Dirbtinių neuroninių tinklų dėstymas Moderniųjų technologijų matematikos studijų programoje

Vadimas Starikovičius

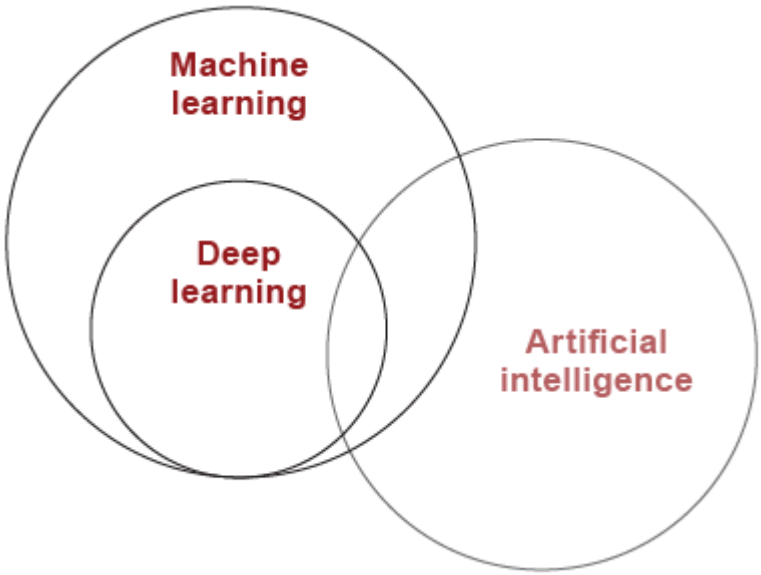
MMK Seminaras, 2020-12-22

Literatūra / Šaltiniai:

1. Michael A. Nielsen. "Neural Networks and Deep Learning", 2015. Free online book: http://neuralnetworksanddeeplearning.com/
2. Andrew W. Trask. "Grokking Deep Learning", Manning Publications, 2019.
3. 3Blue1Brown (youtube.com): Deep learning, chapters 1-4: http://3b1b.co/neural-networks

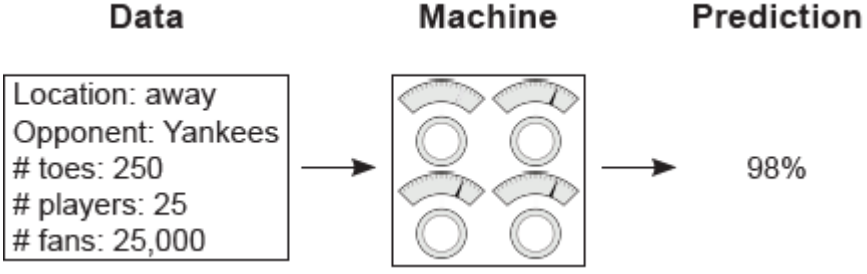
Deep learning

Deep learning is a subset of methods in the machine learning, primarily using artificial neural networks, which are a class of algorithm loosely inspired by the human brain.



Supervised parametric learning:

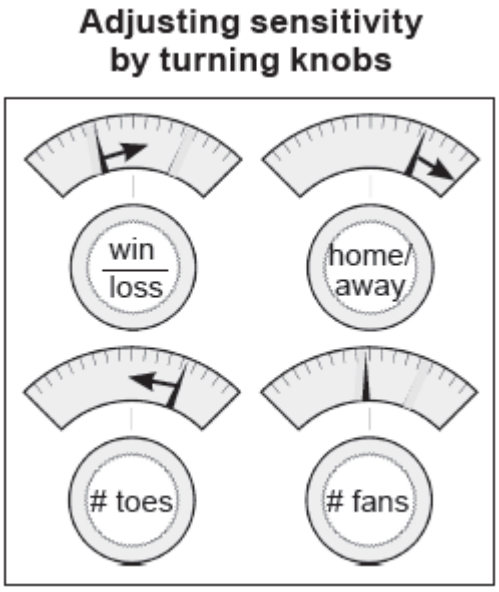
Step 1: Predict (forward propagation)



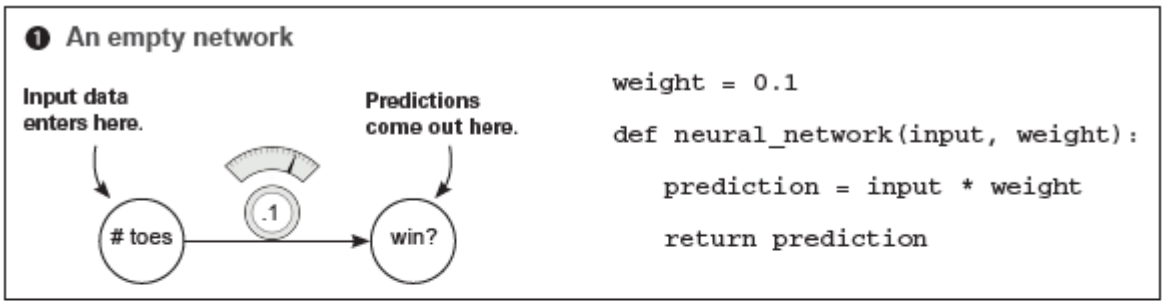
Step 2: Compare to the truth pattern (quatify the loss)

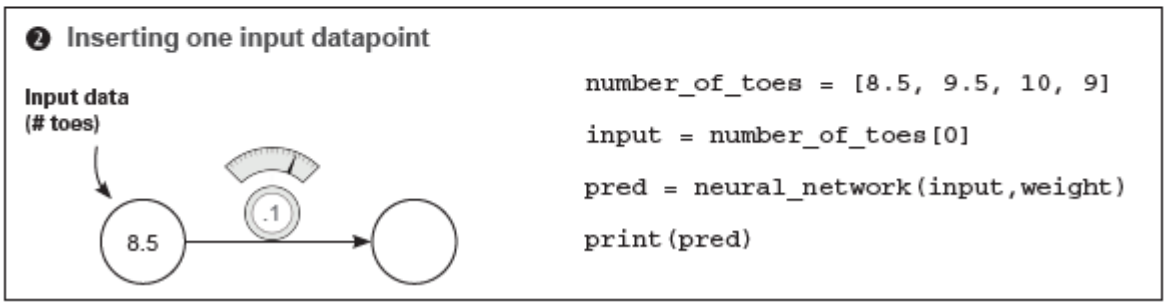
Pred: 98% > Truth: 0%

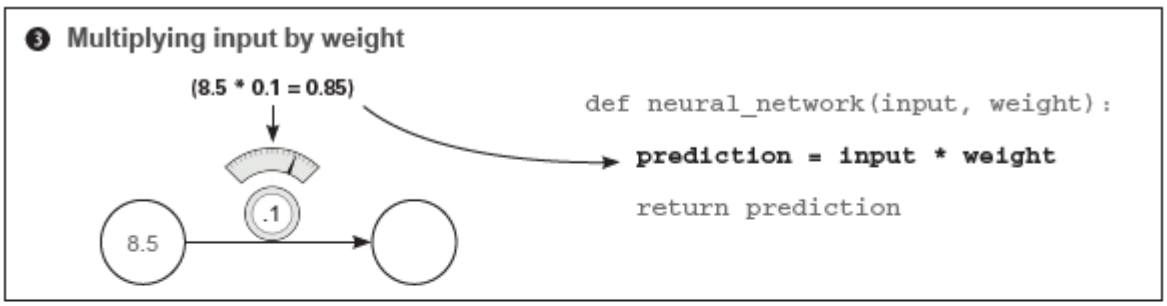
Step 3: Learn the pattern (train the network using gradient descent and backpropagation)

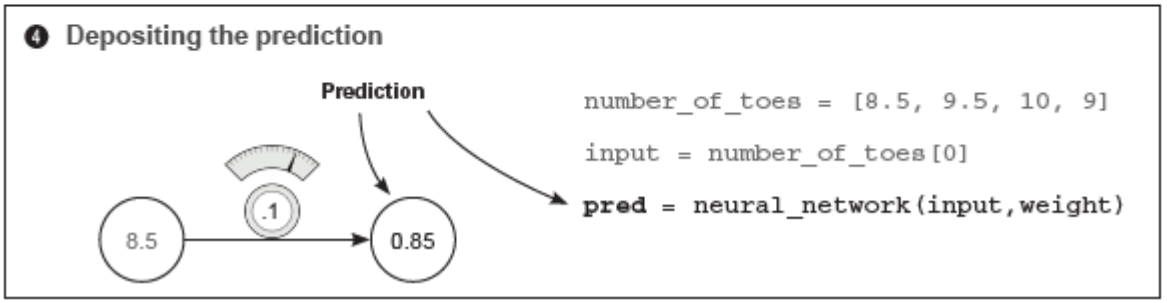


Example. A simple neural network making a prediction (Step 1).



