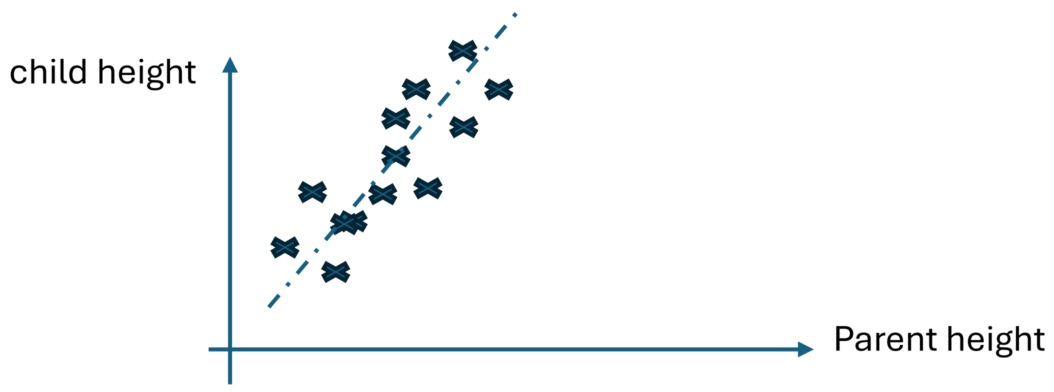
Machine Learning for Robotics: **Regression**

Prof. Navid Dadkhah Tehrani

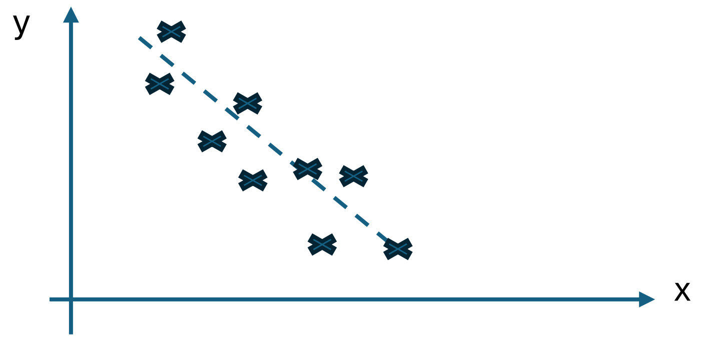
The word regression originates from the work of Sir Francis Galton in 1886.
when he was trying to study relationship between children height and parent height.

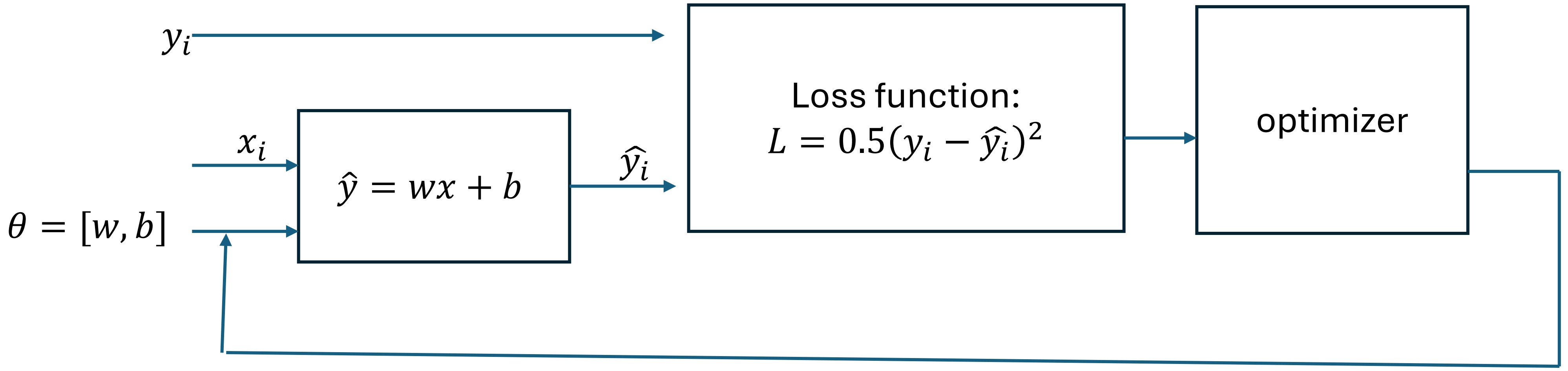


Analyzing the data, he observed that tall parents on average have less tall children.
And short parents, on average have less short children.
In order words, the children height tend to **regress toward the mean**.

