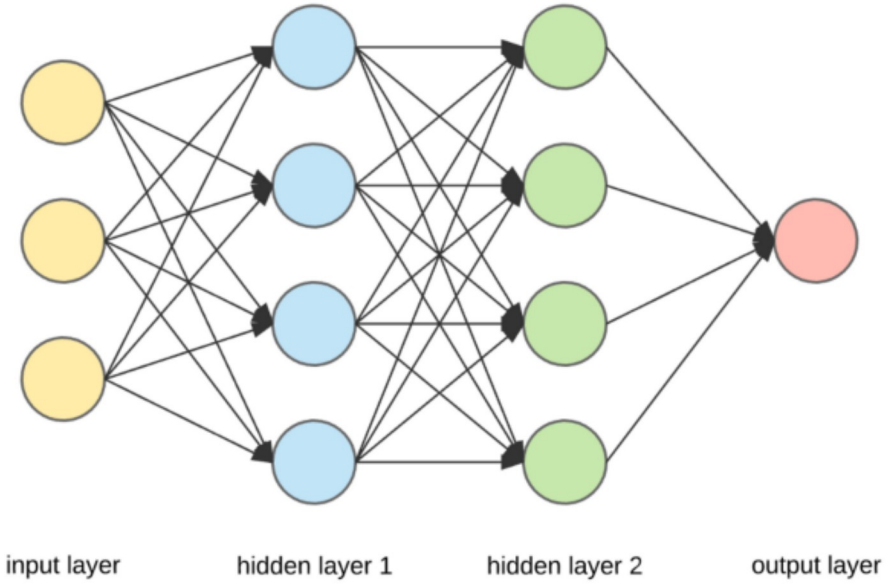
Neural Network Architecture

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Neural Networks

Neural networks are inspired by the human brain and are a powerful tool in machine learning. Here's a simplified breakdown of how they work, with a sprinkle of math for flavor.

In [8]:
```
img("./nn.png", d = 120)
```



Dataset : https://www.kaggle.com/competitions/digit-recognizer MNIST ("Modified National Institute of Standards and Technology") is the de facto "hello world" dataset of computer vision. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine learning techniques emerge, MNIST remains a reliable resource for researchers and learners alike

In [57]:
```
df = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')
```

In [58]:
```
df.head()
```

Out[58]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel7
0	1	0	0	0	0	0	0	0	0	0	...	
1	0	0	0	0	0	0	0	0	0	0	...	
2	1	0	0	0	0	0	0	0	0	0	...	
3	4	0	0	0	0	0	0	0	0	0	...	
4	0	0	0	0	0	0	0	0	0	0	...	

5 rows × 785 columns

In [59]:
```
data = np.array(df)
```

In [60]:
```
m, n = data.shape
```

In [61]:
```
np.random.shuffle(data)
data = data.T
```

In [62]:
```
X, y = data[1:m], data[0]
```

In [63]:
```
X = X/255.
```

Parameter initializer function

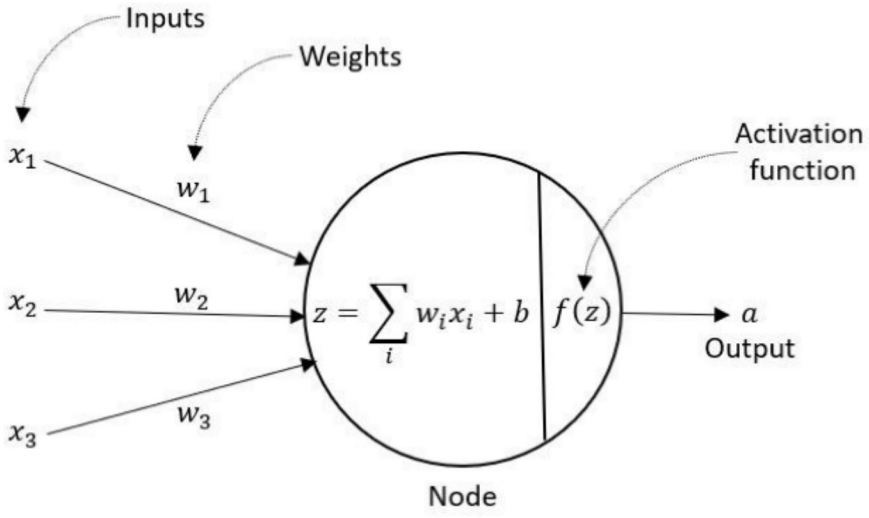
In [64]:
```
def init_params(factor):
    W1 = (np.random.rand(10, 784) - 0.5)*factor
    W2 = (np.random.rand(10, 10) - 0.5)*factor
    b1 = (np.random.rand(10,1) - 0.5)*factor
    b2 = (np.random.rand(10,1) - 0.5)*factor
    return W1, W2, b1, b2
```

