Activation functions

- Linear
- Relu
- Sigmoid
- Softmax
- Tanh

Layers

- Dense
- lInput

Loss functions

- Mean Squared Error
- Categorical Cross-Entropy

Models

Weight initialization

- Random normal
- Random uniform
- Zeros
- Ones
- Xavier normal
- Xavier uniform
- He normal
- He uniform

Activation functions

Activation functions are mathematical functions applied to the output of a neuron in neural network. They introduce non-linearity into the model so it can learn more complex patterns.

An Activation class is Layer subclass that can be attached to a layer (another Layer subclass) in the network (Scroll to a specific activation function for examples).

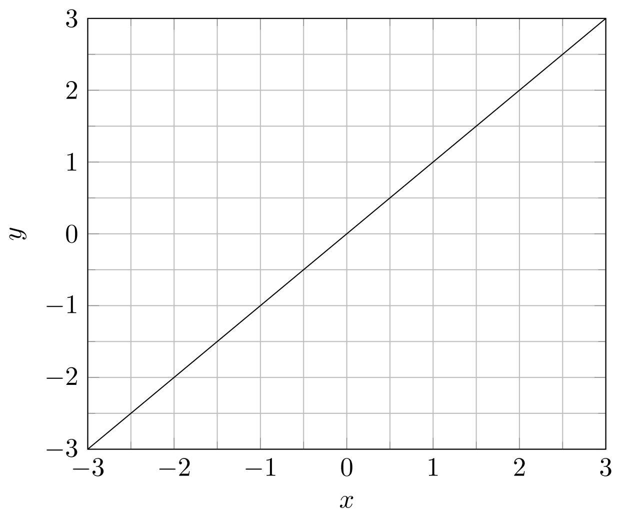
Activation class:

```
class Activation(Layer):
    def __init__(self, activation_function) -> None:
        activation = activation_functions.get_function(
                                              activation_function)
        self.name = activation_function
        self.activation = activation[0]
        self.activation_derivative = activation[1]

    def forward(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    def backward(self, output_error):
        return self.activation_derivative(self.input, output_error)
```

Linear function

Linear activation function returns the value itself. It can be used as a placeholder for an activation function when one isn't needed.

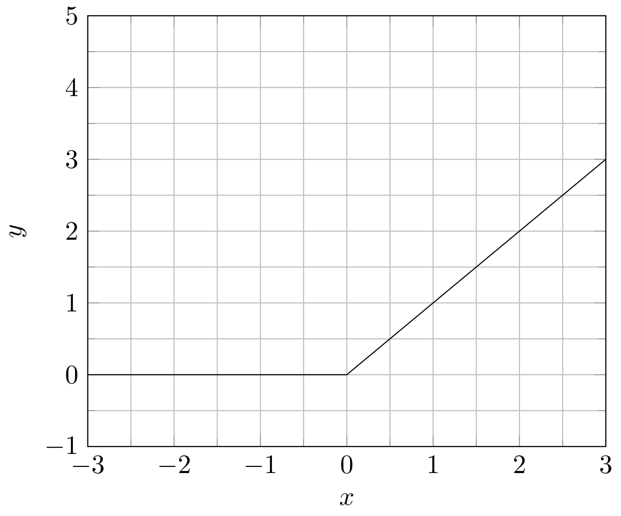
$$f(x) = x$$



<u>ReLU function</u>

ReLU (Rectified Linear Unit) is an activation function that returns 0 for all negative input values and the input value itself for all positive values.

$$f(x) = max(0, x)$$



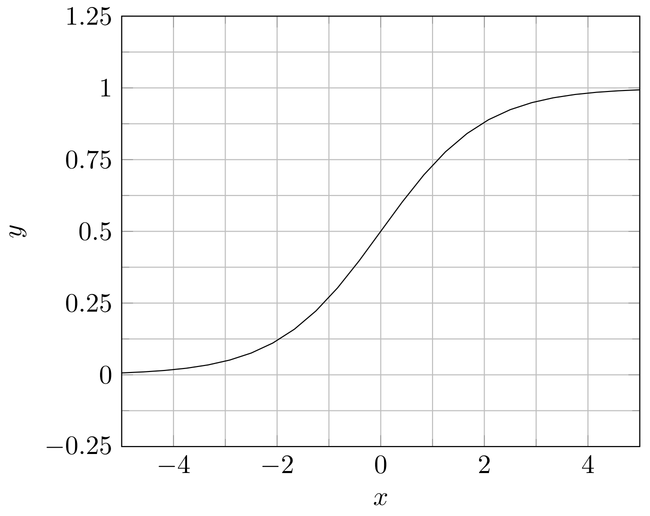
ReLu can be attached as activation function to a layer like this:

```
Dense(input_size =input_size,
      output_size=output_size,
      activation ='relu')
```

Sigmoid function

Sigmoid is an activation function that maps input values to a range between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}}$$



Sigmoid can be attached as activation function to a layer like this:

```
Dense(input_size =input_size,
      output_size=output_size,
      activation ='sigmoid')
```

Softmax function

Softmax is an activation function used for multi-class classification problems. It takes a vector of raw scores (logits) and converts them into a probability distribution over multiple classes meaning that each value will be in range $[0, 1]$ and the sum of those values is 1.

$$y(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

Softmax can be attached as activation function to a layer like this:

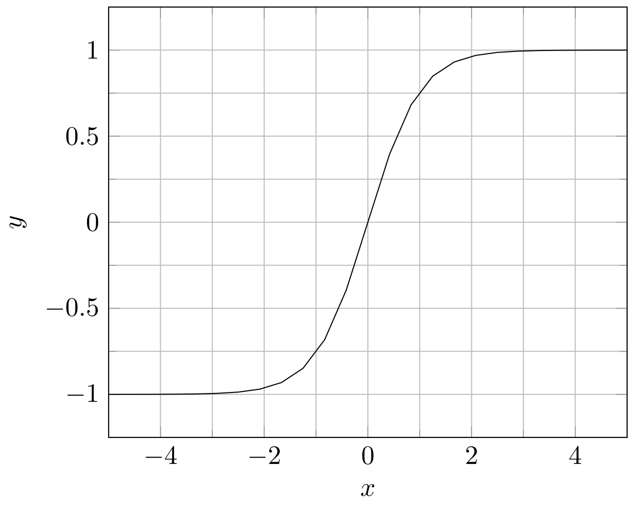
```
    Dense(input_size =input_size,
          output_size=output_size,
          activation ='softmax')
```

NB! Because the derivative for Softmax function is calculated together with Categorical Cross-Entropy loss, they can only be used in combination with each other. The last layer of the model should have Softmax as the activation function and the model should use Categorical Cross-Entropy loss.

Tanh function

Tanh is an activation function that maps input values to a range between -1 and 1.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Tanh can be attached as activation function to a layer like this:

```
Dense(input_size =input_size,
      output_size=output_size,
      activation ='tanh')
```

Layers

Layer is a functional unit that takes input data, applies a mathematical transformation to it and produces an output. Each layer consists of a set of neurons each of which computes a weighted sum of its inputs and applies an activation function to the result.

Base layer class:

```python
class Layer:
    def __init__(self) -> None:
        self.input = None
        self.output = None

    def forward(self, input):
        raise NotImplementedError

    def backward(self, output_error, learning_rate):
        raise NotImplementedError

    def update(self, learning_rate):
        raise NotImplementedError
```

Each layer has a forward() function, backward() function and update() function. Forward function calculates the output given an input, backward function calculates the gradients that will be applied to the weights and update function updates the weights accordingly.

Dense layer

A Dense layer is a type of layer where each neuron is connected to every neuron in the preceding layer. It performs a weighted sum of the inputs from the previous layer and then applies an activation function to produce an output.

The Dense layer has the following functions:

```
def __init__(self,
             output_size         = None,
             activation          = None,
             weights_initializer = None,
             bias_initializer    ='zeros',
             name                = None)

# Initializes a Dense layer and defines:
#       - the number of outputs
#       - the activation function that's going to be used
#       - the initialization technique for the weights
#       - the initialization technique for the biases
#       - the name of the layer
```

```
def __call__(self,
             previous_layer)
# This function is used to stack layers after each other.
# Previous layer argument is added so that on model
# compilation a layer graph can be created. The input size
# for this layer is determined by the output size of the
# previous layer.
# Then the weight matrix is created for this layer.
```

```
def create_weights(self)
# Creates the weight and bias matrices for the layer.
```

```
def forward(self,
            input_data)
# An output is calculated by matrix multiplication between
# an input matrix  and the weights.
# The bias is added and the result is passed through the activation
# function.
```

```
def backward(self,
             output_error)
# Gradients for the weight and bias matrices are calculated by
# first calculating the derivatives of the activation function,
# then the derivatives of the weights and biases. The gradients
# are stored so that the weights and biases can be updated later.
# The input error is returned.
```

```
def update(self,
           learning_rate)
# The calculated gradients for weights and biases are multiplied
# by learning rate and subtracted from weight and bias matrices.
```

```
def validate_init(self,
                  output_size,
                  name)
# Makes sure that the arguments given on initialization are of
# correct type. Also checks that the output size is >0
```

```
def validate_call(self,
                  previous_layer)
# Makes sure the previous layer is of the correct type
```

Input layer

An Input layer is the very first layer of the network. It's only function is to pass on the input data to the next layer.

Input layer has the following functions:

```
def __init__(self,
             shape)
# Shape describes the dimensions of the input data excluding
# the batch dimension.
```

```
def forward(self,
            inputs)
# Forwards the input to the next layer
```

```
def backward(self,
             output_error,
             learning_rate)
# Returns the output error without change.
```

```
def validate_init(self,
                  shape)
# Makes sure that the shape is a tuple and that it consists
# only of positive integers.
```

Loss functions

A loss function is a mathematical function used to measure the error between the predicted outputs of a model and the actual target values. The goal during learning is to minimize this loss.

The Loss class:

```python
class Loss:
    def __init__(self, loss_function) -> None:
        loss = loss_functions.get_loss(loss_function)
        self.name = loss_function
        self.loss = loss[0]
        self.loss_derivative = loss[1]

    def forward(self, y_true, y_pred):
        return self.loss(y_true, y_pred)

    def backward(self, y_true, y_pred):
        return self.loss_derivative(y_true, y_pred)
```

Mean Squared Error

Mean Square Error is a loss function used primarily in regression problems. It measures the average of the squared differences between the predicted values and the actual values.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$$

where y_i is the true value, $\hat{y_i}$ is the predicted value and N is the number of samples.

To use Mean Squared Error as the loss function, it should be added as ”mse” as an argument while compiling the model:

```
model.compile(loss_fn='mse')
```

Categorical Cross-Entropy

Catgeorical Cross-Entropy is a loss function used in multi-class classification problems. It quantifies the dissimilarity between the predicted class probabilities and the true class probabilities.

$$CCE = -\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i)$$

where y_i is the true value, $\hat{y}_i$ is the predicted value and N is the number of samples.

To use Categorical Cross-Entropy as the loss function, it should be added as an argument when compiling the model:

```
model.compile(loss_fn='categorical_cross_entropy')
```

PS! Because the derivatives of Categorical Cross-Entropy and Softmax activation function are calculated together, they should be used in combination. The Softmax function should be the activation function of (only!) the last layer in the model when Categorical Cross-Entropy is used as loss function.

Models

A model is a mathematical or computational structure that comprises layers of interconnected neurons each performing mathematical operations on input data. The model is trained using labeled data to learn patterns and relationships allowing it to make predictions or decisions on new, unseen data.

Model has the following methods:

```
def __init__(self,
             inputs,
             outputs)
# Initializes the model and defines the input and output layers.
```

```
def compile(self,
            loss_fn)
# Defines the loss function and its derivative and creates
# the layer graph starting with the input layer and finishing
# with the output layer.
```

```
def predict(self,
            input_data)
# Performs a forward pass through the network and appends the
# output of the final layer into a list. Does so for each
# sample given and returns the list with outputs.
```

```
def fit(self,
        x_train,
        y_train,
        epochs,
        learning_rate,
        print_loss=True)
# Performs a forward pass and a backward pass through the network
# and updates the layers (in other words: trains the model).
# Calculates the loss and prints it out if print_loss is set
# to True. It does all of that as many times as is defined by
# the variable epochs. Returns the final loss.
```

```
def forward(self,
            x)
# Calls the forward() method for each layer in turn with the
# training sample x as an initial input.
```

```
def backward(self,
             x)
# Calls backward() for each layer in turn starting from the last
                                 one
# and finishing with the first one.
```

```
def update(self,
           learning_rate)
# Calls update() on each layer in turn.
```

A model can be created by defining an input layer and then calling every next layer with the previous one as an argument. Example:

```
inputs = Dense(input_size=2, output_size=16, activation='tanh')
x = Dense(output_size=8, activation='relu')(inputs)
x = Dense(output_size=4, activation='relu')(x)
x = Dense(output_size=2, activation='relu')(x)
outputs = Dense(output_size=1, activation='sigmoid')(x)

model = Model(inputs, outputs)
```

When the model is defined, it can be compiled as follows:

```
model.compile(loss_fn='mse')
```

For training, the fit() method should be called with

- training data
- labels
- number of epochs
- learning rate

as arguments. The parameter print_loss determines whether the loss is printed out after every epoch or not. The fit() method returns the final loss:

```
model.fit(x_train,
          y_train,
          epochs=500,
          learning_rate=0.05,
          print_loss=False)
```

Finally, a predict() method can be called to get some predictions:

```
predictions = model.predict(x_test)
```

Weight initialization

Setting the initial values for network's weights before the training begins is called weight initialization. The goal is to set them in such a way that a reasonable variance is maintained throughout the learning process which helps to prevent the gradients from vanishing or exploding during back-propagation. Proper weight initialization also helps to accelerate learning.

Weight initialization technique can be defined when creating a layer by passing the techinque's name as *weight_initializer* parameter like so:

```
inputs = Input(shape=(2,))
x = Dense(output_size=4,
          activation='tanh',
          weights_initializer='xavier_uniform')(inputs)
```

If *weight_initializer* is not specified, the Random normal initialization is used.

Random normal

The weights are initialized by drawing random values from a normal (Gaussian) distribution with a mean of 0 and standard deviation of 0.05.

It's suitable for layers using ReLU activation functions.

Value to pass as weight_initializer:
"random_normal" or **None**.

Random uniform

The weights are initialized by drawing random values from a uniform distribution within range $[-0.05, 0.05]$.

It's suitable for layers using ReLU activation functions.

Value to pass as weight_initializer:
"random_uniform"

Zeros

All weights are set to zero.

Should be used when initializing biases (not weights).

Value to pass as weight_initializer:
"zeros"

<u>Ones</u>

All weights are set to ones.

In certain networks setting weights to 1 can help establish initial connections that pass information unchanged through the network.

Value to pass as weight_initializer:
"ones"

Xavier normal

Xavier Normal (also known as Glorot Normal) is a weight initialization technique designed to address the vanishing/exploding gradient problem. The weights are initialized by drawing random values from a normal distribution with a meane of 0 and a standard deviation of $\sqrt{\frac{2}{fan\_in+fan\_out}}$ where 'fan_in' is the number of input units and 'fan_out' is the number of output units for the layer.

It's particularly effective when using Sigmoid or Tanh activation functions.

Value to pass as weight_initializer:
"xavier_normal"

Xavier uniform

Xavier Uniform is a weight initialization technique designed to adress vanishing/exploding gradient problem. Weights are initialized by drawing random values from uniform distribution within range $\left[-\sqrt{\frac{6}{fan\_in+fan\_out}}, \sqrt{\frac{6}{fan\_in+fan\_out}}\right]$, where 'fan_in' is the number of input units and 'fan_out' the number of output units for the layer.

Suitable for layers using Sigmoid and Tanh activation functions.

Value to pass as weight_initializer:
"xavier_uniform"

He normal

He Normal is a weight initialization technique that draws random values from normal distribution with a mean of 0 and a standard deviation of $\sqrt{\frac{2}{fan\_in}}$ to initialize weights. 'Fan_in' is the number of input units for the layer.

Designed to work with layers using ReLU activation function.

Value to pass as weight_initializer:
"he_normal"

He uniform

He Uniform is a weight initialization technique that draws random values from uniform distribution in range $\left[-\sqrt{\frac{6}{fan\_in}}, \sqrt{\frac{6}{fan\_in}}\right]$ to initialize the weights. 'Fan_in' is the number of input units for the layer.

Designed to work with layers using ReLU activation function.

Value to pass as weight_initializer:
"he_uniform"