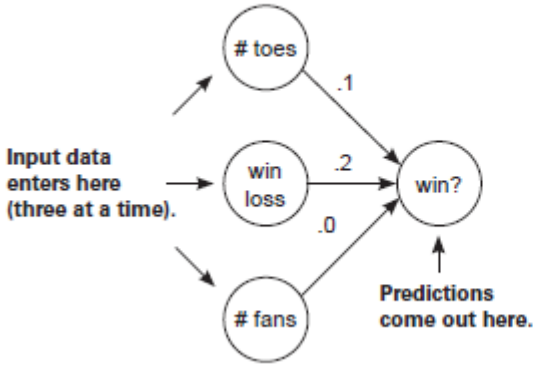






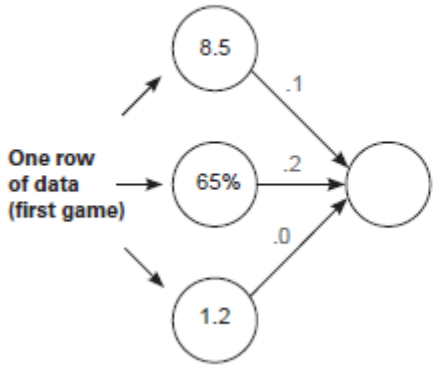
Making a prediction with multiple inputs

❶ An empty network with multiple inputs




```
weights = [0.1, 0.2, 0]

def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

❷ Inserting one input datapoint

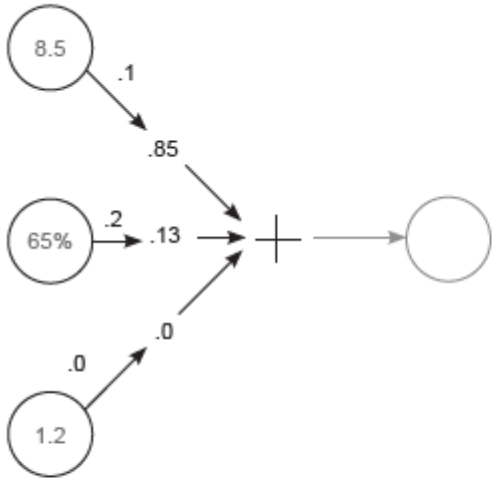


This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
nfans = fan count (in millions).

```
toes = [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)
```

Input corresponds to every entry for the first game of the season.

❸ Performing a weighted sum of inputs



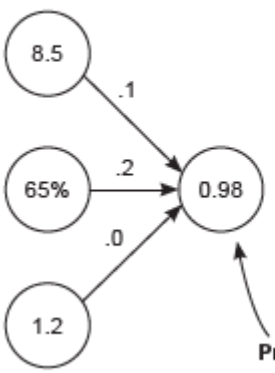

```
def w_sum(a,b):

    assert(len(a) == len(b))

    output = 0

    for i in range(len(a)):
        output += (a[i] * b[i])

    return output


def neural_network(input, weights):

    pred = w_sum(input,weights)

    return pred
```

Inputs		Weights		Local predictions	
(8.50	*	0.1)	=	0.85	= toes prediction
(0.65	*	0.2)	=	0.13	= wlrec prediction
(1.20	*	0.0)	=	0.00	= fans prediction

toes prediction + wlrec prediction + fans prediction = final prediction

0.85 + 0.13 + 0.00 = 0.98

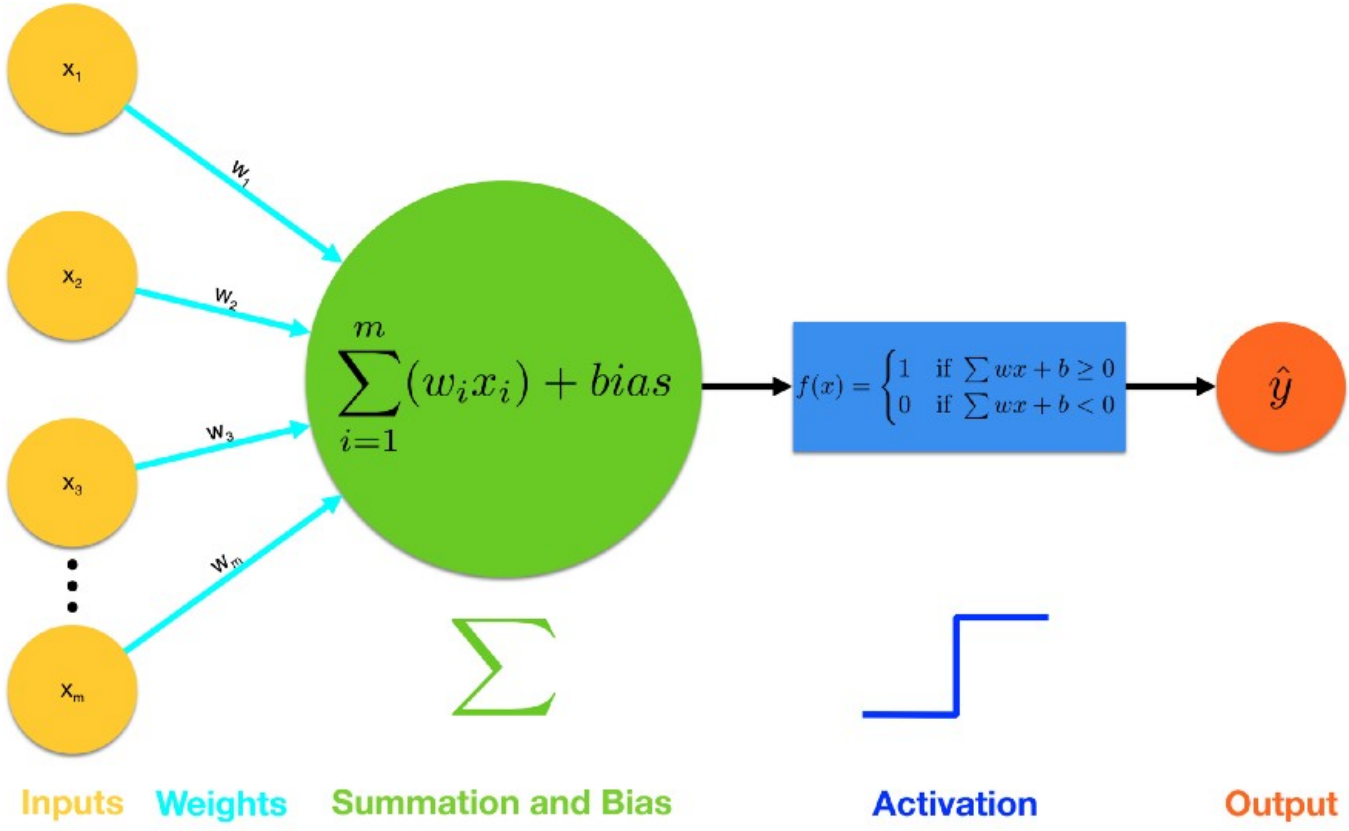
❹ Depositing the prediction



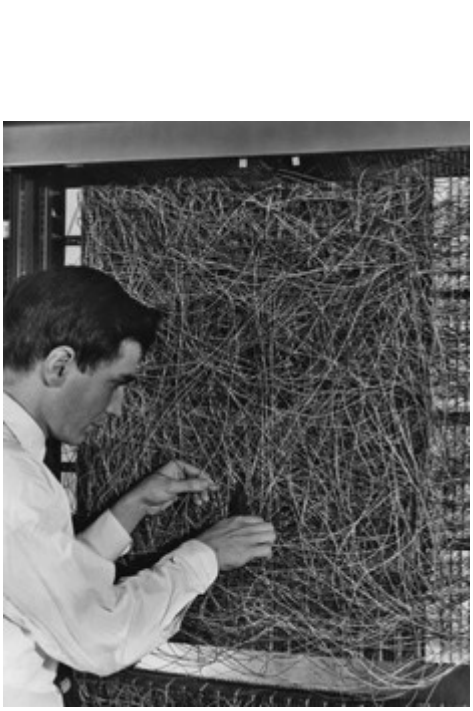
Input corresponds to every entry for the first game of the season.

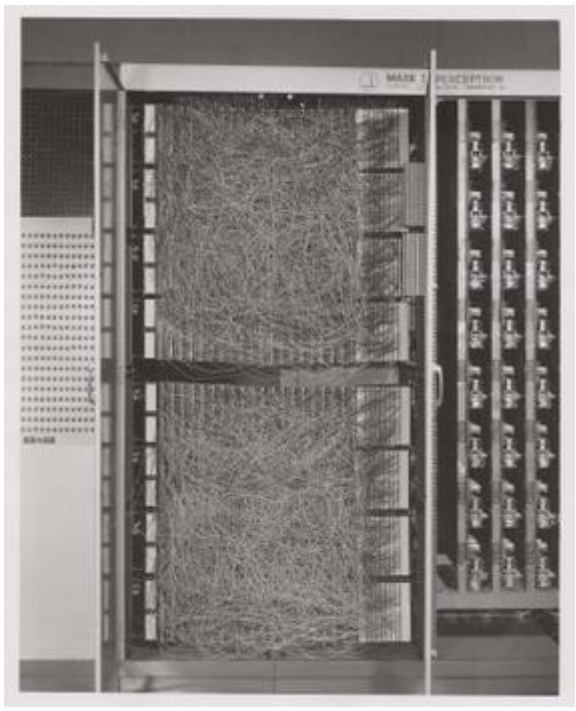
```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)

