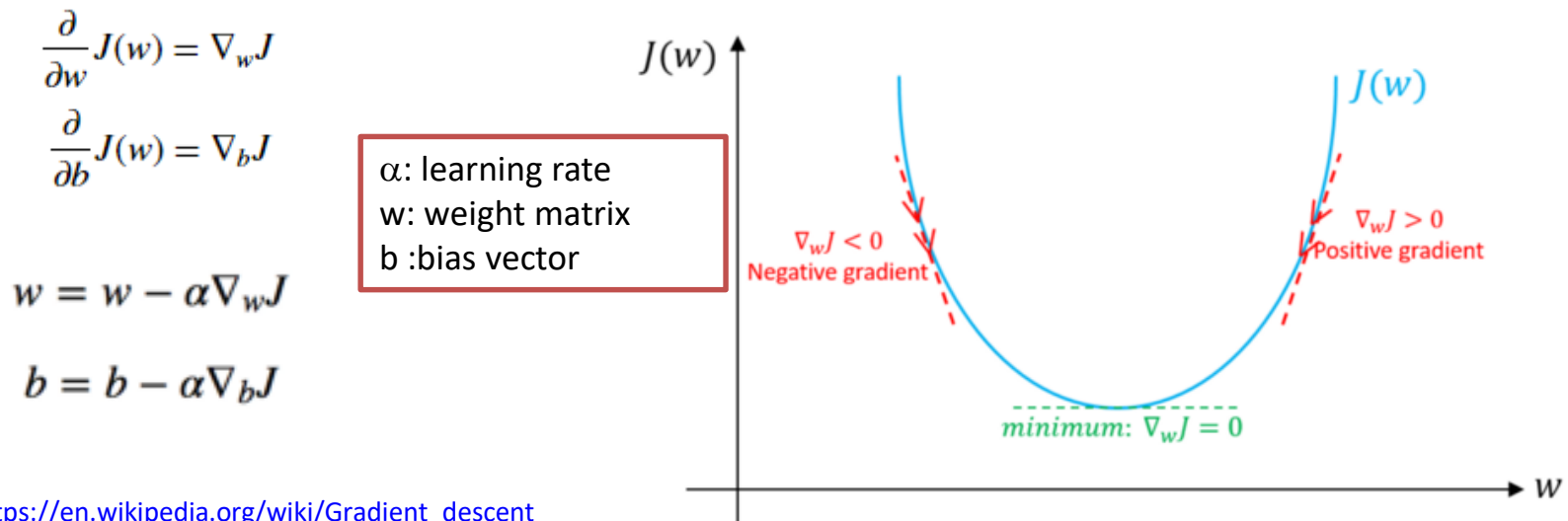


- **Gradient Descent**
- **Back Propagation**
- **Loss functions**

Gradient Descent

- Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable 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.
- Conversely, stepping in the direction of the gradient will lead to a local maximum of that function; the procedure is then known as gradient ascent.
- In machine learning, we use gradient descent to update the parameters of our model.

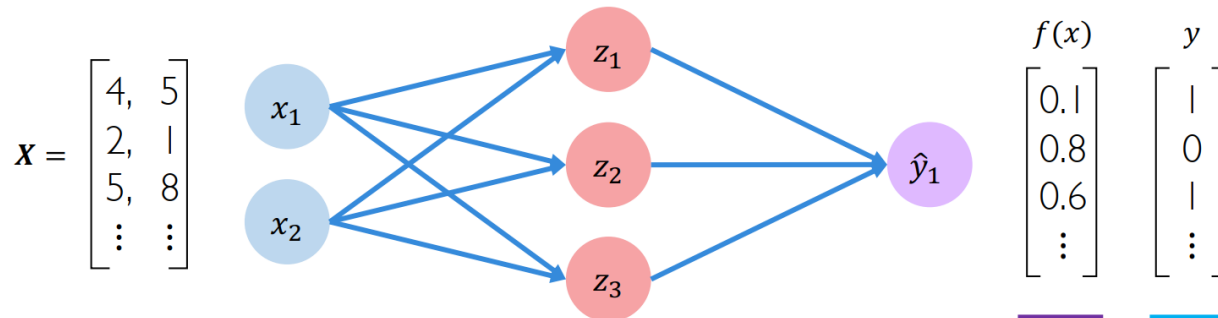


https://en.wikipedia.org/wiki/Gradient_descent

https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

<https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3>

Approaches for computing gradient



Below approaches determines when the weights in the network is updated.

