MLFA-Neural-Net-2

07-Mar-2023 at 6:22 AM

Deep Learning

Generalized Backpropagation:

Earlier, we have talked about back propagation based learning for MLP with one input, one hidden and one output layer. In doing so, we have used chain rule for computing derivatives of the error function with respect to the parameters.

However, for a deep network, the number of hidden layers is arbitrary. This requires a general approach towards computing the gradient (e.g., performing backpropagation).

A feedforward neural network is a computational graph whose nodes are computing units and whose directed edges transmit numerical information from node to node. Such a network represents a **chain of function compositions** that transform an input to an output pattern. The learning problem consists of finding the optimal combination of weights so that the network function ϕ approximates a given function f. Difficulty is that we are not given with the function f explicitly but only given with a set of examples.

For a given training set $X = \{x^t, r^t\}$, let us consider the error function:

$$E = \frac{1}{2} \sum_t (r^t - y^t)^2$$

The weights or parameters are initialized randomly. The backpropagation algorithm is used to find a **local minima** of the error function. The gradient of the errore function is calculated and used to update the weights of the previous iteration. To accomodate arbitrary no of hidden layers the gradients have to computed **recursively**.

Let us consider the weights of the network to be $w_1, w_2, \ldots, w_l$. We need to calculate

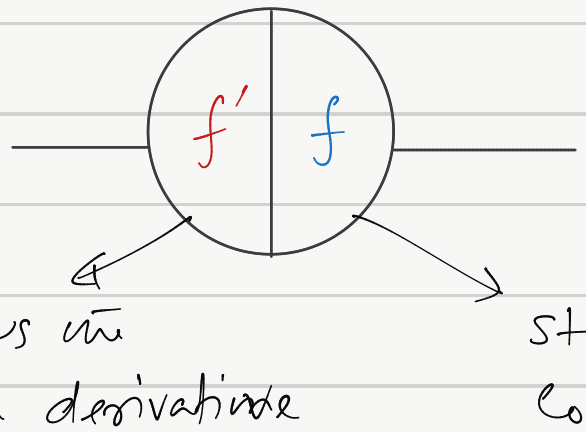
$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_l} \right)$$

The update in weights are

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad i = 1, \ldots, l$$

Derivatives of Network Functions:

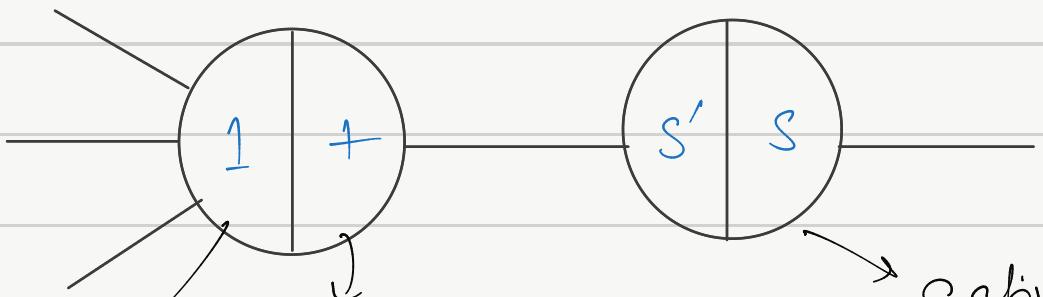
Let us consider a one dimensional network function (having one input and one output). This network is a complex chain of function compositions. We represent the nodes as composite structure to record both forward and gradient computation



Stores the local derivative

stores the forward computation

This can be further detailed as

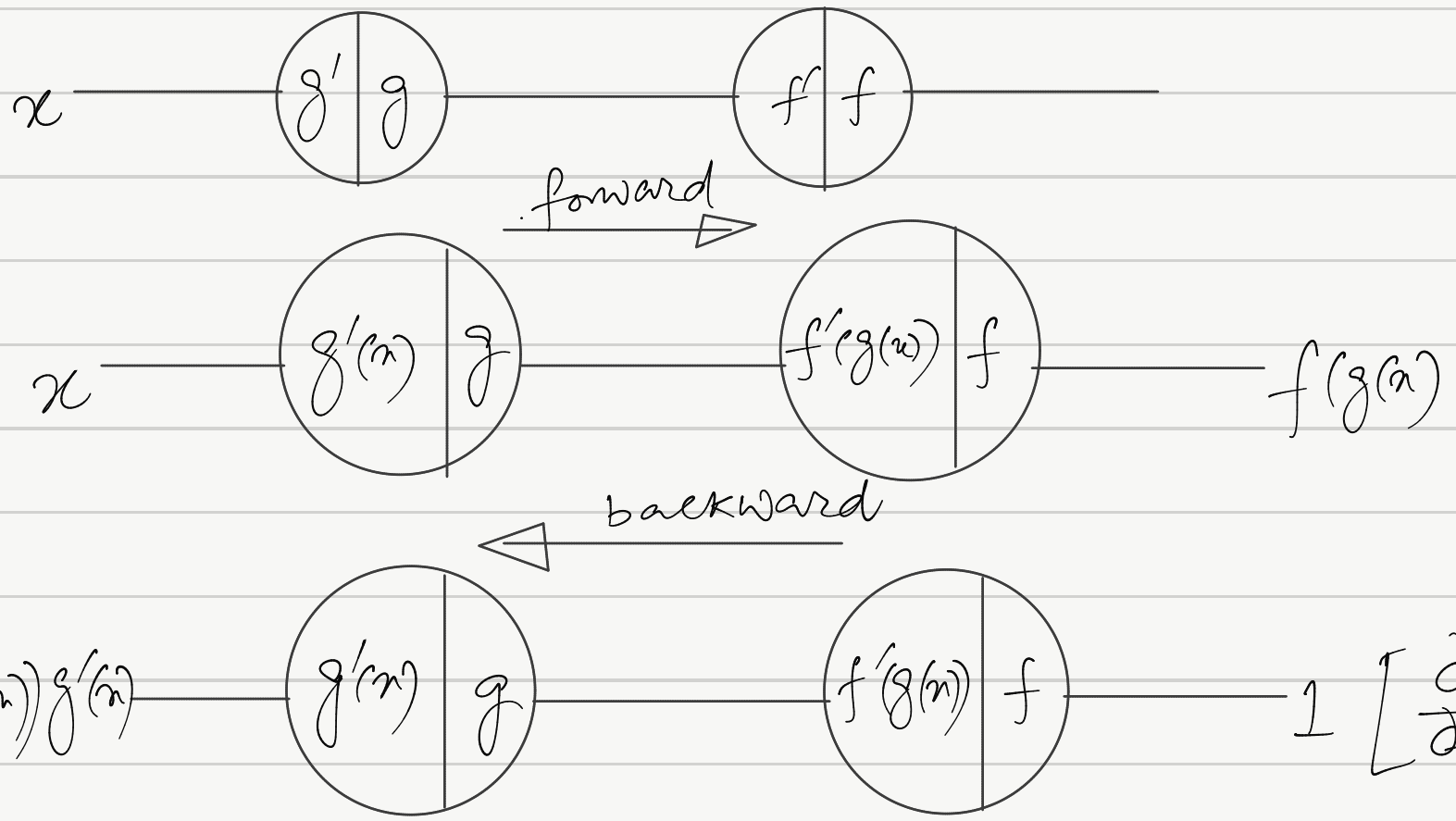


partial derivative with respect to any one of them is 1.

forward aggregation

activation function

Case 1 : Function Composition $[f(g(x))]$

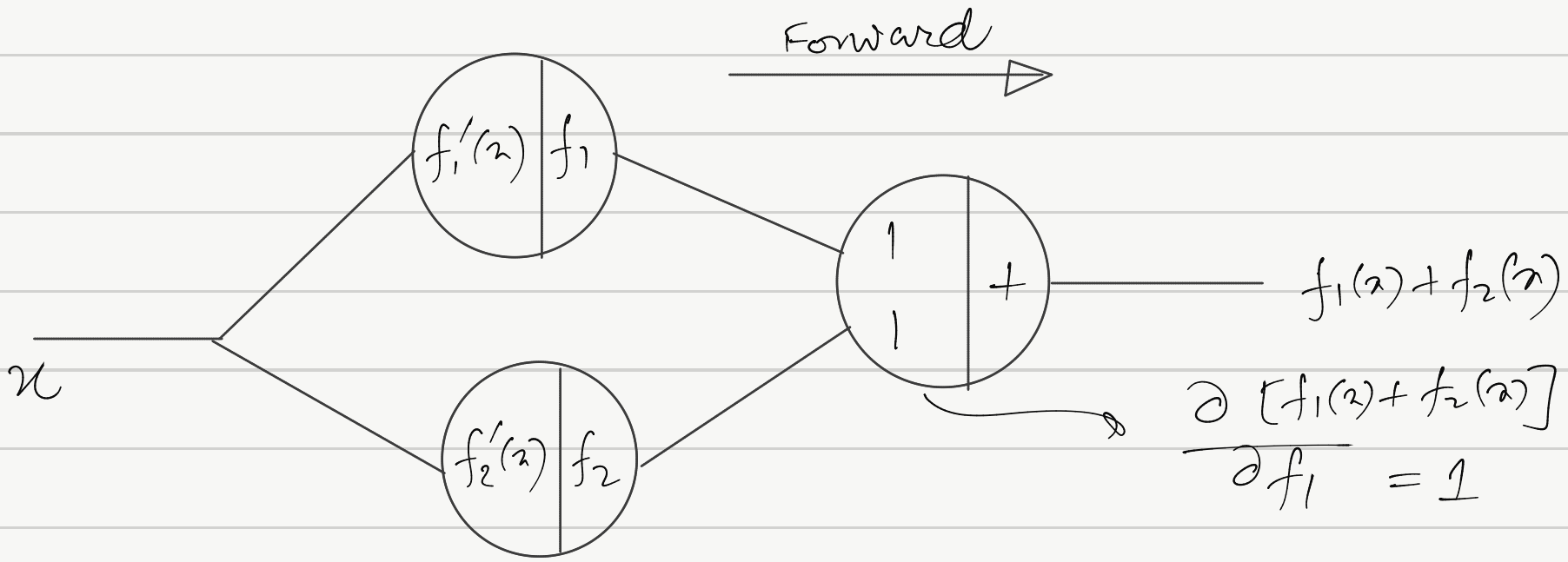


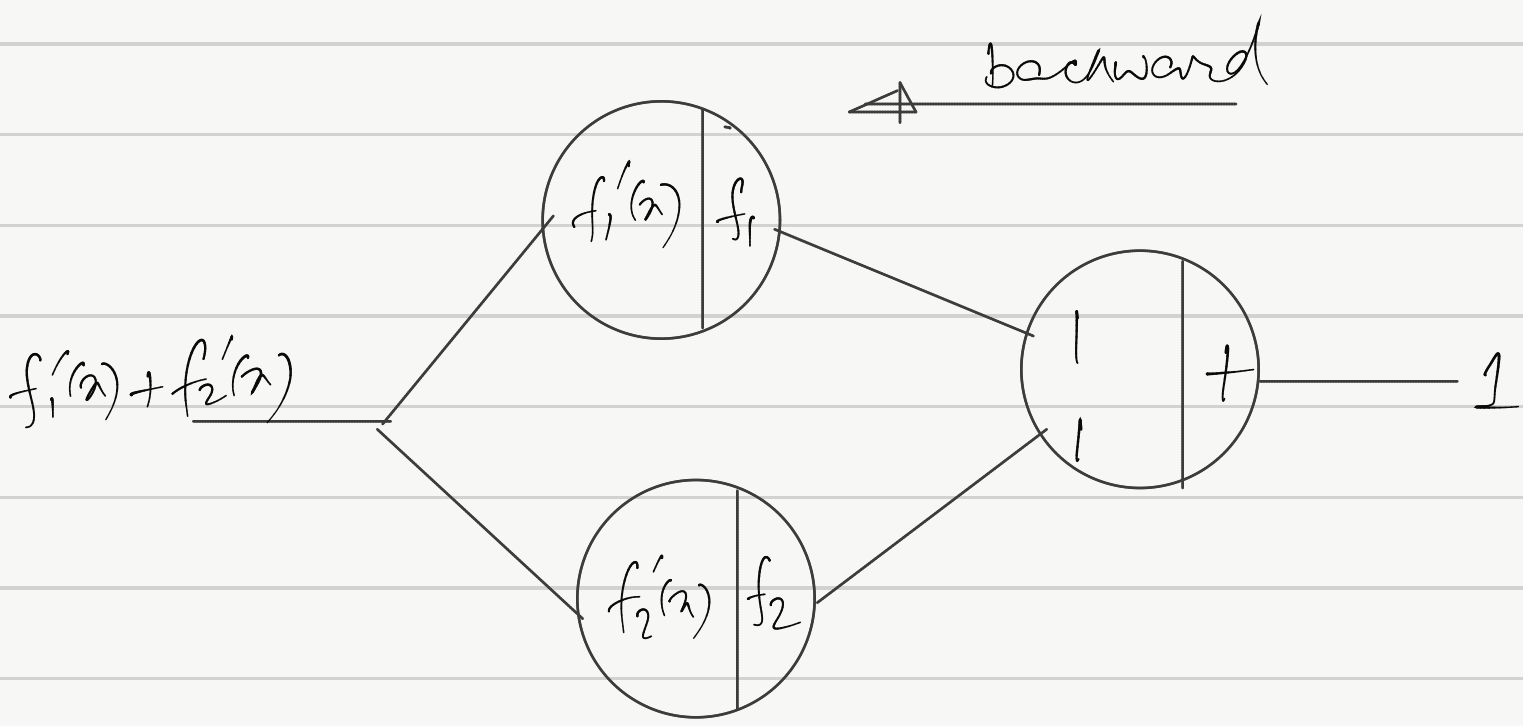
Let $g(x) = y$ $\quad \frac{\partial}{\partial x} f(g(x)) = \frac{\partial}{\partial y} f(y) \cdot \frac{\partial y}{\partial x} = f'(g(x)) \cdot g'(x)$

⇒ The computation in the network is performed backwards with the input 1

⇒ At each node the product with the value in the left side of the node is computed

Case 2: Function Addition $[f_1(x) + f_2(x)]$

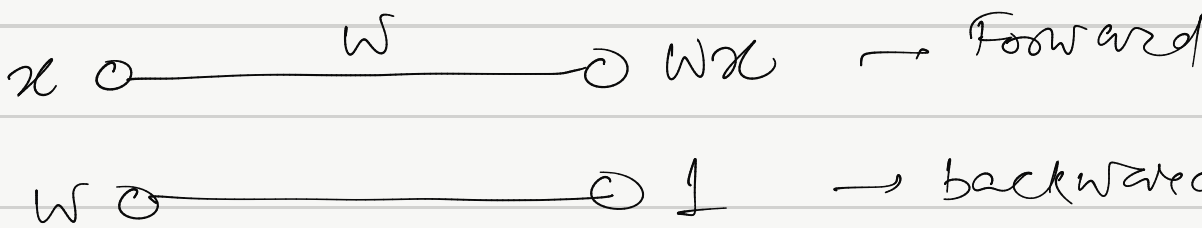




Let $y_1 = f_1(x)$ and $y_2 = f_2(x)$ $\quad z = f_1(x) + f_2(x)$

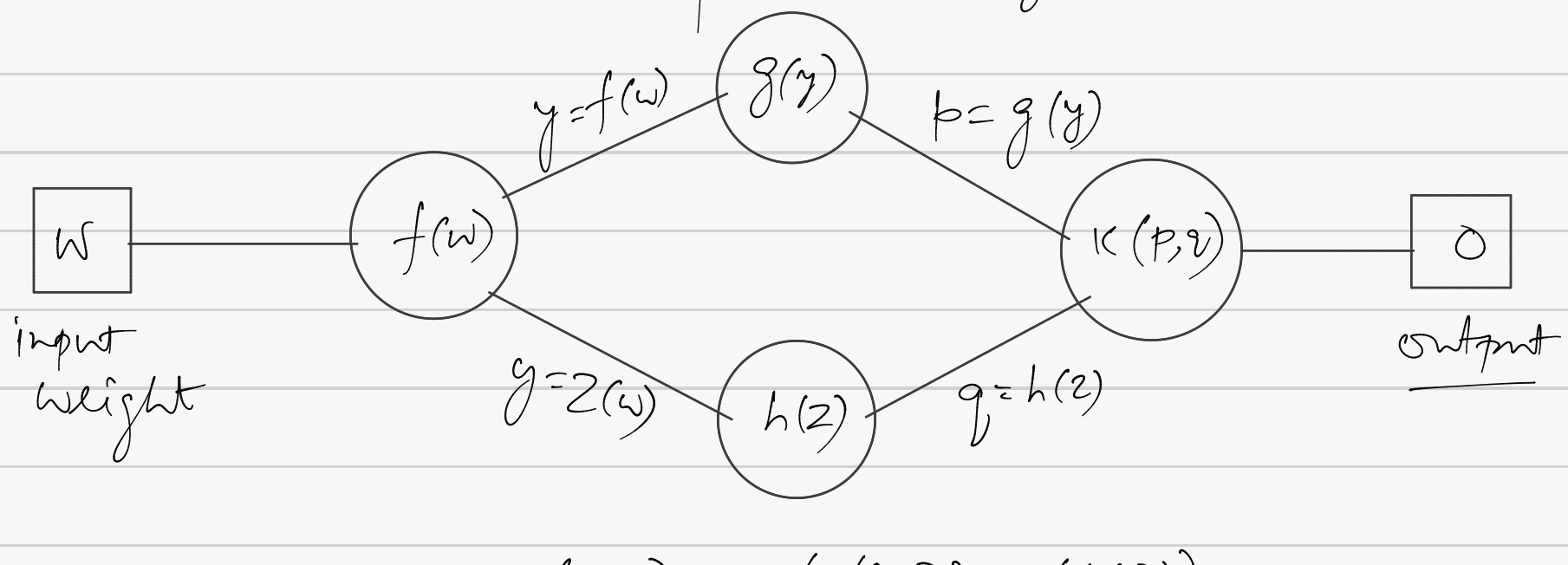
$$\frac{\partial z}{\partial x} = \frac{\partial y_1}{\partial x} + \frac{\partial y_2}{\partial x} = f_1'(x) + f_2'(x)$$

Case 3: Weighted Edges:



Derivative of Function Composition with one weight

* From now on the derivatives in each node will be dropped from the diagram.



$$O = K(p,q) = K(g(f(w)), h(f(w)))$$

$$\frac{\partial O}{\partial w} = \frac{\partial O}{\partial p}\cdot\frac{\partial p}{\partial y}\cdot\frac{\partial y}{\partial w} + \frac{\partial O}{\partial q}\cdot\frac{\partial q}{\partial z}\cdot\frac{\partial z}{\partial w}$$

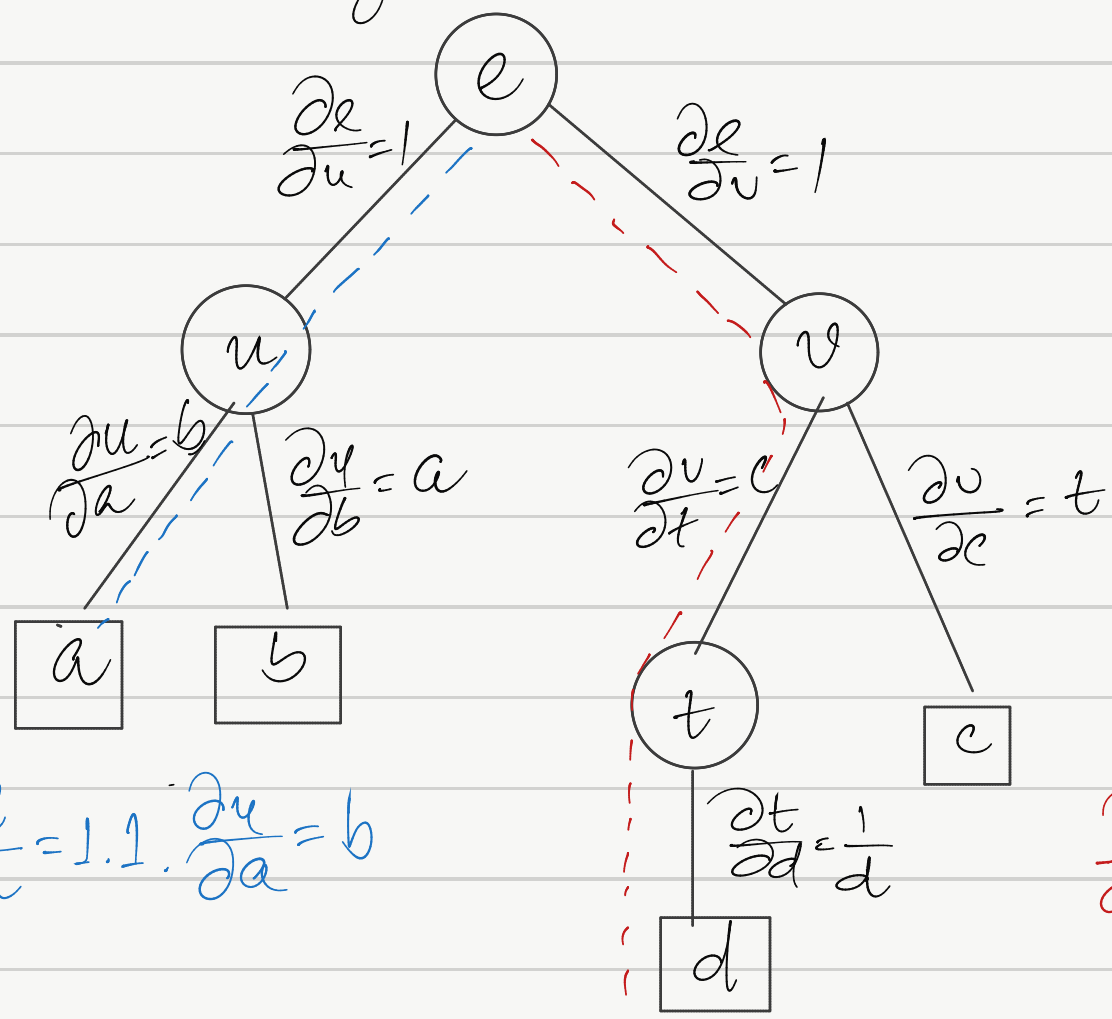
$$= \frac{\partial K(p,q)}{\partial p}\cdot g'(y)\cdot f'(w) + \frac{\partial K(p,q)}{\partial q}\cdot h'(z)\cdot f'(w)$$

Derivative of a composite function with 4 inputs

$$e = a\cdot b + c\log d$$

$$\frac{\partial e}{\partial a} = b,\quad \frac{\partial e}{\partial b} = a,\quad \frac{\partial e}{\partial c} = \log d\quad \frac{\partial e}{\partial d} = \frac{c}{d}$$

Computation graph:



$e = u+v$

$u = a.b$

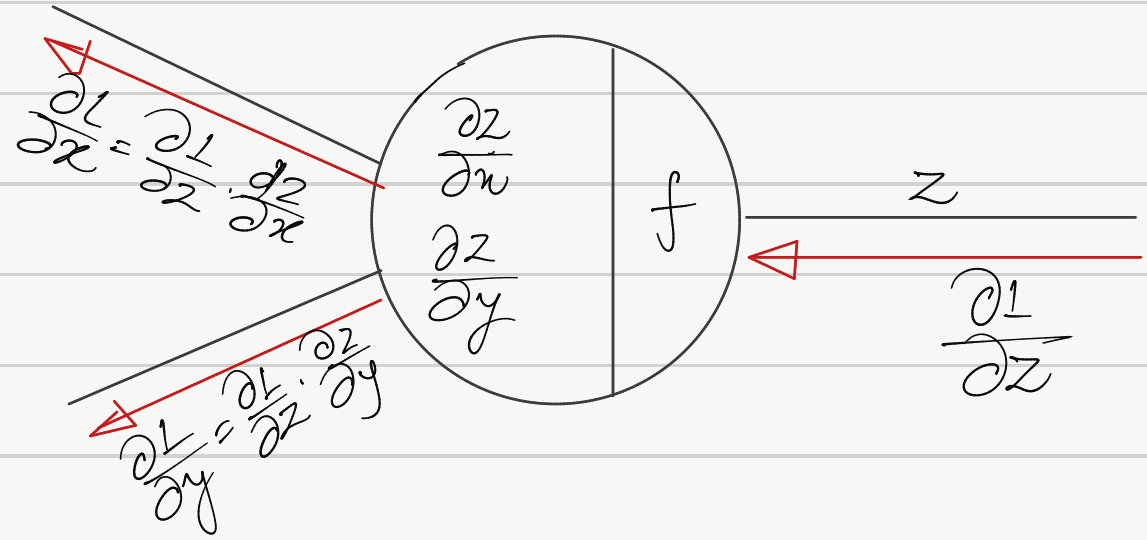
$v = c.t$

$t = \log d$

$\frac{\partial e}{\partial a} = 1.1.\frac{\partial u}{\partial a} = b$

$\frac{\partial e}{\partial d} = 1.1.c.\frac{\partial t}{\partial d} = \frac{c}{d}$

In general derivative for a neuron $z = f(x,y)$



Algo:

→ Choose random weights for the network

→ Feed an example and do forward computation

→ Calculate the error for each node

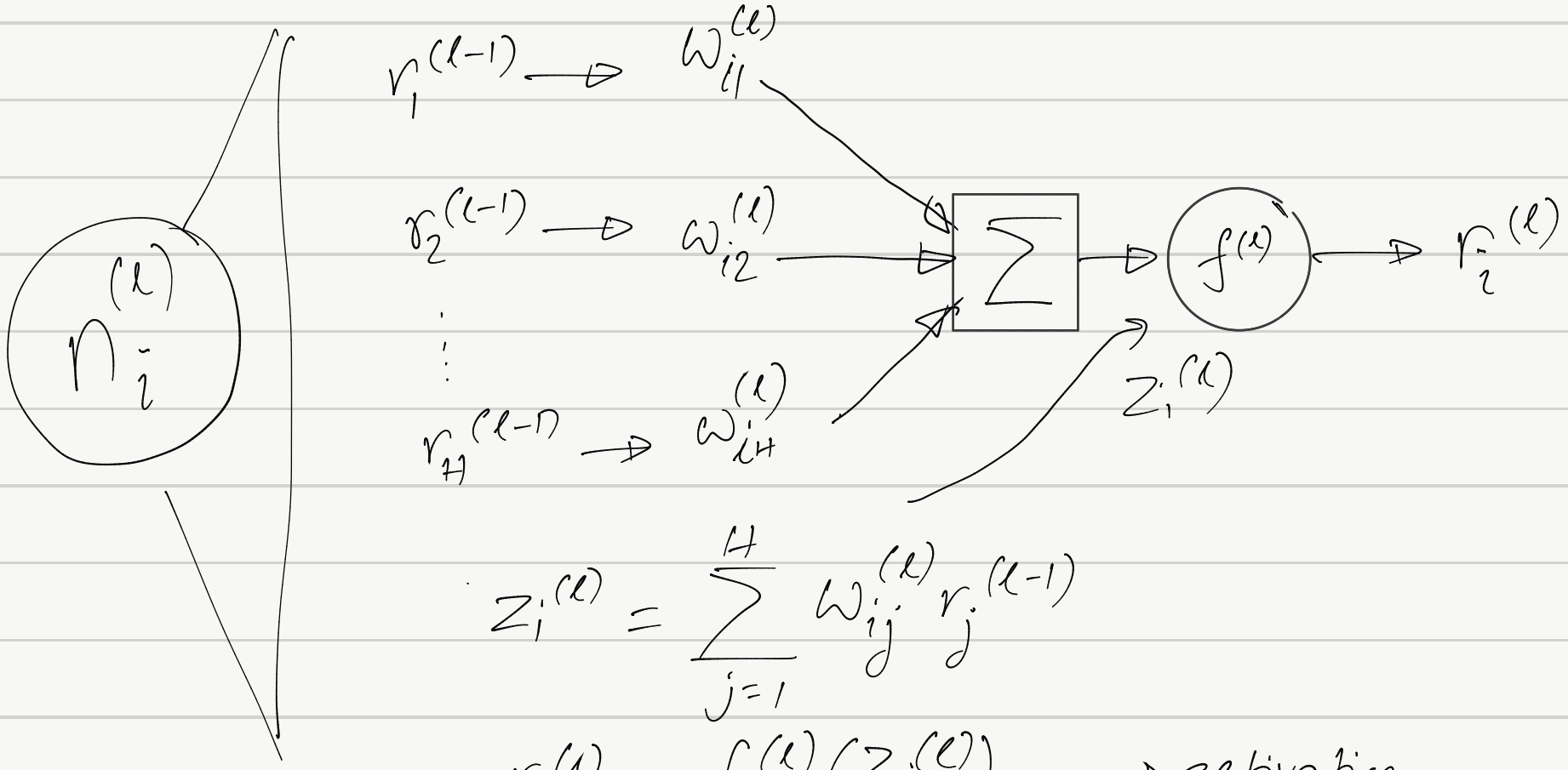
→ starting from the last stage and propagating error backwards

→ update weights

→ Repeat with other examples

Derivation of Backpropagation Rule :

Computation in i^{th} neuron of l^{th} layer :



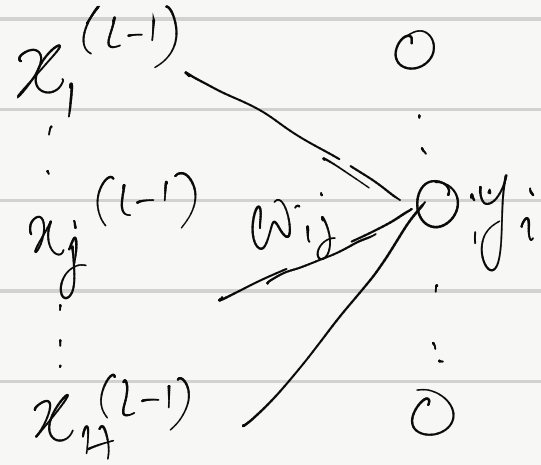
$$z_i^{(l)} = \sum_{j=1}^{H} w_{ij}^{(l)} r_j^{(l-1)}$$

$$r_i^{(l)} = f^{(l)}(z_i^{(l)}) \rightarrow \text{activation}$$

Update Rule :

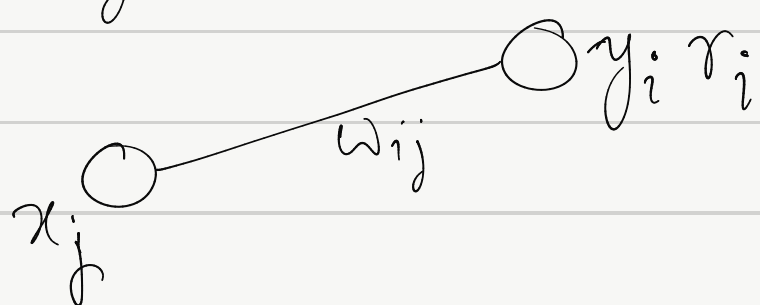
$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial E}{\partial w_{ij}^{(l)}}$$

Consider in simplest case (the output layer)



$$E = \frac{1}{2}(r_i - y_i)^2$$

$$\frac{\partial E}{\partial w_{ij}} = (r_i - y_i) \cdot x_j$$



$$\boxed{\frac{\partial E}{\partial w_{ij}} = \delta_j \, x_j}$$

In general, for the l^{th} layer

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}$$

$\swarrow$ $\delta_i^{(l)}$ — local gradient

$\searrow$ $z_i^{(l)} = \sum_{h=1}^{H} w_{ih}^{(l)} \, r_h^{(l-1)}$

$$\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} = r_j^{(l-1)}$$

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \cdot r_j^{(l-1)}$$

How to calculate $\delta_i^{(l)}$?

$$\delta_i^{(l)} = \frac{\partial E}{\partial z_i^{(l)}} = \frac{\partial E}{\partial r_i^{(l)}} \cdot \frac{\partial r_i^{(l)}}{\partial z_i^{(l)}}$$

$$r_i^{(l)} = f^{(l)}(z_i^{(l)}) \qquad \frac{\partial r_i^{(l)}}{\partial z_i^{(l)}} = f'^{(l)}$$

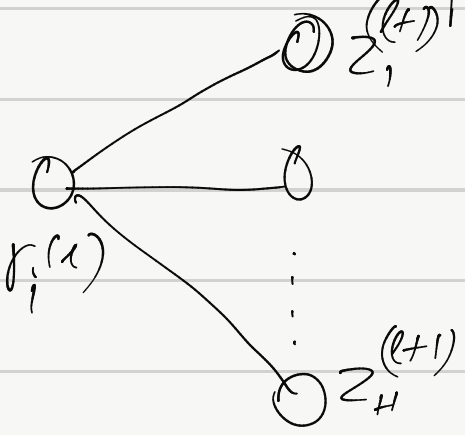
$$\delta_i^{(l)} = \frac{\partial E}{\partial r_i^{(l)}} \cdot f'^{(l)}$$

$\frac{\partial E}{\partial r_i^{(l)}}$ | How does $r_i^{(l)}$ contribute to the error fn E?

↳ $r_i^{(l)}$ is distributed to all the neurons in the next layer.

↳ Contribution of $r_i^{(l)}$ to the error function is through all the neurons that it is connected in the next layer.

↳ Thus variables of the error fn that are of interest to $r_i^{(l)}$ are $z_1^{(l+1)} \dots z_H^{(l+1)}$

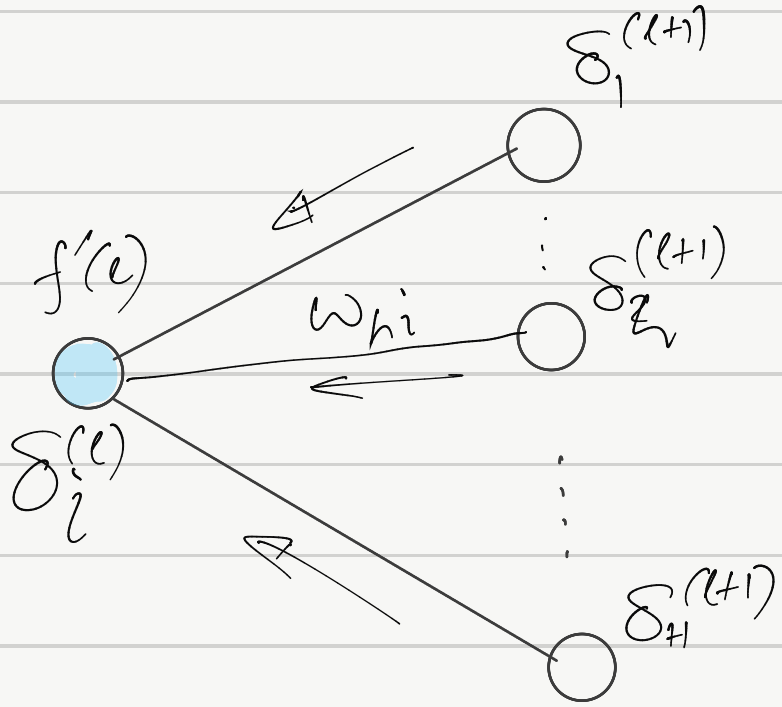


$$\frac{\partial E}{\partial r_i^{(l)}} = \frac{\partial E(z_1^{(l+1)}, \dots z_H^{(l+1)})}{\partial r_i^{(l)}}$$

$$= \sum_{h=1}^{H} \frac{\partial E}{\partial z_h^{(l+1)}} \cdot \frac{\partial z_h^{(l+1)}}{\partial r_i^{(l)}}$$

$$= \sum_{h=1}^{H} \delta_h^{(l+1)} w_{hi}^{(l+1)}$$

$$\delta_i^{(l)} = \begin{cases} \frac{\partial E}{\partial r_i^{(i)}} \cdot f'^{(l)} & \rightarrow \text{if output neuron} \\ \sum_{h=1}^{H} \delta_h^{(l+1)} \cdot w_{hi}^{(l+1)} \cdot f'^{(l)} & \rightarrow \text{For all other neurons} \end{cases}$$



BP Algo :

⇒ Apply input vector x^t to the network and forward propagate

$$z_h^{(l)} = \sum_i w_{hi}^{(l)} r_i^{(l-1)}$$

$$r_h^{(l)} = f^{(l)}(z_h^{(l)})$$

⇒ Evaluate δ_k for all the output units

⇒ Backpropagate δ using

$$\delta_i^{(l)} = f'^{(l)} \sum_h \delta_h^{(l+1)} \cdot w_{hi}$$

⇒ Compute $\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \cdot r_j^{(l-1)}$

Improving Training Convergence

Gradient descent in generally has a slow convergence. We will talk about some simple tricks to make it faster.

Momentum:

Slow convergence may happen due to large oscillations in the successive gradient value for a weight.

The update in the current iteration is made dependent on the updates in the previous iterations. (idea of momentum)

Let t be the time index
- ⇒ epoch no for batch learning
- ⇒ iteration no in online learning or minibatch learning

For each weight w_i,

$$s_i^{(0)} = 0$$

$$s_i^{(t)} = \alpha s_i^{(t-1)} + (1-\alpha)\frac{\partial E^{(t)}}{\partial w_i} \quad \text{[running average]}$$

$$\Delta w_i^{(t)} = -\eta s_i^{(t)}$$

$0 \leq \alpha < 1$ ⇒ decay parameter of running average

Adaptive Learning Rate :

Earlier, the η value was kept constant

* What if we make the learning rate adaptive to the situation.

* In deep network, the contributions of weights to the errors are not the same,

  ⇒ One can adapt the learning factor separately for each weight, depending on the convergence along that direction

  ⇒ Accumulate the past error gradient magnitudes for each weight and make learning factor inversely proportional to that

    ⇒ Learning takes place

  ⇒ In the direction, where the gradient is small, the learning factor is kept larger to move faster and can be kept smaller along the direction where the gradient is large.

RMSProp:

Keep a running average of the past gradients giving more importance to the recent values.

Accumulated past gradients:

$$r_i^{(t)} = \rho\, r_i^{(t-1)} + (1-\rho)\left|\frac{\partial E^t}{\partial w_i}\right|^2$$

$$\Delta w_i^{(t)} = -\frac{\eta}{\sqrt{r_i^{(t)}}}\frac{\partial E^{(t)}}{\partial w_i}$$

Adam (Adaptive Momentum)

It includes the momentum factor along with the accumulated past error gradients

$$s_i^{(t)} = \alpha\, s_i^{(t-1)} + (1-\alpha)\frac{\partial E^{(t)}}{\partial w_i}$$

$$r_i^{(t)} = \rho\, r_i^{(t-1)} + (1-\rho)\left|\frac{\partial E^{(t)}}{\partial w_i}\right|^2$$

$$\tilde{s}_i^{(t)} = \frac{s_i^{(t)}}{1-\alpha^t} \qquad \tilde{r}_i^{(t)} = \frac{r_i^{(t)}}{1-\rho^t}$$

[$\alpha^t \rightarrow$ α raised to power t]

$$\Delta w_i^t = -\eta\,\frac{\tilde{s}_i^{(t)}}{\sqrt{r_i^{(t)}}}$$

Batch Normalization

The values of the input variables may belong to multiple scales. Thus converting them to a standardized scale is a good idea. In this respect all the input values are z-normalized to have zero mean and unit variance.

Batch normalization extends this idea further by normalizing the hidden unit values before applying the activation function.

Let us consider the weighted sum for a hidden unit j is a_j. For each batch or minibatch for j^{th} unit, we calculate the mean m_j and standard deviation s_j (considering the a_j values)

The z-normalized value:

$$\tilde{a}_j = \frac{a_j - m_j}{s_j}$$

Now it can be mapped to have arbitrary mean γ_j and scale β_j

$$\tilde{a}_j = \gamma_j \tilde{a}_j + \beta_j$$

↳ is then fed to the activation fn.

During testing, we use m_j and s_j is calculated over the whole training data.

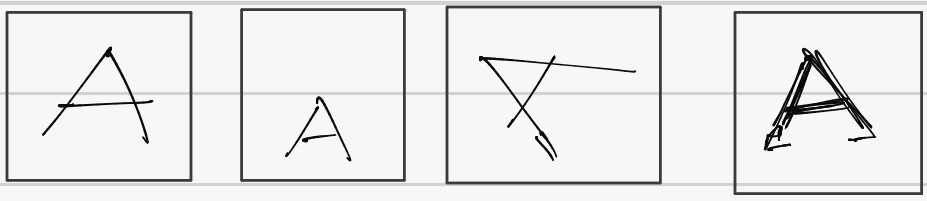
Regularization:

The complexity of an MLP needs to be adjusted to the complexity of the problem reflected by the underlying data.

⇒ Small networks have high bias and large or deep networks have high variances.

We want to deploy large networks, keeping the variance in check at the same time.

Data Augmentation:



⇒ Virtual examples

⇒ We can generate multiple copies of the same training examples at different scale and add them to the training set with same label as the original example.

⇒ increase the training set.

⇒ Augmented example may be included in the training example. Let x' is an augmented example for x. Let $g(x|\theta)$ represent one neural network. We can define a penalty

$$E_h = \sum_{x,x'} [g(x|\theta) - g(x''|\theta)]^2$$

$$E' = E + \lambda_h \cdot E_h$$

Weight Decay:

⇒ If a weight is zero, the input or hidden unit before that weight does not propagate values.

⇒ Traditionally, we initialize the weights to be closer to zero.

⇒ As the learning progresses, more and more weight moves away from zero

⇒ As if complexity of the model increases with training.

⇒ Though the training error may decrease, the validation error may increase

In weight decay, we add some constant force that always pulls a weight towards zero.

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} - \lambda w_i$$

It is equivalent to perform gradient descent on the error function

$$E' = E + \boxed{\frac{\lambda}{2} \sum_i w_i^2}$$

→ Sum over all the weights in all layers

← L2 - Regularization

Ridge Regression

↓ factor for speed of decay

** If we have two networks that have same training error, the simpler one (with fewer weights) has a higher probability of better generalization.

Dropout:

Adding noise to inputs helps in making the network robust as it does not rely too much on particular input.

⇒ Consider an image data set where, by luck a group of pixels always takes similar values for examples of a class.

⇒ Network will learn to base its decision for that class on them.

⇒ If we have additional examples where these pixels are missing, the dependency will be reduced

⇒ We are putting off some inputs

⇒ Similar argument may be extended to the hidden layer inputs.

⇒ Don't train to make the decision dependent on a subset of hidden units

⇒ Distribute the decision.

Corrupting or missing out is modelled by dropping out hidden or input units with a probability p. (set its output to be zero)

For each batch or minibatch, a smaller network is trained

⇒ Dropout is effectively sampling from a pool of networks.