print(pred)
```

Perceptron (algorithm for supervised learning of binary classifiers, Frank Rosenblatt, 1958)



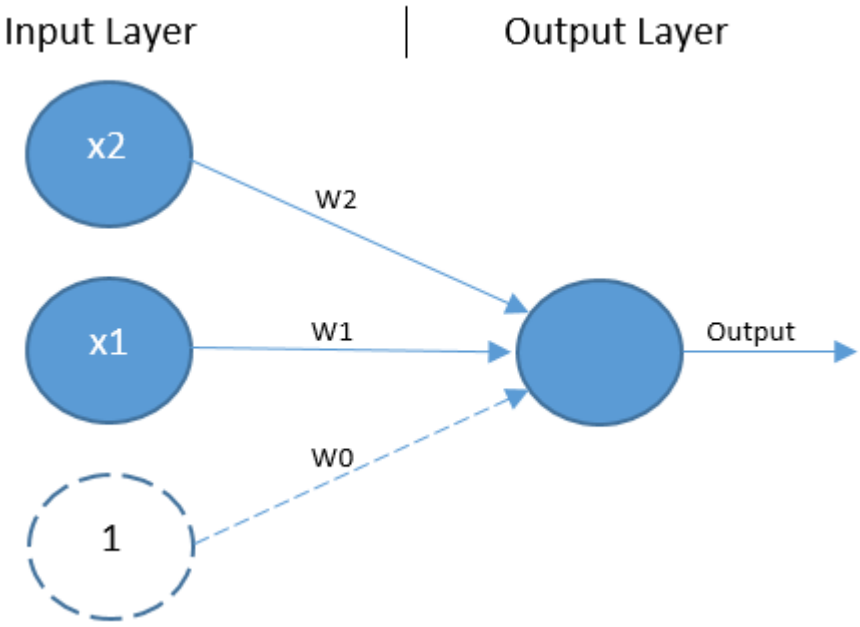
"Mark 1 perceptron". This machine was designed for image recognition.





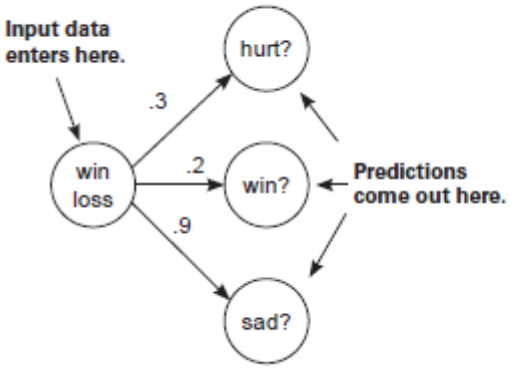
The XOR Problem (Marvin Minsky and Seymour Papert, 1969)

Input 1	Input 2	Output
0	0	0
0	1	1
1	1	0
1	0	1



Making a prediction with multiple outputs

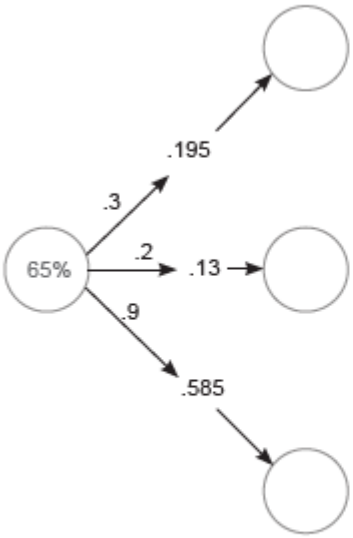
1 An empty network with multiple outputs



Instead of predicting just whether the team won or lost, you're also predicting whether the players are happy or sad and the percentage of team members who are hurt. You make this prediction using only the current win/loss record.

```
weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):
    pred = ele_mul(input,weights)
    return pred
```

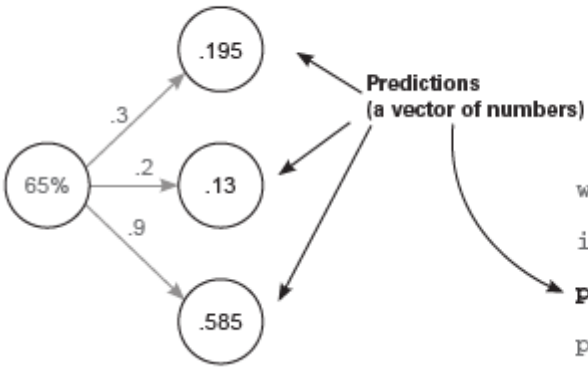
3 Performing elementwise multiplication




```
def ele_mul(number,vector):
    output = [0,0,0]
    assert(len(output) == len(vector))
    for i in range(len(vector)):
        output[i] = number * vector[i]
    return output

def neural_network(input, weights):
    pred = ele_mul(input,weights)
    return pred
```

Inputs		Weights		Final predictions	
(0.65	*	0.3)	=	0.195	= hurt prediction
(0.65	*	0.2)	=	0.13	= win prediction
(0.65	*	0.9)	=	0.585	= sad prediction

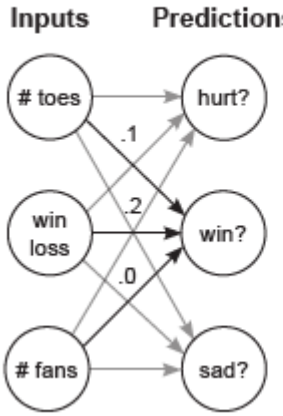
❹ Depositing predictions




```
wlrec = [0.65, 0.8, 0.8, 0.9]

input = wlrec[0]

pred = neural_network(input,weight)

print(pred)
```

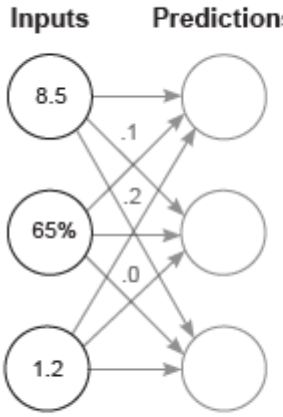
Predicting with multiple inputs and outputs

❶ An empty network with multiple inputs and outputs




```
          # toes % win # fans
weights = [ [0.1, 0.1, -0.3], # hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ] # sad?

def neural_network(input, weights):

    pred = vect_mat_mul(input,weights)

    return pred
```

❷ Inserting one input datapoint

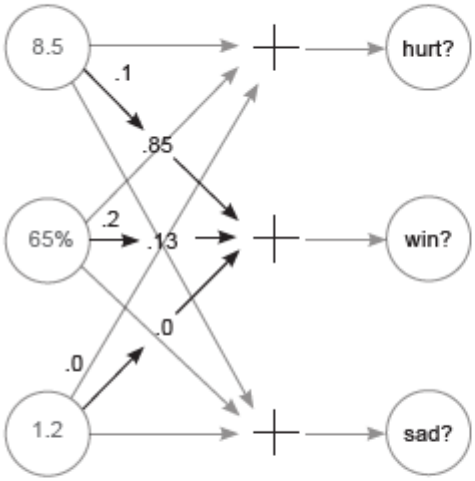


This dataset is the current status at the beginning of each game for the first four games in a season:
toes = current average number of toes per player
wlrec = current games won (percent)
fans = fan count (in millions)

```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)
```

Input corresponds to every entry for the first game of the season.

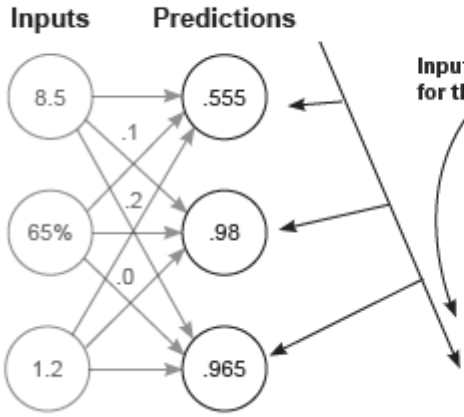
❸ For each output, performing a weighted sum of inputs




```
def w_sum(a,b):
    assert(len(a) == len(b))
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output


def vect_mat_mul(vect,matrix):
    assert(len(vect) == len(matrix))
    output = [0,0,0]

    for i in range(len(vect)):
        output[i]=w_sum(vect,matrix[i])

    return output


def neural_network(input, weights):
    pred=vect_mat_mul(input,weights)
    return pred
```

# toes		% win		# fans			
(8.5 * 0.1)	+	(0.65 * 0.1)	+	(1.2 * –0.3)	=	0.555	= hurt prediction
(8.5 * 0.1)	+	(0.65 * 0.2)	+	(1.2 * 0.0)	=	0.98	= win prediction
(8.5 * 0.0)	+	(0.65 * 1.3)	+	(1.2 * 0.1)	=	0.965	= sad prediction

❹ Depositing predictions

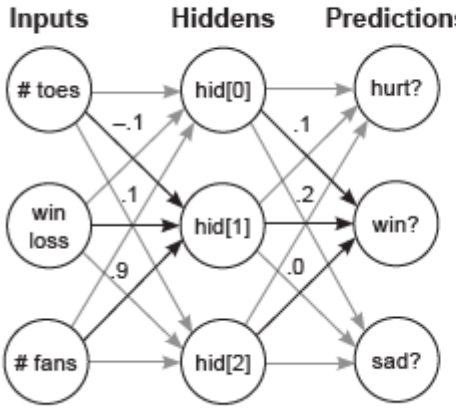


Input corresponds to every entry for the first game of the season.