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini Batch Gradient Descent

Batch gradient descent

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \quad \text{Notation may change in other resources} \quad (w = w - \alpha \nabla_w J(w)) \quad (1)$$

As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory. Batch gradient descent also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):  
    params_grad = evaluate_gradient(loss_function, data, params)  
    params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, we first compute the gradient vector `params_grad` of the loss function for the whole dataset w.r.t. our parameter vector `params`. Note that state-of-the-art deep learning libraries provide automatic differentiation that efficiently computes the gradient w.r.t. some parameters. If you derive the gradients yourself, then gradient checking is a good idea.⁶

We then update our parameters in the direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}) \quad (2)$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Figure [1](#).

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively. Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example. Note that we shuffle the training data at every epoch as explained in Section [6.1](#).

```
for i in range(nb_epochs):  
    np.random.shuffle(data)  
    for example in data:  
        params_grad = evaluate_gradient(loss_function, example, params)  
        params = params - learning_rate * params_grad
```

Mini-batch gradient descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \quad (3)$$

This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used. Note: In modifications of SGD in the rest of this post, we leave out the parameters $x^{(i:i+n)}; y^{(i:i+n)}$ for simplicity.

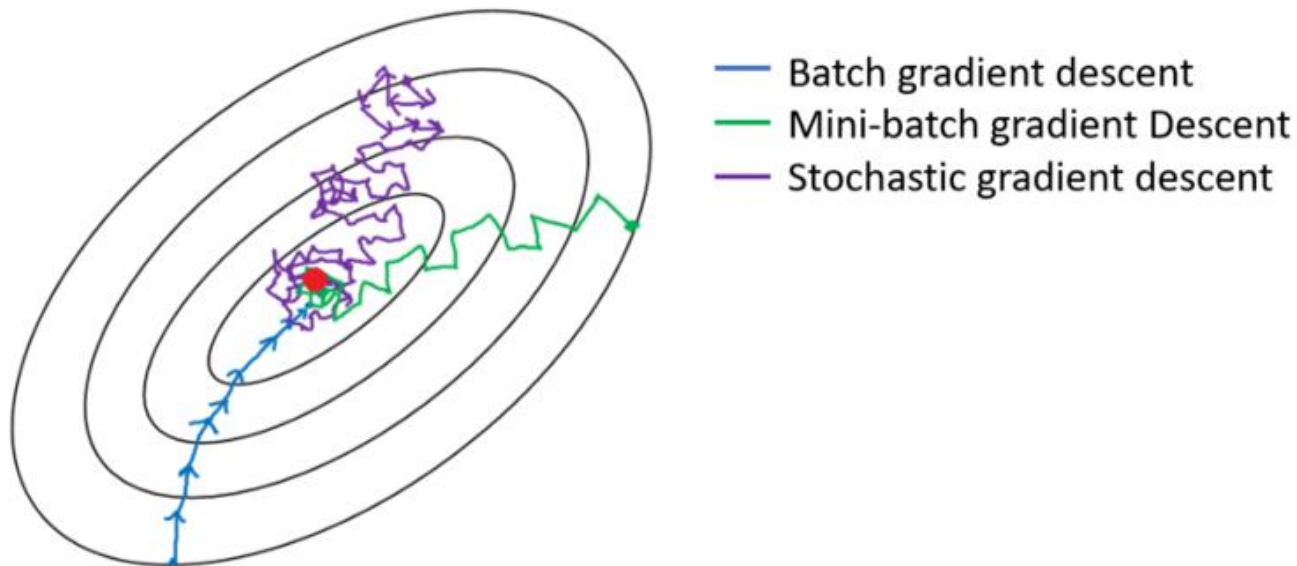
In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):  
    np.random.shuffle(data)  • •  
    for batch in get_batches(data, batch_size=50):  
        params_grad = evaluate_gradient(loss_function, batch, params)  
        params = params - learning_rate * params_grad
```

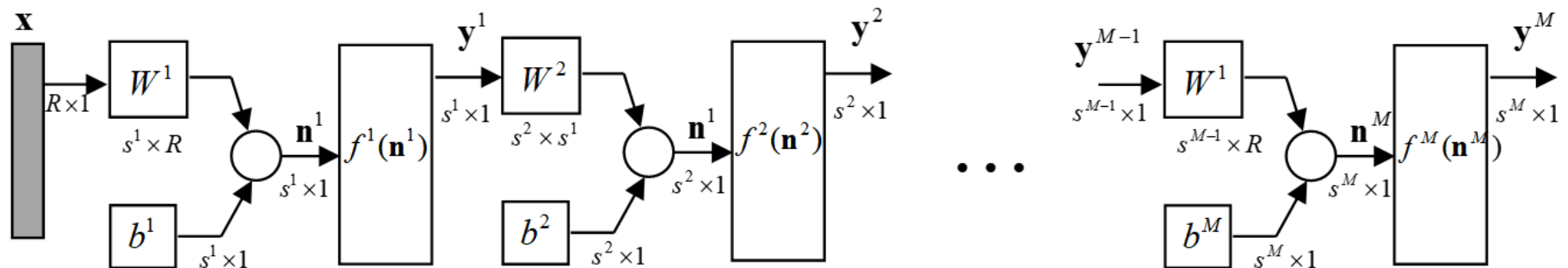
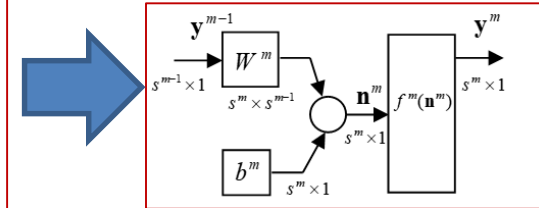
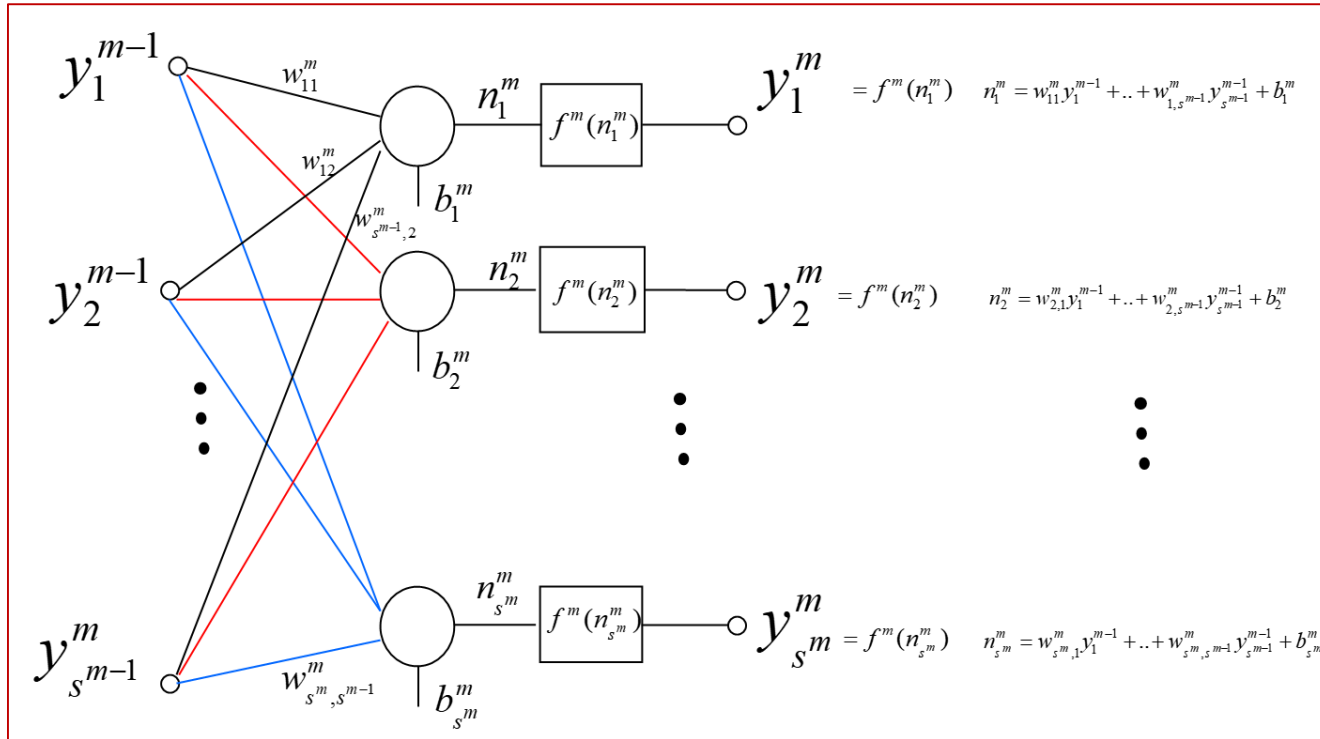
Shuffle the training data set to avoid pre-existing order of examples.

Comparison

- Below is a graph that shows the gradient descent's variants and their direction towards the minimum.
- SGD direction is very noisy compared to mini-batch.



Dense Layer



Dense Layer

$$y_i^m = f^m(w_{i,1}^m y_1^{m-1} + w_{i,2}^m y_2^{m-1} + \dots + w_{i,s^{m-1}}^m y_{s^{m-1}}^{m-1} + b_i^m) \quad i = 1, 2, \dots, s^m$$

Outputs for a dense layer

$$\begin{bmatrix} y_1^m \\ y_2^m \\ \vdots \\ y_{s^m}^m \end{bmatrix} = \mathbf{f}^m \left(\begin{bmatrix} w_{1,1}^m & w_{1,2}^m & \dots & w_{1,s^{m-1}}^m \\ w_{2,1}^m & w_{2,2}^m & \dots & w_{2,s^{m-1}}^m \\ \dots & \dots & \dots & \dots \\ w_{s^m,1}^m & w_{s^m,2}^m & \dots & w_{s^m,s^{m-1}}^m \end{bmatrix} \begin{bmatrix} y_1^{m-1} \\ y_2^{m-1} \\ \vdots \\ y_{s^{m-1}}^{m-1} \end{bmatrix} + \begin{bmatrix} b_1^m \\ b_2^m \\ \vdots \\ b_{s^m}^m \end{bmatrix} \right)$$

In matrix form

$$\mathbf{y}^m = \mathbf{f}^m (\mathbf{W}^m \mathbf{y}^{m-1} + \mathbf{b}^m)$$

Back propagation

Input : $\mathbf{y}^0 = \mathbf{x}$ Output $\mathbf{y}^M = \mathbf{y}$

Output of m th layer

$$\mathbf{y}^m = \mathbf{f}^m(\mathbf{W}^m \mathbf{y}^{m-1} + \mathbf{b}^m)$$

Back propagation

Output layer $\mathbf{d}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{y}) \quad m=M$

For others $\mathbf{d}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{d}^{m+1} \quad m=M-1, \dots, 2, 1$

Loss function (SSE)

$$E = \mathbf{e}^T \mathbf{e} = (\mathbf{t} - \mathbf{y})^T (\mathbf{t} - \mathbf{y}) = \sum_{i=1}^s (t_i - y_i)^2$$

Derivative of loss function

Update weights and biases

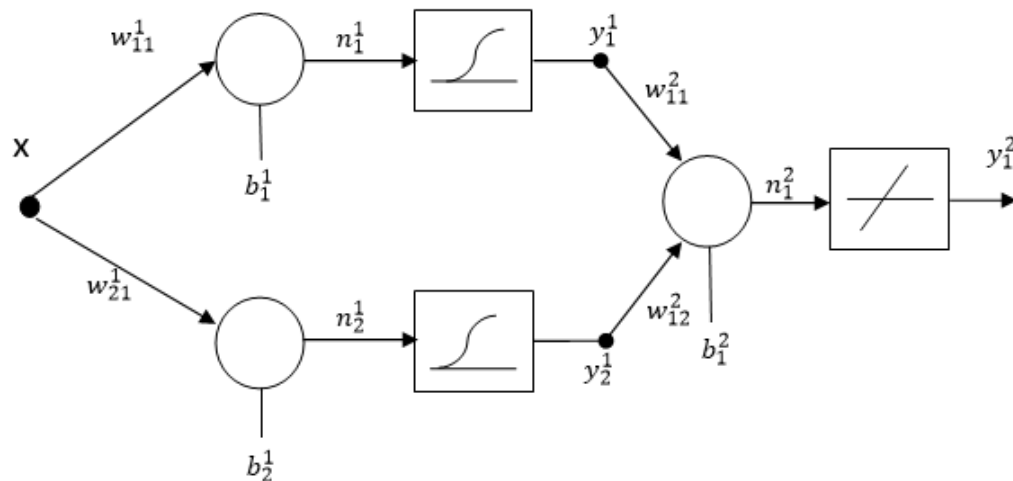
Gradient
Descent

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{d}^m (\mathbf{y}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{d}^m$$

This is applied
to each layer of
the network

Back propagation example



Training set

$$P = [-2, -1.5, -1, -0.5, 0, 0.5, \mathbf{1}, 1.5, 2]$$

$$T = [0, 0.075, 0.292, 0.617, 1.0, 1.382, \mathbf{1.707}, 1.923, 2]$$

$$X = 1 \rightarrow y = y_1^2 = ?, \quad (T = 1.707, \alpha = 0.1)$$

$$W^1 = \begin{bmatrix} w_{11}^1 \\ w_{21}^1 \end{bmatrix} = \begin{bmatrix} -0.3 \\ 0.5 \end{bmatrix} \quad b^1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}$$

$$W^2 = [w_{11}^2 \quad w_{12}^2] = [0.5 \quad 0.6] \quad b^2 = [b_1^2] = [-0.1]$$

Random initial conditions

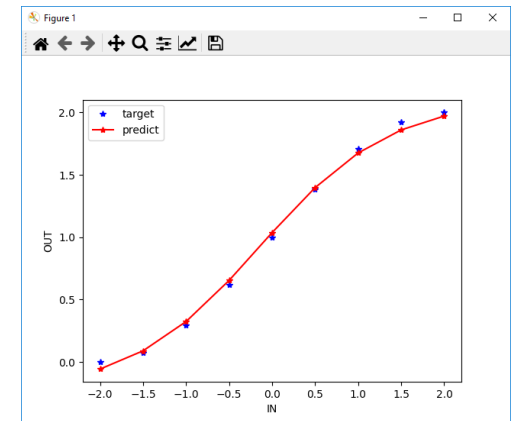
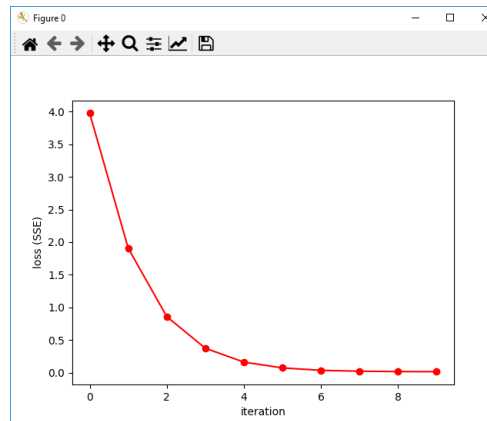
Back propagation example

