Machine Learning

Neural Networks and Deep Learning

Dr Guillermo Hamity

University of Edinburgh

September 27, 2021

European Research Council

Established by the European Commission

Today's outline

1. Neural networks
2. Deep learning
3. Neural network's training
4. Hyperparameter optimisation

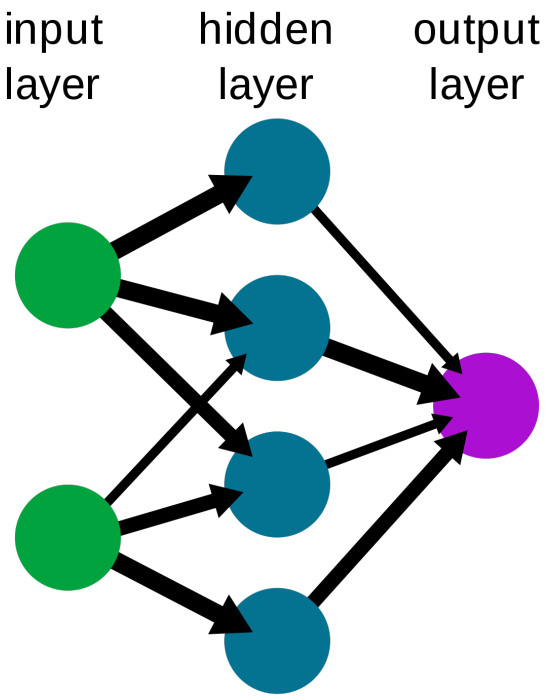
Artificial Neural networks (ANN)

Neural networks

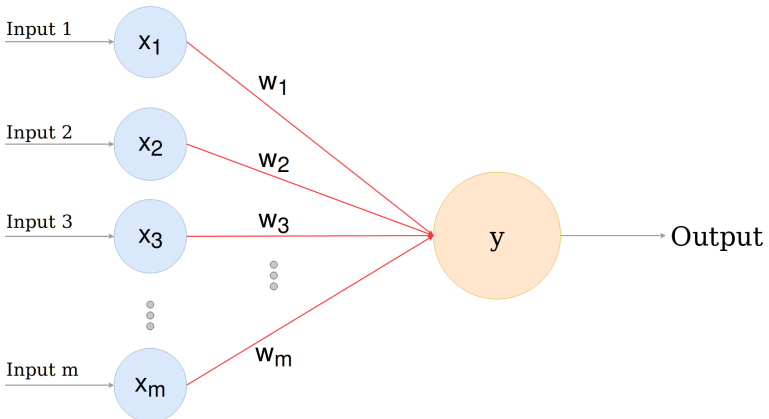
Artificial neural networks (ANNs) are a set of bio-inspired algorithms.

- They are inspired by biological brains:
  - Consist of simple units (**neurons**) connected to each other.
  - They receive, process, and transmit a signal to other neurons, acting like a switch.
- The elements of a neural network are quite simple on their own;
- The complexity and the power of these systems come from the interaction between the elements.

A simple neural network



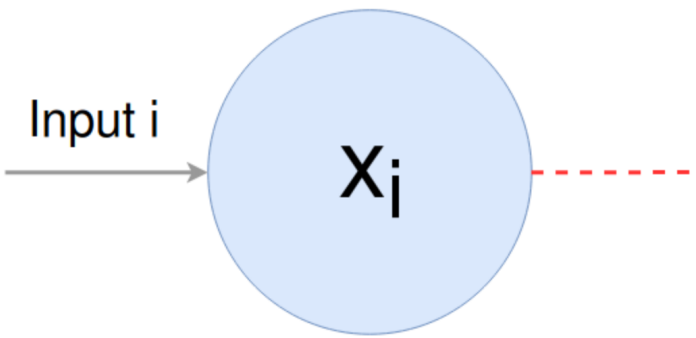
A single layer Neural Network



- **Input layer:** the layer through which we input the features
- **Hidden layer:** the layer which consists of neurons
- **Output layer:** the layer which outputs the predicted value

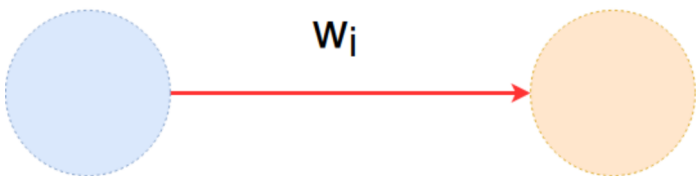
Let's have a look at each layer...

Input nodes



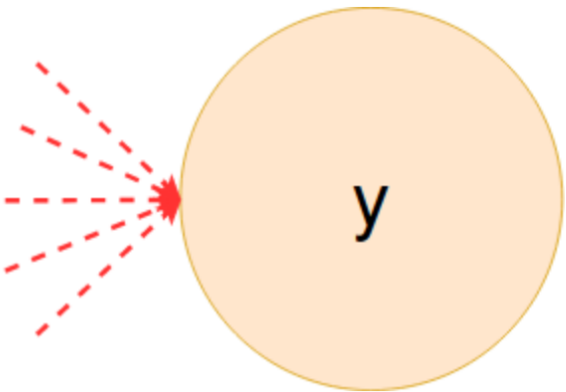
- The blue nodes are the **input nodes**.
- Each such node represents a **feature** of the input problem.
- In the example in the previous slide, we have **m** input features.
- The i-th input feature is denoted by $\{x_i\}$.
- This layer is called the **input layer**.

Weights



- The red arrows connect the input nodes to the orange node.
- These are called the synapses.
- Each one of these synapses has a **weight** attached to it, which is denoted by w_i
  - i.e. the i-th synapse has the weight w_i.
- The weights compose a **layer**.

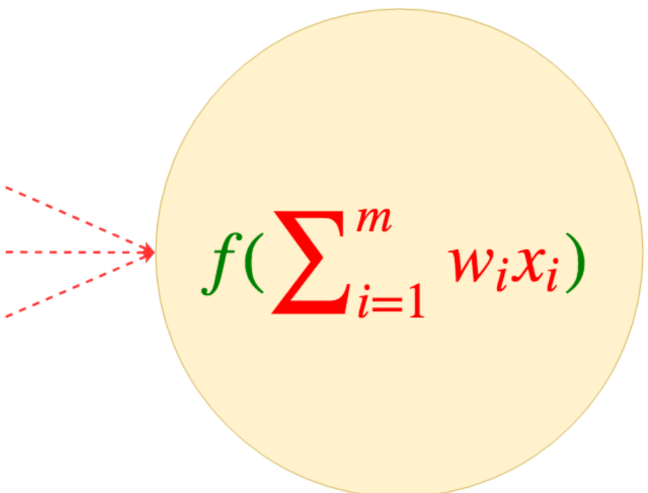
Output node



- The orange node, y, is the **output node**.
- It calculates a score based on the inputs and the weights
  - We introduced the intercept b known as the *bias*
  - The bias is a trainable parameter (will shift the **activation function**)

$$y = \boldsymbol{w} \cdot \boldsymbol{x} + b = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} + b = w_1 x_1 + w_2 x_2 + ... w_m x_m + b$$

Prediction



- Since we have we

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = y \in \mathbb{R}$$

(may) need to change the y definition, depending on the problem

- To make a **prediction** from this calculated score, we have to use an activation function.
- We should choose an activation function σ which maps to our desired output

$$y = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \sigma(b + \sum_{i=1}^{m} w_i x_i)$$

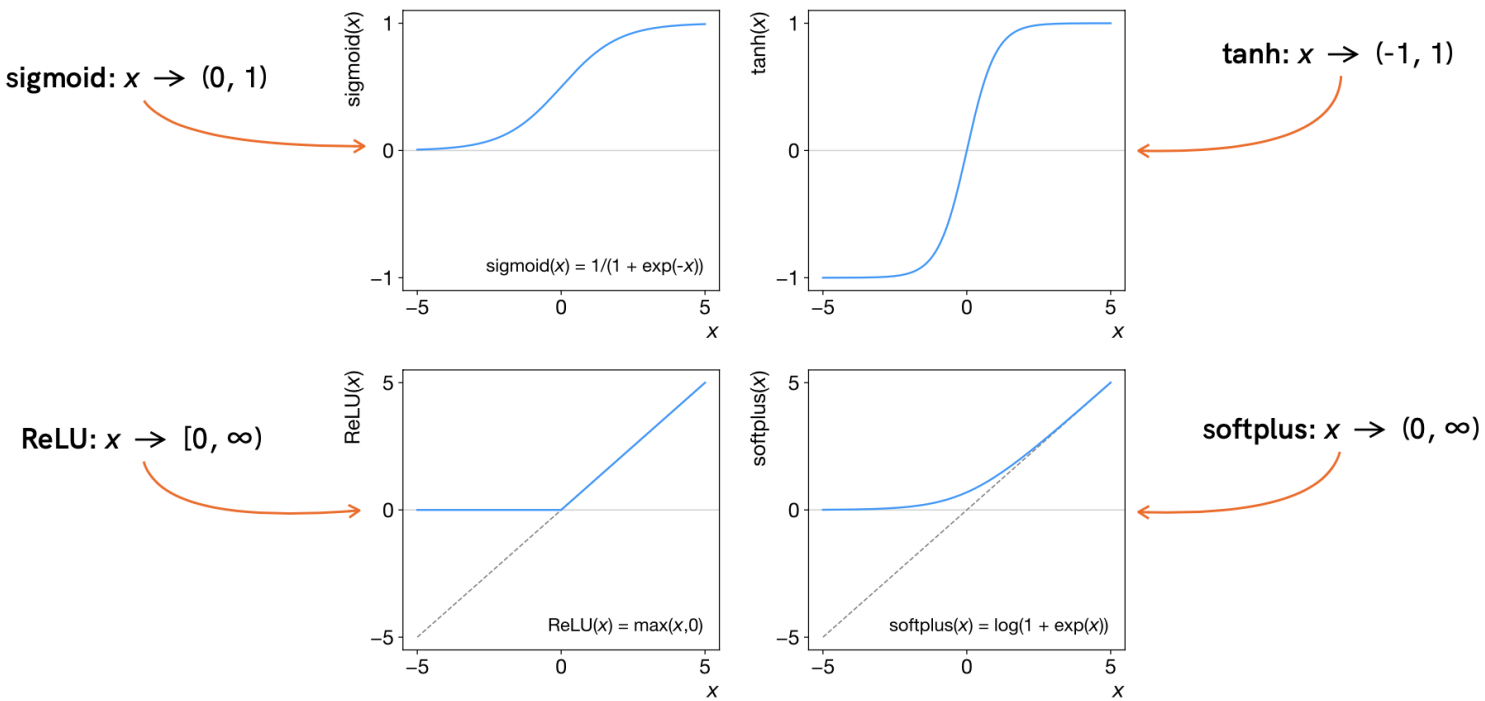
1

[1] The figure uses f for the activation function, but I prefer σ

Activation functions

The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

- There are many kinds of activation functions.
- The most used are: **Sigmoid activation, ReLU, Softmax and tanh**