```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
```

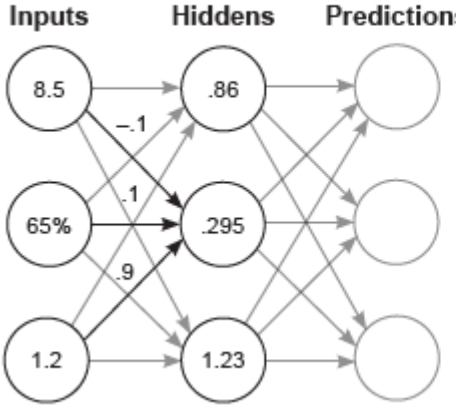
Neural networks can be stacked (hidden layers)

❶ An empty network with multiple inputs and outputs




```
                  # toes % win # fans
ih_wgt = [  [0.1, 0.2, -0.1], # hid[0]
            [-0.1,0.1, 0.9], # hid[1]
            [0.1, 0.4, 0.1] ] # hid[2]

              #hid[0] hid[1] hid[2]
hp_wgt = [  [0.3, 1.1, -0.3], # hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ] # sad?

weights = [ih_wgt, hp_wgt]

def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```

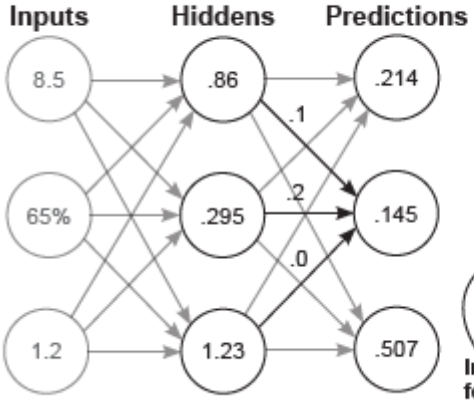
❷ Predicting the hidden layer



Input corresponds to every entry for the first game of the season.

```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)
def neural_network(input, weights):

    hid = vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred
```

❸ Predicting the output layer (and depositing the prediction)




```
def neural_network(input, weights):

    hid=vect_mat_mul(input,weights[0])
    pred = vect_mat_mul(hid,weights[1])
    return pred

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)
print(pred)
```

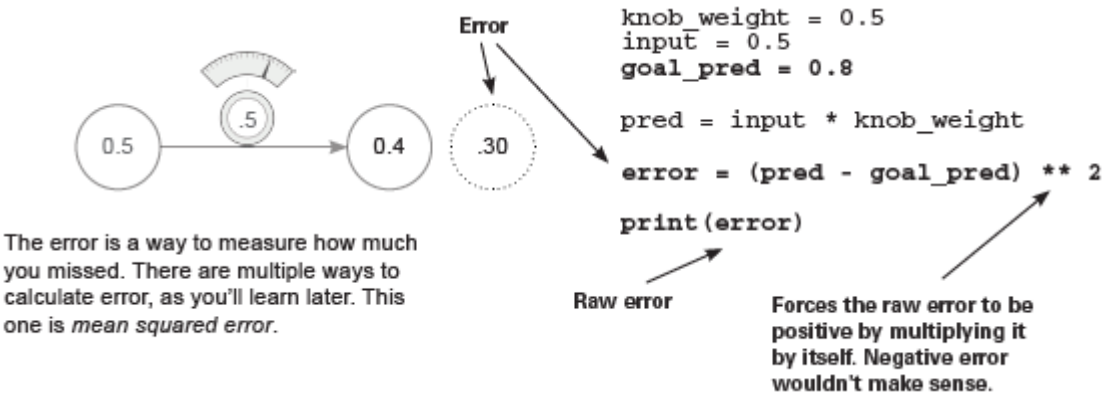
Input corresponds to every entry for the first game of the season.

Predict, compare, and learn

Step 2: Compare

Comparing gives a measurement of how much a prediction

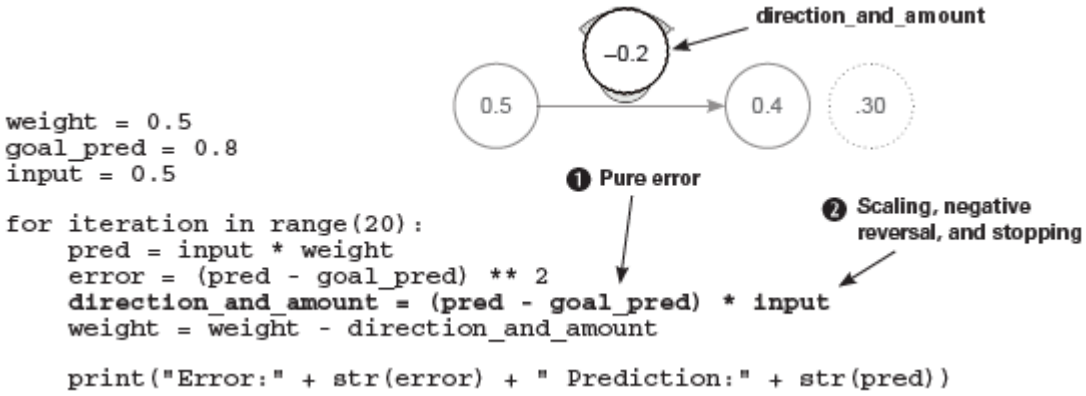
"missed" by.

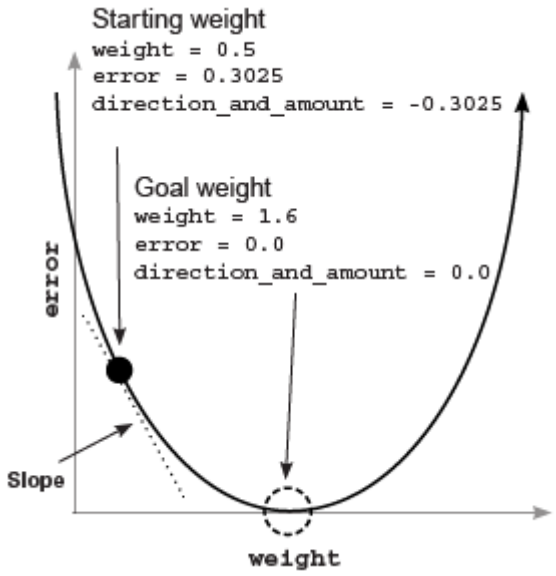


Step 3: Learn

Learning tells each weight how it can change to reduce the error.

Metod of Gradient descent.





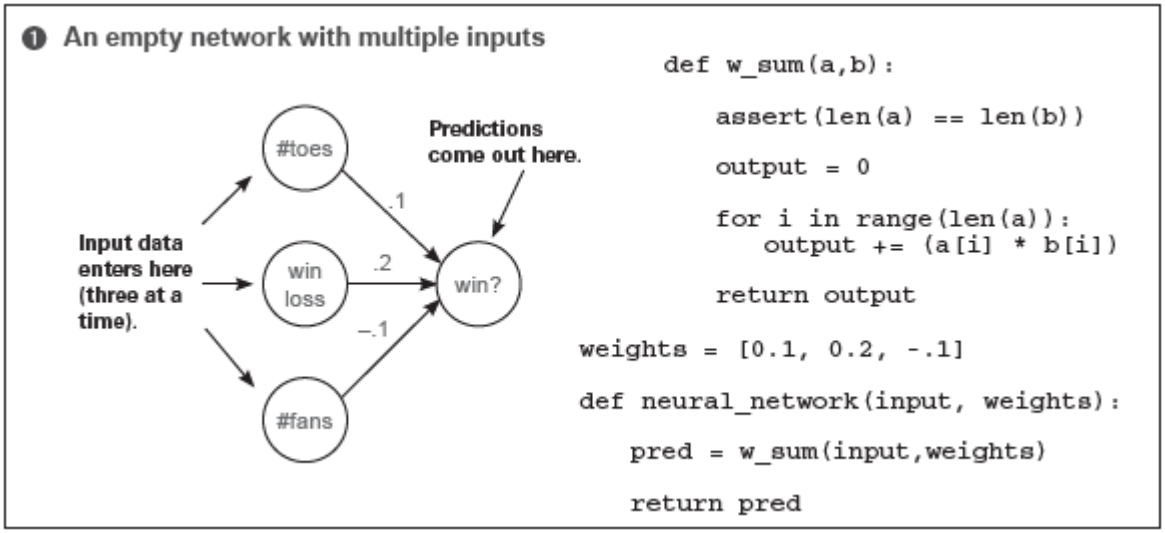
In [2]:

