1 Supervised & unsupervised learning

1.1 Non-linear actiavtion function

Bias is activation function x-Offset **Slope** of activation function is rarely used.

An example sigmoid with **bias** and **slope**.

$$v = \frac{1}{1 + e^{-S(z-b)}}$$

where b is **bias** and S the **slope**. **Bias** can be used as a weight.

$$\begin{aligned} z &= w_1u_1 + w_2u_2 + \ldots + w_nu_n \\ &\rightarrow (z - b) = w_1u_1 + w_2u_2 + \ldots + w_nu_n - b \\ &\rightarrow (z - b) = w_1u_1 + w_2u_2 + \ldots + w_nu_n - (b \cdot -1) \end{aligned}$$

$\rightarrow (b \cdot -1)$ splits to w_{n+1} and u_{n+1}

This results in an additional weighted bias shifting the activation function resulting in

$$z = \sum_{i=1}^{n+1} \omega_i \cdot u_i$$

Each perceptron can implement one **deciscion boundary**. **Deciscion boundaries** seperate inputs into different classes. The boundary can be shifted by adapting the weights.

By adding more perceptrons the **deciscion boundaries** dimension increases. The boundry of 2 neurons results in one-dimensions. 3 Neurons create a 2-Dimensional **deciscion boundary**. More Neurons build more complex spaces.

1.2 Designing a network

- Defined number of inputs
- Defined number of outputs
- Variable hidden layers

Hidden layer depends on linearity of the problem. No general solution to amount of hidden layers. Strategy of trial and error, start with ≈ 100 layers.

Deep Neural Networks: Depth is defined horizontally.

1.3 Convolutional neural networks

Hereby:

- u: Input
- v Output
- x Hidden layer
- w Weight from u to x
- y Weight from x to v

One **Iteration** consists of one forward pass and one backwards pass. One **Epoch** consists of **Iterations** for all Items in the training set.

1.3.1 Forward propagation

1. Set input
2. Calculate for all hidden layers
$$x_j = \sum_i u_i w_{ji}$$
3. Calculate for all output layers
$$v_k = \sum_j x_j w_{kj}$$

1.3.2 Backpropagation

1. Calculate error gradient for all output neurons
$$E_k^0 = v_k(1 - v_k)(t_k - v_k)$$
2. Calculate error gradient for hidden layers
$$E_j^h = x_j(1 - x_j) \sum_k E_K^0 y_{kj}$$
3. Update weights for outputs
$$y\prime_{kj} = y_{kj} + \mu E_k^0 x_j$$
4. Update weights for hidden layers
$$w\prime_{ji} = w_{ji} + \mu E_j^h u_i$$

1.4 Vanishing Gradients

With a great amount of layers the impact of early neurons (close to the input) have less effect on the output error and get changed less resulting in less learning. A high amount of layers does not guarantee better network performance.

Dropout randomly disables neurons and stops updating their weights. This does not guarantee better accuracy only better execution speed.

This only occurs by learning with backpropagation.

Alternative: **NEAT** (*Neuroevolution of augmenting topologies*) using generative algorithms. Possibly (not guaranteed) better performance to optimize output by chaning the entire networks structure. Worst execution speed and memory performance.

1.5 Trainig proceedure

Split trainig dataset into two parts to avoid overfitting. Suggested split:

- 70% training data
- 30% testing data

Initilize weights to random values.

- **Training Datasset**: Adjust weights/learn
- **Testing Dataset**: Testing final solution
- **Validation Dataset**: Minimize overfitting

Always randomize oder of data for every **epoch**, as netoworks easily learn patterns.

More complex splitting algorithms and procesdures:

- **Monte Carlo corss validation** subsamples data randomly into its sets.
- **K-fold corss validataion** divides data into k subsets, trainig it and removing it after training to repeat with the remaining $k-1$ subsets
- **Leave-p-out cross validation** p datasamples, use $n-p$ for training, but test and train $\frac{n!}{p!\cdot(n-p)!}$ times. This presents every datapoint equally often and fairly.