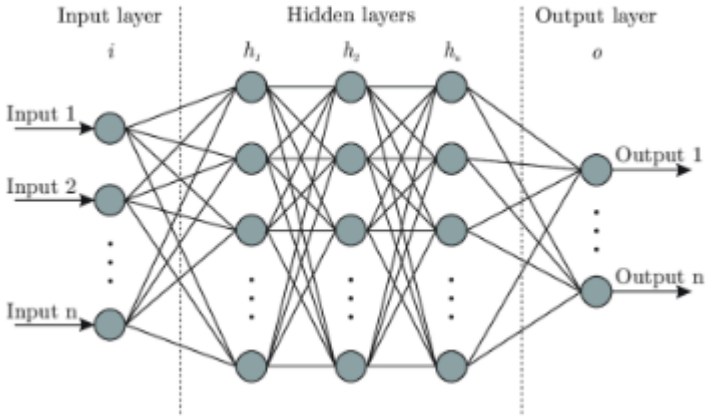
Building a neural netowork

In this notebook we will build a general purpose feed forward neural netowork. It is assumed that the reader is somewhat familiar with neural networks and is fluent in calculus. If not, a breif (and potentially un-enlightening) introduction to feed forward neural networks (ffnns) is presented in the following. The calculus I can't help you with.

Ffnns are baically a whole bunch of connected nodes (neurons) assembled in a layer configuration with an input layer, an arbitrary number of hidden layers, and an output layer. See illustration below:



The networks work by feeding some data you want to model into the input layer, and through the magic of maths, a prediction is made in the output layer. I hope the introduction was enlightening. Now: onto the maths.

Each neuon in each layer (with the exception of the input and output layers) is connected to every neuron in the previous and consecutive layer. So if we input a value into Input 1, this value is sent to each neuron in the first hidden layer, and on the way undergoes a transformation. This is also the case for Input 2, and all inputs up to Input n.

The connections between these neurons are called weights, and take numerical values. We will discuss these values in detail later. Each neuron also has an asscosiated bias.

Let's start to build a more mathematical formulation of this. We can begin my assesing what happens in neuron 1 in the first hidden layer when an array of data $\vec{x}$ of length n is fed into the input layer. Where x_1 enters Input 1 x_2 enters Input 2, and so on. Henceforth I shall entirely dispense with vector notation. This will become somewhat problematic later (opsies)

The values getting passed to neuron 1 is then x_i. But how much of x_i "enters" neuron 1? For each value of x_i we multiply by the weight asscosiated with the connection between the input layer and neuron 1. Mathematically, this is expressed as

$$y_{in} = \sum_{i=1}^{n} w_i x_i$$

When the data is recieved in the neuron, the bias of the neuron is added.

$$y_{neuron} = \sum_{i=1}^{n} w_i x_i + b$$

Now we take a small pause to talk about activation functions. These are functions that dictate how responsive the neuron is to the input. I'll get back to these functions later. At the moment the important thing to remember is that it is adventagous to limit or control how much the input affects the output of the neuron.

With the activation function $f(z)$, the output (or activation) a of the neuron becomes

$$a = f\left(\sum_{i=1}^{n} w_i x_i + b \right)$$

Now, we can generalize this for every neuron in the first hidden layer. Let the index bonanza commence.

For neuron i in layer 1 (first hidden layer), the input is

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1$$

where M is the number of inputs.

The output of neuron i then becomes

$$a_i^1 = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \right)$$

For the hidden layers $l = [2, 3, \ldots, L]$ - L being the output layer - the output of of the i-th neuron becomes

$$a_i^l = f\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l \right)$$

If we want we can have different activation functions for each layer, so that a_i^l becomes

$$a_i^l = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l \right)$$

Now this is all nice and well, but all these indices are honestly too much. Luckily, linear algebra comes to our aid (upper case variables being matrices, and lower case being vectors)

$$a^1 = f^1(X \cdot W^1 + b^1) \quad (1)$$
$$a^l = f^l(a^{l-1} \cdot W^l + b^l) \quad (2)$$

where it is implied that the activation function is applied elementwise to z. These -- (1) and (2) -- are the equations for the feed forward process

Now you might be thing "Congratulations! You have managed to do absolutely nothing in terms of making a data model. And what actually are the weights and biases? WHY ARE WE DOING THIS?"

This is a totally legitimate line of reasoning, and I'm going to pretty much ignore it. What I'll say is that our goal ultimately is to find weights and biases that will transform the input data into the desired data, i.e. make a prediction (that's actually correct) in the output layer. And don't worry fam, I'll get to that now.

The output of the network is the activation of the neurons in the last layer. We write this as

$$\tilde{y} = a^L$$

Now, we might want to evaluate how the netowrk performed. This is done with a cost function $C(W^L)$. The point of the cost function is to tell us how bad (or) good the netowork did. After feeding the data through the network once, you'd might excpect the prediction to be absolutely useless, and I think you should keep on to that intuition. But we can use this garbage prediction.

By finding the derivative and thus the minimum of the cost, we can update the weights and biases in such a way that the next prediction is better. Let's do some maths

We define a cost function $C(W^L) = C(f^L(a^{L-1}W^L + b^L))$. We also remind remember that

$$a^L = f^L(a^{L-1}W^L + b^L) = f^L(z^L)$$

We then find the derivative (assuming for the moment that all variables and "functions" are one-dimential, e.g. $C(W) = e^W, \quad W \in \mathbb{R}$) of the cost function

$$\frac{\partial C}{\partial W^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial W^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}a^{L-1}$$

We write this (in matrix form) as

$$\nabla_{W^L} C = (a^{L-1})^T\left(f'(z^L) \odot \nabla_{a^L} C\right) = (a^{L-1})^T\delta^L \tag{3}$$

where we define δ^L as the error in the output layer. We're kinda doing matrix calculus here (not really, I'm just presenting you with a result), so it might be of value to note that we're getting some transposes and hadamar products all of a sudden; and the order of the products are mirrored. I'm not going into any detail about this, but it's easy (not really, but it's not too hard to work through to show the same result) to see where these come from if you instead work with the explicit form

$$C(W^L) = C\left(f^L\left(\sum_{j=1}^{N_{L-1}} W_{ij}^L a_j^{L-1} + b_i^L\right)\right)$$

We can find the error made by an arbitrary layer with (again assuming variables $\in \mathbb{R}$, not $\mathbb{R}^{n\times m}$)

$$\delta^l = \frac{\partial C}{\partial z^l} = \frac{\partial C}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial z^l} = \delta^{l+1}\frac{\partial z^{l+1}}{\partial z^l}$$

Now we remember that

$$z^{l+1} = a^l W^{l+1} + b^{l+1}$$

thus

$$\frac{\partial z^{l+1}}{\partial z^l} = f'(z^l)W^{l+1}$$

Yielding

$$\delta^l = \delta^{l+1} f'(z^l)W^{l+1}$$

Or in matrix form

$$\delta^l = (\delta^{l+1}(W^{l+1})^T) \odot f'(z^l)$$

$$\rightarrow \nabla_{W^l} C = a^{l-1}\delta^l \tag{4}$$

By using gradient descent we can find the optimal weights for our network

$$W^l \leftarrow W^l - \eta \nabla_{W^l} C$$

Where η is some tunable parameter we use to scale the gradient if it is too big, so that we don't overshoot the minumum of the cost function

We can do the same thing with the bias bee seeing that

$$\delta^L = \frac{\partial C}{\partial b^L}\frac{\partial b^L}{\partial z^L}$$

So now we can also update the biases

$$b^l \leftarrow b^l - \eta\delta^l$$

In summation, we have four equations we use to find the error of each layer, and update the weights and biases

$$\delta^L = f'(z^L) \odot \nabla_{a^L} C$$
$$\delta^l = (\delta^{l+1}(W^{l+1})^T) \odot f'(z^l)$$
$$W^l \leftarrow W^l - \eta(a^{l-1})^T\delta^l$$
$$b^l \leftarrow b^l - \eta\delta^l$$

Now we can soon start writing a neural network. The last thing we need to do is decide what task we shall set our network to perform, define the appropriate activation functions and cost funtion, and their derivatives.