Nowadays, in statistic and machine learning, the word regression is used every time we’re trying to estimate a numerical value from an input data.

Let consider a simple case with a linear model.





The easiest optimizer is based on the gradient descent rule:

$$b_i = b_{i-1} - \alpha \frac{\partial L}{\partial b} = \mathrm{b}_{i-1} + \alpha(y_i - \widehat{y_i})$$
$$w_i = w_{i-1} - \alpha \frac{\partial L}{\partial w} = w_{i-1} + \alpha x_i(y_i - \widehat{y_i})$$

If we process each data point individually like above, it's called stochastic gradient descent. Stochasticity comes from picking a random pair (x_i, y_i) in each iteration.

```
• Initialize $w$ and b
• For M training epoch
    • Shuffle the data.
    • For all the data
        • pick a $(x_i, y_i)$
        • calculate gradient w.r.t $w$ , b
        • calculate the update rule
        • update $w$ , $b$
```

Full Batch mode (batch gradient descent):

Updates to the parameters are applied at the end of going through all the data

The loss is also defined as average loss over the entire batch:

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

- For number of epochs
    - predictions = model(data)
    - loss = loss_function(predictions, ground_truth)
    - gradients = compute_gradients(model, loss)
    - update_parameters(model, gradients)

Mini-batch Mode

This is what used in practice in deep learning.

```
• For number of epochs
    • Shuffle the data
    • For all the batches
        • Get the batch_data :
        • predictions = model(batch_data)
        • loss = loss_function(predictions, ground_truth_batch)
        • gradients = compute_gradients(model, loss)
        • update_parameters(model, gradients)
```

(after each epoch the data is randomly shuffled, therefore in each batches are different in each epoch.)
If we don’t shuffle the data after each batch, the network forgets about the old data and start memorizing the new data. So shuffling in each epoch helps the network to be robust to the order of training data.

- B can be anything between 1 to total number of data. In practice we use batch size of 32, 64 for image data.
- Note that since all the data is loaded to GPU, maximum batch size you can use is function of your hardware.

Regression update rule for one layer Perceptron:

$$y = \sigma(xW + b)$$

The only thing that changes is the calculation of gradient of the $W \ and \ b$ w.r.t the loss function.

We need to review differential calculus.

Review of differential Calculus

	Function	Derivative
Sum Rule	$f(x) + g(x)$	$f'(x) + g'(x)$
Difference Rule	$f(x) - g(x)$	$f'(x) - g'(x)$
Product Rule	$f(x)g(x)$	$f'(x)g(x) + f(x)g'(x)$
Quotient Rule	$f(x)/g(x)$	$[g(x)f'(x) - f(x)g'(x)]/[g(x)]^2$
Reciprocal Rule	$1/f(x)$	$-[f'(x)]/[f(x)]^2$
Chain Rule	$f(g(x))$	$f'(g(x))g'(x)$

Gradient of multi-variable functions: $f(x, y, z)$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}$$

$$f\left(g(x), h(x)\right)$$

Derivative of the outerpart

Derivative of the innerpart

$$\frac{d}{dx}\left[f(g(x), h(x))\right] = \frac{\partial f}{\partial g}\cdot\frac{dg}{dx} + \frac{\partial f}{\partial h}\cdot\frac{dh}{dx} = \begin{bmatrix}\frac{\partial f}{\partial g}\\ \frac{\partial f}{\partial h}\end{bmatrix}\cdot\begin{bmatrix}\frac{\partial g}{\partial x}\\ \frac{\partial h}{\partial x}\end{bmatrix}$$

$$\mathbf{f}(x_1, x_2, \ldots, x_m) = \begin{bmatrix} f_1\left(x_1, x_2, x_3, \cdots x_m\right)\\ f_2\left(x_1, x_2, x_3, \cdots x_m\right)\\ f_3\left(x_1, x_2, x_3, \cdots x_m\right)\\ \vdots\\ f_m\left(x_1, x_2, x_3, \cdots x_m\right)\end{bmatrix}$$