In [66]:
```
def softmax(a):
    return np.exp(a) / sum(np.exp(a))
```

Function for Forward Propagation

Imagine a neuron as a simple processing unit. It receives inputs (x1, x2, etc.) from other neurons or the external world. These inputs are multiplied by weights (w1, w2, etc.) that act like control knobs, determining the influence of each input. A bias (b) is added, and everything is summed. Finally, an activation function (f) applies a non-linear transformation, introducing a threshold for firing. Here's the equation:

In [9]:
```
img("./nn_maths_1.jpg", 130)
```



This "firing" output (y) is then sent to other neurons in the network. The activation function ensures the network isn't just a linear mess, allowing it to learn complex patterns.

In [65]:
```
def relu(x):
    return np.maximum(x,0)
```

In [67]:
```
def forward_pass(W1, W2, b1, b2, X):
    z1 = W1.dot(X) + b1
    a1 = relu(z1)
    z2 = W2.dot(a1) + b2
    a2 = softmax(z2)
    return a1, a2, z1, z2
```

Neurons are organized in layers. The first layer takes raw data as input, the middle layers (often called hidden layers) process it, and the final layer produces the output. Information flows forward through the network, with each layer's output becoming the next layer's input.

In [68]:
```
def one_hot_encode(y):
    y_ = np.zeros((y.size, y.max()+1))
    y_[np.arange(y.size), y] = 1
    return y_.T
```

In [69]:
```
def d_relu(x):
    return x>0
```

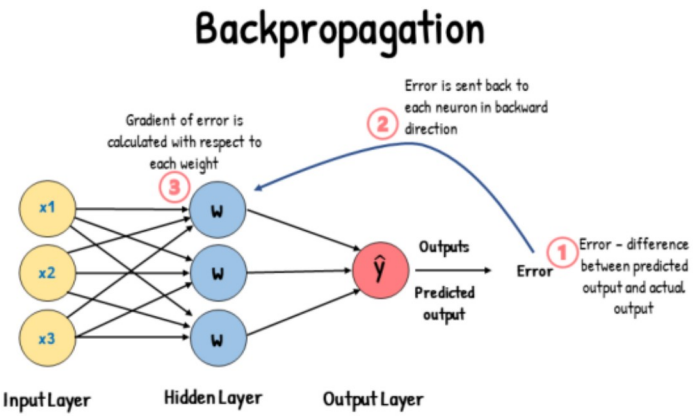
In [70]:
```
def predict(a2):
    return np.argmax(a2, 0)

def accuracy(pred, true):
    return np.sum(pred == true)/m
```

Function for Backward Propagation

Backpropagation trains neural networks by iteratively adjusting internal connections (weights) to minimize errors. Imagine a game where you adjust dials (weights) based on the distance from a target (error). By backpropagating the error through the network, the adjustments become fine-tuned, enabling the network to learn complex patterns.

In [10]:
```
img("./grad_descent.png", 140)
```



In [71]:
```
def back_pass(a1, a2, z1, z2, W2, X, y):
    Y = one_hot_encode(y)
    dz2 = a2 - Y
    dW2 = 1/m*dz2.dot(a1.T)
    db2 = 1/m*np.sum(dz2)
    dz1 = W2.T.dot(dz2) * d_relu(z1)
    dW1 = 1/m*dz1.dot(X.T)
    db1 = 1/m*np.sum(dz1)

    return dW1, dW2, db1, db2
```

In [72]:
```
def update_params(W1, W2, b1, b2, dW1, dW2, db1, db2, alpha):
    W1 = W1 - (alpha*dW1)
    W2 = W2 - (alpha*dW2)
    b1 = b1 - (alpha*db1)
    b2 = b2 - (alpha*db2)
    return W1, W2, b1, b2
```

Function of gradient descent

In [73]:
```
def train(training_data, labels, iters = 500, alp=.1, factor = 1):
    accuracy_timeline = []
    W1, W2, b1, b2 = init_params(factor)
    alp = .10
    for i in range(iters+1):
        alp = alp
        a1, a2, z1, z2 = forward_pass(W1, W2, b1, b2, training_data)
        dW1, dW2, db1, db2 = back_pass(a1, a2, z1, z2, W2, training_data, labels
        W1, W2, b1, b2 = update_params(W1, W2, b1, b2, dW1, dW2, db1, db2, alpha

        if i%50 == 0:
            print(f"Iteration {i}/{iters}  ---  accuracy:{accuracy(predict(a2),
        if i%5 == 0:
            accuracy_timeline.append(accuracy(predict(a2), labels))
    return accuracy_timeline
```

Training various models and comparing them

In [104…
```
m_str = "weights at start: {ran},   alpha: {alpha}"
models = []
```

In [105…
```
model0= train(X, y, alp = .1, factor = 2)
models.append(m_str.format(ran= "-1 to 1", alpha=0.1))
```

```
Iteration 0/500  ---  accuracy:0.072
Iteration 50/500  ---  accuracy:0.20714285714285716
Iteration 100/500  ---  accuracy:0.2992142857142857
Iteration 150/500  ---  accuracy:0.382547619047619
Iteration 200/500  ---  accuracy:0.445547619047619
Iteration 250/500  ---  accuracy:0.49585714285714283
Iteration 300/500  ---  accuracy:0.5461666666666667
Iteration 350/500  ---  accuracy:0.5896904761904762
Iteration 400/500  ---  accuracy:0.6273809523809524
Iteration 450/500  ---  accuracy:0.6296666666666667
Iteration 500/500  ---  accuracy:0.6575952380952381
```

In [106…
```
model1= train(X, y, alp = .1, factor = 1)
models.append(m_str.format(ran= "-0.5 to 0.5", alpha=0.1))
```

```
Iteration 0/500  ---  accuracy:0.13566666666666666
Iteration 50/500  ---  accuracy:0.4845952380952381
Iteration 100/500  ---  accuracy:0.6532619047619047
Iteration 150/500  ---  accuracy:0.7334285714285714
Iteration 200/500  ---  accuracy:0.7717857142857143
Iteration 250/500  ---  accuracy:0.7954523809523809
Iteration 300/500  ---  accuracy:0.8113095238095238
Iteration 350/500  ---  accuracy:0.8233333333333334
Iteration 400/500  ---  accuracy:0.8329285714285715
Iteration 450/500  ---  accuracy:0.8390952380952381
Iteration 500/500  ---  accuracy:0.846
```

In [107…
```
model2= train(X, y, alp = .1, factor = 0.5)
models.append(m_str.format(ran= "-0.25 to 0.25", alpha=0.1))
```

```
Iteration 0/500  ---  accuracy:0.098
Iteration 50/500  ---  accuracy:0.4662142857142857
Iteration 100/500  ---  accuracy:0.6476190476190476
Iteration 150/500  ---  accuracy:0.7505238095238095
Iteration 200/500  ---  accuracy:0.8012857142857143
Iteration 250/500  ---  accuracy:0.8267380952380953
Iteration 300/500  ---  accuracy:0.8422142857142857
Iteration 350/500  ---  accuracy:0.8525952380952381
Iteration 400/500  ---  accuracy:0.8593333333333333
Iteration 450/500  ---  accuracy:0.8665714285714285
Iteration 500/500  ---  accuracy:0.8721904761904762
```

In [108…
```
model3= train(X, y, alp = .2, factor = 2)
models.append(m_str.format(ran= "-1 to 1", alpha=0.2))
```

```
Iteration 0/500  ---  accuracy:0.15854761904761905
Iteration 50/500  ---  accuracy:0.371452380952381
Iteration 100/500  ---  accuracy:0.4553333333333333
Iteration 150/500  ---  accuracy:0.5299047619047619
Iteration 200/500  ---  accuracy:0.5968333333333333
Iteration 250/500  ---  accuracy:0.6437619047619048
Iteration 300/500  ---  accuracy:0.6756428571428571
Iteration 350/500  ---  accuracy:0.6989047619047619
Iteration 400/500  ---  accuracy:0.7179523809523809
Iteration 450/500  ---  accuracy:0.7351666666666666
Iteration 500/500  ---  accuracy:0.7499285714285714
```

In [109…
```
model4= train(X, y, alp = .2, factor = 1)
models.append(m_str.format(ran= "-0.5 to 0.5", alpha=0.2))
```

```
Iteration 0/500  ---  accuracy:0.16204761904761905
Iteration 50/500  ---  accuracy:0.5036190476190476
Iteration 100/500  ---  accuracy:0.6666666666666666
Iteration 150/500  ---  accuracy:0.7406428571428572
Iteration 200/500  ---  accuracy:0.7776904761904762
Iteration 250/500  ---  accuracy:0.8024761904761905
Iteration 300/500  ---  accuracy:0.8190714285714286
Iteration 350/500  ---  accuracy:0.8318571428571429
Iteration 400/500  ---  accuracy:0.841
Iteration 450/500  ---  accuracy:0.8491904761904762
Iteration 500/500  ---  accuracy:0.8543571428571428
```

In [110…
```
model5= train(X, y, alp = .2, factor = 0.5)
models.append(m_str.format(ran= "-0.25 to 0.25", alpha=0.2))
```

```
Iteration 0/500  ---  accuracy:0.08561904761904762
Iteration 50/500  ---  accuracy:0.3093095238095238
Iteration 100/500  ---  accuracy:0.5697857142857143
Iteration 150/500  ---  accuracy:0.7477380952380952
Iteration 200/500  ---  accuracy:0.8071190476190476
Iteration 250/500  ---  accuracy:0.8314047619047619
Iteration 300/500  ---  accuracy:0.8460714285714286
Iteration 350/500  ---  accuracy:0.8562142857142857
Iteration 400/500  ---  accuracy:0.8643809523809524
Iteration 450/500  ---  accuracy:0.8706904761904762
Iteration 500/500  ---  accuracy:0.8748571428571429
```

Plotting the results of training

On observing the following graphs:

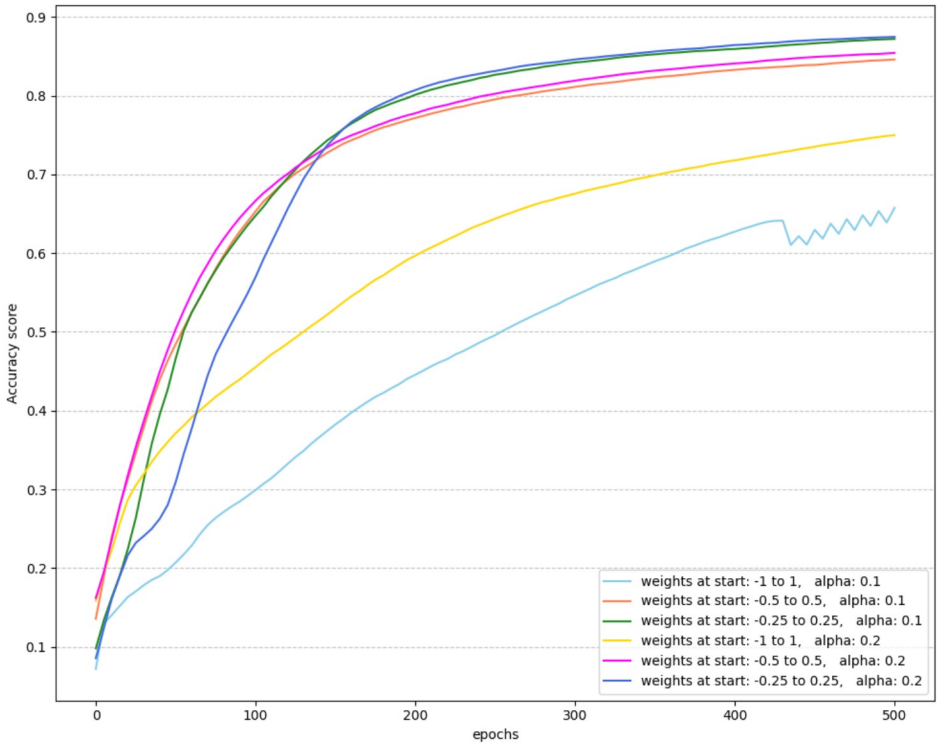
- Weights initialized closer to 0 produce much better results
- Larger aplha values perform significantly worse than all other models
- Larger learning rate helped the models during initial phase of the training a lot
- A lower learning rate leads to better accuracy at longer training sessions
- After around 200 epochs most models only got marginal improvements in accuracy
- The dark blue model had a period of deviation from the trends during the intial training epochs
- The skyblue model during the ultimate training epochs had unstable results
- Weight decay seems like a good technique after observing the effects of alpha values at different stages of training.

In [111…

```
plt.figure(figsize=(10, 8))
epoch = np.arange(0, 501, 5)
colors = ['skyblue', 'coral', 'forestgreen', 'gold', 'magenta', 'royalblue']
plt.plot(epoch, model0, label = models[0], color=colors[0])
plt.plot(epoch, model1, label = models[1], color=colors[1])
plt.plot(epoch, model2, label = models[2], color=colors[2])
plt.plot(epoch, model3, label = models[3], color=colors[3])
plt.plot(epoch, model4, label = models[4], color=colors[4])
plt.plot(epoch, model5, label = models[5], color=colors[5])

plt.xlabel("epochs")
plt.ylabel("Accuracy score")
plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

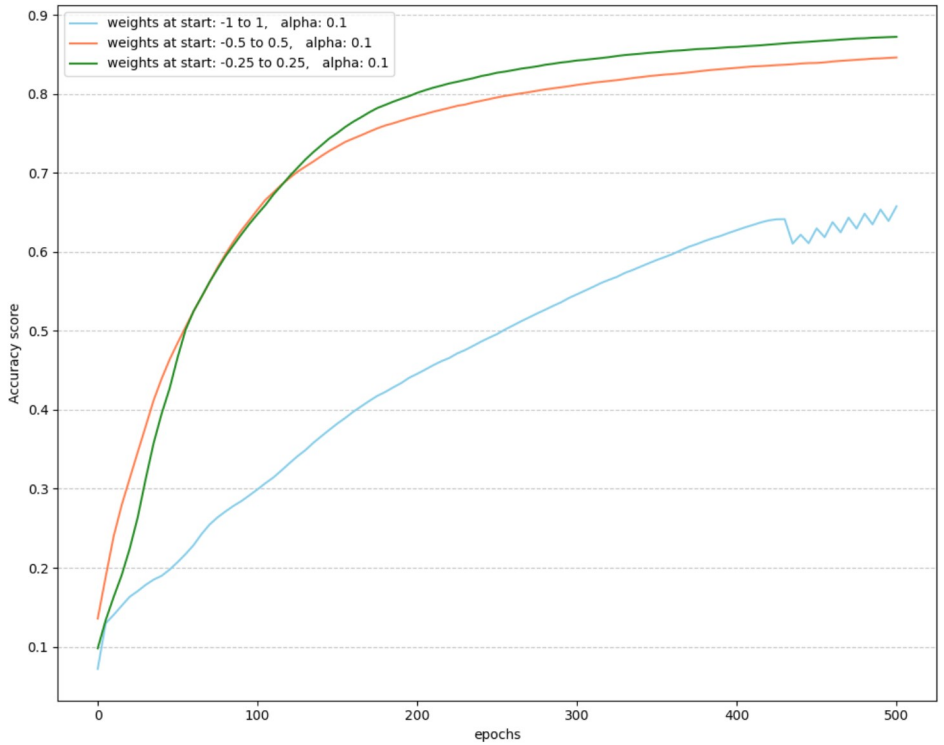


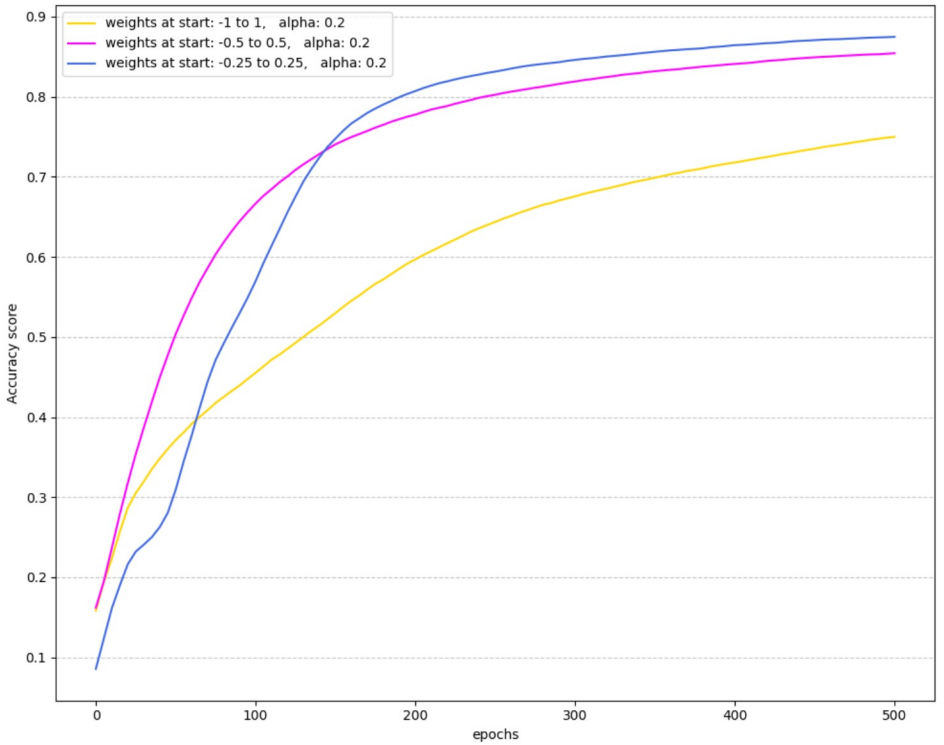
In [112…

```
plt.figure(figsize=(10, 8))
epoch = np.arange(0, 501, 5)
colors = ['skyblue', 'coral', 'forestgreen', 'gold', 'magenta', 'royalblue']
plt.plot(epoch, model0, label = models[0], color=colors[0])
plt.plot(epoch, model1, label = models[1], color=colors[1])
plt.plot(epoch, model2, label = models[2], color=colors[2])

plt.xlabel("epochs")
plt.ylabel("Accuracy score")
plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```





In [113…

```
plt.figure(figsize=(10, 8))
epoch = np.arange(0, 501, 5)
colors = ['skyblue', 'coral', 'forestgreen', 'gold', 'magenta', 'royalblue']
plt.plot(epoch, model3, label = models[3], color=colors[3])
plt.plot(epoch, model4, label = models[4], color=colors[4])
plt.plot(epoch, model5, label = models[5], color=colors[5])

plt.xlabel("epochs")
plt.ylabel("Accuracy score")
plt.legend()

plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

Thanks a lot for checking this out