```
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from matplotlib.pyplot import plot
```

```
7 def sigmoid(x):
8     y=1/(1+np.exp(-x))
9     return y
10 def dsigmoid(x):
11     y=(1-x)*x
12     return y
13 # SGD (online training)
14 #DATA SET -----
15 #X=np.linspace(-2,2,9)
16 X=np.array([-2,-1.5,-1,-0.5, 0,0.5,1,1.5,2],dtype='float32')
17 #T=1+np.sin(X*np.pi/4)
18 T=np.array([0, 0.075,0.292,0.617,1.0, 1.382,1.707,1.923,2],dtype='float32')
19 # Initialize trainable parameters -----
20 W1=np.random.rand(2,1)
21 b1=np.random.rand(2,1)
22 W2=np.random.rand(1,2)
23 b2=np.random.rand(1)
24 alfa=0.3#learning rate
25 epoch=10
```

```
27 SSE=np.empty(epoch)
28 for k in range(epoch):
29     for i in range(X.size):
30         #Forward propagation
31         #layer 1
32         y1=sigmoid( W1*X[i]+b1)
33         #layer 2
34         y2=np.matmul(W2,y1)+b2
35
36         # Back propagation
37         F2=1
38         d2=-2*F2*(T[i]-y2 )
39
40         F1=np.array([[ dsigmoid(y1[0]) , 0],
41                     [ 0 , dsigmoid(y1[1]) ] ])
42
43         d1= np.matmul(F1.astype('float32'), W2.reshape(2,1))*d2
44
45         # update weights in layer2
46         W2=W2-alfa*d2*y1.reshape(1,2) #y1'
47         b2=b2-alfa*d2
48         # update weights in layer1
49         W1=W1-alfa*d1*X[i]
50         b1=b1-alfa*d1
51     #print loss at the end of the epoch
52     #forward propagation
53     err=0
54     for i in range(len(X)):
55         #layer 1
56         Y1=sigmoid( W1*X[i]+b1)
57         #layer 2
58         Y2=np.matmul(W2,Y1)+b2
59         err+= (T[i]-Y2)**2
60     SSE[k]=err
```

```
62 plt.figure(0)
63 plt.plot(range(epoch),SSE,'r-o')
64 plt.xlabel("iteration")
65 plt.ylabel("Loss (SSE)")
66 print("W1=",W1,"\nW2",W2)
67 print("b1=",b1,"\nb2",b2)
68 #test network with trained weights
69 Y=np.empty(len(X))
70 for i in range(len(X)):
71     #layer 1
72     Y1=sigmoid( W1*X[i]+b1)
73     #layer 2
74     Y[i]=np.matmul(W2,Y1)+b2
75
76 plt.figure(1)
77 plt.plot(X,T,'b*')
78 plot(X,Y,'r-*')
79 plt.xlabel("IN")
80 plt.ylabel("OUT")
81 plt.legend(['target','predict'])
```



Loss functions

- Loss functions are used to calculate the amount of error in the back propagation algorithm.
- Loss function is chosen according to the problem type.
- Some loss functions:
 - Problem type: regression:
 - MSE
 - MAE
 - Problem type: classification
 - binary cross entropy
 - categorical cross entropy

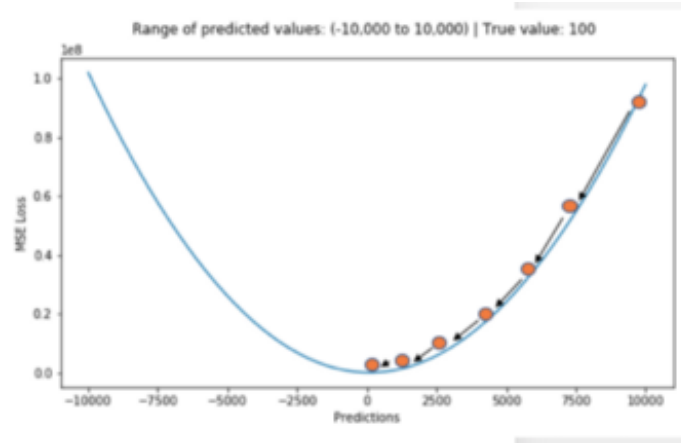
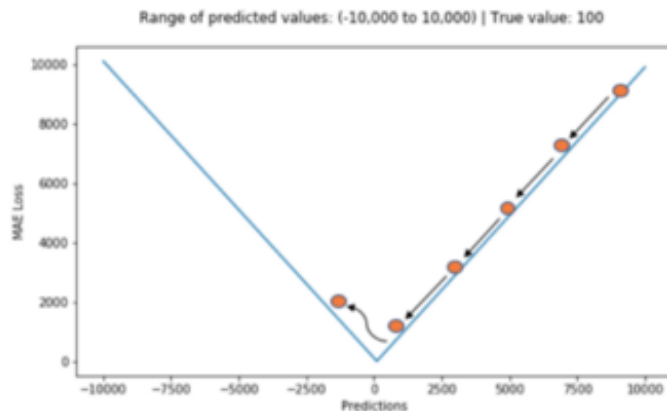
MSE loss ve MAE loss

- MSE (Mean Square Error), L2 loss

$$MSE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}$$

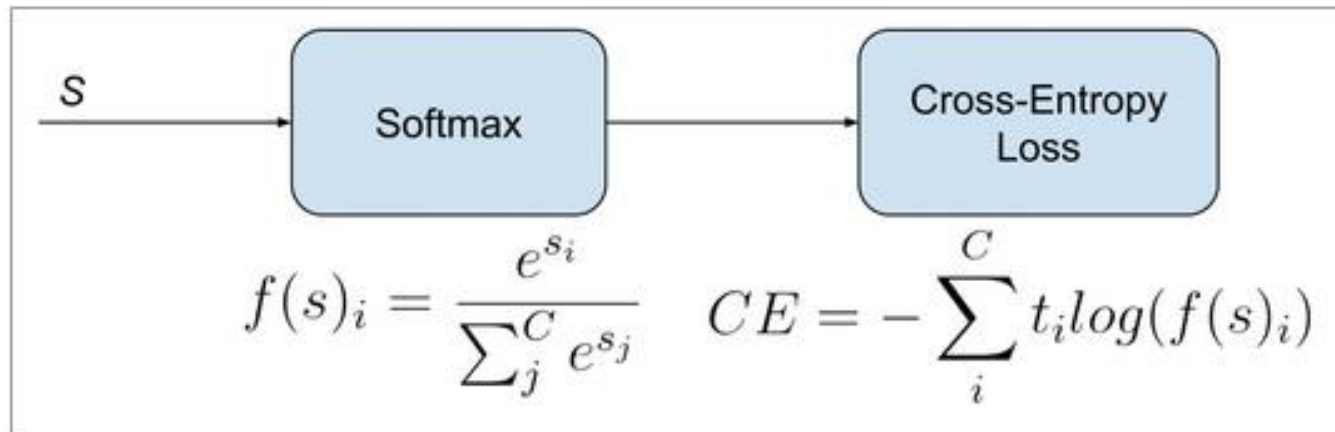
- MAE (Mean Absolute Error), L1 loss

$$MAE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{n}$$



Cross Entropy Loss

- Categorical cross entropy:
 - Also called **Softmax Loss**. It is a **Softmax activation** plus a **Cross-Entropy loss**. If we use this loss, we will train a CNN to output a probability over the C classes for each image. It is used for multi-class classification.

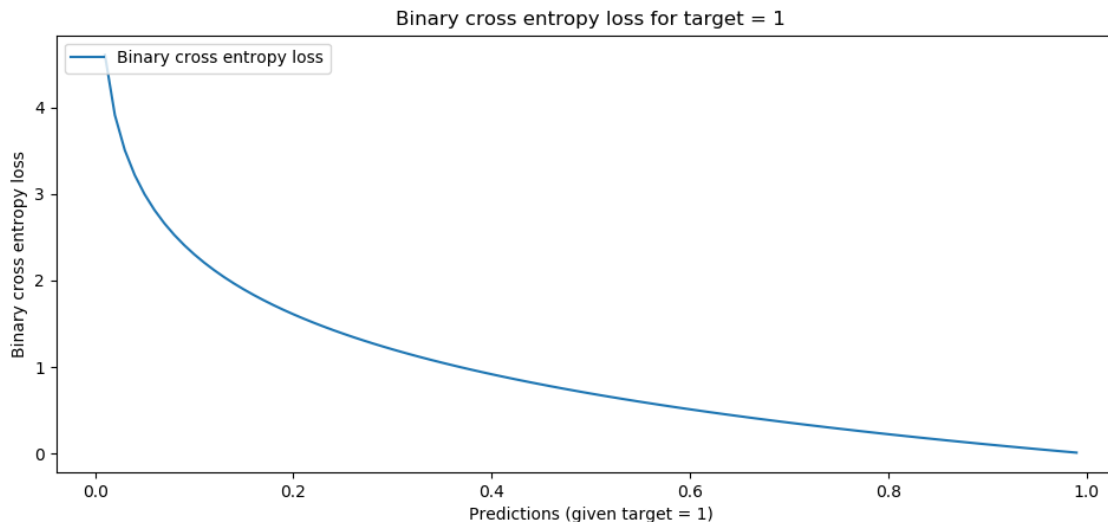


- 0: [1, 0, 0]
- 1: [0, 1, 0]
- 2: [0, 0, 1]

Cross Entropy Loss

- **Binary crossentropy:**
- Binary classification

$$BCE(t, p) = -(t * \log(p) + (1 - t) * \log(1 - p))$$



MAPE, RMSE, Logcosh and Huber

- Mean Absolute Percentage Error (MAPE)

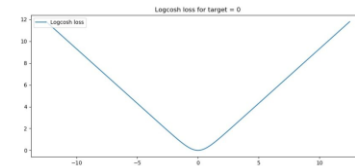
$$M = \frac{100\%}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right|,$$

- Root Mean Squared Error

$$\text{RMSD}(\hat{\theta}) = \sqrt{\text{MSE}(\hat{\theta})}$$

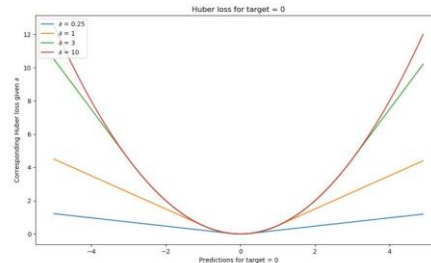
- Logcosh

$$\text{Logcosh}(t) = \sum_{p \in P} \log(\cosh(p - t))$$




- Huber loss, (Smooth Mean Absolute Error)

$$\text{Huber loss}(t, p) = \begin{cases} \frac{1}{2}(t - p)^2, & \text{when } |t - p| \leq \delta \\ \delta|t - p| - \frac{\delta^2}{2}, & \text{otherwise} \end{cases}$$



Keras loss functions

- **keras.losses**
- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- huber_loss
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity
- ...



A custom loss can be defined in Keras.

Creating custom losses

Any callable with the signature `loss_fn(y_true, y_pred)` that returns an array of losses (one of sample in the input batch) can be passed to `compile()` as a loss. Note that sample weighting is automatically supported for any such loss.

Here's a simple example:

```
def my_loss_fn(y_true, y_pred):  
    squared_difference = tf.square(y_true - y_pred)  
    return tf.reduce_mean(squared_difference, axis=-1) # Note the 'axis=-1'  
  
model.compile(optimizer='adam', loss=my_loss_fn)
```