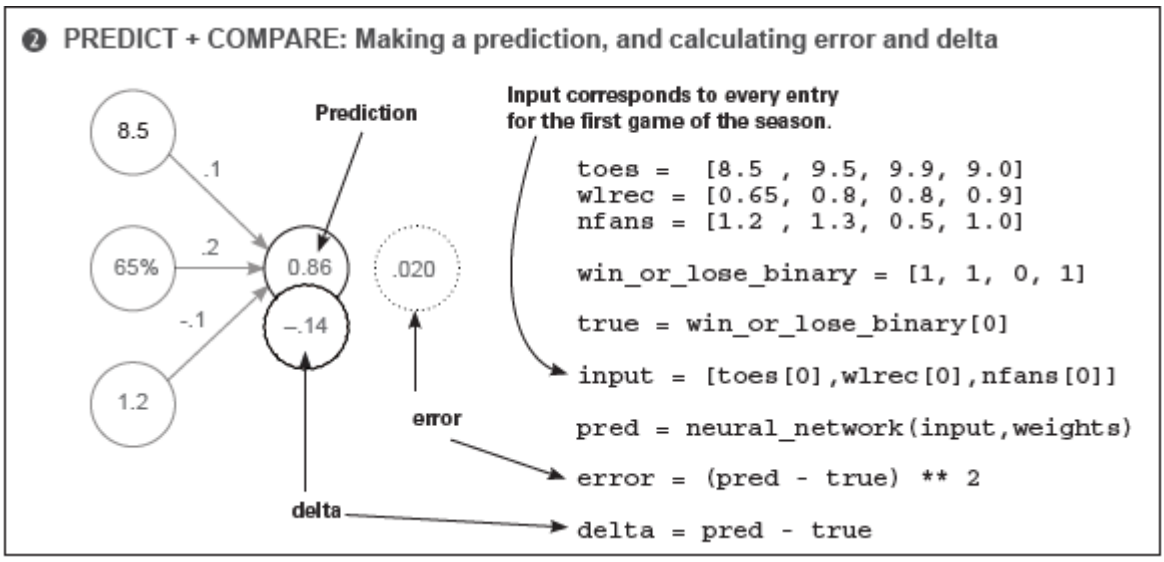
```
weight = 0.5
goal_pred = 0.8
input = 0.5
#alpha = 0.1

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount

    print("Error:"+str("{0:10.3e}").format(error)+"\tPrediction:"+str("{0:10.6f}").format(pred)
          +"\tWeight:"+str("{0:10f}").format(weight))
```

```
Error: 3.025e-01        Prediction:  0.250000   Weight:  0.775000
```

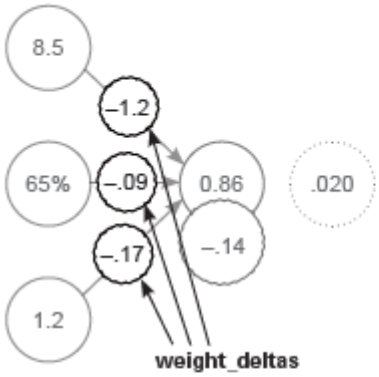
```
Error: 1.702e-01        Prediction:  0.387500   Weight:  0.981250
Error: 9.571e-02        Prediction:  0.490625   Weight:  1.135938
Error: 5.384e-02        Prediction:  0.567969   Weight:  1.251953
Error: 3.028e-02        Prediction:  0.625977   Weight:  1.338965
Error: 1.703e-02        Prediction:  0.669482   Weight:  1.404224
Error: 9.582e-03        Prediction:  0.702112   Weight:  1.453168
Error: 5.390e-03        Prediction:  0.726584   Weight:  1.489876
Error: 3.032e-03        Prediction:  0.744938   Weight:  1.517407
Error: 1.705e-03        Prediction:  0.758703   Weight:  1.538055
Error: 9.593e-04        Prediction:  0.769028   Weight:  1.553541
Error: 5.396e-04        Prediction:  0.776771   Weight:  1.565156
Error: 3.035e-04        Prediction:  0.782578   Weight:  1.573867
Error: 1.707e-04        Prediction:  0.786934   Weight:  1.580400
Error: 9.604e-05        Prediction:  0.790200   Weight:  1.585300
Error: 5.402e-05        Prediction:  0.792650   Weight:  1.588975
Error: 3.039e-05        Prediction:  0.794488   Weight:  1.591731
Error: 1.709e-05        Prediction:  0.795866   Weight:  1.593799
Error: 9.615e-06        Prediction:  0.796899   Weight:  1.595349
Error: 5.408e-06        Prediction:  0.797674   Weight:  1.596512
```

Gradient descent learning with multiple inputs





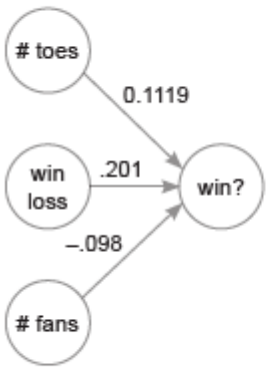
❸ LEARN: Calculating each weight_delta and putting it on each weight




```
def ele_mul(number,vector):

    output = [0,0,0]

    assert(len(output) == len(vector))

    for i in range(len(vector)):
        output[i] = number * vector[i]

    return output

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)

error = (pred - true) ** 2

delta = pred - true

weight_deltas = ele_mul(delta,input)
```

8.5 * -0.14 = -1.19 = **weight_deltas[0]**
0.65 * -0.14 = -0.091 = **weight_deltas[1]**
1.2 * -0.14 = -0.168 = **weight_deltas[2]**

❹ LEARN: Updating the weights



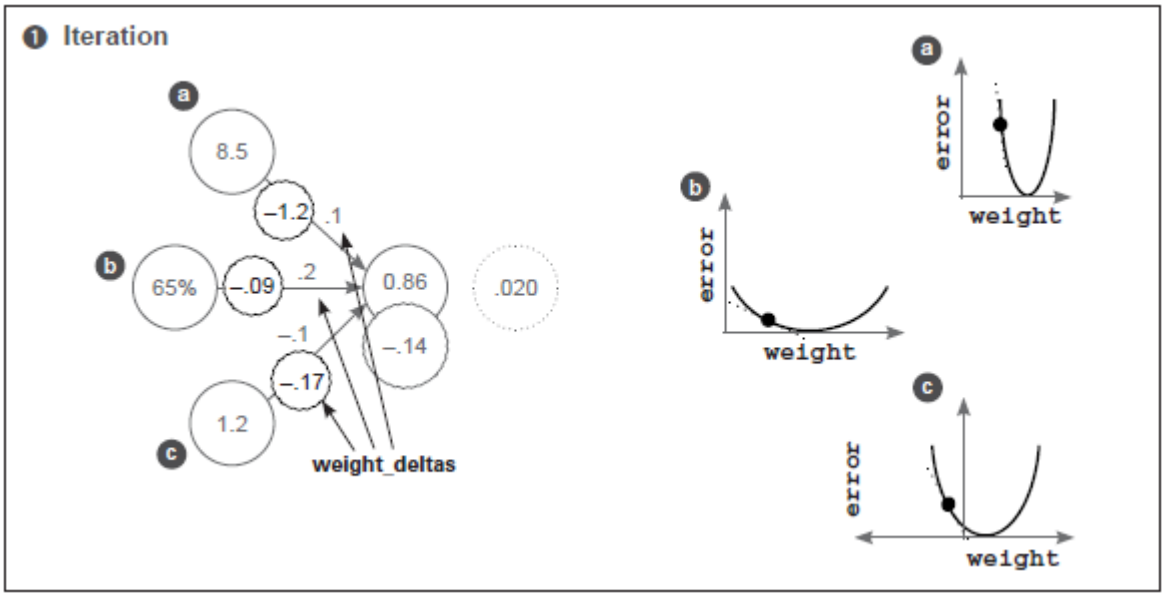

```
input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weight)
error = (pred - true) ** 2
delta = pred - true

weight_deltas = ele_mul(delta,input)

alpha = 0.01

for i in range(len(weights)):
    weights[i] -= alpha * weight_deltas[i]
print("Weights:" + str(weights))
print("Weight Deltas:" + str(weight_deltas))
```

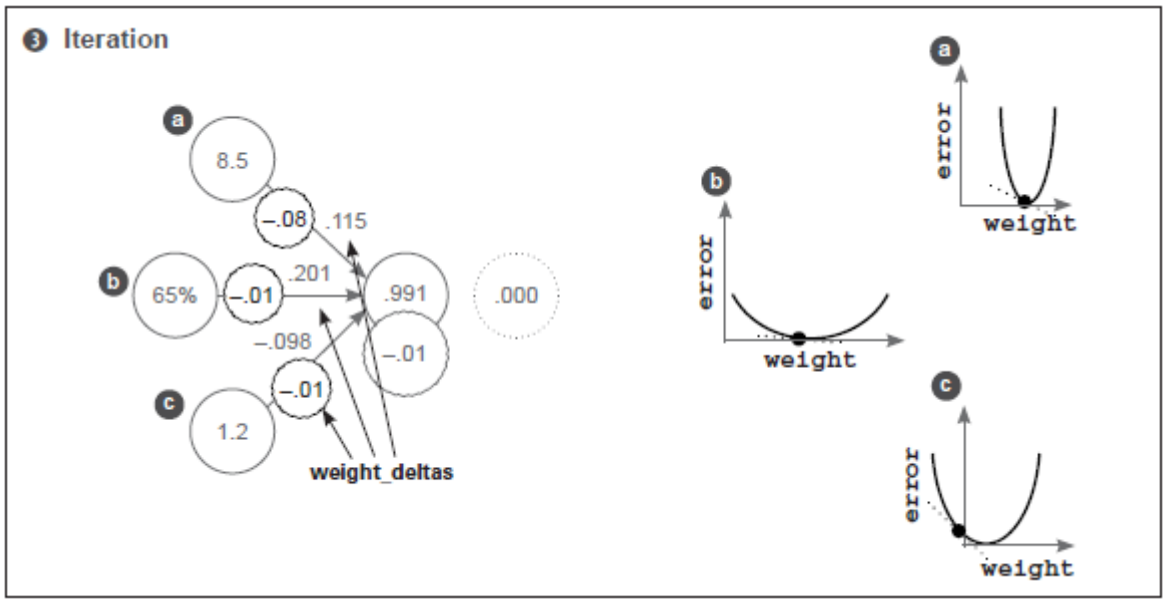
0.1 - (-1.19 * 0.01) = 0.1119 = **weights[0]**
0.2 - (-.091 * 0.01) = 0.2009 = **weights[1]**
-0.1 - (-.168 * 0.01) = -0.098 = **weights[2]**

In [ ]:

```
def neural_network(input, weights):
  out = 0
  for i in range(len(input)):
    out += (input[i] * weights[i])
  return out

def ele_mul(scalar, vector):
  out = [0,0,0]
  for i in range(len(out)):
    out[i] = vector[i] * scalar
  return out

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

win_or_lose_binary = [1, 1, 0, 1]
true = win_or_lose_binary[0]

alpha = 0.01
weights = [0.1, 0.2, -.1]
print("Weights:" + str(weights))

input = [toes[0],wlrec[0],nfans[0]]

for iter in range(3):
```

```
  pred = neural_network(input,weights)

  error = (pred - true) ** 2
  delta = pred - true

  weight_deltas=ele_mul(delta,input)

  print("\nIteration:" + str(iter+1))
  print("Pred:" + str(pred))
  print("Error:" + str(error))
  print("Delta:" + str(delta))
  print("Weight_Deltas:" + str(weight_deltas))

  for i in range(len(weights)):
    weights[i]-=alpha*weight_deltas[i]

  print("Weights:" + str(weights))
```





Gradient descent learning with multiple outputs

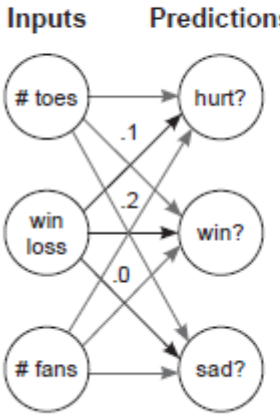
In [ ]:

```
weights = [0.3, 0.2, 0.9]

def neural_network(input, weights):
    pred = ele_mul(input,weights)
    return pred

wlrec = [0.65, 1.0, 1.0, 0.9]

hurt  = [0.1, 0.0, 0.0, 0.1]
win   = [  1,   1,   0,   1]
```

```
sad   = [0.1, 0.0, 0.1, 0.2]

input = wlrec[0]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):
    error[i] = (pred[i] - true[i]) ** 2
    delta[i] = pred[i] - true[i]

def scalar_ele_mul(number,vector):
    output = [0,0,0]

    assert(len(output) == len(vector))

    for i in range(len(vector)):
        output[i] = number * vector[i]

    return output

weight_deltas = scalar_ele_mul(input,delta)

alpha = 0.1

for i in range(len(weights)):
    weights[i] -= (weight_deltas[i] * alpha)

print("Weights:" + str(weights))
print("Weight Deltas:" + str(weight_deltas))
```

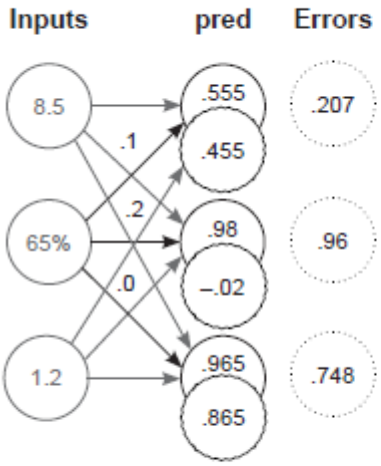
Gradient descent with multiple inputs and outputs

❶ An empty network with multiple inputs and outputs



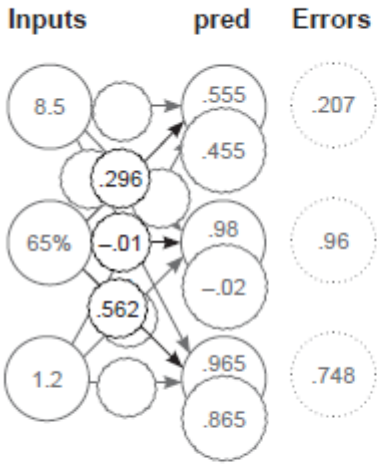

```
                  # toes %win # fans
weights = [ [0.1, 0.1, -0.3],# hurt?
            [0.1, 0.2, 0.0], # win?
            [0.0, 1.3, 0.1] ]# sad?

def vect_mat_mul(vect,matrix):
    assert(len(vect) == len(matrix))
    output = [0,0,0]
    for i in range(len(vect)):
       output[i] = w_sum(vect,matrix[i])
    return output

def neural_network(input, weights):

   pred = vect_mat_mul(input,weights)

   return pred
```

❷ PREDICT: Making a prediction and calculating error and delta




```
toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65,0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

hurt  = [0.1, 0.0, 0.0, 0.1]
win   = [  1,   1,   0,   1]
sad   = [0.1, 0.0, 0.1, 0.2]

alpha = 0.01

input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]


pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

   error[i] = (pred[i] - true[i]) ** 2
   delta = pred[i] - true[i]
```

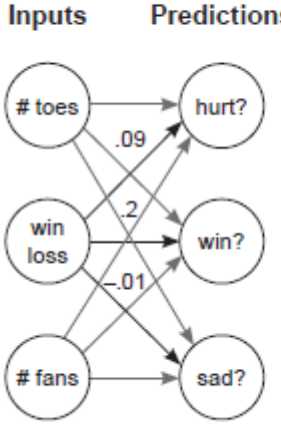
❸ COMPARE: Calculating each weight_delta and putting it on each weight



(weight_deltas are shown for only one input, to save space.)

```
def outer_prod(vec_a, vec_b):

    out = zeros_matrix(len(a),len(b))

    for i in range(len(a)):
        for j in range(len(b)):
            out[i][j] = vec_a[i]*vec_b[j]

    return out

input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = outer_prod(input,delta)
```

❹ LEARN: Updating the weights

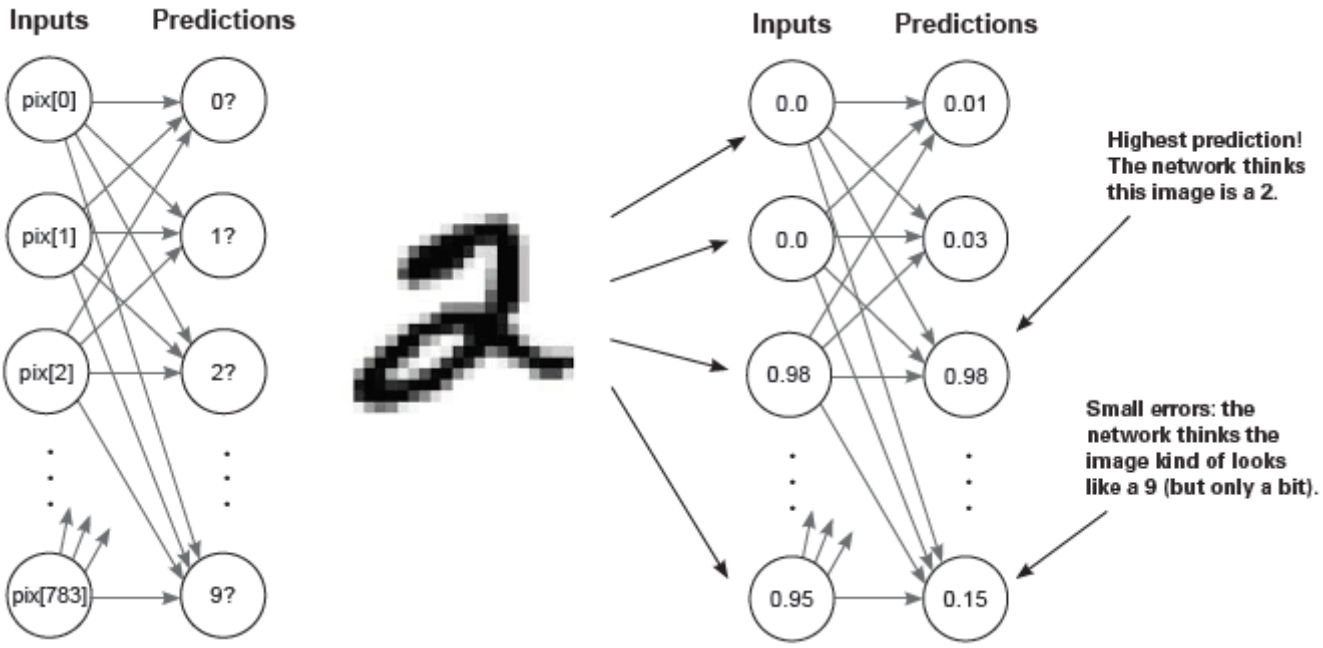


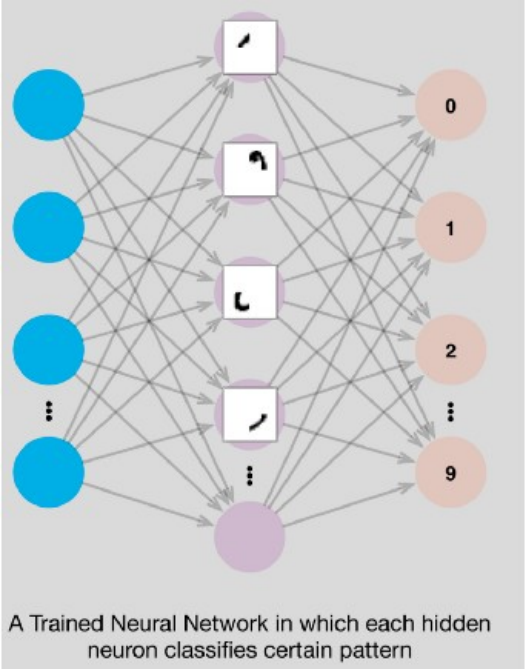

```
input = [toes[0],wlrec[0],nfans[0]]
true = [hurt[0], win[0], sad[0]]

pred = neural_network(input,weights)

error = [0, 0, 0]
delta = [0, 0, 0]

for i in range(len(true)):

    error[i] = (pred[i] - true[i]) ** 2
    delta = pred[i] - true[i]

weight_deltas = outer_prod(input,delta)

for i in range(len(weights)):
  for j in range(len(weights[0])):
    weights[i][j] -= alpha * \
                     weight_deltas[i][j]
```

'Hello World' example: Recognition of hand-written digits

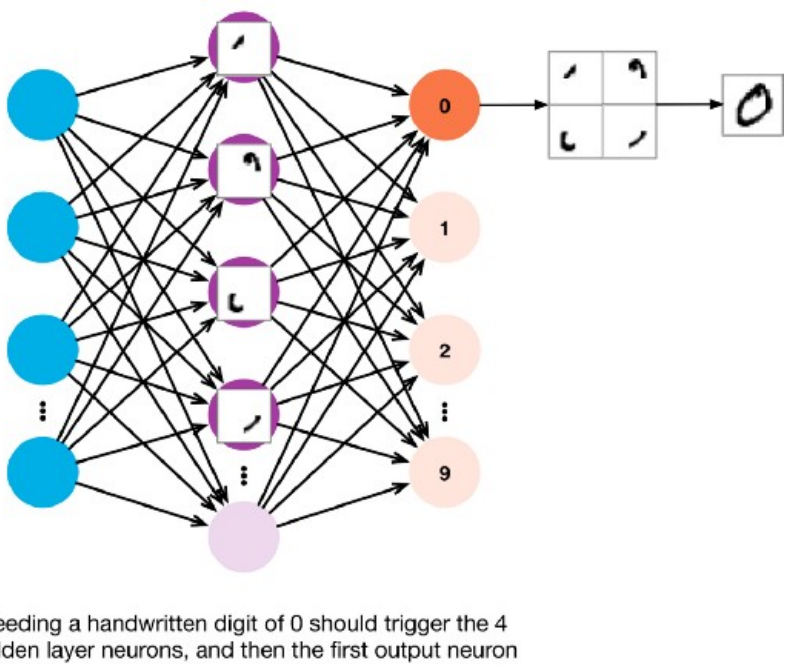
Modified National Institute of Standards and Technology (MNIST) dataset

It contains 70,000 examples of digits written by human hands. Each of these digits is a picture of 28x28 pixels. So in total each image of a digit has 28*28=784 pixels. Each pixel takes a value beween 0 and 255 (grayscale). 0 means the color is white and 255 means the color black.





A Trained Neural Network in which each hidden neuron classifies certain pattern



Feeding a handwritten digit of 0 should trigger the 4 hidden layer neurons, and then the first output neuron

Create and train neural network with one hidden layer

In [56]:

```
import sys, numpy as np
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

images, labels = (x_train[0:1000].reshape(1000,28*28) / 255, y_train[0:1000])

one_hot_labels = np.zeros((len(labels),10))
for i,l in enumerate(labels):
    one_hot_labels[i][l] = 1
labels = one_hot_labels

test_images = x_test.reshape(len(x_test),28*28) / 255
test_labels = np.zeros((len(y_test),10))
for i,l in enumerate(y_test):
    test_labels[i][l] = 1

np.random.seed(1)
relu = lambda x:(x>=0) * x # returns x if x > 0, return 0 otherwise
relu2deriv = lambda x: x>0 # returns 1 for input > 0, return 0 otherwise
alpha, iterations, hidden_size, pixels_per_image, num_labels = (0.005, 350, 40, 784, 10)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0, 0)

    for i in range(len(images)):
        # forward propagation
        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)

        # calculate the error/loss
        error += np.sum((labels[i:i+1] - layer_2) ** 2)
        correct_cnt += int(np.argmax(layer_2) == np.argmax(labels[i:i+1]))

        # backpropagation for calculation of the gradient
        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)* relu2deriv(layer_1)

        # update weights using the gradient
        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
```

```
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

    sys.stdout.write("\r I:"+str(j)+ \
                     " Train-Err:" + str(error/float(len(images)))[0:5] +\
                     " Train-Acc:" + str(correct_cnt/float(len(images))))

    if(j % 10 == 0 or j == iterations-1):
        error, correct_cnt = (0.0, 0)

        # evaluate the fitted model using test set from MNIST
        for i in range(len(test_images)):

            layer_0 = test_images[i:i+1]
            layer_1 = relu(np.dot(layer_0,weights_0_1))
            layer_2 = np.dot(layer_1,weights_1_2)

            error += np.sum((test_labels[i:i+1] - layer_2) ** 2)
            correct_cnt += int(np.argmax(layer_2) == np.argmax(test_labels[i:i+1]))
        sys.stdout.write(" Test-Err:" + str(error/float(len(test_images)))[0:5] +\
                         " Test-Acc:" + str(correct_cnt/float(len(test_images))))
        print()
```

```
I:0 Train-Err:0.729 Train-Acc:0.527 Test-Err:0.593 Test-Acc:0.6493
I:10 Train-Err:0.198 Train-Acc:0.94 Test-Err:0.321 Test-Acc:0.8559
I:20 Train-Err:0.126 Train-Acc:0.972 Test-Err:0.293 Test-Acc:0.8638
I:30 Train-Err:0.091 Train-Acc:0.991 Test-Err:0.291 Test-Acc:0.8614
I:40 Train-Err:0.071 Train-Acc:0.994 Test-Err:0.295 Test-Acc:0.86
I:50 Train-Err:0.057 Train-Acc:0.995 Test-Err:0.299 Test-Acc:0.8584
I:60 Train-Err:0.047 Train-Acc:0.997 Test-Err:0.304 Test-Acc:0.8571
I:70 Train-Err:0.040 Train-Acc:0.998 Test-Err:0.307 Test-Acc:0.8555
I:80 Train-Err:0.034 Train-Acc:0.998 Test-Err:0.311 Test-Acc:0.8524
I:90 Train-Err:0.030 Train-Acc:0.998 Test-Err:0.314 Test-Acc:0.8513
I:100 Train-Err:0.026 Train-Acc:0.998 Test-Err:0.318 Test-Acc:0.8505
I:110 Train-Err:0.023 Train-Acc:0.998 Test-Err:0.321 Test-Acc:0.8491
I:120 Train-Err:0.020 Train-Acc:0.998 Test-Err:0.323 Test-Acc:0.8467
I:130 Train-Err:0.018 Train-Acc:0.998 Test-Err:0.326 Test-Acc:0.8445
I:140 Train-Err:0.016 Train-Acc:0.998 Test-Err:0.327 Test-Acc:0.8437
I:150 Train-Err:0.014 Train-Acc:0.998 Test-Err:0.329 Test-Acc:0.8427
I:160 Train-Err:0.013 Train-Acc:0.998 Test-Err:0.331 Test-Acc:0.8414
I:170 Train-Err:0.012 Train-Acc:0.998 Test-Err:0.333 Test-Acc:0.8405
I:180 Train-Err:0.010 Train-Acc:0.998 Test-Err:0.335 Test-Acc:0.839
I:190 Train-Err:0.009 Train-Acc:0.999 Test-Err:0.337 Test-Acc:0.8377
I:200 Train-Err:0.009 Train-Acc:0.999 Test-Err:0.338 Test-Acc:0.8368
I:210 Train-Err:0.008 Train-Acc:0.999 Test-Err:0.340 Test-Acc:0.8365
I:220 Train-Err:0.007 Train-Acc:0.999 Test-Err:0.341 Test-Acc:0.8356
I:230 Train-Err:0.007 Train-Acc:0.999 Test-Err:0.343 Test-Acc:0.8346
I:240 Train-Err:0.006 Train-Acc:0.999 Test-Err:0.344 Test-Acc:0.8344
I:250 Train-Err:0.006 Train-Acc:0.999 Test-Err:0.345 Test-Acc:0.8336
I:260 Train-Err:0.005 Train-Acc:0.999 Test-Err:0.347 Test-Acc:0.8342
I:270 Train-Err:0.005 Train-Acc:0.999 Test-Err:0.348 Test-Acc:0.8332
I:280 Train-Err:0.005 Train-Acc:0.999 Test-Err:0.349 Test-Acc:0.8326
I:290 Train-Err:0.004 Train-Acc:0.999 Test-Err:0.350 Test-Acc:0.832
I:300 Train-Err:0.004 Train-Acc:0.999 Test-Err:0.351 Test-Acc:0.8313
I:310 Train-Err:0.004 Train-Acc:0.999 Test-Err:0.352 Test-Acc:0.831
I:320 Train-Err:0.004 Train-Acc:0.999 Test-Err:0.353 Test-Acc:0.83
I:330 Train-Err:0.003 Train-Acc:0.999 Test-Err:0.354 Test-Acc:0.8294
I:340 Train-Err:0.003 Train-Acc:0.999 Test-Err:0.354 Test-Acc:0.8293
I:349 Train-Err:0.003 Train-Acc:0.999 Test-Err:0.355 Test-Acc:0.829
```

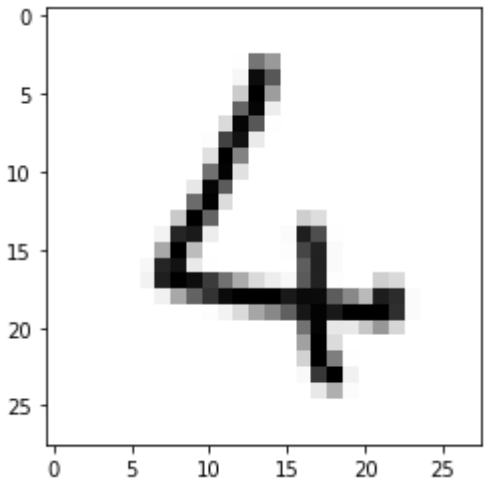
Test the single image from MNIST test set

In [62]:

```
import matplotlib.pyplot as plt
image_index = -1 # image index for testing

if image_index > -1:
    plt.imshow(x_test[image_index],cmap='Greys')
    test_image = x_test[image_index].reshape(1,28*28) / 255
    test_label = np.zeros((1,10))
    test_label[0][y_test[image_index]] = 1
else:
    from PIL import Image
```

```
    img = Image.open('number4_s5.png').convert("L")
    number = 4
    test_label = np.zeros((1,10))
    test_label[0][number] = 1
    #img.show()
    img = img.resize((28,28))
    im2arr1 = np.array(img)
    im2arr2 = 255 - im2arr1
    im2arr3 = im2arr2.astype('float32')
    im2arr3 /=255
    plt.imshow(im2arr3,cmap='Greys')
    test_image = im2arr3.reshape(1,28*28)

error, correct_cnt = (0.0, 0)

layer_0 = test_image[0]
layer_1 = relu(np.dot(layer_0,weights_0_1))
layer_2 = np.dot(layer_1,weights_1_2)

error += np.sum((test_label[0] - layer_2) ** 2)
correct_cnt += int(np.argmax(layer_2) == np.argmax(test_label[0]))
sys.stdout.write(" Test-Err:" + str(error/float(len(test_image)))[0:5])
print()
print(layer_2)
print(np.argmax(layer_2))
print(np.argmax(test_label[0]))
```

```
 Test-Err:0.362
[-1.56759421e-04 -1.53665567e-03 -7.54447080e-03  2.03631110e-03
  4.74070631e-01 -2.01224524e-01  1.89258957e-01  1.10369250e-02
  9.13311018e-02  3.14043202e-02]
4
4
```



MNIST test with Tensorflow / Keras

In [ ]:

```
import tensorflow as tf

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Reshaping the array to 4-dims so that it can work with the Keras API
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)
# Making sure that the values are float so that we can get decimal points after division
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
# Normalizing the RGB codes by dividing it to the max RGB value.
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print('Number of images in x_train', x_train.shape[0])
print('Number of images in x_test', x_test.shape[0])
```

```
# Importing the required Keras modules containing model and layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D
# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten()) # Flattening the 2D arrays for fully connected layers
model.add(Dense(128, activation=tf.nn.relu))
model.add(Dropout(0.2))
model.add(Dense(10,activation=tf.nn.softmax))

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=10, batch_size=200)
# Save the model
# model.save('models/mnistCNN1.h5')

# Final evaluation of the model
metrics = model.evaluate(x_test, y_test, verbose=0)
print("Metrics(Test loss & Test Accuracy): ")
print(metrics)
```

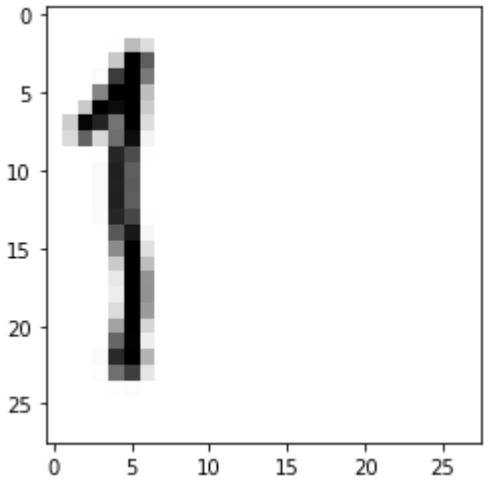
MNIST test with saved Tensorflow / Keras models

In [63]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

import logging
tf.get_logger().setLevel(logging.ERROR)

# load data: MNIST data set
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Importing the Keras libraries
from keras.models import load_model
model1 = load_model('models/mnistCNN1.h5')
model1.summary()
model2 = load_model('models/mnistCNN2b.h5')
model2.summary()

image_index = -1 # image index for testing

if image_index > -1:
    plt.imshow(x_test[image_index],cmap='Greys')
    #plt.imsave('number_test.png',x_test[image_index],cmap='Greys')
    im2arr = x_test[image_index].reshape(1,28,28,1).astype('float32')
    im2arr /=255
else:
    from PIL import Image
    img = Image.open('number1_s5.png').convert("L")
    #img.show()
    img = img.resize((28,28))
    im2arr1 = np.array(img)
    im2arr2 = 255 - im2arr1
    im2arr3 = im2arr2.astype('float32')
    im2arr3 /=255
    plt.imshow(im2arr3,cmap='Greys')
    im2arr = im2arr3.reshape(1,28,28,1)
    #im2arr = im2arr2.reshape(1,28,28,1).astype('float32')

im2arr=tf.convert_to_tensor(im2arr, dtype=tf.float32)

# Predicting the Test set results
```

```
print("Prediction of Model 1: ")
y_pred1 = model1.predict(im2arr)
print(y_pred1)
print(y_pred1.argmax())

print("Prediction of Model 2: ")
y_pred2 = model2.predict(im2arr)
print(y_pred2)
print(y_pred2.argmax())
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 26, 26, 28)        280
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 13, 13, 28)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 4732)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               605824
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
=================================================================
Total params: 607,394
Trainable params: 607,394
Non-trainable params: 0
_________________________________________________________________
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 32)        9248
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 32)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 800)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               102528
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 113,898
Trainable params: 113,898
Non-trainable params: 0
_________________________________________________________________
Prediction of Model 1:
[[1.4738780e-02 7.8341291e-06 1.8078618e-03 9.6822809e-03 2.6916400e-09
  4.5331709e-02 2.5474331e-03 9.2588401e-01 4.3217153e-08 1.0716258e-07]]
7
Prediction of Model 2:
[[2.9971883e-02 1.6248816e-03 3.3303763e-04 1.8803219e-05 1.0304848e-02
  2.8730949e-04 9.4877541e-01 1.5124185e-04 7.9154773e-03 6.1712466e-04]]
6
```



In [ ]: