Lecture 4

Deep learning: more than two layers

During the training (learning) process the algorithm searches for correlations (paterns) between input and output datasets.

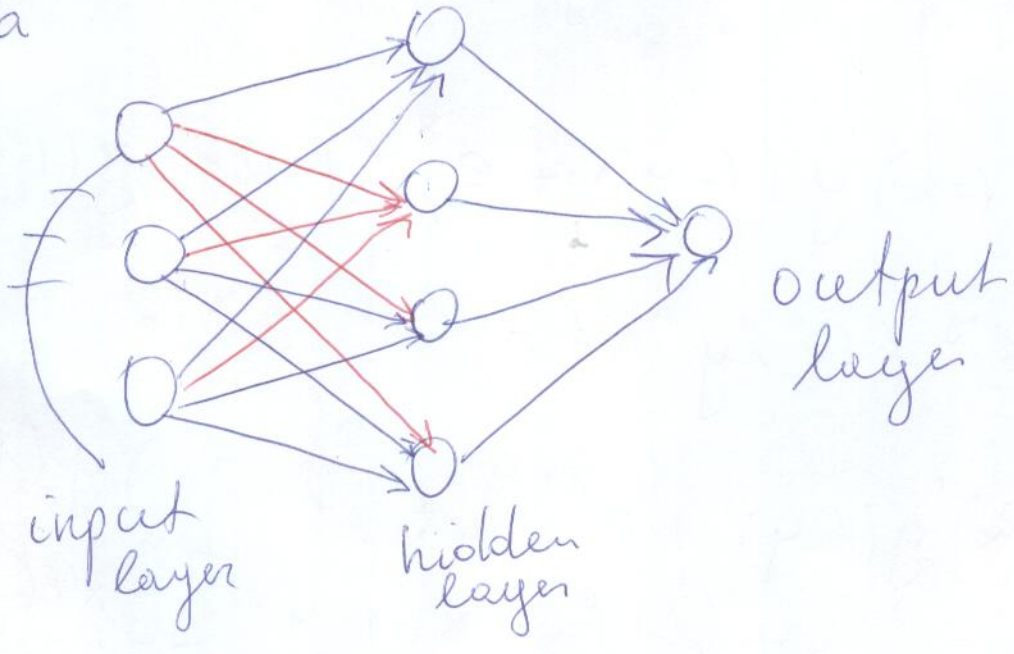
In fact for more general ANN the network searches correlations between their input and output layers.

Because the new input datasets doesn't necessary correlate with the output data set.

What we do in such situations.

A good idea is to use the input dataset to create an intermediate layer that does have a correlation with the output.

We create a virtual correlation by using this hidden layer of artificial neurons situated between the input and output layers, responsible for learning complex (non-linear) patterns of data



The presented ANN depends on the following data structures

$$weights\_01 = [\,[w_{00}^{01}, w_{01}^{01}, w_{02}^{01}, w_{03}^{01}],\ [w_{10}^{01}, w_{11}^{01}, w_{12}^{01}, w_{13}^{01}],\ [w_{20}^{01}, w_{21}^{01}, w_{22}^{01}, w_{23}^{01}]\,]$$ size $[3\times 4]$

$$weights\_12 = [[w_{00}^{12}], [w_{10}^{12}], [w_{20}^{12}], [w_{30}^{12}]]$$ size $[4\times 1]$.

The loss function

$$error(W) = \frac{1}{2}\sum_{m=0}^{M-1}\left(pred_m(W) - goal[m]\right)^2$$

M is the size of dataset

$$pred_m(W) = \sum_{p=0}^{P-1} w_{p0}^{12}\, layer^{1,m}[p]$$

$$layer^{1,m}[p] = \sum_{n=0}^{N-1} w_{np}^{01}\, layer^{0,m}[n]$$

For our NN:

input size $N = 3$, hidden layer 1 size $P = 4$

Next we consider the case

$M = 1$ (one input dataset).

The weights $W = \{W^{01}, W^{12}\}$ are defined by solving the minimization problem

$$\tilde{W} = \arg\min_{W} \text{error}(W)$$

$$\text{input} = [\text{inp}_0, \cdots, \text{inp}_{N-1}]$$

$$\text{output} = [\text{goal}_0].$$

$$pred(W) = \sum_{p=0}^{P-1} W_p^{12} \times pred_p^{1}$$

where

$$weights\_1\_2 = [W_0^{12}, \dots, W_{P-1}^{12}]$$

A short motivation to introduce an additional layer:

1. The number of free parameters is increased and it can be expected to get a better accuracy of predictions
2. Additional correlations between input and output data are possible
3. In general we expect that gradient descent method still works

Remarke. Such hopes should be restricted due to the obvious conclusion

In order to predict results connecting Level_0 and Level_1 nodes of the NN architecture we construct standard weighted sums

$$pred_p^1 = \sum_{n=0}^{N-1} w_{np}^{01} * inp_n ,$$

$$p = 0, \dots, P-1.$$

Similarly in order to predict the result on level_2

$$pred(W) = \sum_{p=0}^{P-1} w_p^{12} \times pred_p^1$$

There exists a single weighted sum with identical behaviour

$$pred(W) = \sum_{n=0}^{N-1} W_n^{0,2} \times inp_n$$

(layer°[n])

Anything that the three-layer network can do, the two-layer network can also do.

Thus such NN still doesn't works.

In general each node in the middle layer have some (may be small) amount of correlation with each input node.

Questions:

1. How to update the weights between layer_0 and layer 1 (between input and middle layers)?
2. What error measure we can use?

3. We want to know delta values at layer_1 to help layer 2 make accurate predictions.

Backpropagation

a) If we know the error of the prediction to the goal at layer 2

b) and if we have the weights ^{1,2} we can compute how much each node of layer_1 contributes

Remember this information

$$layer2\_delta = pred(W) - goal$$

$$pred(W) = \sum_{p=0}^{P-1} W_{p0}^{12} \, layer^1[p]$$

$$layer^1[p] = \sum_{n=0}^{N-1} W_{np}^{01} \, layer^0[n]$$

$$\| \; inp[n]$$

Now the value of $layer1\_delta$ can be predicted as ~~delta~~ given by two-layer NN but prediction is done by using a logic in reverse, or backpropagation

$$layer1\_delta[p] = W_{p0}^{12} * layer2\_delta$$

The weights ^{12} are updated in a standard way

$$\text{weights}^{12}[p] = \text{weight}^{1,2}[p] - \text{alpha} \times \text{grad}(W^{12})[p]$$

$$err = \frac{1}{2}\left(\sum_{p=0}^{P-1} \text{layer}1[p] \times w_{p0}^{12} - \text{goal}\right)^2$$

$$\text{grad}(W^{12})_p = \frac{\partial err}{\partial w_p^{12}} = \text{layer}2\_\text{delta} \times \text{layer}^1[p]$$

Since we have information (due to backpropagation technique) on layer1_delta, the update of weights^{01} again can be done in a standard way

$$\text{weights}_{np}^{01}[n] = \text{weights}_{np}^{01}[n] - \text{alpha} \times \text{layer}1\_\text{delta}[p]\ \text{layer}0[n]$$

This update formula follows from the gradient expression:

$$er_p = \frac{1}{2}\left(\sum_{n=0}^{N-1} layer0[n] \times weight^{01}_{np} - \tilde{g}_p\right)^2$$

$p = 0, 1, \dots, P-1$

$$grad(W^{01})_{np} = \frac{\partial er_p}{\partial W^{01}_{np}}$$

$$= layer1\_delta \times layer0[n]$$

It is clear, that layer1_delta was computed during backpropagation stage.

[Iterative procedure is expected to be applied in order to find W parameters. Bad news - in most cases such iterations are not converging

Let us make a test and apply three-level NN to solve the following small problem

```
inputs = np.array([[1, 0, 1],
                   [0, 1, 1],
                   [0, 0, 1],
                   [1, 1, 1]])   # N = 3, M = 4.

goals = np.array([[1, 1, 0, 0]])

hidden_size = 4   # P = 4

weights_01 = ....   # random [N × P]

weights_12 = ....   # random [P × 1]

for iteration in range(40):
    layer_2_error = 0
```

```
layer2_error = 0
for i in range(len(inputs)):
    layer0 = inputs[i:i+1]
    layer1 = np.dot(layer0, weights_01)
    layer2 = np.dot(layer1, weights_12)

    layer2_error += np.sum(
        (goals[i:i+1] - layer2)**2)

    layer2_delta = (layer2 - goals[i:i+1])
                                    **2)
    # Backpropagation
    layer1_delta = layer2_delta.dot
                    (weights_12.T)
    weights_12 -= alpha*layer1.T.
                    dot(layer2_delta)
    weights_01 -= alpha*layer0.T.
                    dot(layer1_delta)
    if (iteration % 10 == 9):
        print("Error:" + str(layer2_error))
```

Let's start experiments - the following results are obtained

Error: 1.50447
Error: 1.41939
Error: 1.39579
Error: 1.38190

The iterative process is not converging.

Thus three-layer NN gives no improvement in comparison with two-layer NN.

Some new ideas should be added to the given NN.