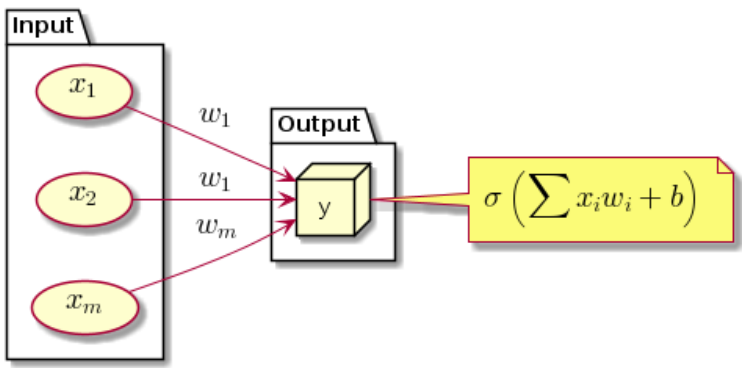


Multidimensional output

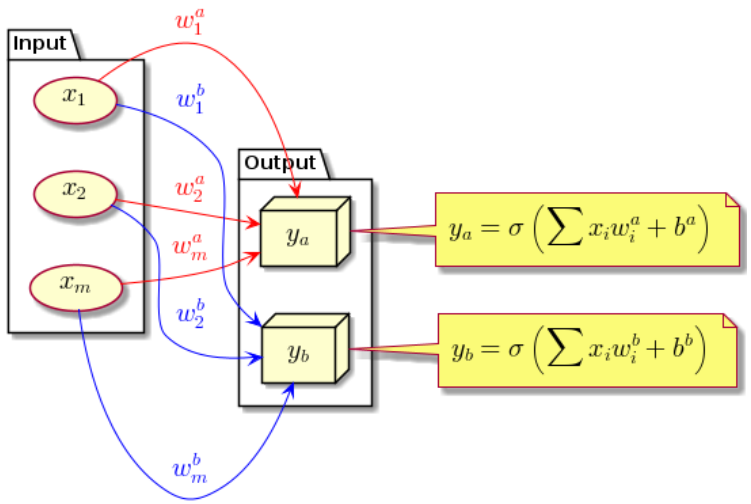
Can define more than one output, then activation acts as a vector function

- Regressing multiple numbers: x,y coordinates
- Multiclass classification: One hot encoding (more on this later)

One output node



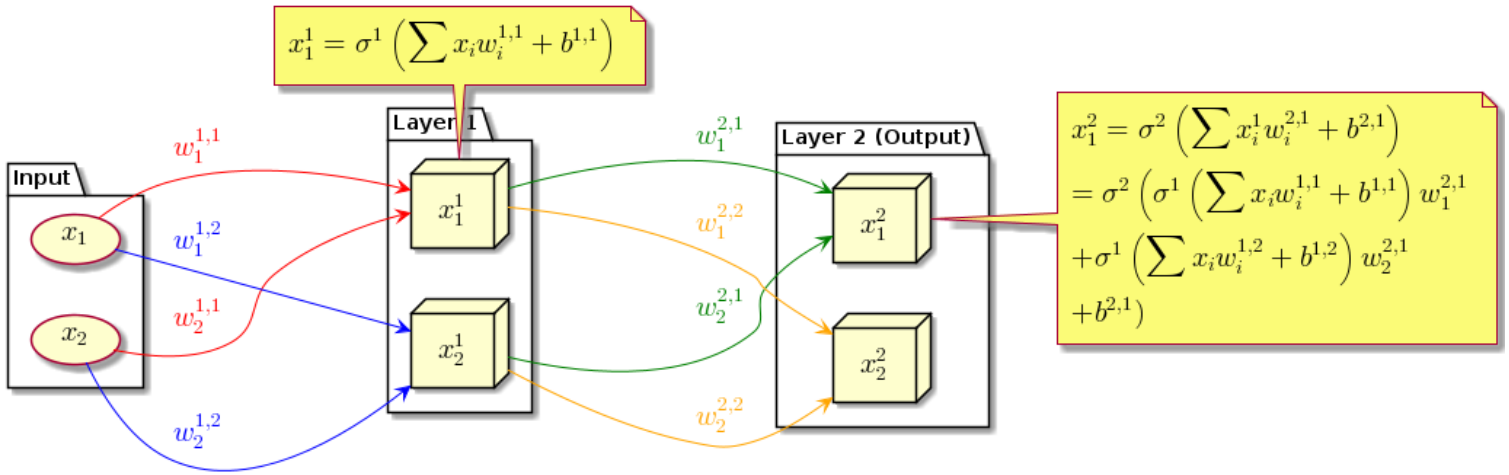
Two output nodes



Note that each output node has its own bias, (b^a or b^b) and own set of weights.

Hidden layers

In order to build more complex models, we introduce **hidden layers**

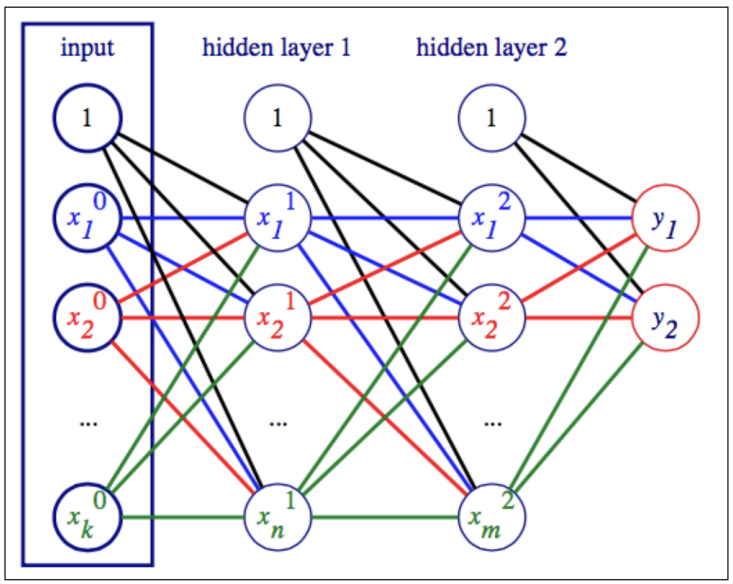


Spoiler Alert

We wrapped layer 1 node in **activation functions**. This is to avoid linear structure. More in deep learning section.

Neural networks: Hidden layers

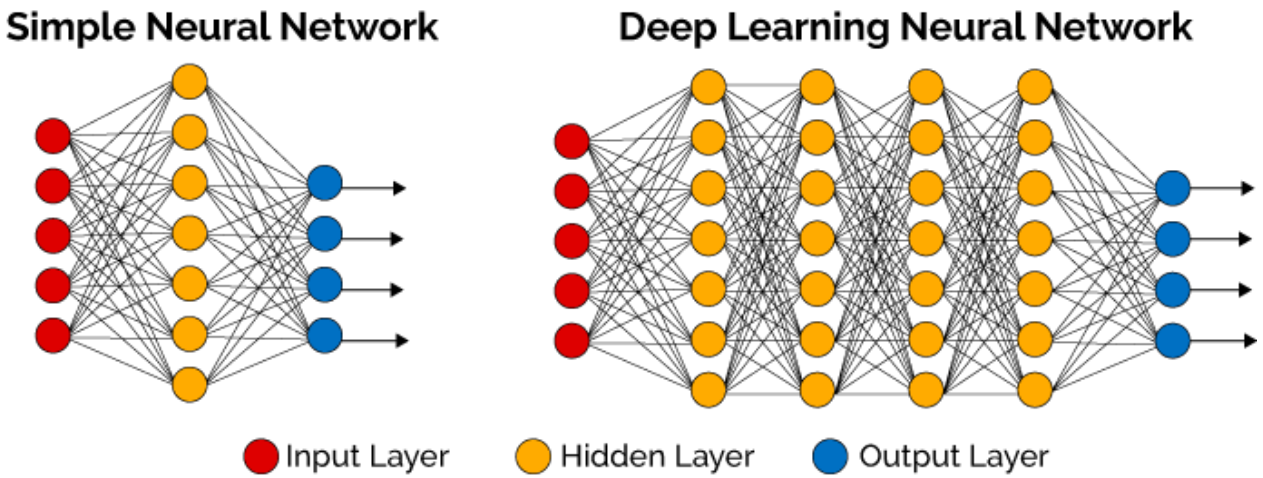
- The scheme introduced in the previous slides represents a Neural Network that has only 1 **hidden layer**:
  - A Neural network with just 1 hidden layer is also called **shallow** Neural Network
- More layers can be introduced between the input and the output



Deep Neural networks (DNN)

Deep Neural Networks

Deep Neural Networks are Neural Networks with at least 2 or more hidden layers.

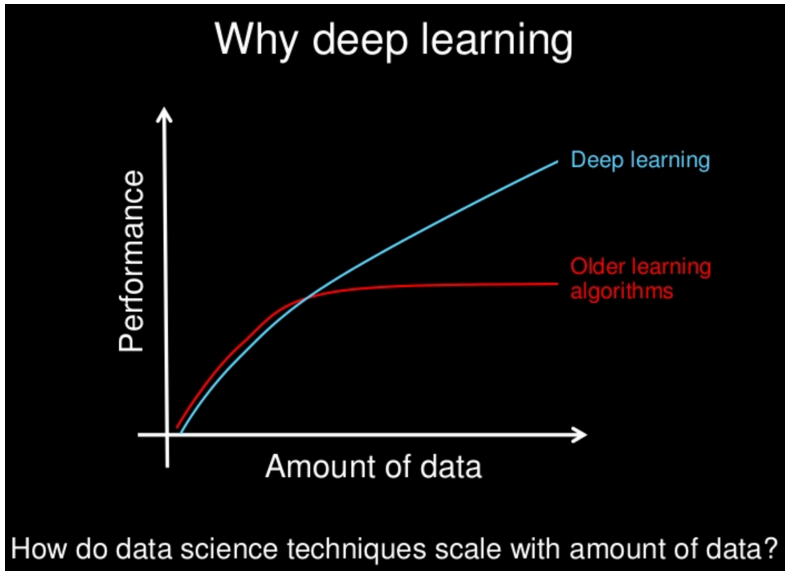


A Neural network with a single hidden layer and infinite hidden nodes is a **Universal Approximator**, i.e. can (in principle) approximate any continuous function.

Why do we need Deep Neural Networks?

Why Deep Neural Networks

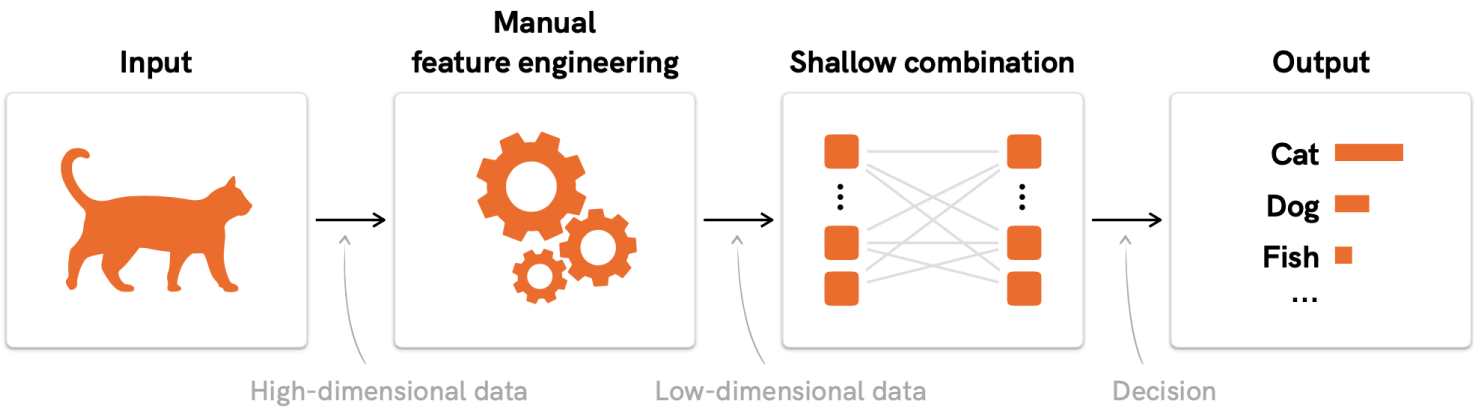
- Deep of deep learning refers not just to how many layers the neural net has, but to the level of "learning":
  - The network does not simply learn to predict an output Y given an input X, but it also understands basic features of the input.
  - The neural network is able to make abstractions of the features of the input and to make predictions based on those characteristics.
- This level of abstraction is missing in other ML algorithms



Performances of Deep Learning algorithms increase with the amount of data!

Deep vs shallow learning

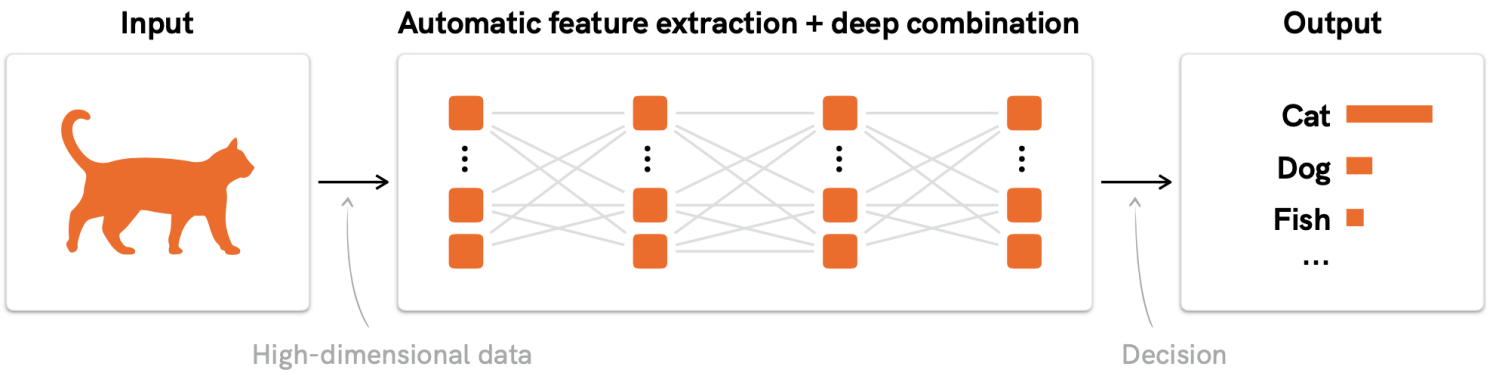
- Traditional Shallow learning



- Unable to exploit high-dimensional data
- Manual feature engineering
- Combine using simple (relatively) algorithms to get the output

Deep vs shallow learning

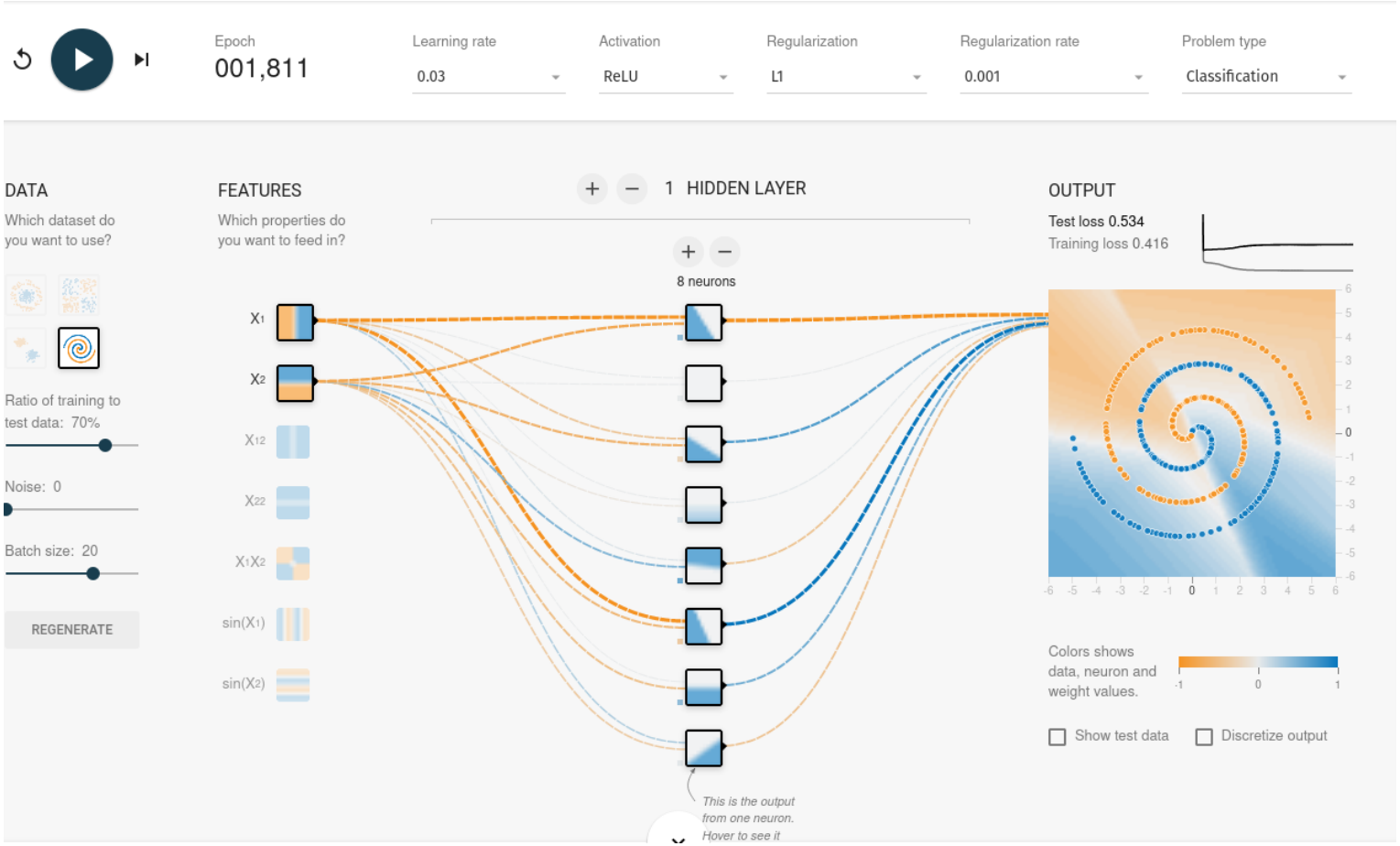
- Deep learning



- Capacity to exploit high-dimensional data
- The network is able to learn relevant features
- Perform feature extraction and deep combination simultaneously
- **Feature selection** refers to find the best subset of features
- In fact, a large number of features is not always better

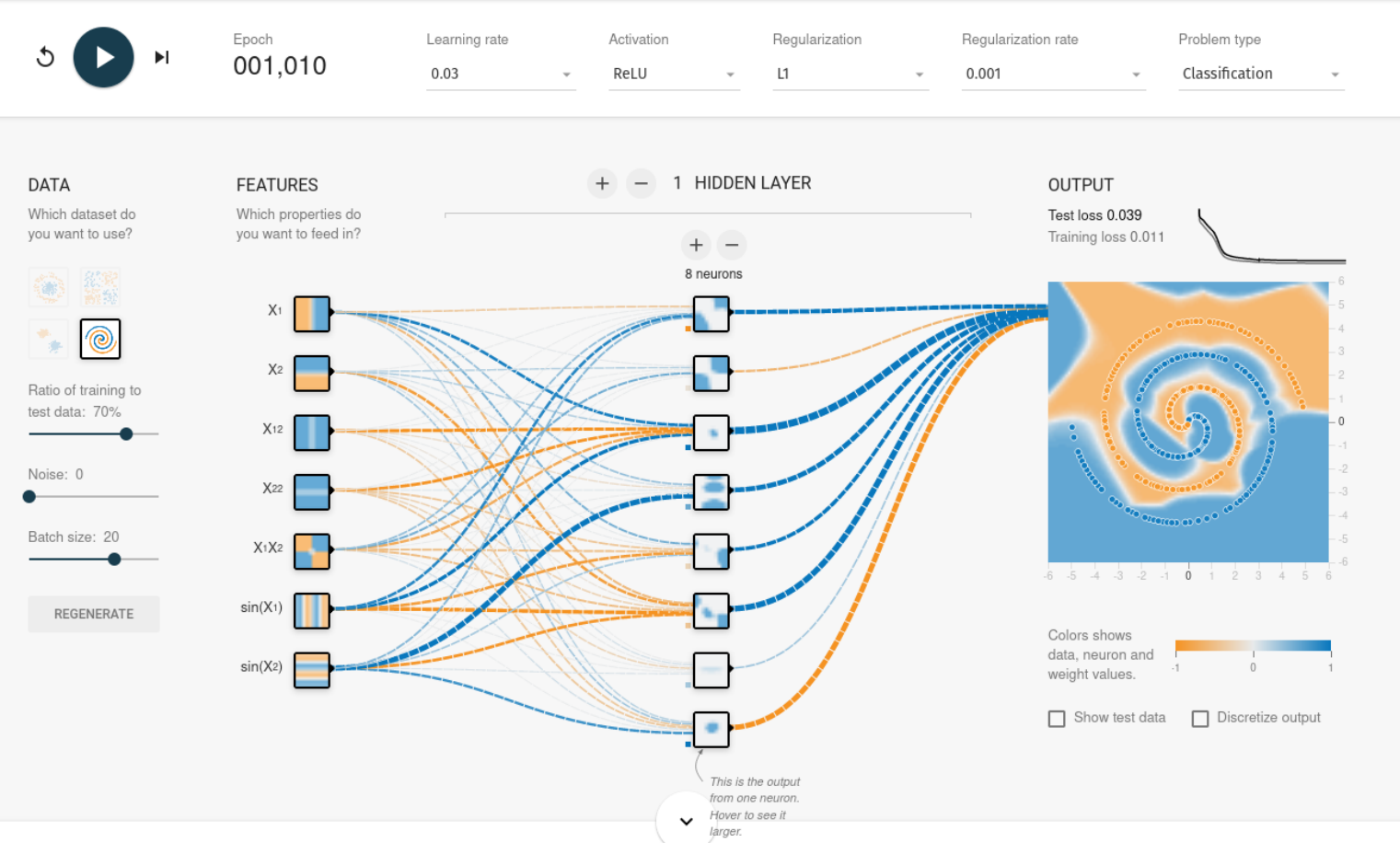
Deep vs shallow learning

- A great way to visualise the feature importance is playing with Tensorflow playground
- Consider a shallow NN on **2 input features**, after 1000s of epochs no good performance



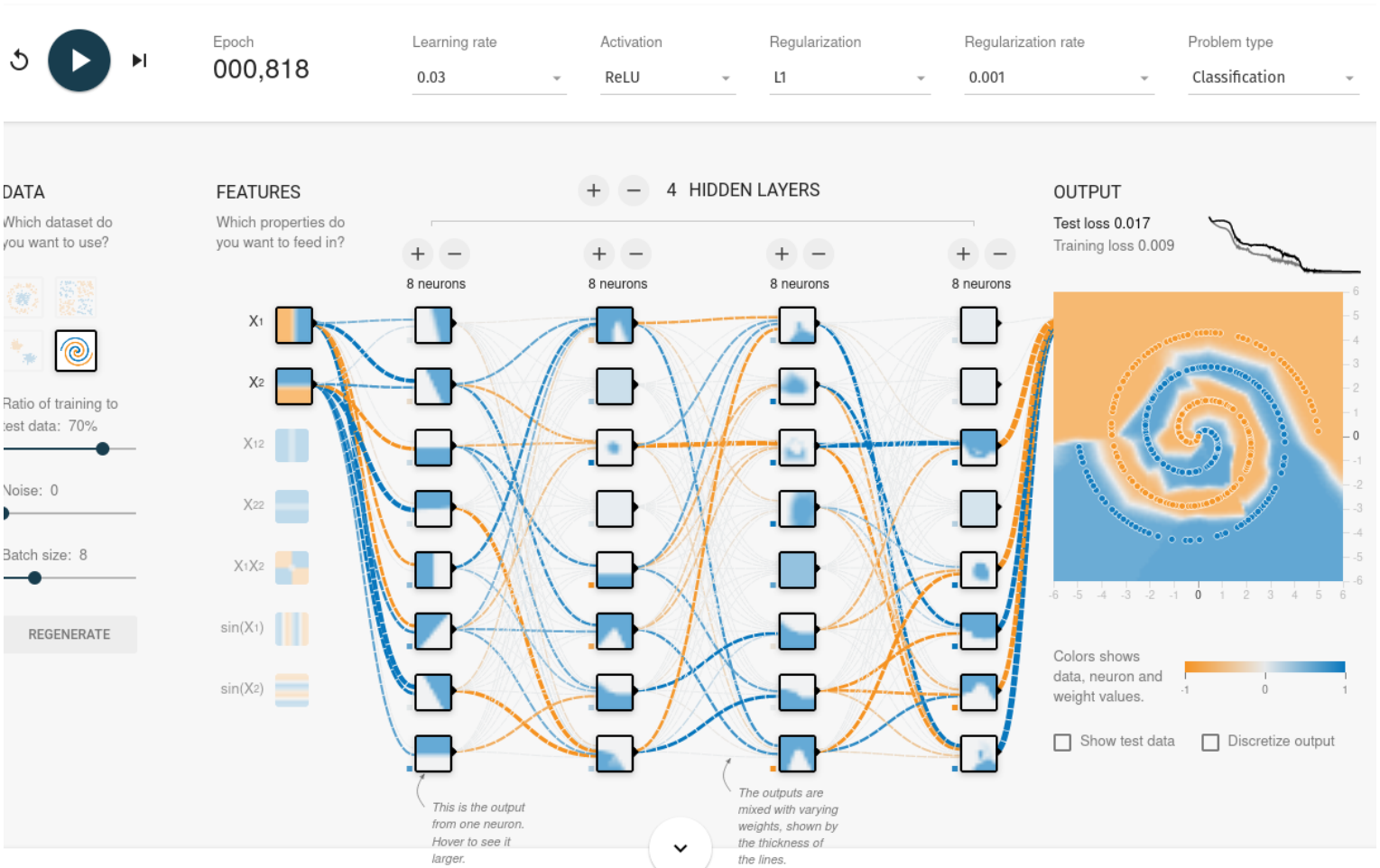
Deep vs shallow learning

- Combining features in different ways we get good performance
- But how do we know which combinations?



Deep vs shallow learning

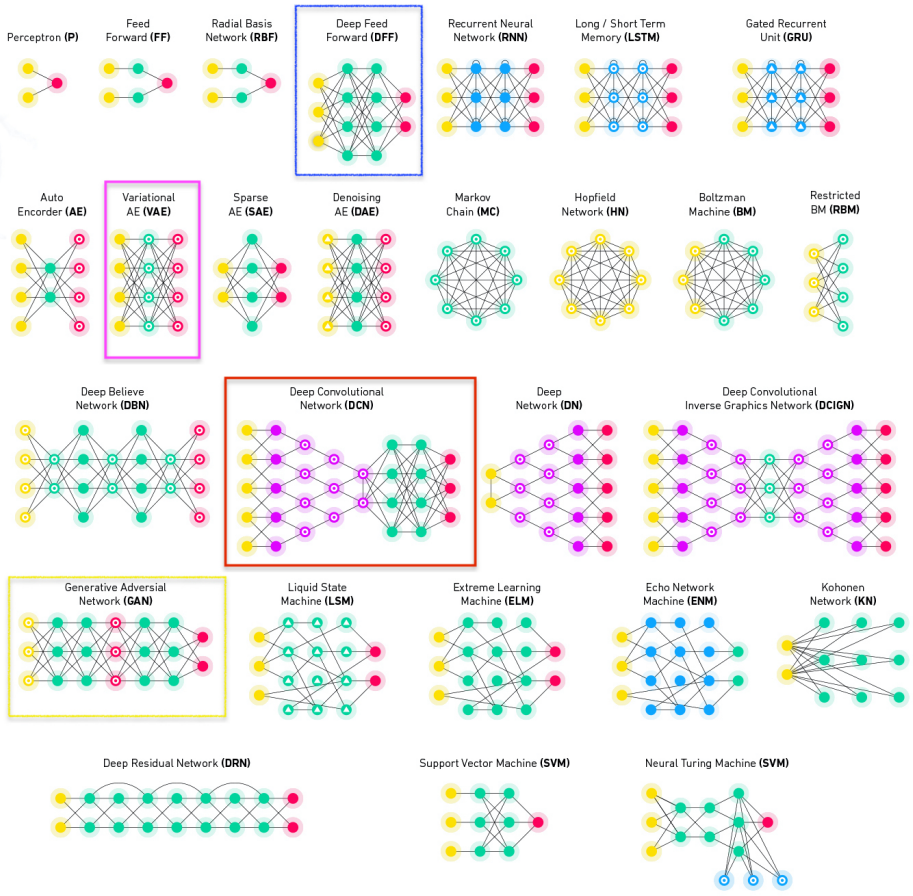
- The Deep NN finds a similar solution from the original two features



Deep Learning models

We will work with many different architectures during this course.

- Deep Feed-Forward
- Variational AutoEncoders
- Convolutional Neural Networks
- Generative Adversarial Networks



Activation functions

The activation function is a mathematical "gate" between the input feeding the current neuron and its output goint to the next layer.

- The main purpose of the activation function is to introduce **non-linearity** into the output of a neuron.

Why do we need non linearity into the neural network?

1. Hidden layers become useless if we use linear activation functions.
   - This because the composition of linear functions is itself a linear function!
2. It is not possible to use **backpropagation** (gradient descent) to train a model using linear activation functions
   - The derivative of the function is a costant and has no relation to the input X
   - This means that is not possible to go back and undestand which weights can provide a better prediction

Activation functions in hidden layers

Apply differentiable non linear activation functions allow multi layer models to learn non linear relations!

$$f(x) = W^{(2)} \cdot h(W^{(1)} \cdot x) \neq W'x$$

- sigmoid

$f(a) = \frac{1}{1+e^a}$, $f'(a) = f(a)(1 - f(a))$

- tanh

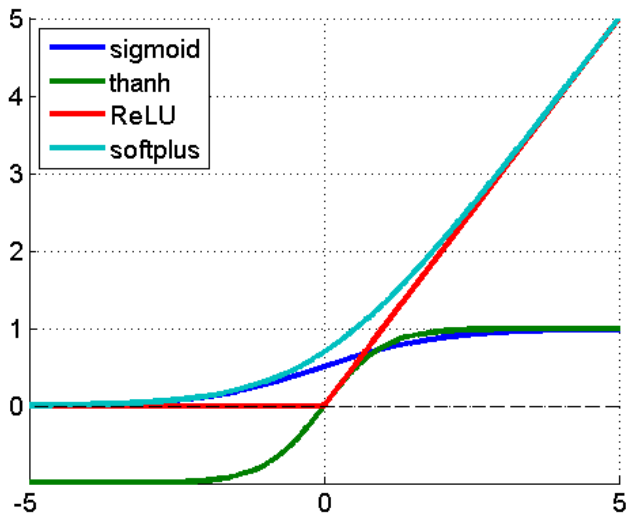
$f(a) = \tanh a$, $f'(a) = 1 - f(a)^2$

- softplus

$f(a) = \log(1 + e^a)$, $f'(a) = \frac{1}{(1+e^{-a})}$

- ReLU

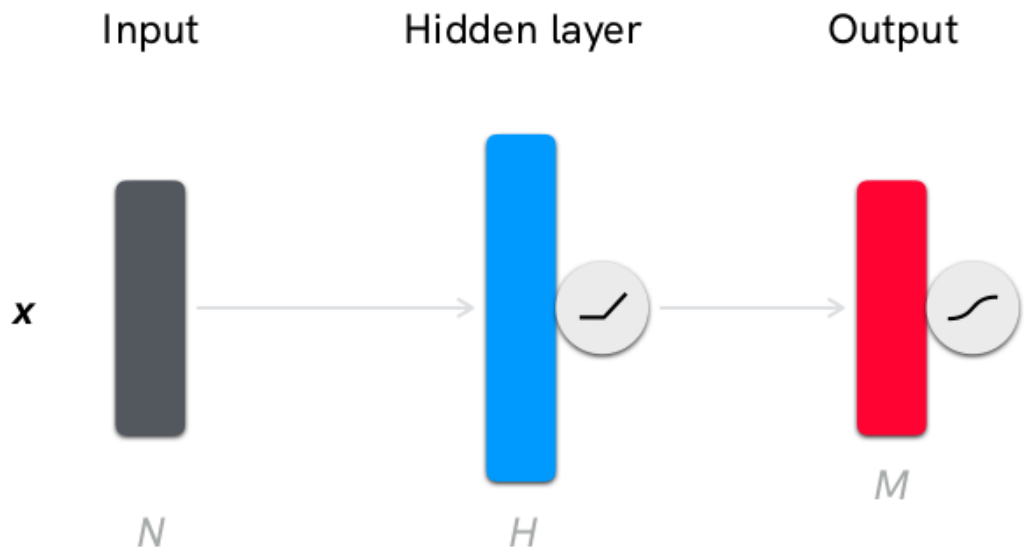
$f(a) = \max(0, a)$, $f'(a) = (0, 1)$



- `sigmoid` and `tanh` neurons can saturate, ie. have vanishing gradients, and are expensive calcualations
- `ReLU` provides fast convergence, fast to compute, wont send to saturate
  - Nodes can die if not activated, and never update. `Leaky ReLU` is a solution.

Activation functions in output layers

- As mentioned before, activation functions are suitable for specific problems:
  - Regressing non-negative quantity (e.g energy): **ReLU**
  - Binary classification probability p $\in$ (0,1): **Sigmoid**
  - Multiclass classification probability, p: **Softmax**
- Build our NN using different activations for different layers.



Neural Network's Training

Loss functions

Neural networks are trained using an optimization process that requires a **loss function** to calculate the model error.

- The choice of loss function is directly related to the activation function used in the output layer of the neural network
- The choice of the loss function has to be appropriate to the type of problem you are solving
  - **Regression problem** Mean squared error (MSE)
    $L_{MSE}(\theta) = \frac{1}{n}(y - f_\theta(x))^2$
  - **Binary classification** Binary cross-entropy (BCE)
    $L_{BCE}(\theta) = -ylogf_\theta(x) - (1-y)log(1-f_\theta(x))$
  - **Multiclass classification** Multiclass cross-entropy (MCCE)
    $L_{MCCE}(\theta) = \sum_{i=0}^{|y|-1} -y_i log f_{\theta,i}(x)$
- **Goal:** Use the loss to evaluate and diagnose how well the model is learning.

Loss function: Regression

Regression (MSE)

$$\frac{1}{n}\sum(y - h(x))^2$$

- Regression model $h(x)$, target y

```
y   = np.array([0.875, 0.426, 0.213, 0.348])
h_x = np.array([0.900, 0.526, 0.721, 0.348])
```

- Loss

```
(np.sum((y-h_x)**2))/4
```

```
0.06727252749518241
```

- sk-learn

```
from sklearn.metrics import
    mean_squared_error
mean_squared_error(y,h_x)
```

```
0.06727252749518241
```

Loss function: Binary Classification

Classification (Binary cross-entropy, aka. Log Loss)

$$-\frac{1}{n}\sum y_i \log\left(p(y_i)\right) + (1-y_i)\log\left(1-p(y_i)\right)$$

Probability from sigmoid activation

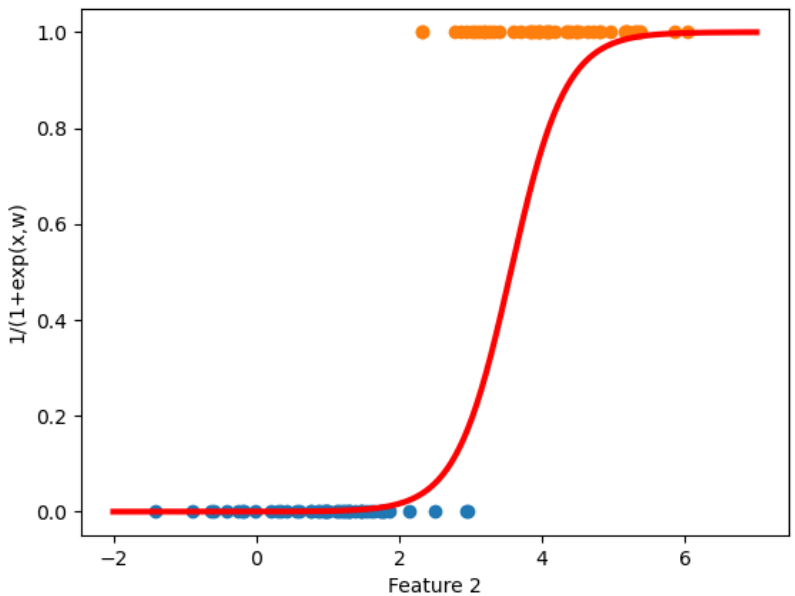
- Classification probability $p(x)$, target $y \in (0, 1)$

```
y = np.array([0  ,1  ,1  ,1  ,0])
p = np.array([0.1,0.7,0.5,0.8,0.2])
```

- sk-learn

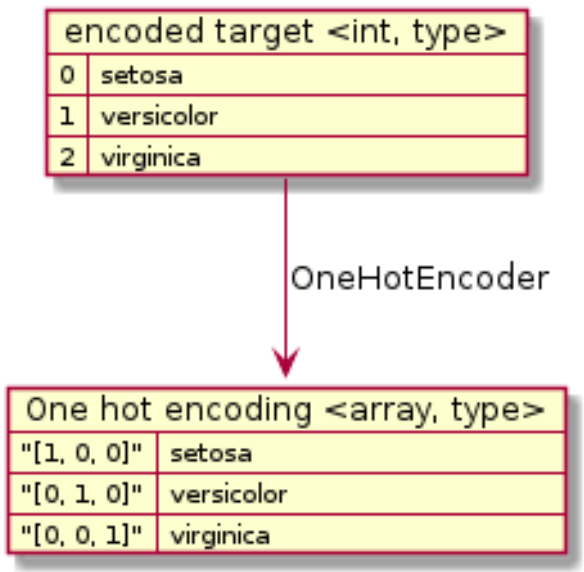
```
from sklearn.metrics import log_loss
log_loss(y,p)
```

```
0.3202939485569847
```



Multi-class Classification: One Hot Encoding

- Sometimes in datasets, we encounter columns that contain numbers of no specific order of preference.
- **Example:** Iris dataset, we take 5 random points with targets:




```
from sklearn.preprocessing import OneHotEncoder
target = np.array([[0],[1],[2],[0],[0]])
OneHotEncoder(sparse=False).fit_transform(target)
```

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{matrix}$$

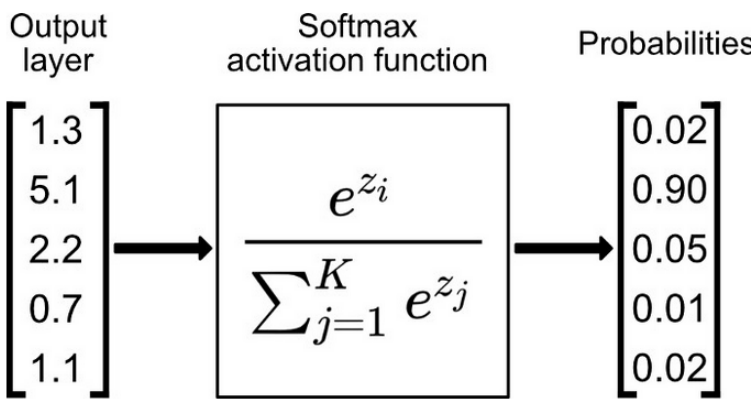
- Convert categorical data to numerical data with one hot encoding
  - categorical input features become standardized,
  - and used in multiclass output layers

Loss function: Multi-class Classification

Classification (cross entropy)

$$-\frac{1}{n}\sum y_i \log\left(p(y_i)\right)$$

Activation function form multiclass classification is `softmax`




```
target = np.array([
    [0., 0., 1.],
    [1., 0., 0.],
    [1., 0., 0.],
    [1., 0., 0.],
    [0., 1., 0.]])
prediction = np.array([
    [0.  , 0.05, 0.95],
    [0.98, 0.02, 0.  ],
    [0.98, 0.02, 0.  ],
    [0.95, 0.05, 0.  ],
    [0.03, 0.6 , 0.37]])

log_loss(target,prediction)
```

- Output values are normalised to probabilities summing to 1
- Top probability is the

Loss function: Multi-class Classification

```
target = np.array([
    [0., 0., 1.],
    [1., 0., 0.],
    [1., 0., 0.],
    [1., 0., 0.],
    [0., 1., 0.]])
prediction = np.array([
    [0.  , 0.05, 0.95],
    [0.98, 0.02, 0.  ],
    [0.98, 0.02, 0.  ],
    [0.95, 0.05, 0.  ],
    [0.03, 0.6 , 0.37]])

log_loss(target,prediction)
```

0.130763525435227

```
a = target*np.log(prediction)
np.round(a,decimals=2)
```

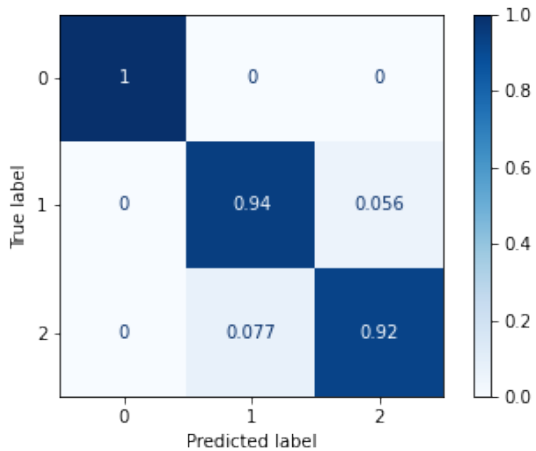
nan	0	-0.05
-0.02	0	nan
-0.02	0	nan
-0.05	0	nan
0	-0.51	0

```
#handle nan values
a[np.isnan(a)]=0
#loss
-np.sum(a)/5
```

0.1307635254352261 5

Confusion matrix

- A confusion matrix is a technique for summarizing the performance of a classification algorithm.
- Classification accuracy alone can be misleading if you have more than two classes in your dataset.
- scikit learn provides both the objects needed to calculate and plot the confusion matrix




```
from sklearn.metrics import
    plot_confusion_matrix
matrix = plot_confusion_matrix(
    clf, X_test, y_test,
    cmap=plt.cm.Blues,
    normalize='true')
```

- Predicted labels on the x-axis and the true labels on the y-axis. The blue cells contain the number of samples that the model accurately predicted. The white cells contain the number of samples that were incorrectly predicted. We can normalise by row or column.

Stochastic gradient descent

- **Backpropagation** is a training method used for a multilayer neural network. It is a generalization of the chain rule
- Weights are modified so as to **minimise the loss** the network's prediction and the actual target value.
- Calculate for all parameters (weights, biases)
  - $$\delta_w = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial h(w)} \frac{\partial h(w)}{\partial w}$$
- These modifications are made in the **backward** direction that is from the output layer through each hidden layer down to the first hidden layer. Hence the name **backpropagation**
- The weight's output δ and input activation are multiplied to find the **gradient of the weight**
- Adjust weights, where x_i is the activation function and η is the learning rate:
  - $W_{i,j} \leftarrow W_{i,j} + \eta \delta_j x_i$
- See backup for basic example
- Computationally simplified since activation gradients are easy to compute

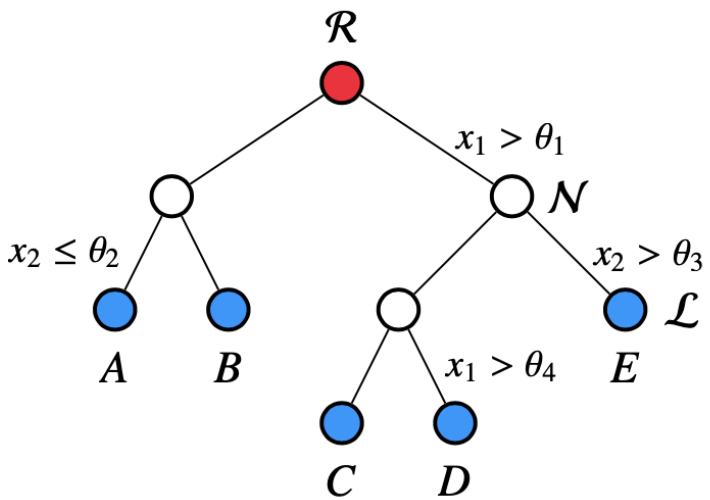
Stochastic gradient descent

- Stochastic gradient descent: Compute loss, perform gradient updates in batches, with typical batch sizes of O(10–1000) examples:
  - Non-deterministic; yields different results depending on composition, order of batches (shuffle!)
  - Different (stochastic) gradient descent algorithms ("optimisers") can be used
- One pass through all training examples gives N / (batch size) gradient updates. One such pass is called an epoch
- NN training then proceeds for a predefined number of epochs, typically O(10–1000)
- We can set some early stopping criteria for if e.g. training saturates
- Don't need to stick with final NN. Saving intermediate steps allows to get best NN.

Initialization: Decision trees vs Neural networks

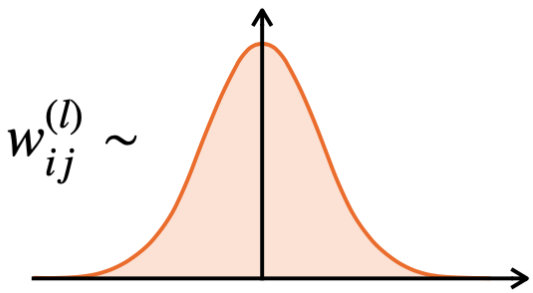
Decision trees

- Start from root note and build iteratively a tree
- No ambiguity, completely deterministic



Neural networks

- Set weights to 0 or 1: will lead neurons to learn the same features during the training
- Uniform distribution: initialize weights randomly from the uniform distribution.
  - Every number in the uniform distribution has an equal probability to be picked.

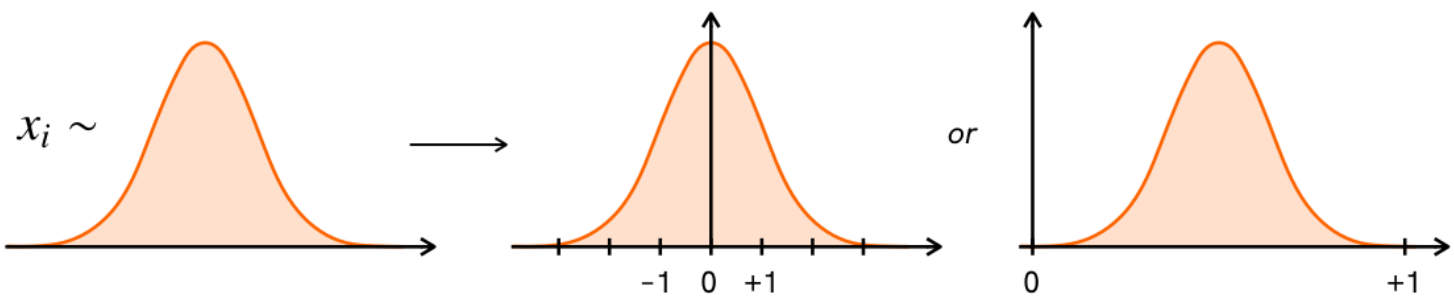


Data preprocessing: Standardize and Normalize

- Unlike decision trees, neural networks can't handle all inputs equally well
- In NNs, value **magnitude** $\approx$ **importance**. Features on very different scales (e.g. particle charge `e` vs energies `MeV`) assigned very different importance

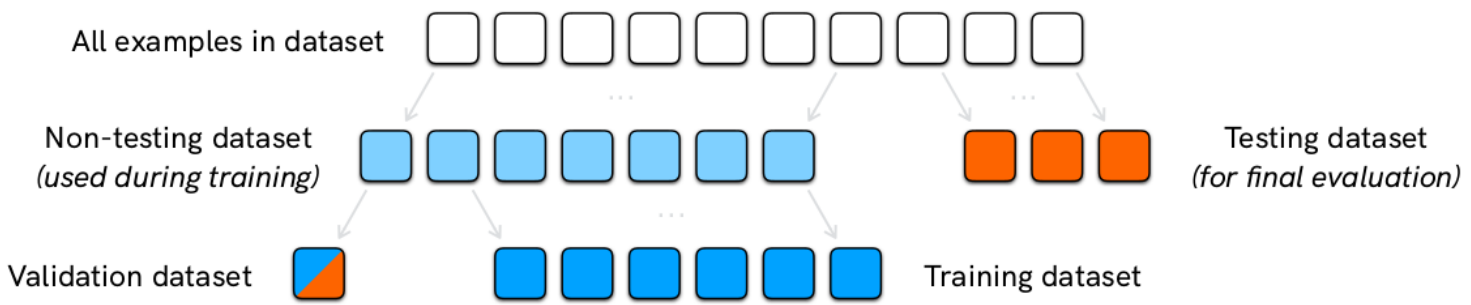
Solution: Feature standardisation

- **Standardising**: Input distribution of values is rescaled so that the *mean* of observed values is 0 and the *Standard deviation* is 1.
  - scikit-learn has the object StandardScaler that is used to standardize the dataset
- **Normalising**: It normalises data to the range min = 0 and max = 1.
  - scikit-learn has the object MinMaxScaler that is used to normalize the dataset



Training dataset split

- Any dataset (▢) of N examples ("rows") each with F features ("columns") is a **random sample of an underlying population**
- We want to train an ML algorithm on this sample, making sure that it **generalises** well to the full population
- Typical to split dataset into **training** (▢) and **testing** (▢) parts
  - The testing set is used only for final, to **avoid bias**
  - Additionally, some of the training dataset may be held out for on-the-fly **validation** (▢)



Neural Network in scikit-learn

- Scikit learn allows to construct also NN.
- scikit-learn uses Multilayer perceptron to construct NN
  - MLPClassifier for classification
  - MLPRegressor for regression

Example (sk-learn NN)

```
clf = MLPClassifier(max_iter=1000, alpha=0, hidden_layer_sizes=(100,),
    batch_size=16 )
```

- `max_iter` is the Maximum number of iterations. It determines the number of **epochs** (how many times each data point will be used)
- `batch_size` is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.
- Many more: `activation`, `learning_rate`, `solver`, ...

Neural Network in Keras

- Keras is the most common NN library
- Keras allow to use both sequential API and functional API models.
- Functional API models (that we will use) are more powerful because allow to have multiple outputs (suitable for multiclassification problems).

Example (Keras NN)

```
input = Input(shape=(8,))
x = Dense(8, activation='relu')(input)
output = Dense(4, activation='softmax')(x)
model = Model(input, output)
```

- The model expects rows of data with 8 variables (shape)
- The hidden layer has 8 nodes and uses the relu activation function.
- The output layer has 4 noded and uses the softmax activation function.

Fitting a model

Example (sklearn)

```
clf.fit(X_train, y_train)
```

Example (Keras NN)

```
mdl.compile('adam', loss='mean_squared_error',
    metrics=[tf.keras.metrics.MeanAbsoluteError()])
mdl.fit(X_train, y_train, epochs=100, batch_size=32, shuffle=True,
    validation_split=0.3)
```

- Optimiser, eg. 'adam', optimized is a stochastic gradient descent method, sets learning rate, etc.
- loss is the loss function on which network performance is evaluated.
- Metrics are additional loss functions, except that the results from evaluating a metric are not used when training the model
- See https://keras.io/api/

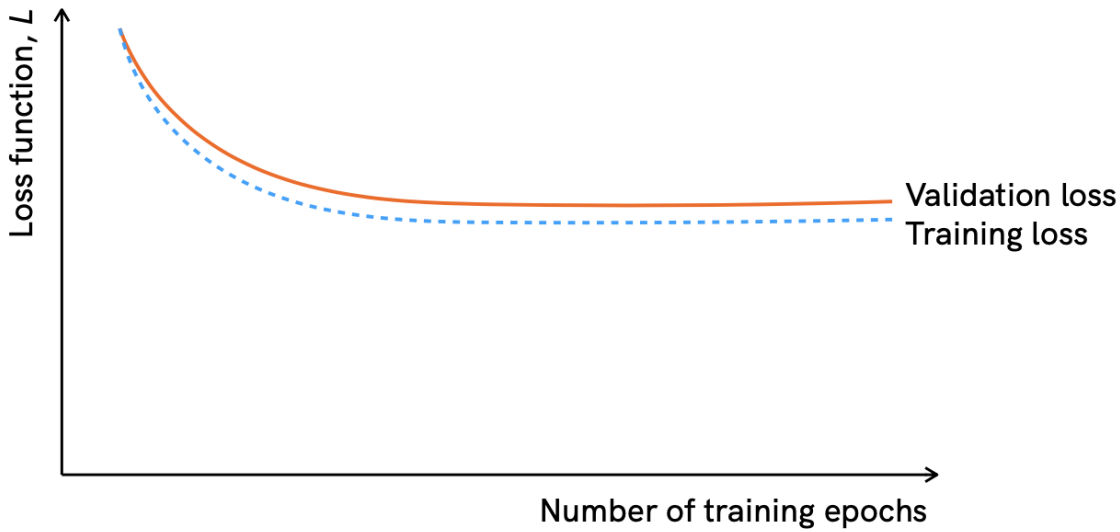
Bias variance trade off

Bias-variance trade-off is a fundamental principle for understanding the generalization of predictive learning models

- **Ideal model**: A model with low variance and low bias
- **Overfitting**: A model with low bias and high variance.
- **Underfitting**: A model with high bias and low variance.
- **worst model**: A model with high bias and high variance

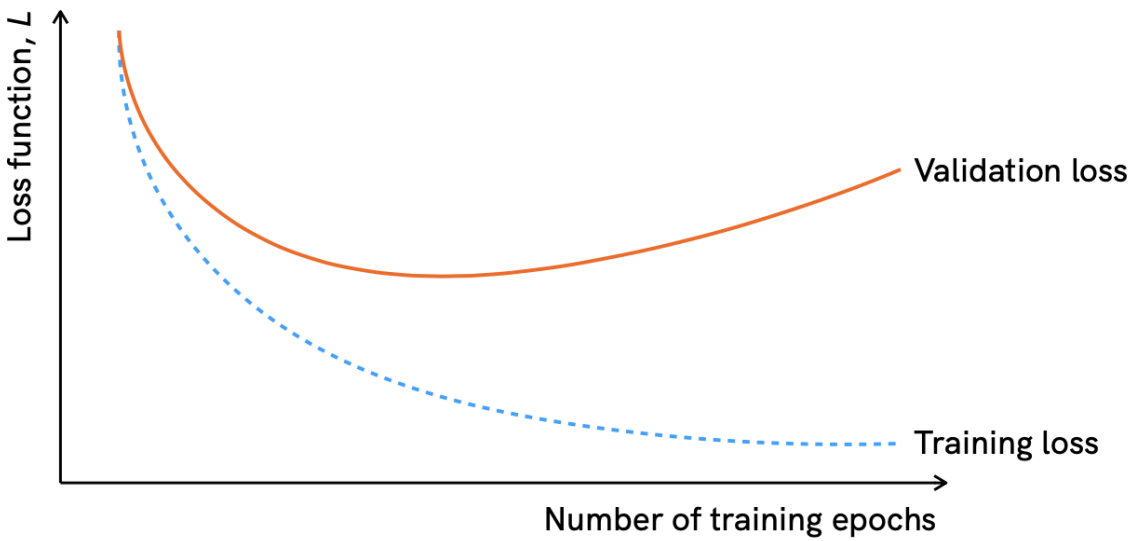
Bias-variance trade-off: Underfitting

- The Model has insufficient capacity/flexibility
- A model with poor performance on training data has **high bias**
- It is unable to learn relevant structure in dataset: **high bias**, **underfitting**
- In the example, training loss has saturated



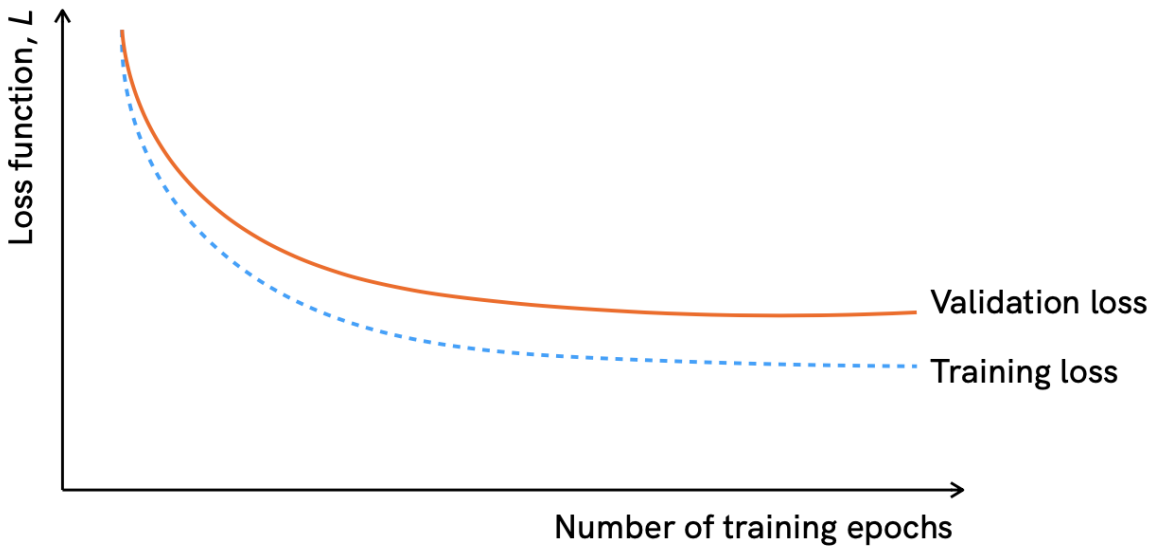
Bias-variance trade-off: Overfitting

- The Model has to much capacity/flexibility
- A model shows low loss on training, but high loss on testing datasets. The errors have **high variance**.
- Learns random structure in dataset that doesn't represents a generalisation: **high variance**, **overfitting**
- In the example, training loss is minimised, while validation loss saturates



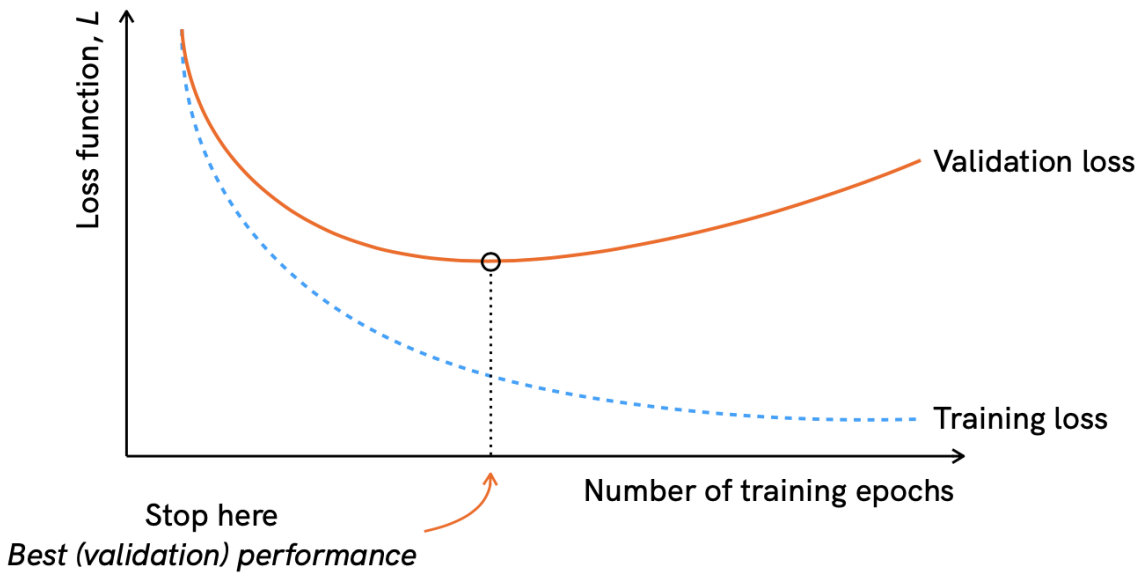
Bias-variance trade-off: Ideal model

- The Model has sufficient capacity/flexibility
- Learns relevant structure in dataset, good generalization: Ideal model



Stopping criteria

- Training needs to be stopped after a number of epochs
- Possibility to monitor the validation loss function to decide where to stop the training.



Regularization

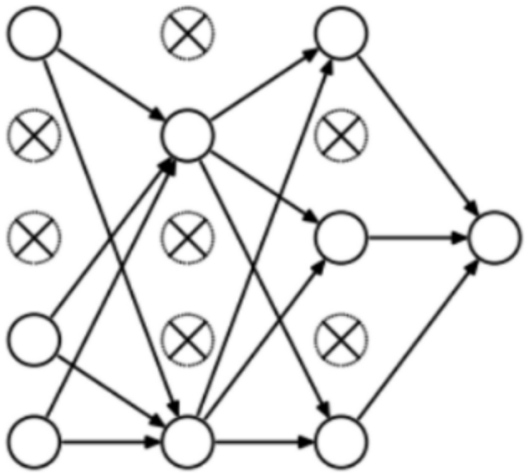
Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better, thus it reduces overfitting.

Most common types of regularization: L_1 (Lasso) and L_2 (Euclid norm)

- These updates the general cost function by adding a **regularization term**
- Due to the addition of the regularization term, the values of weight matrices decrease and therefore it reduces overfitting
- L2: Cost function = Loss + $\frac{\alpha}{\cdot} \sum \| w \|^2$
- L1: Cost function = Loss + $\frac{\alpha}{\cdot} \sum \| w \|$
- L1/L2: Cost function = Loss + $\frac{\alpha}{\cdot} \sum \| w \| + \frac{\alpha}{\cdot} \sum \| w \|^2$

Regularization: Dropout

- **Dropout** is another frequently used regularization technique
- Dropout at every iteration randomly selects some nodes and removes them along with all of their incoming and outcoming connections.



- Therefore each iteration has a different set of nodes and this results in a different set of outputs (similar as ensemble techniques).
- Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

Checkpoint1

Time to practice

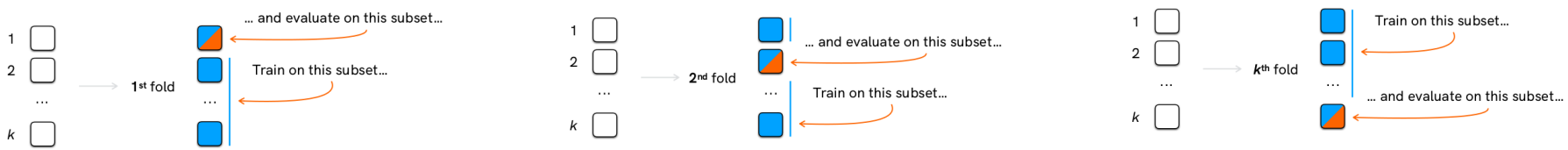
- Checkpoint 1 on weather dataset
- Classification problem on weather prediction (Clear, Cloudy, Raining)

Title	Exercise nos.	Number of marks
1. Conceptual questions	1–5	2.5
2. Data preprocessing and RandomForest	6–9	2.5
3. Neural networks in `scikit-learn`	10–11	1.5
4. Neural networks in `Keras`	12–13	2
5. Regularisation	14–15	1.5
6. Bonus: Hyperparameter optimisation	16	1.0 (*bonus*)
Total		**10 + 1**

- Submit to Learn by Friday 10am

Cross validation

- When optimising a network architecture and hyperparameters we will generate biased estimators due to running multiple trainings on the same data
- One solution is cross validation, i.e. cycle the training and validation data
  - For each hyperparameter configuration to be tested, randomly split training data into k partitions (typically 3–10)



- Distributional information (mean, std. dev.) from these k folds provide unbiased basis for e.g. optimising hyperparams!

Hyperparameters

Hyperparameters

Hyperparameters are the variables that determines the network structure (e.g. number of hidden units) and the variables which determine how the network is trained (e.g. learning rate)

- **Hyperparameters related to Network structure**: N. hidden layers, dropout, weight inizialization, activation function.
- **Hyperparameters related to training algorithm**: N. learning rate, N. epochs, batch size.
- Various ways to optimize them

Hyperparameter optimization

Tuning or optimizing hyperparameters involves finding the values of each hyperparameter which will help the model provide the most accurate predictions.

Hyperparameter optimization methods:

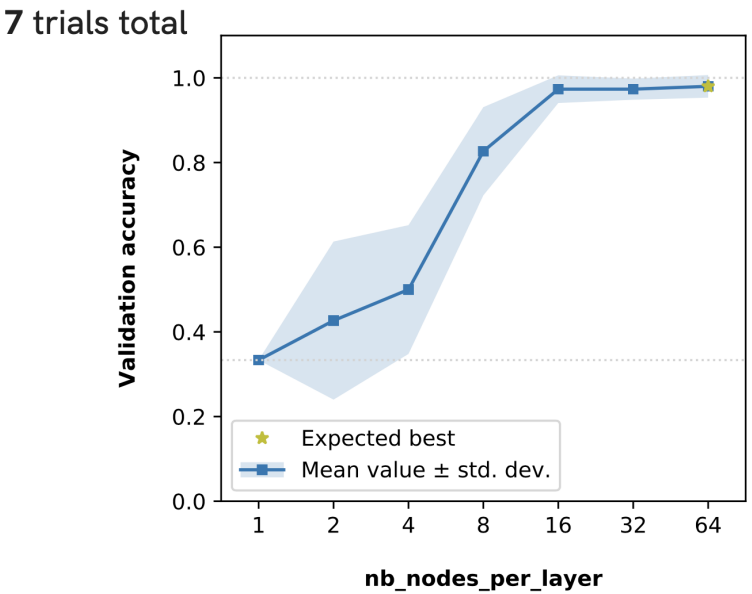
1. Manual Hyperparameter tuning
   - Pros: Very simple and effective with skilled operators Cons: Not scientific, unknown if you have fully optimized hyperparameters
2. Grid search
3. Random search
4. Bayesian optimization

Grid search

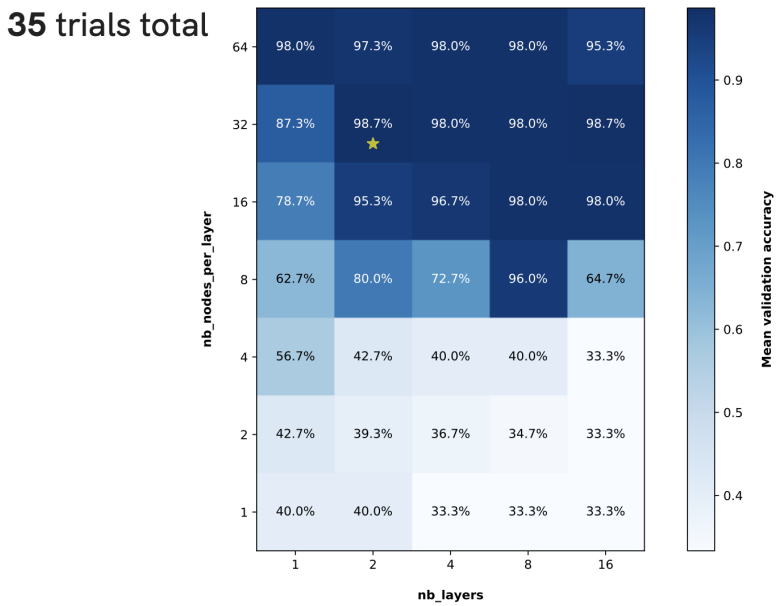
- Grid search is slightly more sophisticated than manual tuning.
- It involves systematically testing multiple values of each hyperparameter, by automatically retraining the model for each value of the parameter.
- **Example** you can perform a grid search for the optimal batch size by automatically training the model for batch sizes between 10-100 samples, in steps of 20.
  - The model will run 5 times and the batch size selected will be the one which yields highest accuracy.
- **Pros**: Maps out the problem space and provides more opportunity for optimization
- **Cons**: Can be slow to run for large numbers of hyperparameter values

Grid search: 1D and 2D

- Choose number of layers D=3, scan grid nodes H $\in \{1, 2, 4, 8, 16, 32, 64\}$



- Grid D x H $\in \{1, 2, 4, 8, 16\}$ x $\{1, 2, 4, 8, 16, 32, 64\}$



Random search

- Testing randomized values of hyperparameters is actually more effective than manual search or grid search.
- It conists in testing randomnly values drawn from the entire problem space instead of testing systematically to cover "promising areas" of the problem space.
- **Pros**: According to the study, provides higher accuracy with less training cycles, for problems with high dimensionality
- **Cons**: Results are unintuitive, difficult to understand "why" hyperparameter values were chosen

Bayesian optimization

- Bayesian optimization is a technique which tries to approximate the trained model with different possible hyperparameter values.
- It trains the model with different hyperparameter values, and observes the function generated for the model by each set of parameter values.
- It does this over and over again, each time selecting hyperparameter values that are slightly different.
- The algorithm ends up with a list of possible hyperparameter value sets and model functions, from which it predicts the optimal function across the entire problem set.
- **Pros**: bayesian optimization results in significantly higher accuracy compared to random search.
- **Cons**: Like random search, results are not intuitive and difficult to improve on, even by trained operators

Conclusions

- Given an introduction to Neural Networks and Deep learning algorithms
- Overview of Neural Network training
  - Both scikit learn and Keras models introduced
- Hyperparameter and Hyperparameter optimization
- Enjoy Checkpoint 1!

Thanks for your attention!

Batch normalization

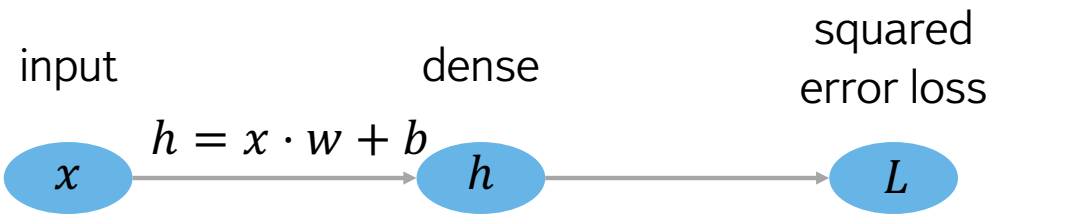
- Batch Normalization converts interlayer outputs into a standard format, called normalizing.
  - This effectively 'resets' the distribution of the output of the previous layer to be more efficiently processed by the subsequent layer.

Advantages of Batch Normalization:

- Leads to faster learning rates since normalization ensures there's no activation value that's too high or too low, as well as allowing each layer to learn independently of the others.
- Normalizing inputs reduces the "dropout" rate, or data lost between processing layers which significantly improves accuracy throughout the network.

Back propagation

The simplest NN ever

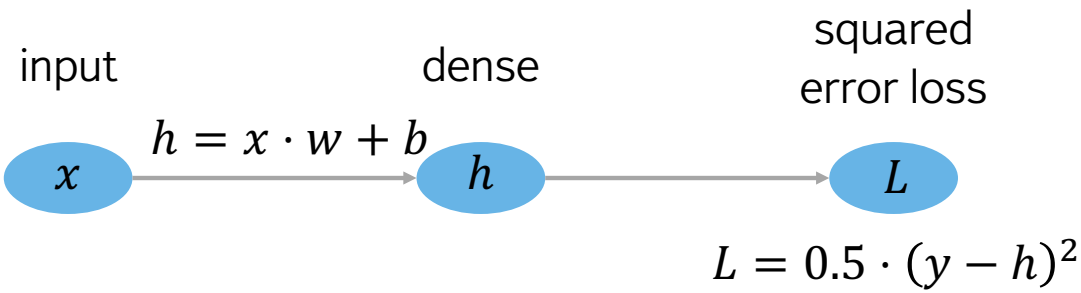


- Parameters:
  - Weight w and bias b
- Input: x
- Target: y

Also known as
least squares linear regression

Back propagation

The simplest NN ever



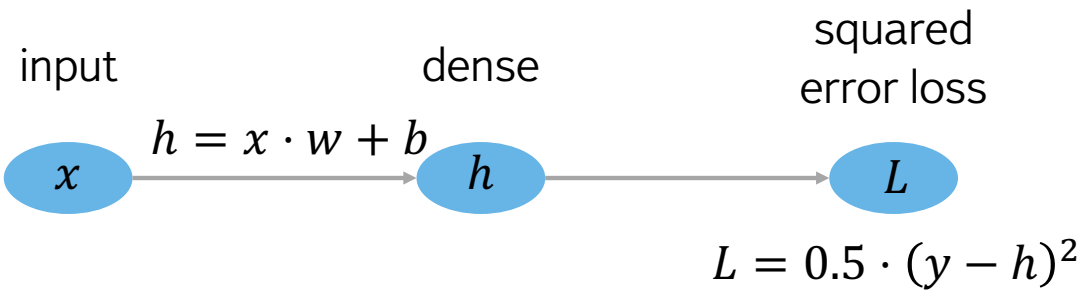
- Parameters:
  - Weight w and bias b
- Input: x
- Target: y

L is just a function of parameters, features and target:

$$L = f\big(y, g(x, w, b)\big)$$

Back propagation

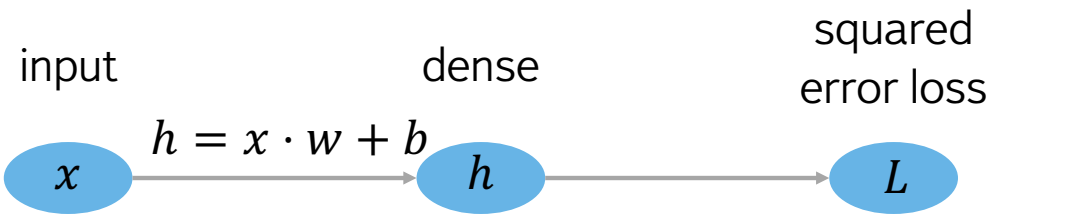
The simplest NN ever



- Gradient?
- $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$

Back propagation

The simplest NN ever



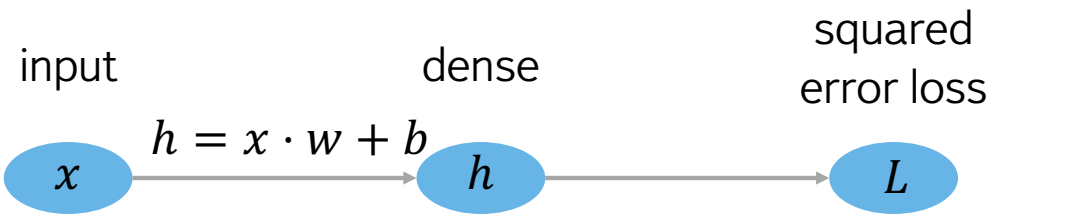
$$L = 0.5 \cdot (y - h)^2$$

- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b

Back propagation

Forward pass

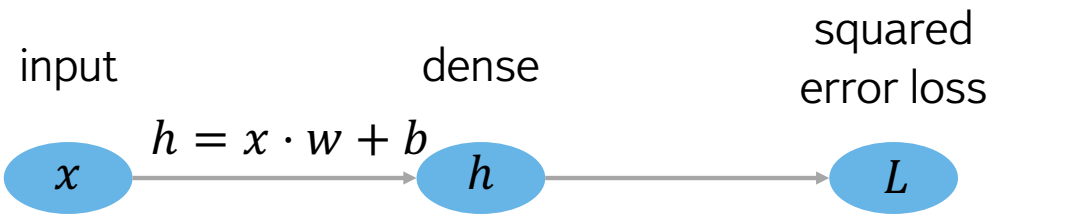


- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80					

Back propagation

Backward pass



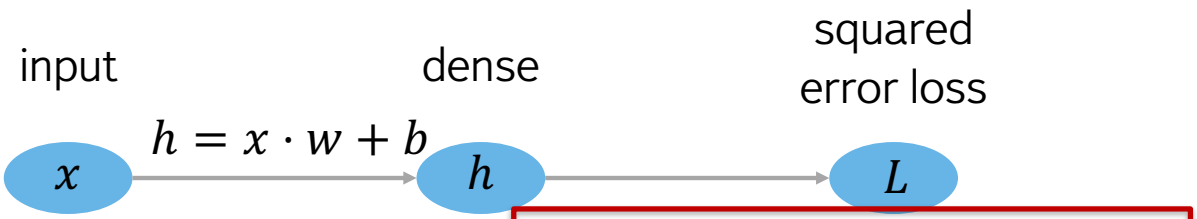
$$L = 0.5 \cdot (y - h)^2$$

- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Back propagation

Backward pass

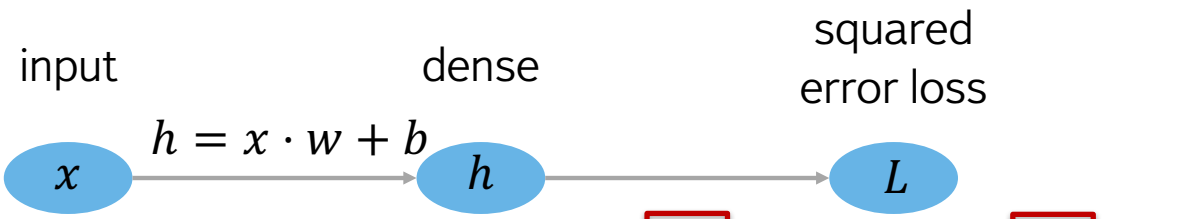


- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Back propagation

Backward pass



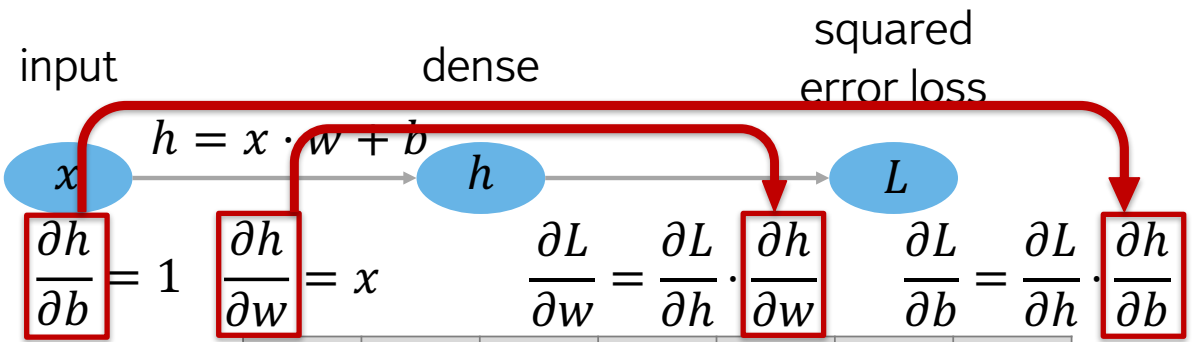
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Back propagation

Backward pass

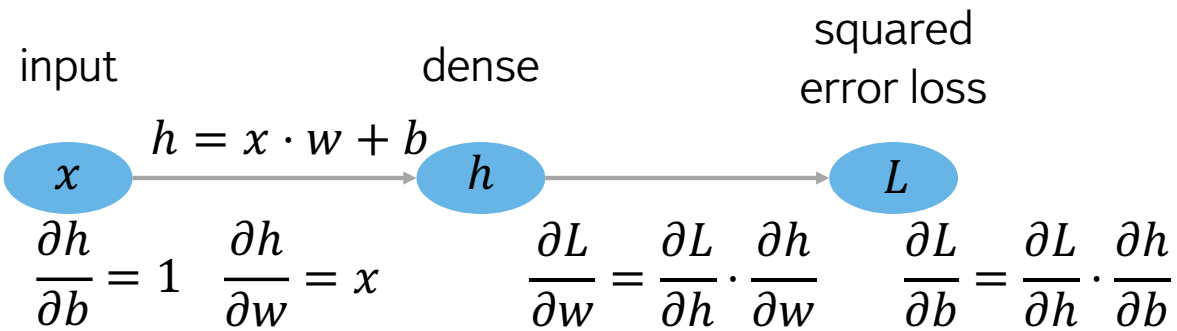


- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9				

Back propagation

Backward pass

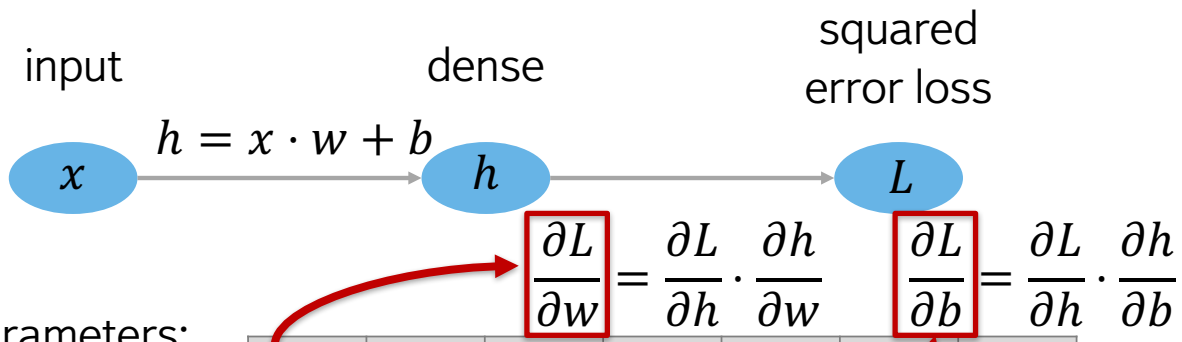


- Let's fit
  - $y = 3, x = 1$
- Initial
  - $w = 0.1, b = 1$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9		

Back propagation

Backward pass



- Update parameters:

$$w \mathrel{-}= \eta \frac{\partial L}{\partial w}$$

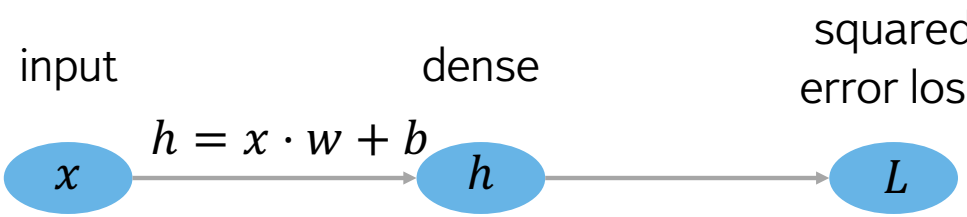
$$b \mathrel{-}= \eta \frac{\partial L}{\partial b}$$

$$\eta = 0.2$$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9		

Back propagation

Backward pass



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Update parameters:

$$w \mathrel{-}= \eta \frac{\partial L}{\partial w}$$

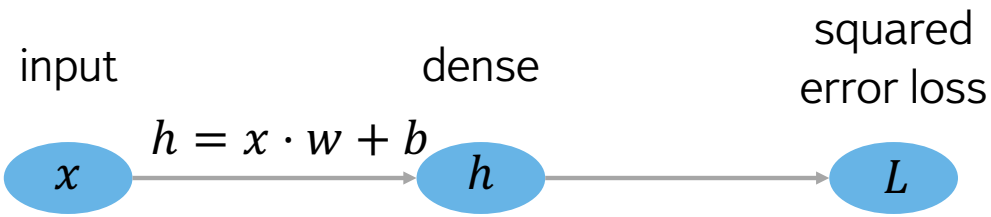
$$b \mathrel{-}= \eta \frac{\partial L}{\partial b}$$

$$\eta = 0.2$$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9	0.48	1.38

Back propagation

After a few more updates...



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial w} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial b}$$

- Update parameters:

$$w \mathrel{-}= \eta \frac{\partial L}{\partial w}$$

$$b \mathrel{-}= \eta \frac{\partial L}{\partial b}$$

$$\eta = 0.2$$

h	L	$\frac{\partial L}{\partial h}$	$\frac{\partial L}{\partial w}$	$\frac{\partial L}{\partial b}$	w	b
1.1	1.80	-1.9	-1.9	-1.9	0.48	1.38
1.86	0.65	-1.14	-1.14	-1.14	0.71	1.61
2.32	0.23	-0.68	-0.68	-0.68	0.84	1.75