Jacobina matrix

$$J\left(x_1, x_2, x_3, \cdots x_m\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots & \frac{\partial f_1}{\partial x_m}\\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_2}{\partial x_m}\\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \cdots & \frac{\partial f_3}{\partial x_m}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial x_3} & \cdots & \frac{\partial f_m}{\partial x_m}\end{bmatrix}$$

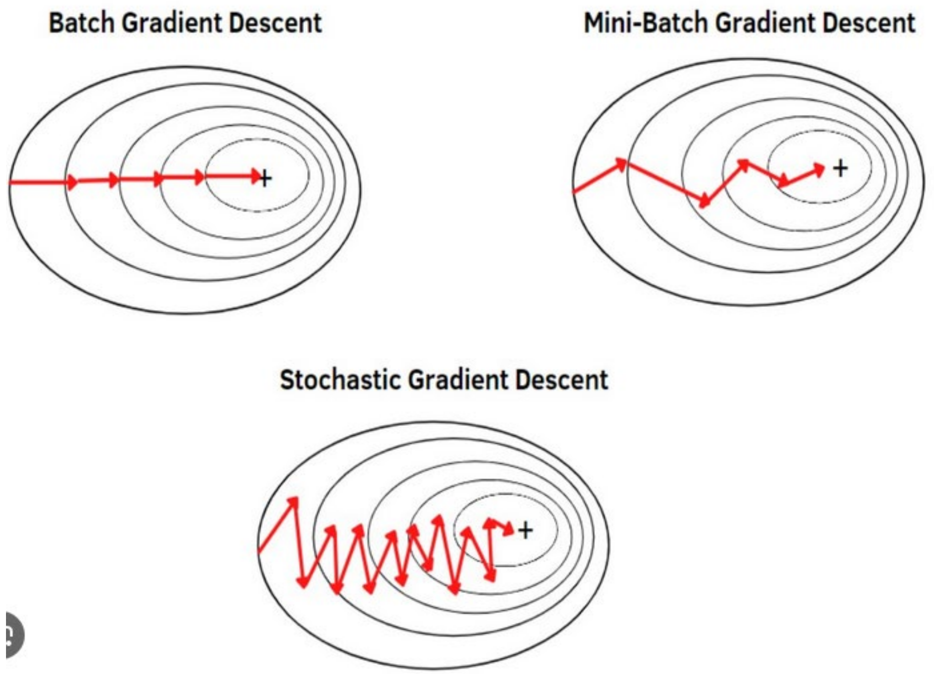
Derivation of perceptron loss derivative

Let’s ignore bias for now. The loss is called MSE loss (mean square error)

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$

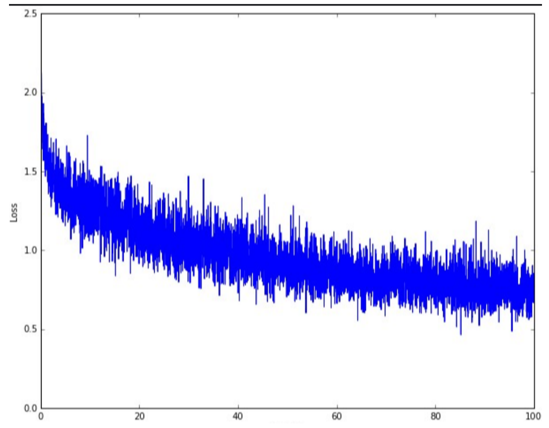
$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\
&= \frac{\partial}{\partial w_j} \sum_i \frac{1}{2n} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]})^2 \\
&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}^{[i]} \\
&= \frac{1}{n} \sum_i (\sigma(\mathbf{w}^T \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^T \mathbf{x}^{[i]})} x_j^{[i]}
\end{aligned}$$

Geometrical Interpretation



Stochastic gradient descend and mini-batch gradient descent updates are noisy,
because each minibatch approximates
the overall loss on the training set.

As a result, the loss over time looks something like this:

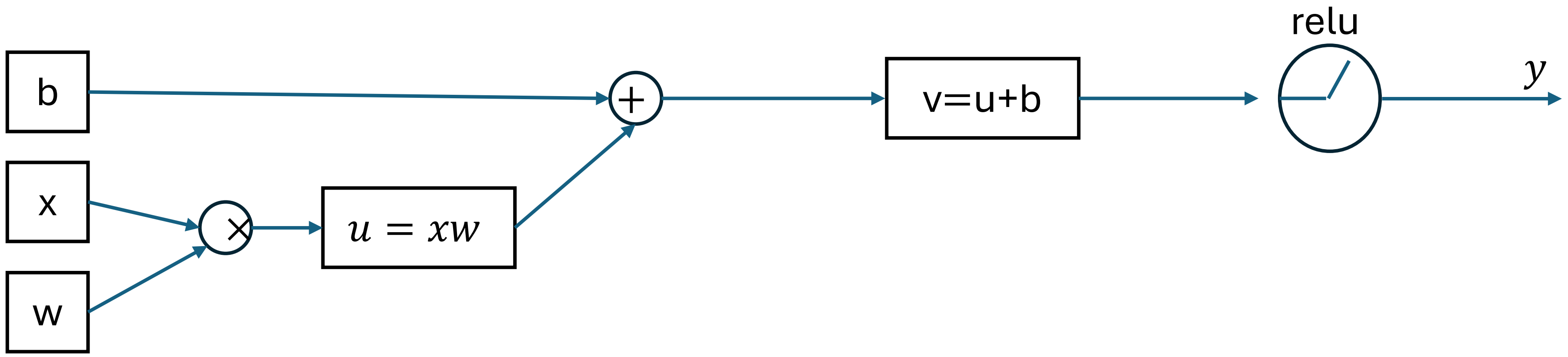


Automatic differentiation via Computation Graph

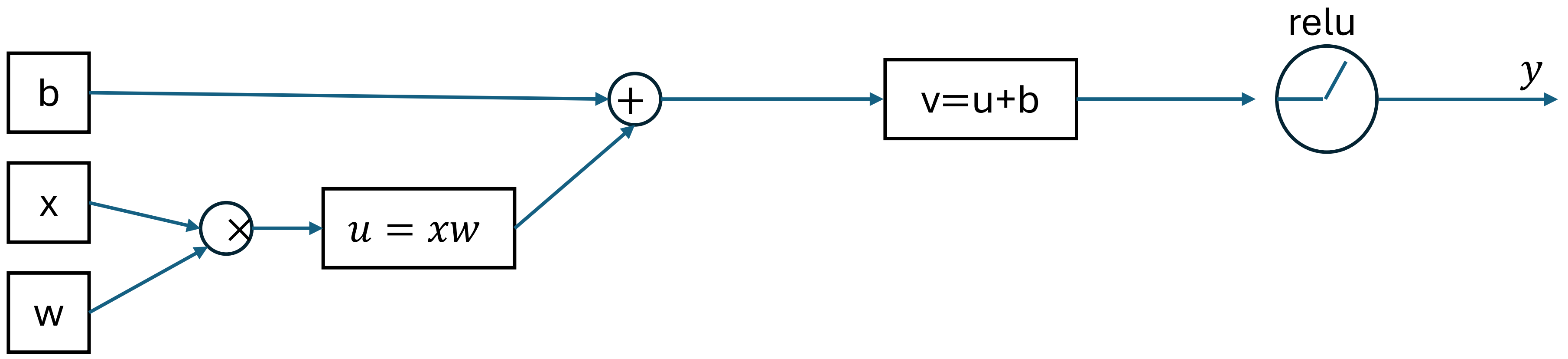
As you can imagine, computation of the gradient of the loss function w.r.t parameters of the NN gets complicated when the NN have many hidden layers.

Pytorch and other deep learning training tools, use computation graph to to calculate the gradient of the loss w.r.t the parameters.

Neural networks are computation graphs:



We need $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ so we can update w and b via gradient descent

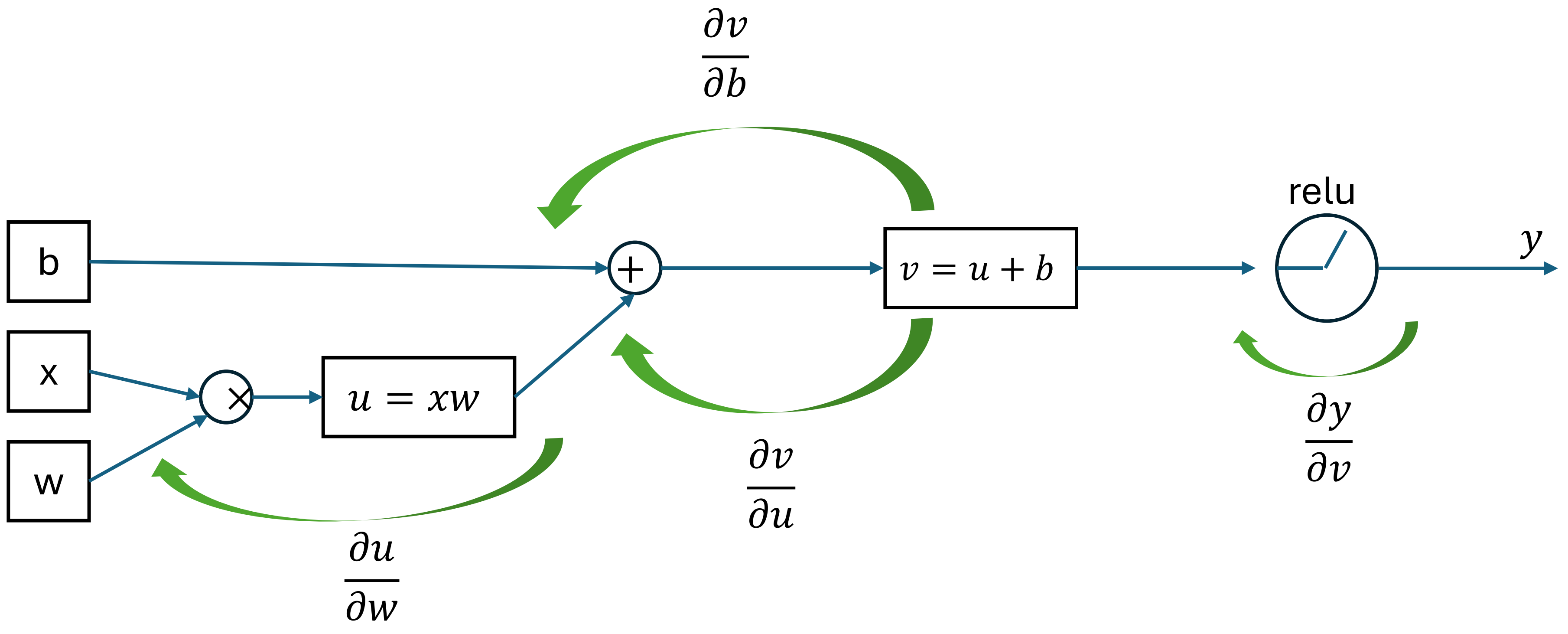


$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial w}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b}$$

$\frac{\partial L}{\partial y}$ can be calculated via the definition of the loss function. For example, for MSE loss, $L = (y - y_{truth})^2$

So the most important part is to calculate $\frac{\partial y}{\partial w}$ and $\frac{\partial y}{\partial b}$

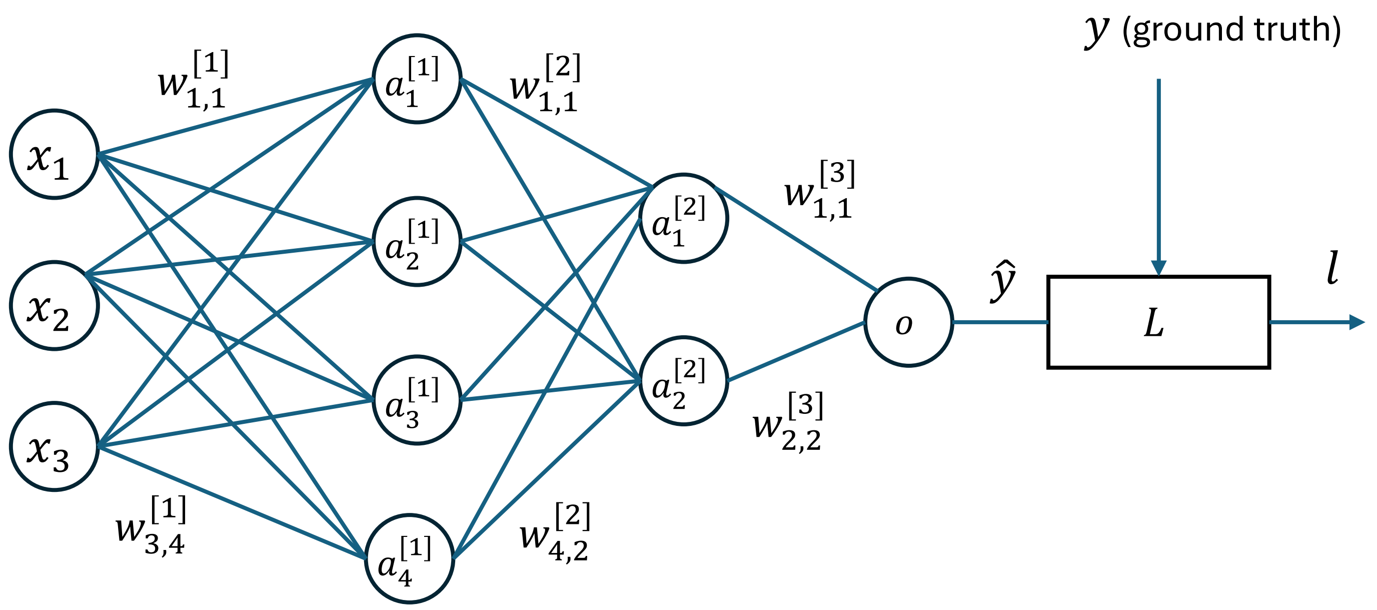

$\frac{\partial v}{\partial b}$
relu
b
$+$
$v = u + b$
y
x
$\times$
$u = xw$
w
$\frac{\partial y}{\partial v}$
$\frac{\partial v}{\partial u}$
$\frac{\partial u}{\partial w}$

To calculate $\frac{\partial y}{\partial w}$ and $\frac{\partial y}{\partial b}$ we multiply the backward paths that leads to the parameter.

$$\frac{\partial y}{\partial w} = \frac{\partial y}{\partial v} * \frac{\partial v}{\partial u} * \frac{\partial u}{\partial w}$$

$$\frac{\partial y}{\partial b} = \frac{\partial y}{\partial v} * \frac{\partial v}{\partial b}$$

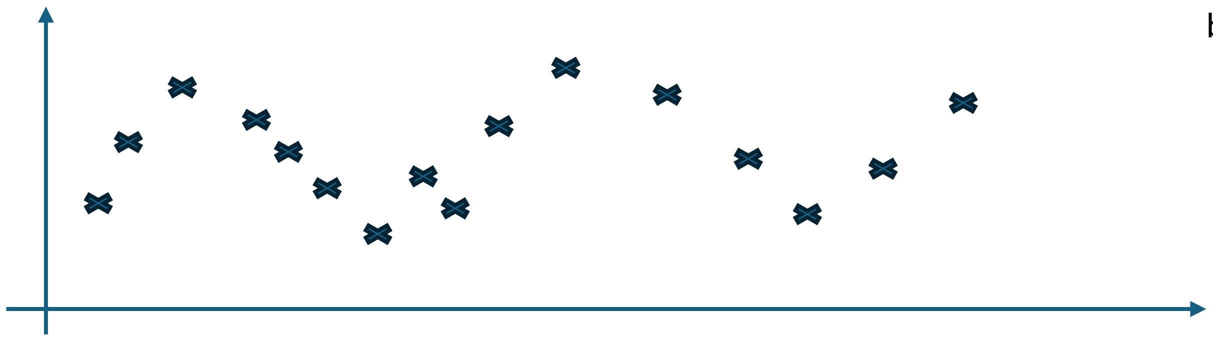
Computation graph for fully connected NN

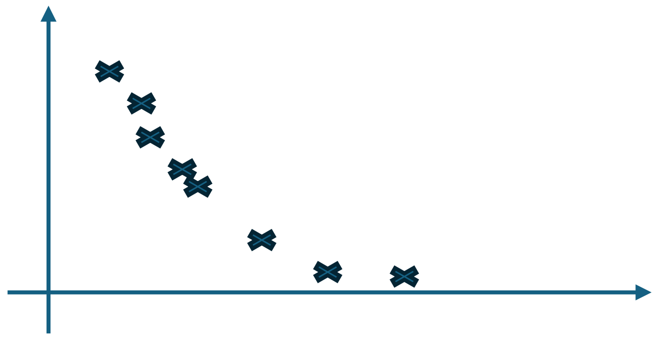


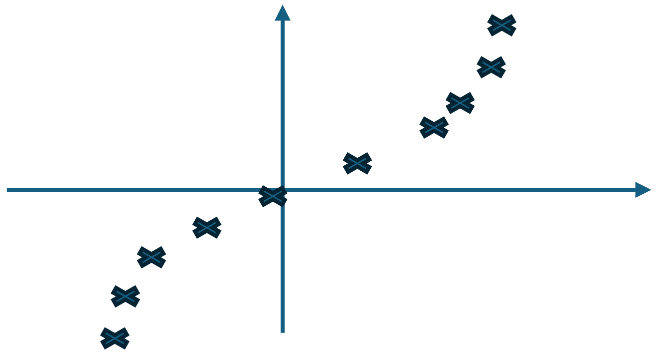
$$\frac{\partial l}{\partial w_{1,1}^{[1]}} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial a_1^{[2]}} * \frac{\partial a_1^{[2]}}{\partial a_1^{[1]}} * \frac{\partial a_1^{[1]}}{\partial w_{1,1}^{[1]}} + \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial a_2^{[2]}} * \frac{\partial a_2^{[2]}}{\partial a_1^{[1]}} * \frac{\partial a_2^{[1]}}{\partial w_{1,1}^{[1]}}$$

Neural networks are universal approximators

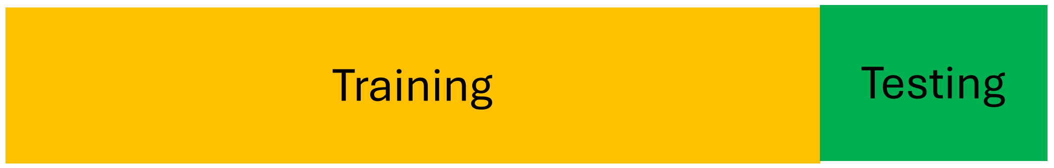
Originated from research in 80s: given enough nodes and the right set of weights, NN can model any function.


best fit with sinusoidal functions

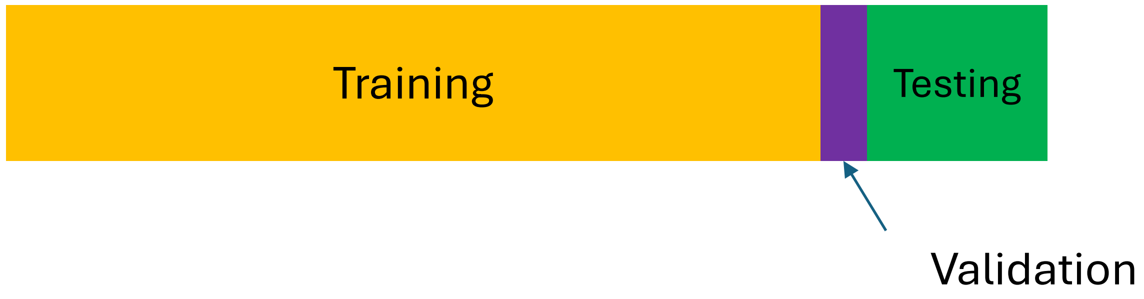

best fit with exponential functions


best fit with polynomial functions

Training set, Test set, Validation set



There's another split that often used is called validation set that is used for **tuning** the model:


Training
Testing
Validation