I could spend a lot of time discussing which activation functions we'll use, and what cost function, but I'll make another notebook for that (I'll link it here when it's done). So I hereby proclaim that we use the softmax function as the activation function in the output layer, and categorical cross entropy as the cost function. Why? We'll use the famous (and frankly overused) MNIST dataset to do some multinomial classification. The reason for this is that it's super easy to get the data on any system (Windows, Linux, etc): we'll just import it from ScikitLearn's datasets module. Also: Read the other notebook when it comes out. Let's define these functions

$$f(z^L) = \frac{e^{z^L}}{\sum_{m=1}^{K} e^{z_m^L}}$$

where f is the softmax function

$$C(W) = -\sum_{i=1}^{n} y_i \log a_i^L$$

and C is the cost function

Now we COULD go through the whole process of evaluating $f'(z^L) \odot \frac{\partial C}{\partial a^L}$ for these functions, but I'll leave that for another time, and just show you the result

$$f'(z^L) \odot \nabla_{a^L} C = a^L - y$$

A little note before we press on to the actual coding: We can (and will) forgo the whole analytical evaluation of the derivatives and just use autograd. This is because the derivative of the activation function (sometimes, and in the case of softmax) has to be evaluated as a Jacobian (example:https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1 (https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1)), i.e. we have to evaluate

$$\mathcal{J}(f(x)) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

So that δ^L becomes

$$\delta^L = \mathcal{J}(f(z^L)) \cdot \nabla_{a^L} C$$

This can be demonstrated by a few lines of code (Remember to install Autograd if you want to try to run the code for yourself):

In [2]:
```
#pip install autograd
```

```
Collecting autograd
  Using cached autograd-1.3-py3-none-any.whl
Collecting future>=0.15.2
  Using cached future-0.18.2-py3-none-any.whl
Requirement already satisfied: numpy>=1.12 in /srv/conda/envs/notebook/li
b/python3.9/site-packages (from autograd) (1.21.2)
Installing collected packages: future, autograd
Successfully installed autograd-1.3 future-0.18.2
Note: you may need to restart the kernel to use updated packages.
```

In [20]:
```
import autograd.numpy as np
from autograd import jacobian
from autograd import elementwise_grad as egrad

#Softmax activation function
def softmax(z):
    return np.exp(z)/np.sum(np.exp(z))
#Categorical cross-entroy cost function
def C(a, y):
    return -np.sum(y*np.log(a))

#Random input from previous layer with values in [0, 1)
z = np.random.random(10)
#Activation
a = softmax(z)
#Target
y = np.zeros(10)
y[3] = 1

#Calculating Jacobian
dsdz = jacobian(softmax)
#"Regular" derivative
dcda = egrad(C, 0)

#Numerically calculated delta^L
delta_num = dsdz(z) @ dcda(a, y)
#Analytical solution
delta_an = a-y
#Absolute difference between numerical and analytical solution
print(np.abs(delta_num-delta_an))
```

```
[0.00000000e+00 1.38777878e-17 1.38777878e-17 2.22044605e-16
 6.93889390e-18 1.38777878e-17 1.38777878e-17 1.38777878e-17
 1.38777878e-17 1.38777878e-17]
```

We have a more than acceptable numerical precision, with errors being between 0 and $\sim 10^{-16}$

With other activation- and cost functions, this is not necessary. Let's take an example from regression using NN's. It is common to use the mean squared error as cost function, and let the activation in the output be linear ($f(z^L) = z^L$). In this case we don't need to calculate the Jacobian, because (obviously) $f'(z^L) = 1$. But we can still demonstrate that

$$\mathcal{J}(f(z^L)) \cdot \nabla_{a^L} C = f'(z^L) \odot \nabla_{a^L} C$$

In [21]:
```
def lin(x):
    return x
def MSE(a, y):
    return np.mean(y-a)

y = np.random.randn(10)

#Calculating Jacobian
dfdx = jacobian(lin)(z)
#Linear activation
a = lin(z)
#Differentiating cost function
dmdx = egrad(MSE)(a, y)

#Jacobian times derivative of MSE
print(dfdx @ dmdx)

#"Regular" Hadamard product
dfdx = egrad(lin)(z)

print(dfdx*dmdx)
```

```
[-0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1]
[-0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1 -0.1]
```

Mathematically this can be shown as

$$\mathcal{J}(f(z^L)) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & 0 & \cdots & 0 \\ 0 & \frac{\partial f_2}{\partial x_2} & \cdots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

So it is easy to see that also in this case

$$\mathcal{J}(f(z^L)) \cdot \nabla_{a^L} C = f'(z^L) \odot \nabla_{a^L} C$$

because

$$\mathcal{J}(f(x)) \cdot \frac{\partial C}{\partial a^L} = \begin{pmatrix} \frac{\partial f_1}{\partial z_1^L} & 0 & \cdots & 0 \\ 0 & \frac{\partial f_2}{\partial z_2^L} & \cdots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & \frac{\partial f_n}{\partial z_n^L} \end{pmatrix} \cdot \left[\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_2^L}, \cdots, \frac{\partial C}{\partial a_n^L} \right]^T$$

$$= \left[\frac{\partial f_1}{\partial x_1} \frac{\partial C}{\partial a_1^L}, \cdots, \frac{\partial f_n}{\partial x_n} \frac{\partial C}{\partial a_n^L} \right]^T = f'(z^L) \odot \nabla_{a^L} C$$

So for ease of computation, we will use

$$\delta^L = \mathcal{J}(f(z^L)) \cdot \nabla_{a^L} C$$

Might might turn your nose up at this, because the Jacobian of a single variable function is just the derivative of the function, i.e. the jacobian $\mathcal{J}(f(x))$ where $f(x) = x$ is simply (1). But this is not what we are doing. Remember: z^L is a vector (so to speak. I really don't want to begin talking about vector spaces and all that now. Either way, there is nothing particularly rigorous about any of this regardless), so that $f(z^L)$ COULD be expressed as

$$f(z^L) = \sum_{i=1}^{n} z_i^L \hat{z}_i^L$$

Where

$$f_i = z_i^L \hat{z}_i^L$$

and $\hat{z}_i^L$ is a basis vector.

I hope you now (if not before) can see why the Jacobian then becomes

$$J(f(z^L)) = \begin{pmatrix} \frac{\partial f_1}{\partial z_1^L} & 0 & \cdots & 0 \\ 0 & \frac{\partial f_2}{\partial z_2^L} & \cdots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & \frac{\partial f_n}{\partial z_n^L} \end{pmatrix}$$

All of this Jacobian stuff, however, comes at a significant computational cost. Let's compare how much faster the analytical solution is compared to the numerical one:

In [22]:
```
#If not you don't have autograd.numpy, install with command "pip install au
from time import perf_counter

def speed_test():

    z = np.random.random(10)
    a = softmax(z)
    y = np.zeros(10)
    y[3] = 1

    t_num_start = perf_counter()
    dsdz = jacobian(softmax)
    dcda = egrad(C, 0)

    delta_num = dsdz(z) @ dcda(a, y)
    t_num_stop = perf_counter()

    t_an_start = perf_counter()
    delta_an = y-a
    t_an_stop = perf_counter()

    num_time = t_num_stop - t_num_start
    an_time = t_an_stop - t_an_start
    return num_time, an_time
```

We check how much faster the analytical solution is by performing the calculation 50 times, and

taking the average time.

In [23]:
```
s = 0
for i in range(50):
    t_num, t_an = speed_test()
    s += t_num/t_an
print(s/50)
```

```
1287.1321986218459
```

The analytical solution is roughly three orders of magnitude faster. This means that training a network using the Jacobian calculated by autograd will be significantly slower than if we were to use the analytical solution. The trade-off here is that we can now use which ever activation- cost function combination we desire, without having to do any analytical evaluation.

Now, I have cheated a bit, and I wrote the code for the neural network earlier. Here you go:

In [24]:
```
class NeuralNet:

    def __init__(self):

        #Lists for holding the weight, bias, etc, matrices/ vectors.
        #Call them (for the time being) "empty" tensors, if you're so incli
        self.layers = []
        self.act_funcs = []
        self.d_act_funcs = []
        self.weights = []
        self.biases = []
        self.Z = []
        self.A = []
        self.delta = []
    def add(self, n_neurons, act_func, input_size = None):
        """
        Sequantially adds layer to network in the order (in, hidden_1, ...,
        input size must be specified.
        """

        if isinstance(n_neurons, int) and n_neurons >= 1:
            self.layers.append(n_neurons)

        else:
            #Should be obvious to anyone attempting to use this class. Stil
            raise TypeError("n_neurons must be of type int and greater than

        if isinstance(input_size, int):
            self.weights.append(np.random.randn(input_size, n_neurons)*0.01


        elif isinstance(input_size, type(None)):
            self.weights.append(np.random.randn(self.layers[-2], n_neurons)
        #Errrrr
        else:
            raise TypeError("Errr")

        if isinstance(act_func, str):
            function = self.activation_function(act_func)
            self.act_funcs.append(function)
        else:
            raise TypeError("act_func argument must be of type str")

        #Making lists for holding the necessary vectors and matrices
        #Works OK, but not very "pretty"
        self.biases.append(np.random.randn(n_neurons)*0.01)
        self.A.append(0)
        self.Z.append(0)
        self.delta.append(0)

    def activation_function(self, act):
        """
        NOT DOC STRING.
        Note to self:
        Not happy with this method.
        """
```

```
        if act == "sigmoid":
            def activ(x):
                return 1/(1+np.exp(-x))

        elif act == "RELU":
            def activ(x):
                return np.maximum(x, 0)

        elif act == "leaky_RELU":
            def activ(x):
                return np.maximum(x, 0.01 * x)

        elif act == "softmax":
            def activ(x):
                return np.exp(x)/np.sum(np.exp(x))

        elif act == "linear":
            def activ(x):
                return x

        #Yes, formatting
        else:
            print("-------------------------------------")
            print(" ")
            print(str(act) + " is an invalid activation function name")
            print(" ")
            print("-------------------------------------")

            return

        return activ

    def loss_function(self, loss):
        """Under developement. Will be adding more loss functions."""

        if isinstance(loss, str):
            if loss == "MSE":
                def func(x, y):
                    return np.mean((x - y)**2, axis = 1, keepdims = True)

            elif loss == "categorical_cross":
                def func(x, y):
                    return -np.sum(y*np.log(x), axis = 1)

            else:
                raise ValueError("Invalid loss function name")

        else:
            raise TypeError("Loss function argument must be of type str")

        return func

    def feed_forward(self, X):

        #Feeding in feature matrix
        self.Z[0] = X @ self.weights[0] + self.biases[0].T
```

```
        #Activation in first hidden layer
        self.A[0] = self.act_funcs[0](self.Z[0])

        for i in range(1, len(self.weights)):
            #Feeding forward
            self.Z[i] = self.A[i-1] @ self.weights[i] + self.biases[i].T
            self.A[i] = self.act_funcs[i](self.Z[i])

    def diff(self, C, A):
        """
        Not sure this method is of any real use
        """
        dCda = egrad(C)
        dAdz = jacobian(A)

        return dCda, dAdz

    def back_prop(self, y, diff):

            #Assigning Jacobian and derivative functions as variables
            dC, da = diff
            #"Empty" (Zeros) array to hold Jacobian
            d_act = np.zeros(len(self.Z[-1]))
            #Empty array to hold derivative of cost function
            dcda = d_act.copy()
            #Empty array to hold delta^L
            self.delta[-1] = np.zeros((len(self.Z[-1]), self.layers[-1]))
            for i in range(len(self.Z[-1])):
                #Calculate Jacobian and derivative for each training exampl
                d_act = da(self.Z[-1][i])
                dcda = dC(self.A[-1][i], y)
                #Jacobian of activation times derivative of cost function (
                self.delta[-1][i] = d_act @ dcda

            for i in range(len(self.weights)-2, -1, -1):
                #Gradient of activation function of hidden layer i. No need
                dfdz = egrad(self.act_funcs[i])
                #Equation 2 is calculated in 2 parts. Just for ease of read
                t1 =  self.delta[i+1] @ self.weights[i+1].T
                self.delta[i] = np.multiply(t1, dfdz(self.Z[i]))

    def optimizer(self, X, eta):
        """
        For the moment only supports mini-batch SGD. More will come (maybe)
        """

        self.weights[0] -= eta * (X.T @ self.delta[0])
        self.biases[0] -= eta * np.sum(self.delta[0], axis = 0)

        for i in range(1, len(self.weights)):
            self.weights[i] -= eta * (self.A[i-1].T @ self.delta[i])
            self.biases[i] -= eta * np.sum(self.delta[i], axis = 0)

    def train(self, X, y, epochs, loss, metric, batch_size = 10, num_iters

        """
        Takes args: X (feature matrix), y (targets), and epochs (type int).
```

```
        Takes kwargs: batch_size, num_iters, eta_init, decay. The "standard
        has been found by testing on one dataset. You should probably not u
        """

        diff = self.diff(self.loss_function(loss), self.act_funcs[-1])

        data_indices = len(X)
        #eta function (not the Dirichlet one): for decreasing learning rate
        eta = lambda eta_init, iteration, decay: eta_init/(1+decay*iteratio

        for i in range(1, epochs+1):

            for j in range(num_iters):
                eta1 = eta(eta_init, j, decay)
                #Randomly choose datapoints to use as mini-batches
                chosen_datapoints = np.random.choice(data_indices, size = b
                #Making mini-batches
                X_mini = X[chosen_datapoints]
                y_mini = y[chosen_datapoints]
                #Feed forward
                self.feed_forward(X_mini)
                #Backprop
                self.back_prop(y_mini, diff)
                #Update weights and biases
                self.optimizer(X_mini, eta(eta_init, j, decay))

            #Make a prediction and print mean of performance of mini-batch
            predicted = self.predict(X_mini)
            performance = self.metrics(predicted, y_mini, metric)
            print(metric +" is " + str(np.mean(performance))+ " at epcoh "


    def metrics(self, y_hat, y, a):
        """
        Takes args: y_hat, y, a (prediction, targets, activation in layer L
        """
        if a == "accuracy":
            s = 0
            for i in range(len(y)):
                true = np.argmax(y[i])
                pred = np.argmax(y_hat[i])
                if true == pred:
                    s += 1
                else:
                    continue

            return s/len(y_hat)

        elif a == "MSE":
            return np.mean((y-y_hat)**2, axis = 0)


    def predict(self, X):
        """
        Takes arg: X
        Does one feed forward pass and returns the output of last layer
        """
```

```
        self.feed_forward(X)
        return self.A[-1]
```

Now, let's prepare the data and train the network

In [31]:
```
from sklearn import datasets
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def fix_data():
    # download MNIST dataset
    digits = datasets.load_digits()

    # define inputs and labels
    inputs = digits.images
    labels = digits.target

    # one-hot encoding the targest
    def to_categorical_numpy(integer_vector):
        n_inputs = len(integer_vector)
        n_categories = np.max(integer_vector) + 1
        onehot_vector = np.zeros((n_inputs, n_categories))
        onehot_vector[range(n_inputs), integer_vector] = 1

        return onehot_vector


    n_inputs = len(inputs)
    inputs = inputs.reshape(n_inputs, -1)
    X = inputs
    Y = to_categorical_numpy(labels)
    return X, Y

X, Y = fix_data()

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.2 )

in_size = len(X[0])
out_size = len(y_test[0])

Net = NeuralNet()
#We don't need anything fancy for this demonstration. 1 hidden layer is eno
Net.add(in_size, "sigmoid", input_size = in_size)
Net.add(10, "softmax")

t_train_start = perf_counter()

#We're leaving batch size at a modest 10 as to not having to spend all day
Net.train(X_train, y_train, 100, "categorical_cross", "accuracy", batch_siz

t_train_stop = perf_counter()

print(t_train_stop-t_train_start)

#Reminder of args and kwargs
#train(self, X, y, epochs, loss, metric, batch_size = 10, num_iters = 50, e


pred = Net.predict(X_test)
s = 0
```

```
for i in range(len(X_test)):
    true = np.argmax(y_test[i])
    guess = np.argmax(pred[i])
    if true == guess:
        s += 1
print("test accuracy is " + str(s/len(y_test)))
```

```
accuracy is 0.2 at epcoh 1
accuracy is 0.1 at epcoh 2
accuracy is 0.3 at epcoh 3
accuracy is 0.2 at epcoh 4
accuracy is 0.1 at epcoh 5
accuracy is 0.1 at epcoh 6
accuracy is 0.3 at epcoh 7
accuracy is 0.6 at epcoh 8
accuracy is 0.4 at epcoh 9
accuracy is 0.6 at epcoh 10
accuracy is 0.3 at epcoh 11
accuracy is 0.2 at epcoh 12
accuracy is 0.7 at epcoh 13
accuracy is 0.8 at epcoh 14
accuracy is 0.6 at epcoh 15
accuracy is 0.7 at epcoh 16
accuracy is 0.3 at epcoh 17
accuracy is 0.8 at epcoh 18
accuracy is 0.5 at epcoh 19
accuracy is 0.9 at epcoh 20
accuracy is 1.0 at epcoh 21
accuracy is 0.9 at epcoh 22
accuracy is 0.6 at epcoh 23
accuracy is 0.9 at epcoh 24
accuracy is 0.8 at epcoh 25
accuracy is 1.0 at epcoh 26
accuracy is 0.9 at epcoh 27
accuracy is 0.9 at epcoh 28
accuracy is 0.9 at epcoh 29
accuracy is 0.7 at epcoh 30
accuracy is 0.9 at epcoh 31
accuracy is 0.7 at epcoh 32
accuracy is 0.8 at epcoh 33
accuracy is 0.9 at epcoh 34
accuracy is 0.7 at epcoh 35
accuracy is 0.8 at epcoh 36
accuracy is 0.9 at epcoh 37
accuracy is 0.7 at epcoh 38
accuracy is 1.0 at epcoh 39
accuracy is 0.8 at epcoh 40
accuracy is 1.0 at epcoh 41
accuracy is 0.7 at epcoh 42
accuracy is 1.0 at epcoh 43
accuracy is 1.0 at epcoh 44
accuracy is 0.9 at epcoh 45
accuracy is 0.9 at epcoh 46
accuracy is 1.0 at epcoh 47
accuracy is 1.0 at epcoh 48
accuracy is 0.9 at epcoh 49
accuracy is 1.0 at epcoh 50
accuracy is 1.0 at epcoh 51
```

```
accuracy is 1.0 at epcoh 52
accuracy is 0.8 at epcoh 53
accuracy is 0.9 at epcoh 54
accuracy is 1.0 at epcoh 55
accuracy is 1.0 at epcoh 56
accuracy is 0.9 at epcoh 57
accuracy is 0.9 at epcoh 58
accuracy is 1.0 at epcoh 59
accuracy is 1.0 at epcoh 60
accuracy is 0.8 at epcoh 61
accuracy is 1.0 at epcoh 62
accuracy is 0.9 at epcoh 63
accuracy is 0.9 at epcoh 64
accuracy is 1.0 at epcoh 65
accuracy is 1.0 at epcoh 66
accuracy is 1.0 at epcoh 67
accuracy is 1.0 at epcoh 68
accuracy is 0.9 at epcoh 69
accuracy is 1.0 at epcoh 70
accuracy is 0.9 at epcoh 71
accuracy is 0.9 at epcoh 72
accuracy is 1.0 at epcoh 73
accuracy is 1.0 at epcoh 74
accuracy is 0.9 at epcoh 75
accuracy is 1.0 at epcoh 76
accuracy is 1.0 at epcoh 77
accuracy is 0.9 at epcoh 78
accuracy is 1.0 at epcoh 79
accuracy is 0.9 at epcoh 80
accuracy is 1.0 at epcoh 81
accuracy is 1.0 at epcoh 82
accuracy is 0.9 at epcoh 83
accuracy is 1.0 at epcoh 84
accuracy is 1.0 at epcoh 85
accuracy is 1.0 at epcoh 86
accuracy is 0.8 at epcoh 87
accuracy is 1.0 at epcoh 88
accuracy is 1.0 at epcoh 89
accuracy is 1.0 at epcoh 90
accuracy is 1.0 at epcoh 91
accuracy is 1.0 at epcoh 92
accuracy is 1.0 at epcoh 93
accuracy is 0.9 at epcoh 94
accuracy is 1.0 at epcoh 95
accuracy is 1.0 at epcoh 96
accuracy is 1.0 at epcoh 97
accuracy is 1.0 at epcoh 98
accuracy is 1.0 at epcoh 99
accuracy is 1.0 at epcoh 100
287.3286320306361
test accuracy is 0.9416666666666667
```

We get a good accuracy (94%), but this takes forever (287s). There are of course stuff that could be done to improve speed of execution, but let´s just do the same with Tensorflow:

In [32]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense#, Flatten

model = Sequential()
model.add(Dense(units = 512, activation = "sigmoid", input_dim = len(X_trai
model.add(Dense(units = 512, activation  = "sigmoid"))
model.add(Dense(units = 10, activation = "softmax"))

model.compile(loss = "categorical_crossentropy", optimizer = "sgd", metrics
t_train_start_tf = perf_counter()
model.fit(X_train, y_train, epochs = 400, batch_size = 32)
t_train_stop_tf = perf_counter()
```

```
Epoch 1/400
45/45 [==============================] - 0s 2ms/step - loss: 2.3282 - acc
uracy: 0.1364
Epoch 2/400
45/45 [==============================] - 0s 2ms/step - loss: 2.1682 - acc
uracy: 0.3009
Epoch 3/400
45/45 [==============================] - 0s 2ms/step - loss: 2.0375 - acc
uracy: 0.5451
Epoch 4/400
45/45 [==============================] - 0s 2ms/step - loss: 1.9216 - acc
uracy: 0.6396
Epoch 5/400
45/45 [==============================] - 0s 2ms/step - loss: 1.7836 - acc
uracy: 0.7560
Epoch 6/400
45/45 [==============================] - 0s 2ms/step - loss: 1.6607 - acc
uracy: 0.8047
Epoch 7/400
```

In [33]:
```
print(t_train_stop_tf-t_train_start_tf)
y_hat = model.predict(X_test)

s = 0
for i in range(len(X_test)):
    true = np.argmax(y_test[i])
    guess = np.argmax(y_hat[i])
    if true == guess:
        s += 1
print("test accuracy is " + str(s/len(y_test)))
```

```
39.43308077007532
test accuracy is 0.9666666666666667
```

Wow, better accuracy, and speed. Not to mention it took only a couple of minutes coocking op the Tensorflow code, whereas my ffnn took a bit longer.
This may seem like an exercise of futility, but I believe it´s good to know how NNs work and how to code them (albeit primitively), before you start using Tensorflow or something similar.

Also please do excuse any typo in this notebook. There is no spellcheck here.

Peace out!

In [ ]: