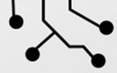
[10]
[columns=2, title=**IAF Alphabetical Index**]

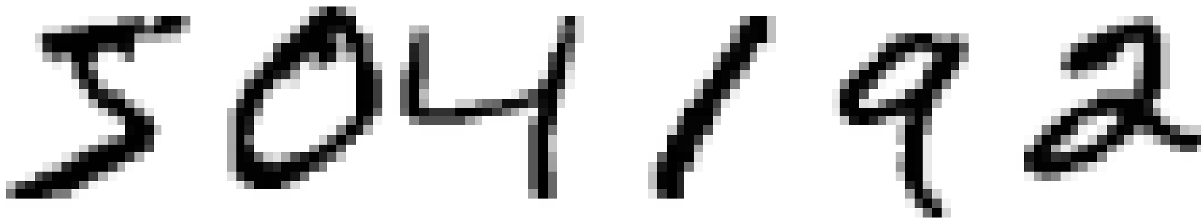
Chapter 0

k.hemant@samsung.com

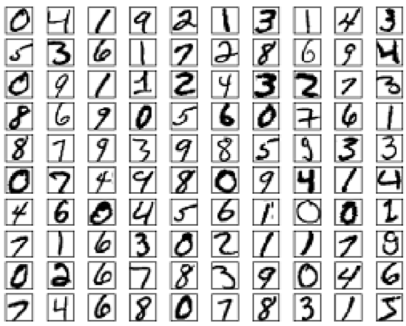


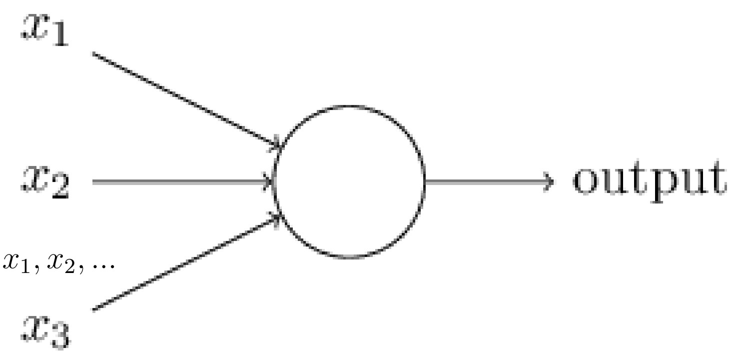
We live among machines that are becoming increasingly intelligent – often able to see, speak or even imitate patterns of human thinking. This is called deep learning.

REF: EPA037



V_1
V_1





$x_1, x_2, ...$

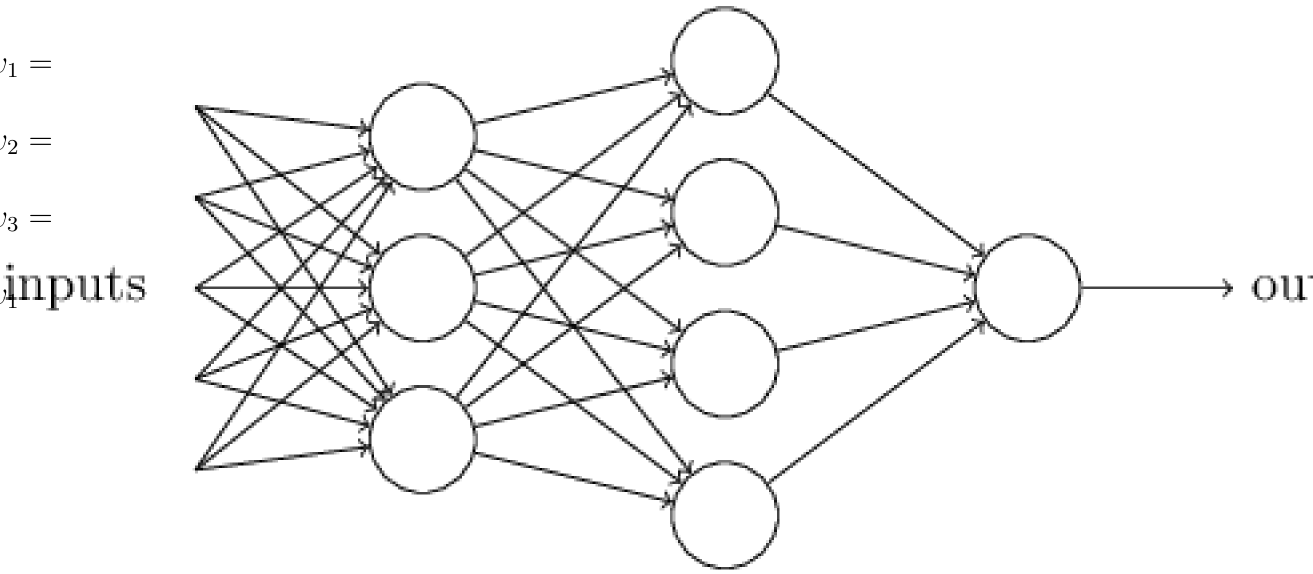
x_1
x_2
x_3
w_1
w_2
$\Sigma w_j x_j$

$$output = \begin{cases} 0 \sum w_j x_j \leq \\ 1 \sum w_j x_j > \end{cases}$$

x_1
$x_2 x_3$
$x_1 =$
1
$x_1 =$
0
$x_2 =$
1
$x_2 =$
0
x_3

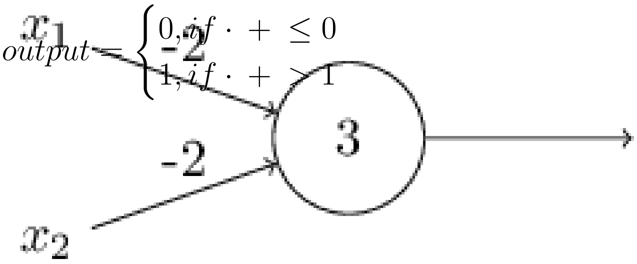
$w_1 =$
6
$w_2 =$
2
$w_3 =$
2
w_1

3

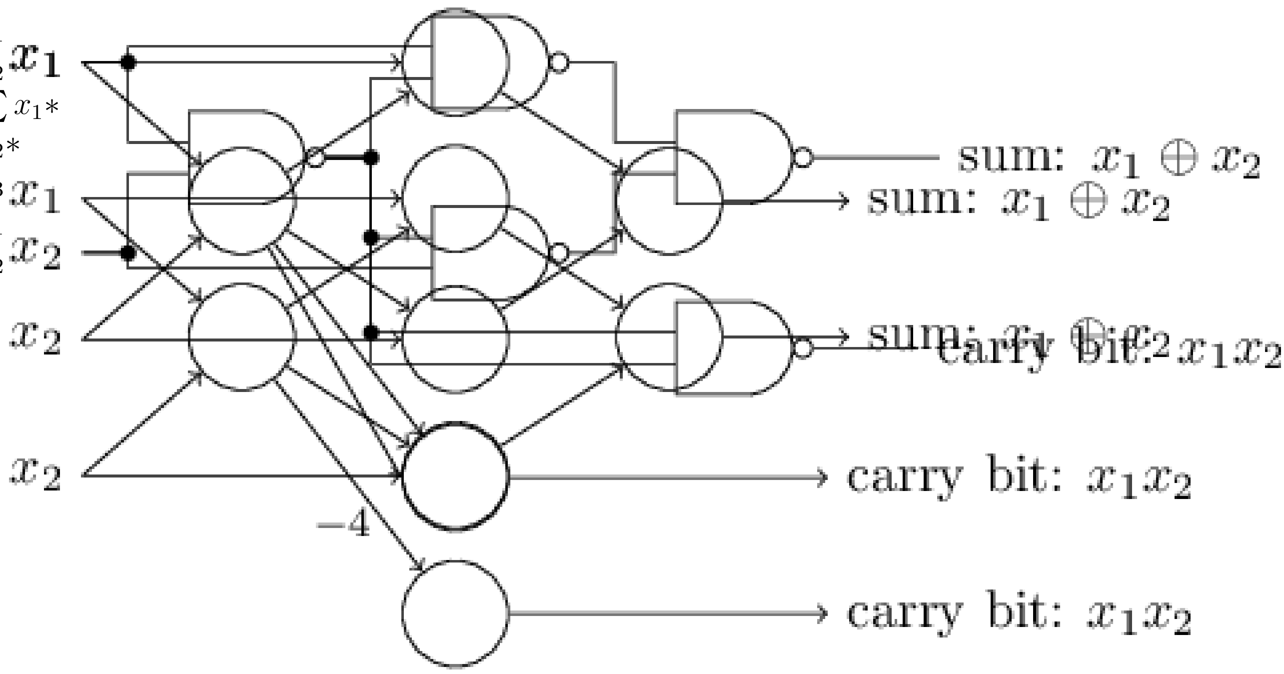


$\sum_j w_j x_j >$
$threshold$
$\sum_j w_j x_j$
$\cdot$
$\equiv$
$\sum w_j x_j$
$b \equiv$
$-threshold$

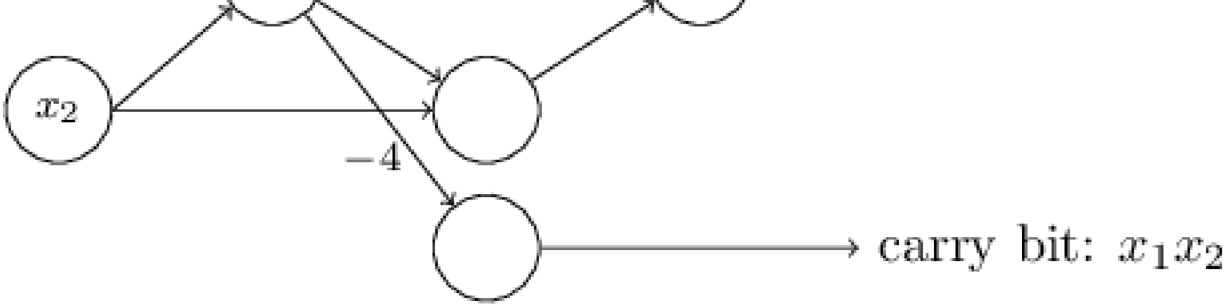
$$output = \begin{cases} 0, if \cdot + \leq 0 \\ 1, if \cdot + > 1 \end{cases}$$

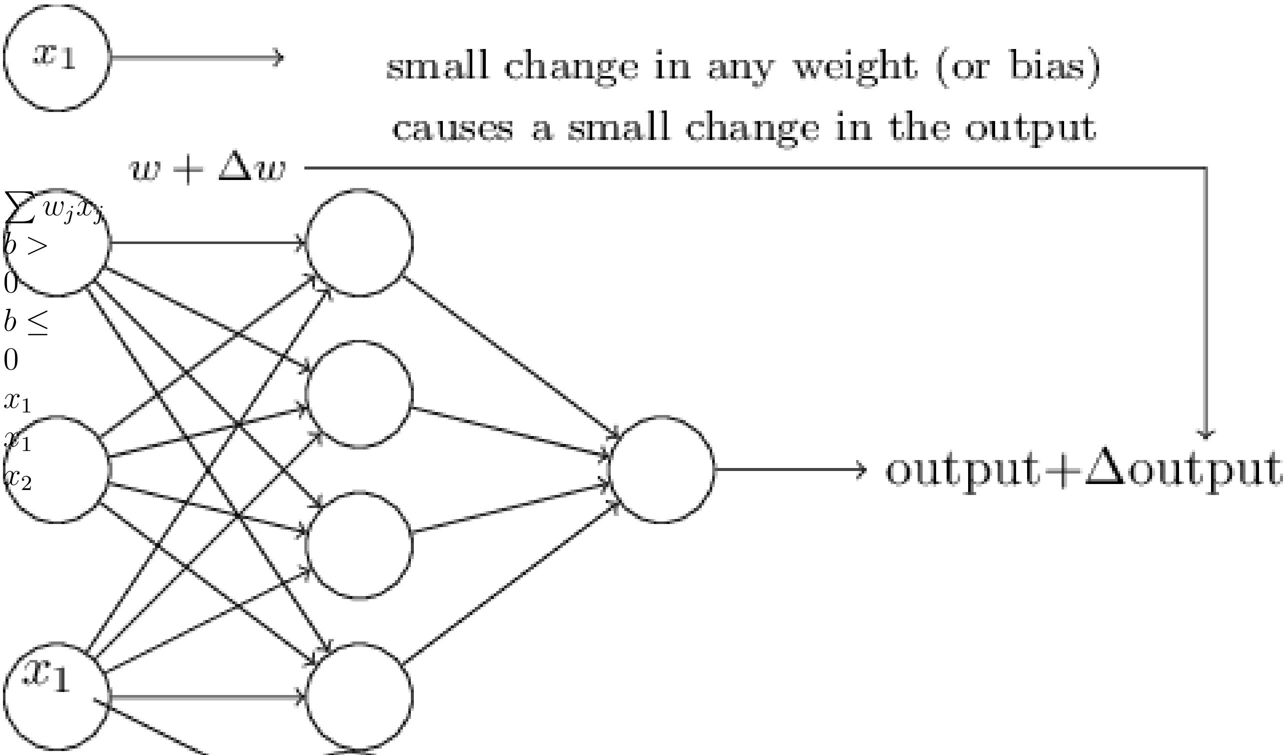


x_1
x_2
$\sum x_1 *$
$x_2 *$
x_3
x_1
x_2
$\cdot$

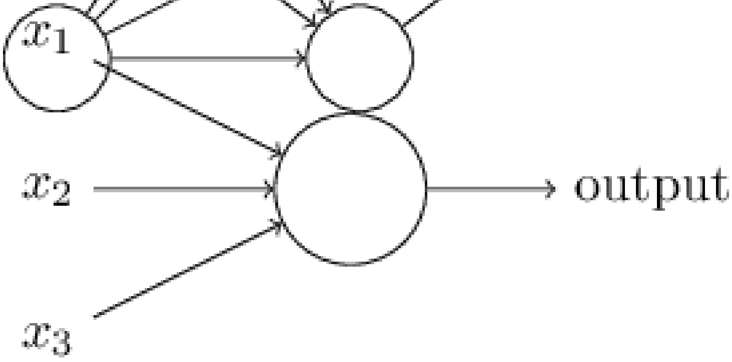


1
2





$\sum w_j x_j$
$b >$
0
$b \leq$
0
x_1
x_1
x_2



x_1
x_2
w_1
w_2
$\sigma(w*$
$x+$
$b)$
σ

$$\sigma(z)\frac{1}{1+e^{-z}}$$

σ

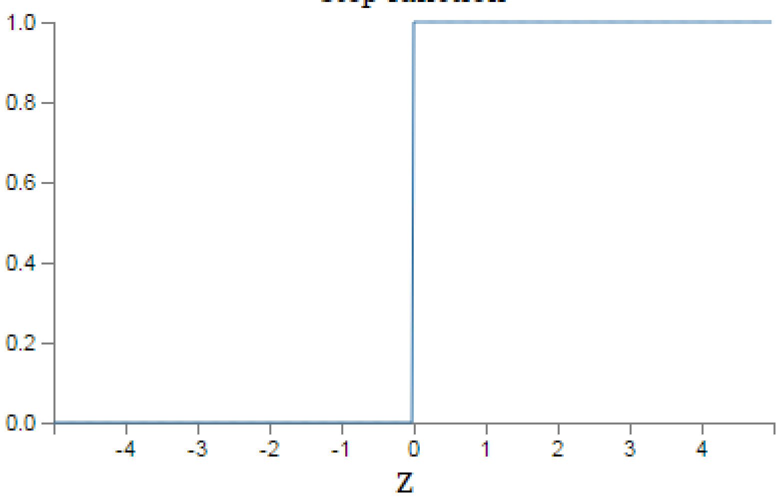
x_1
x_2
w_1
w_2

$$\frac{1}{1 + (\sum_j w_j x_j b)}$$

$z \equiv$
$w*$
$x+$
b
$ez0$
$(z)1$
$z =$
$w*$
$x+$
b
$z =$
$w*$
$x+$
b
$e-$

sigmoid function

step function



σ
$w*$
$x+$
b

w_j
b
$output$
$output$

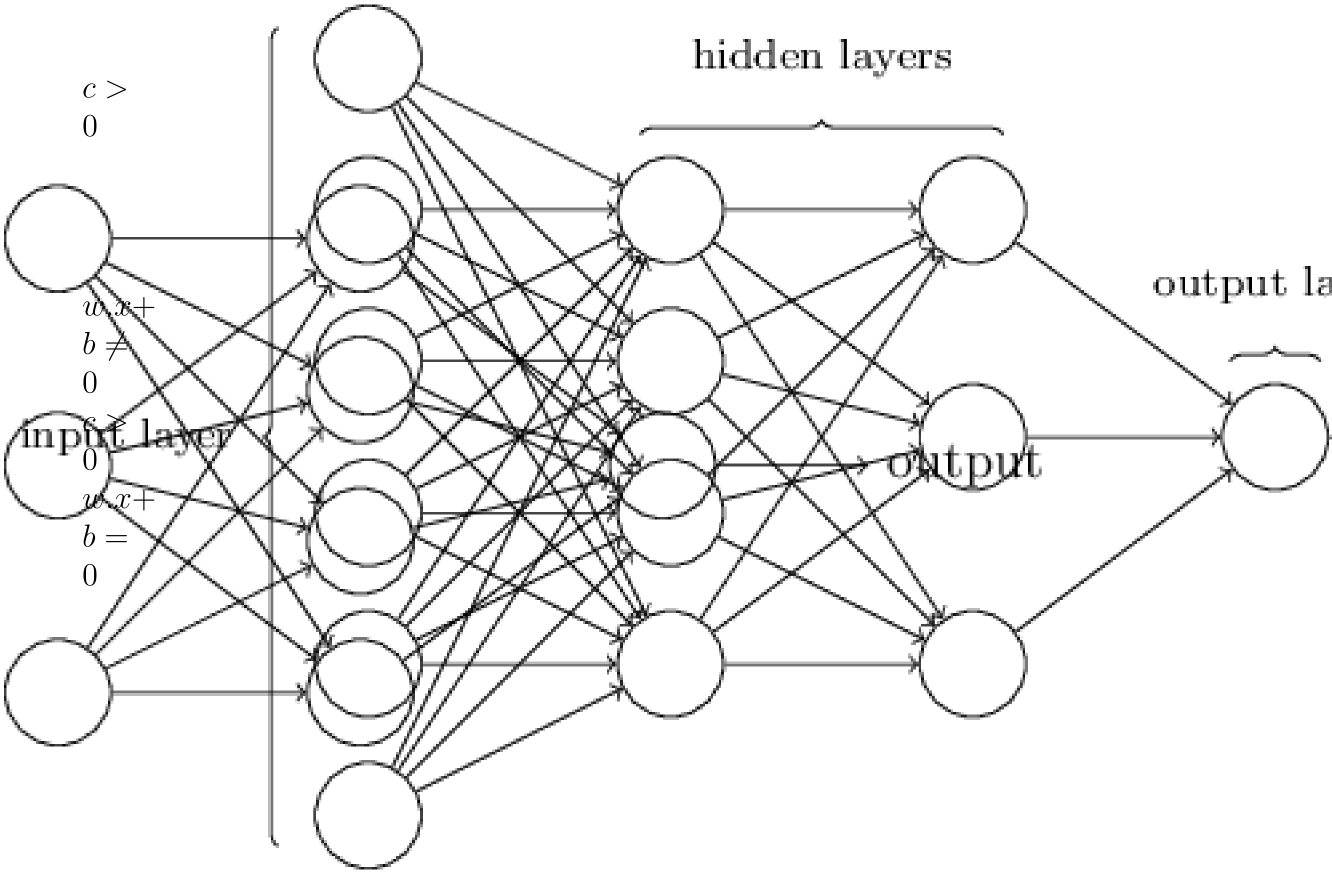
$$\Delta output \approx \Sigma_j \frac{\partial output}{\partial w_j} \triangle w_j + \frac{\partial output}{\partial b} \triangle b$$

w_j
$\frac{\partial output}{\partial w_j}$
$\frac{\partial output}{\partial b}$
$output$
w_j
b
$output$
w_j
b

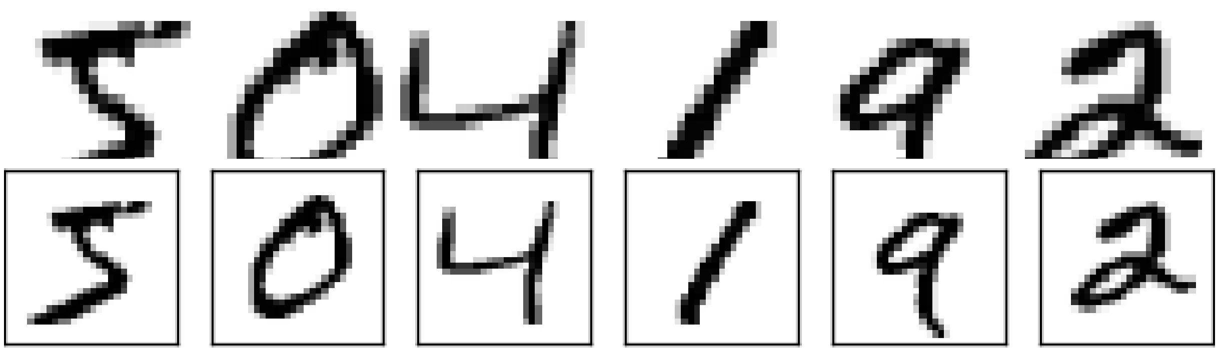
σ
$f(w.x+$
$b)$
$f(\cdot)$

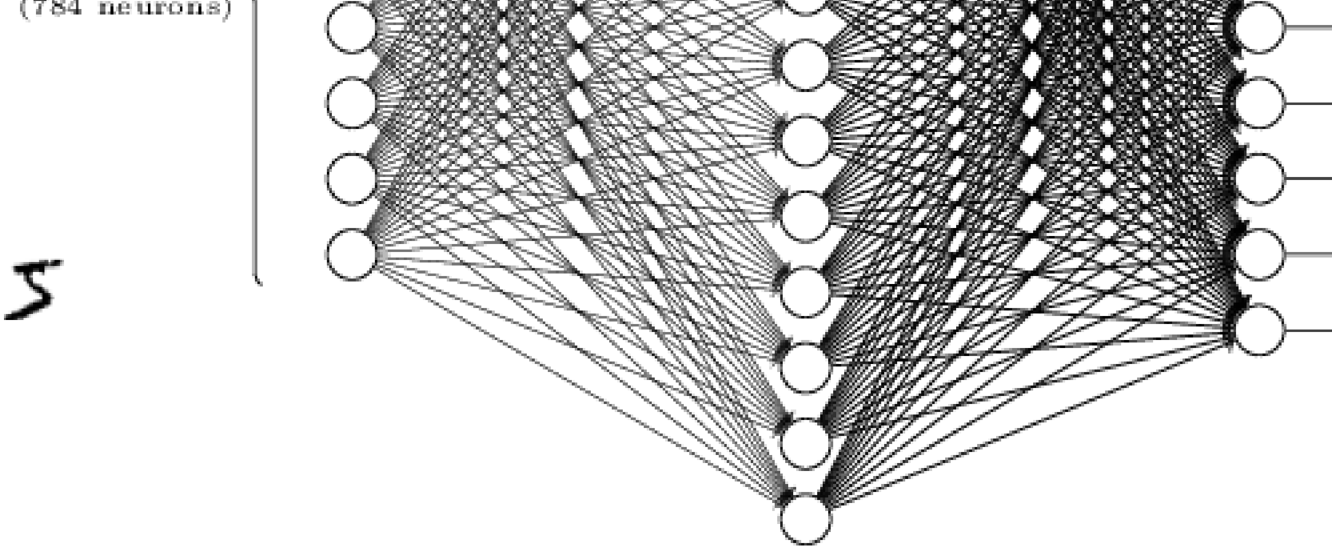
0.173...
0.689...
0.5
0.5

$w*$
$x+$
$b =$
0



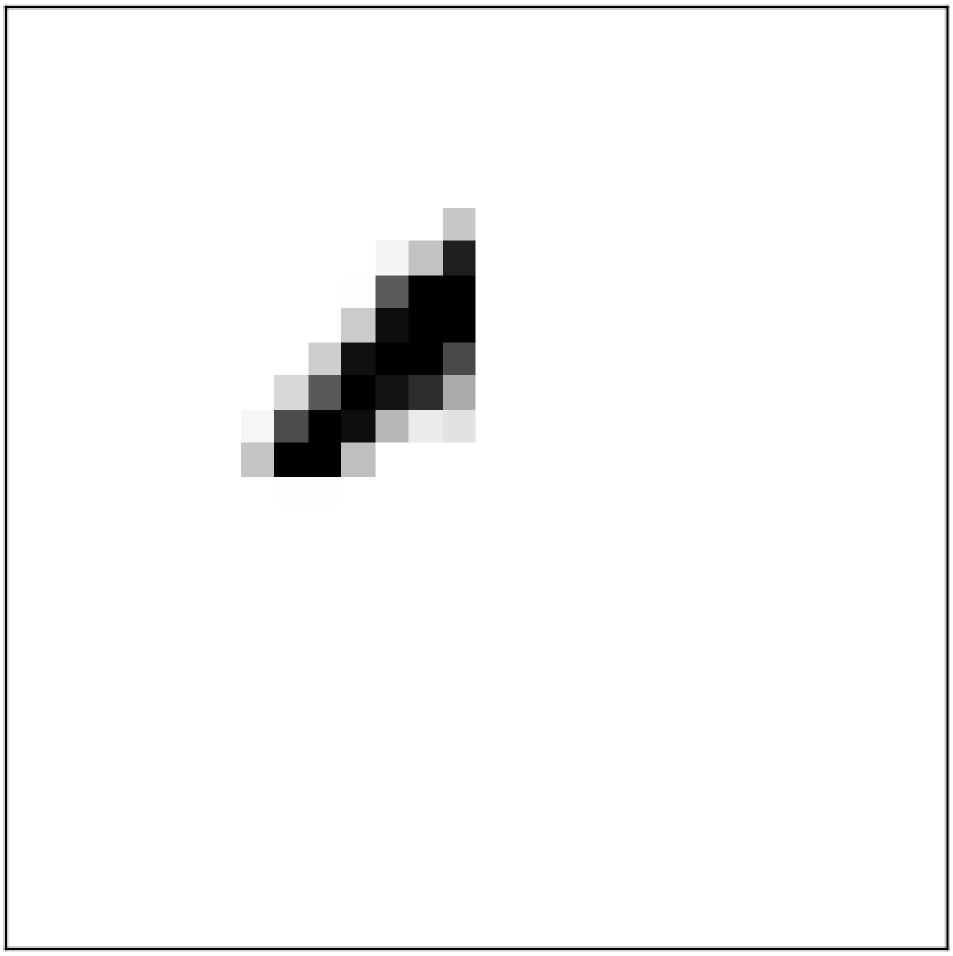
64
64
$4,096 =$
$64\times$
64
0
1
0.5
0.5

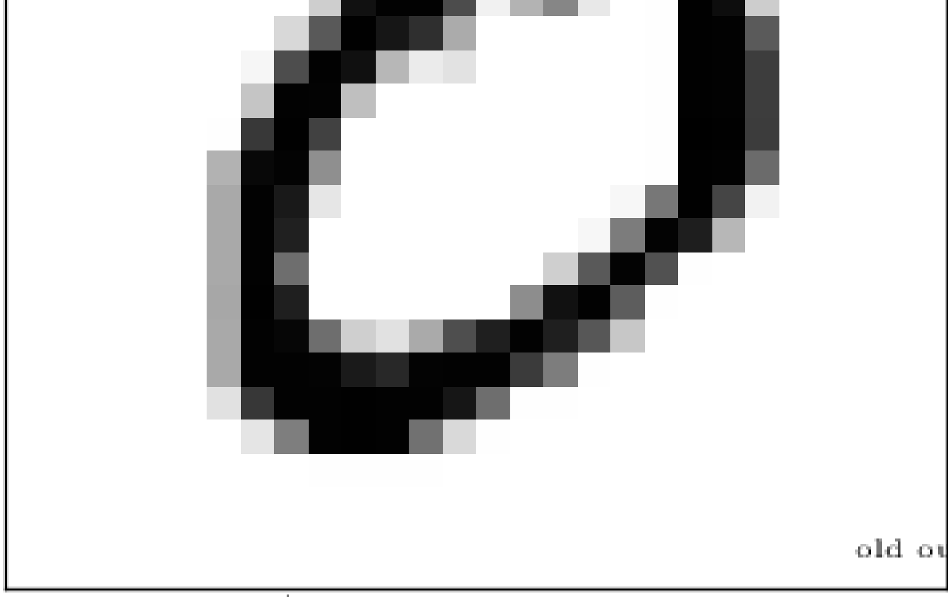


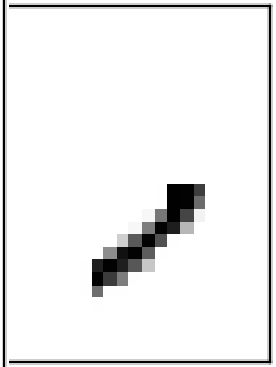


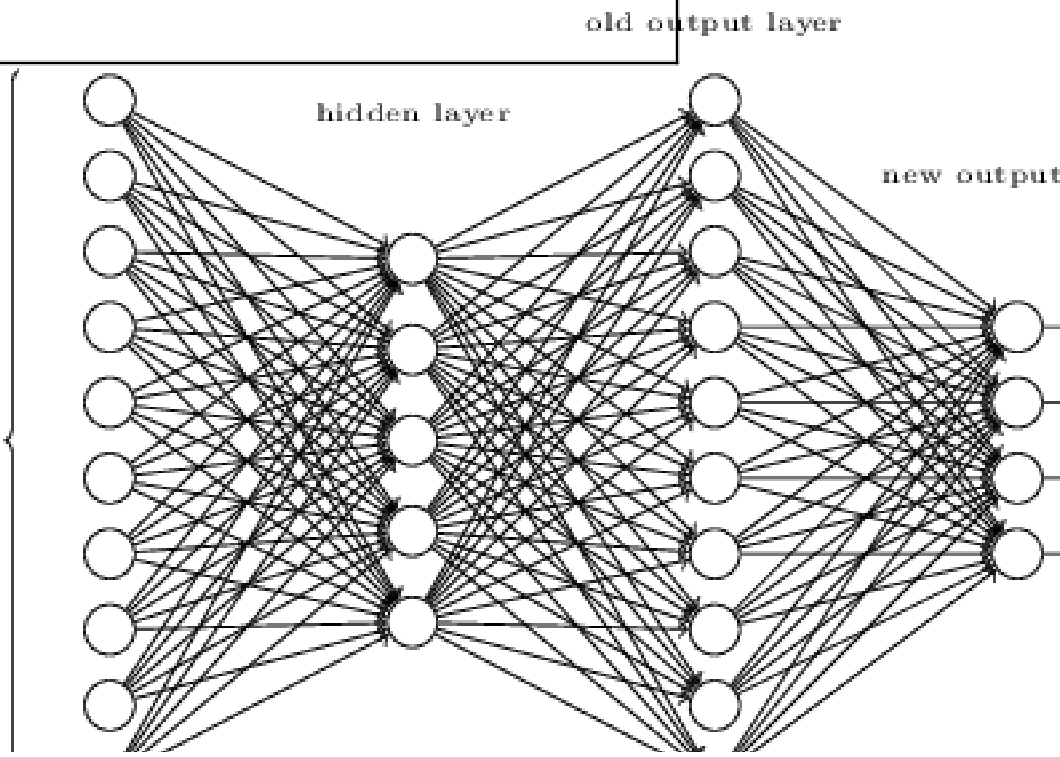
$28 by 28$
$784 =$
$28\times$
28

n
n
$n =$
15







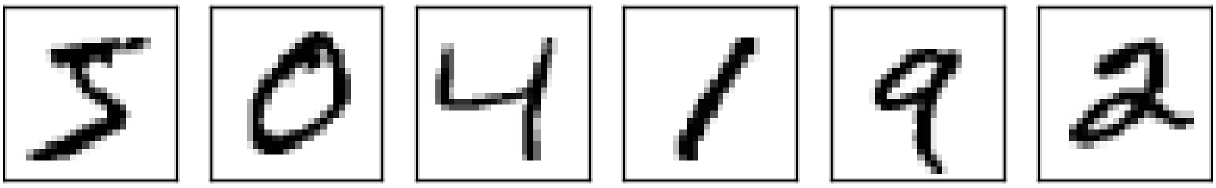


0
0
0
0

10
4
4

4

3
0.99
0.01



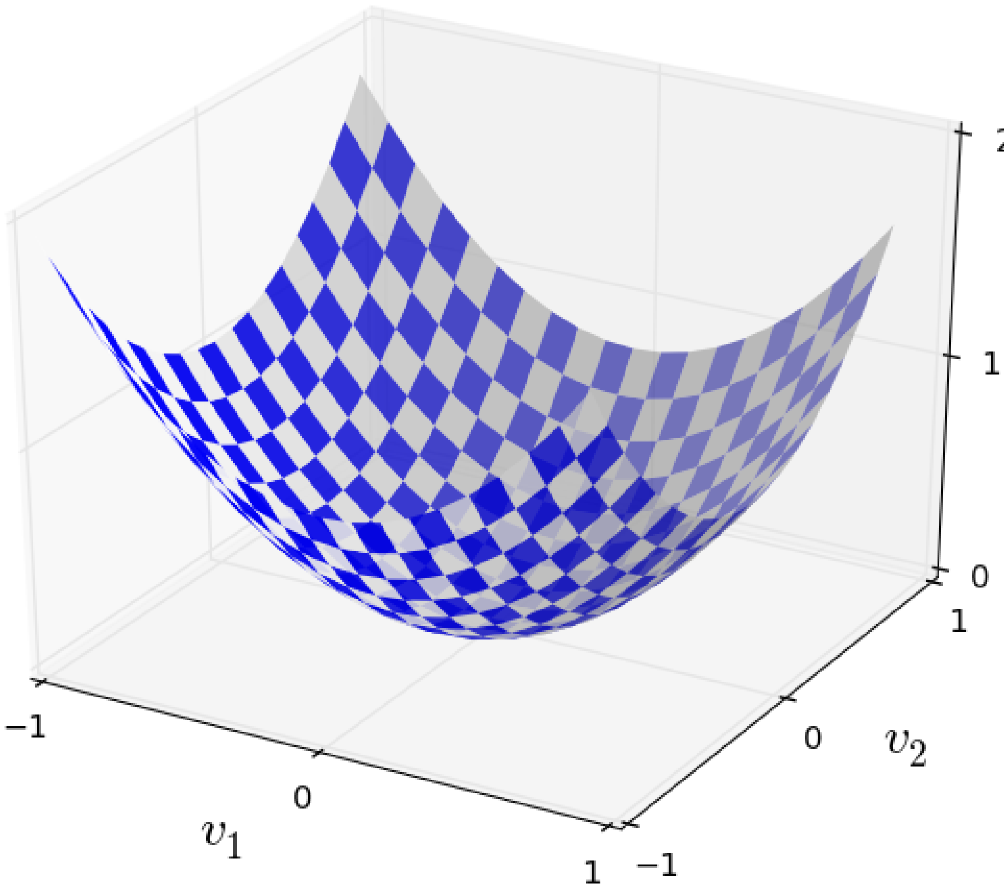
$28x28 =$
784
$y =$
$y(x)$
y
10
6
$y(x) =$
$(0, 0, 0, 0, 0, 0, 1, 0, 0, 0)T$
T

$()$

$$C(w,b) \equiv \frac{1}{2n}\sum_x ||y(x)a||^2$$

$||v||$

MSE
$(,)$
$(,)$



C
C

C
C

C
C

C

C

Δv_1
v_1
Δv_2
v_2
C

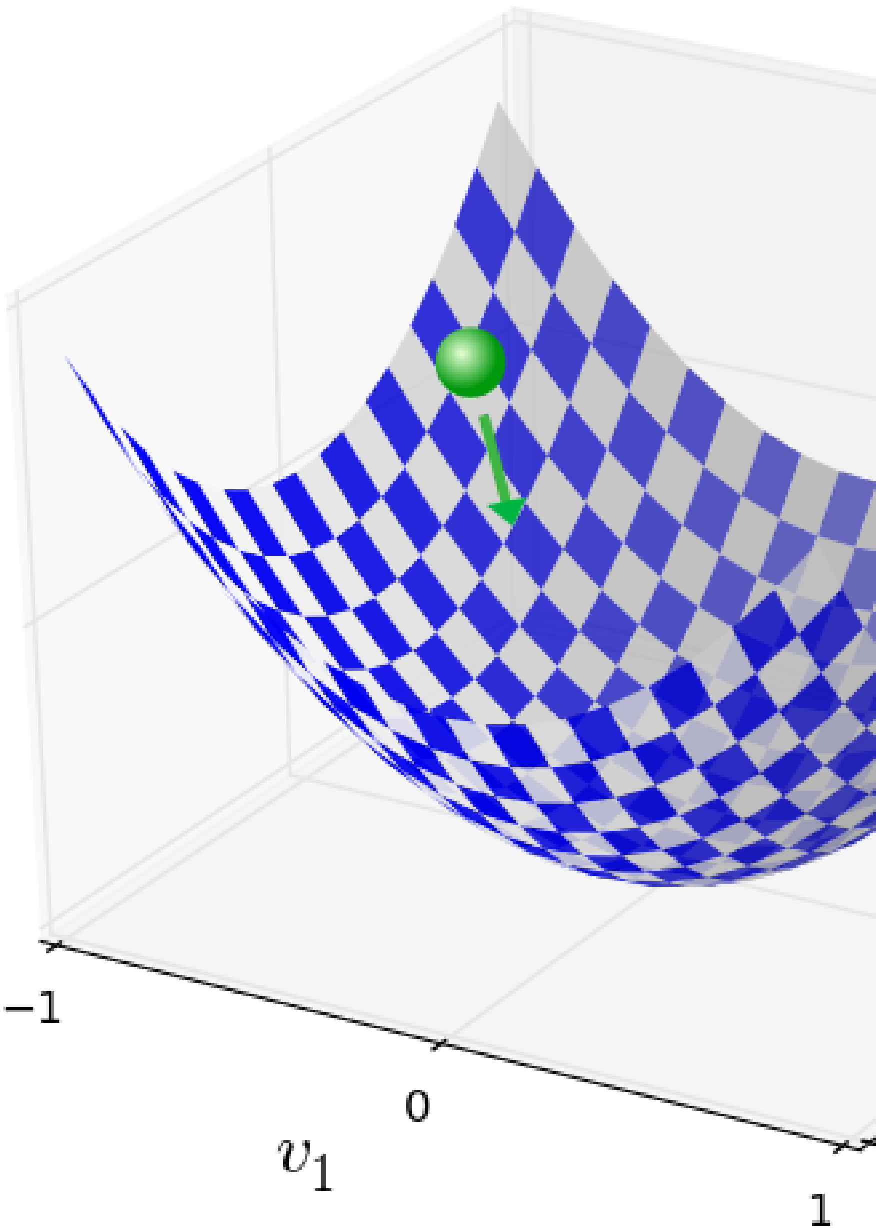
$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Δv_1
Δv_2
ΔC
Δv
v
$\Delta v \equiv$
$(\Delta v_1, \Delta v_2)^T$
T
C
$(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$
∇C

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

ΔC
∇C

∇C
∇
∇



Δv

η
$\Delta C >$
0
η
Δv
η

C
C
C
m
$v_1, ..., v_m$
ΔC
C
$\Delta v =$
$(\Delta v_1, ..., \Delta v_m)^T$

$\Delta C \approx \nabla C \cdot \Delta v$

∇C

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_m}\right)^T$$

$\Delta v = \eta \nabla C$

ΔC
C

$v \rightarrow v' = v - \eta \nabla C$

v
C
C

Δv

C
$\Delta C \approx$
$\nabla C \cdot$
Δv
$||\Delta v|| =$
ϵ
$\epsilon >$
0
C
Δv
$\nabla C \cdot$
Δv
$\Delta v =$
$\eta \nabla C \Delta v =$
$\eta \nabla C$
$\eta =$
$\frac{\epsilon}{||\nabla C||}$
$||\Delta v|| =$
ϵ
C

C
C

C
$\frac{\partial^2 C}{\partial v_j \partial v_k}$
v_j

w_k
b_l
v_j
w_k
b_l

$\frac{{}^2 C}{v_j v_k} =$
$\frac{{}^2 C}{v_k v_j}$

∇C
$\frac{C}{w_k}$
$\frac{C}{b_l}$

$$
\begin{aligned}
w_k \rightarrow w_k' &= w_k - \eta \frac{\partial C}{\partial w_k} \\
b_l \rightarrow b_l' &= b_l - \eta \frac{\partial C}{\partial b_l}
\end{aligned}
$$

$C =$
$\frac{1}{n}\sum_x C_x$
$C_x \equiv$
$\frac{||y(x)a||^2}{2}$
∇C
∇C_x
$\nabla C =$
$\frac{1}{n}\sum_x \nabla C_x$

∇C
∇C_x
∇C

$X_1, X_2, ..., X_m$
∇C_{X_j}
∇C_x

$$\frac{\sum_1^{m_j} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n}$$

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}$$

w_k
b_l

$$
\begin{aligned}
w_k \rightarrow w_k' &= w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \\
b_l \rightarrow b_l' &= b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}
\end{aligned}
$$

X_j

$\frac{1}{n}$
$\frac{1}{n}$
η

$n =$
$60,000$
$m =$
10
$6,000$
C

x

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

20

C
ΔC
C

```
git

   clone

   https
   ://
   github
   .
   com
   /
```

```
mnielsen/neural-networks-and-deep-learning.git
```

```
class Network(object):

    def __init__(self, sizes):
```

```
self
.
num_layers

=

len
(
sizes
)

self
.
sizes

=

sizes

self
.
biases

=

[
np
.
random
.
randn
(
y
,

1)
```

```
for

y

in

sizes
[1:]]








self
.
weights

=

[
np
.
random
.
randn
(
y
,

x
)
```

```
for
x
,
y
in
zip
(
sizes
[:-1],
sizes
[1:])
]
```

```
net
=
Network
([2,
3,
1])
```

np.random.randn
0
1
$Numpy$

k^{th}
j^{th}
j
k

j
k

$a = \sigma(wa+b)$

σ
σ

```
def sigmoid(z):
    return 1.0/(1.0 + np.exp(-z))
```

```
def feedforward(self,
```

```
a
)
:



"""
Return

the

output

of

the

network

if

"
a
"

is

input
.
"""



for

b
,

w

in

zip
(
self
.
biases
,

self
.
```

```
weights):
    a = sigmoid(np.dot(w, a) + b)
    return a

def SGD(self, training_data, epochs
```

```
,
mini_batch_size
,
eta
,
test_data
=
None
)
:
"""
Train
the
neural
network
using
mini
-
batch
stochastic
gradient
descent
.
The
"
training_data
"
is
a
```

```
list of tuples "(x, y)" representing the training inputs and the desired outputs. The other non-optional parameters are
```

```
self
-
explanatory
.

If

"
test_data
"

is

provided

then

the

network

will

be

evaluated

against

the

test

data

after

each

epoch
,

and

partial
```

```
progress
printed
out
.
This
is
useful
for
tracking
progress
,
but
slows
things
down
substantially
.
"""
if
test_data
:
n_test
=
len
(
test_data
```

```
)
n = len(training_data)
for j in xrange(epochs):
    random.shuffle(training_data)
    mini_batches
```

```
=

[
training_data
[
k
:
k
+
mini_batch_size
]

for

k

in

xrange
(0,

n
,

mini_batch_size
)
]

for

mini_batch

in
```

```
mini_batches
:
self
.
update_mini_batch
(
mini_batch
,
eta
)
if
test_data
:
print
"
Epoch
```

```
{0}:
{1}
/
{2}
"
.
format
(
j
,
self
.
evaluate
(
test_data
)
,
n_test
)
else
:
print
"
Epoch
{0}
```

```
complete"
.format(j)
```

η

```
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and
```

```
biases
by
applying
gradient
descent
using
backpropagation
to
a
single
mini
batch
.
The
"
mini_batch
"
is
a
list
of
```

```
tuples
"(
x
,
y
)
",
and
"
eta
"
is
the
learning
rate
.
"""
nabla_b
=
[
np
.
zeros
(
b
.
shape
)
```

```
for
b
in
self
.
biases
]

nabla_w
=
[
np
.
zeros
(
w
.
shape
)
for
w
in
self
.
weights
]

for
```

```
x
,

y

in

mini_batch
:

delta_nabla_b
,

delta_nabla_w

=

self
.
backprop
(
x
,

y
)

nabla_b

=
```

```
[
nb
+
dnb

for

nb
,

dnb

in

zip
(
nabla_b
,

delta_nabla_b
)
]









nabla_w

=

[
nw
+
dnw

for

nw
,

dnw

in

zip
(
```

```
nabla_w
,

delta_nabla_w
)
]








self
.
weights

=

[
w
-(
eta
/
len
(
mini_batch
)
)
*
nw
```

```
for
w
,
nw
in
zip
(
self
.
weights
,
nabla_w
)
]
self
.
biases
=
[
b
-(
eta
/
len
(
mini_batch
)
)
*
nb
```

```
for
b
,
nb
in
zip
(
self
.
biases
,
nabla_b
)
]
```

```
delta_nabla_b
```

```
,
delta_nabla_w
=
self
.
backprop
(
x
,
y
)
```

σ

```
"""
network
.
py
~~~~~~~~~~
A
module
to
implement
the
stochastic
gradient
descent
learning
algorithm
for
```

a feedforward neural network. Gradients are calculated using backpropagation. Note that I have focused on making the code simple, easily readable, and easily modifiable.

```
It is not optimized, and omits many desirable features.
"""

# ### Libraries
# Standard library
import random

# Third-party libraries
import numpy as
```

```
np
class Network(object):
    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in
```

```
the
respective
layers
of
the
network
.
For
example
,
if
the
list
was
[2,
3,
1]
then
it
would
be
a
three
-
layer
network
,
with
the
```

first

layer

containing

2

neurons
,

the

second

layer

3

neurons
,

and

the

third

layer

1

neuron
.

The

biases

and

weights

for

the

network

are

initialized

randomly

,
using
a
Gaussian
distribution
with
mean
0,
and
variance
1.

Note
that
the
first
layer
is
assumed
to
be
an
input
layer
,
and
by
convention
we

```
won
'
t

set

any

biases

for

those

neurons
,

since

biases

are

only

ever

used

in

computing

the

outputs

from

later

layers
.
"""









self
```

```
.
num_layers

=

len
(
sizes
)








self
.
sizes

=

sizes








self
.
biases

=

[
np
.
random
.
randn
(
y
,

1)

for

y
```

```
in

sizes
[1:]]








self
.
weights

=

[
np
.
random
.
randn
(
y
,

x
)
```

```
for x, y in zip(sizes[:-1], sizes[1:])]

def feedforward(self, a):
    """Return the output of the
```

```
network
if
``
a
``
is
input
.
"""
for
b
,
w
in
zip
(
self
.
biases
,
self
.
weights
)
:
```

```
a = sigmoid(np.dot(w, a)+b)
return a
def SGD(self, training_data, epochs, mini_batch_size, eta,
```

```
test_data
=
None
)
:

"""
Train

the

neural

network

using

mini
-
batch

stochastic

gradient

descent
.

The

``
training_data
``
```

is a list of tuples ``(x, y)`` representing the training inputs and the desired outputs.

The other non-optional parameters are self-explanatory.

If ``test_data`` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out.

This is useful

```
for
tracking
progress
,
but
slows
things
down
substantially
.
"""
if
test_data
:
n_test
=
len
(
test_data
)
n
=
len
(
```

```
training_data
)
for
j
in
xrange
(
epochs
)
:
random
.
shuffle
(
training_data
)
mini_batches
```

```
=
[
training_data
[
k
:
k
+
mini_batch_size
]
for
k
in
xrange
(0,
```

```
n
,
mini_batch_size
)
]
for
mini_batch
in
mini_batches
:
self
.
update_mini_batch
(
mini_batch
,
eta
)
```

```
if
test_data
:
print
"
Epoch
{0}:
{1}
/
{2}
"
.
format
(
```

```
j
,
self
.
evaluate
(
test_data
)
,
n_test
)
else
:
```

```
print "Epoch {0} complete".format(j)

def update_mini_batch(self, mini_batch, eta):
    """Update the network's
```

weights

and

biases

by

applying

gradient

descent

using

backpropagation

to

a

single

mini

batch

.

The

``

mini_batch

``

is

a

list

of

tuples

``(

x

,

y

)

``,

and

```
``
eta
``

is

the

learning

rate
.
"""

nabla_b

=

[
np
.
zeros
(
b
.
shape
)

for

b

in

self
.
biases
]
```

```
nabla_w = [np.zeros(w.shape) for w in self.weights]
for x, y in mini_batch:
```

```
delta_nabla_b
,
delta_nabla_w
=
self
.
backprop
(
x
,
y
)
nabla_b
=
[
nb
+
dnb
for
nb
,
dnb
in
zip
(
nabla_b
,
```

```
delta_nabla_b
)
]

nabla_w
=
[
nw
+
dnw
for
nw
,
dnw
in
zip
(
nabla_w
,
delta_nabla_w
)
]

self
.
weights
```

```
=

[
w
-(
eta
/
len
(
mini_batch
)
)
*
nw





















for

w
,

nw

in

zip
(
self
.
weights
,
```

```
nabla_w
)
]

self
.
biases
=
[
b
-(
eta
/
len
(
mini_batch
)
)
*
nb
```

```
for b, nb in zip(self.biases, nabla_b)]

def backprop(self, x, y):
    """Return a tuple ``(
```

nabla_b
,
nabla_w
)
``
representing
the
gradient
for
the
cost
function
C_x
.

``
nabla_b
``
and
``
nabla_w
``
are
layer
-
by
-
layer
lists
of
numpy
arrays
,
similar
to

```
``
self
.
biases
``

and

``
self
.
weights
``.
"""

nabla_b

=

[
np
.
zeros
(
b
.
shape
)

for

b

in

self
.
biases
]
```

```
nabla_w = [np.zeros(w.shape) for w in self.weights]
# feedforward
activation = x
```

```
# list to store all the activations, layer by layer
activations = [x]
zs =
```

```
[]
# list to store all the z vectors, layer by layer
for b, w in zip(self.biases, self.weights):
```

```
z
=
np
.
dot
(
w
,
activation
)
+
b
zs
.
append
(
z
)
```

```
activation

=

sigmoid
(
z
)

activations
.
append
(
activation
)

#

backward

pass
```

```
delta

=

self
.
cost_derivative
(
activations
[-1],

y
)

*

\

sigmoid_prime
(
zs
[-1])

nabla_b
[-1]

=

delta
```

```
nabla_w
[-1]

=

np
.
dot
(
delta
,

activations
[-2].
transpose
()
)

# Note that the variable l in the loop below is used a
```

little

#

differently

to

the

notation

in

Chapter

2

of

the

book

.

#

Here

,

l

=

1

means

the

last layer of neurons, l = 2 is the

second-last layer, and so on. It's a renumbering of

```
the

#
scheme
in
the
book
,
used
here
to
take
advantage
of
the
fact

#
that
Python
can
use
```

```
negative
indices
in
lists
.

for
l
in
xrange
(2,
self
.
num_layers
)
:

z
=
zs
[-
l
]
```

```
sp = sigmoid_prime(z)
delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
```

```
nabla_b
[-
l
]

=

delta









nabla_w
[-
l
]

=

np
.
dot
(
delta
,

activations
[-
l
-1].
transpose
()
)
```

```
return
(
nabla_b
,
nabla_w
)

def
evaluate
(
self
,
test_data
)
:

"""
Return
the
number
of
test
inputs
```

for which the neural network outputs the correct result. Note that the neural network's output is assumed to

```
be
the
index
of
whichever
neuron
in
the
final
layer
has
the
highest
activation
.
"""
test_results
=
[(
np
.
argmax
(
```

```
self
.
feedforward
(
x
)
)
,

y
)
```

```
for

(
x
,

y
)

in

test_data
]
```

```
return sum(int(x == y) for (x, y) in test_results)

def cost_derivative(self, output_activations, y):
```

```
"""
Return
the
vector
of
partial
derivatives
\
partial
C_x
/
\
partial
a
for
the
output
activations
.
"""
return
```

```
(output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(
```

```
z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

```
>>> import mnist_loader
>>> training_data, validation_data
```

```
,
test_data
=
\
...
mnist_loader
.
load_data_wrapper
()
```

```
>>>
import
network
>>>
net
=
network
.
Network
([784,
30,
10])
```

$\eta =$
3.0

```
>>>
net
.
SGD
(
training_data
```

```
,
30,
10,
3.0,
test_data
=
test_data
)
```

$9,129$
$10,000$

```
Epoch
0:
9129
/
10000
Epoch
1:
9295
/
10000
Epoch
2:
9348
/
10000
...
Epoch
```

```
27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

95
95.42

100

```
>>> net = network.Network([784, 100, 10])
```

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

96.59

$\eta = 0.001$

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data
```

```
,
30,
10,
0.001,
test_data
=
test_data
)
```

```
Epoch
0:
1139
/
10000
Epoch
1:
1136
/
10000
Epoch
2:
1135
/
10000
...
Epoch
27:
2101
```

```
/ 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

=
0.01
=
1.0
3.0

=
100.0

```
>>> net = network.Network([784, 30, 10])
```

```
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
```

```
Epoch
    3:
    1009
    /
    10000
...
Epoch
    27:
    982
    /
    10000
Epoch
    28:
    982
    /
    10000
Epoch
    29:
    982
    /
    10000)
```

```
"""
```

```
mnist_loader
~~~~~~~~~~~~

A library to load the MNIST image data. For details of the data
structures that are returned, see the doc strings for ``load_data``
```

```
and
``
load_data_wrapper
``.
In
practice
,
``
load_data_wrapper
``
is
the
function
usually
called
by
our
neural
network
code
.
"""
#
###
Libraries
#
Standard
library
import
cPickle
```

```
import gzip

# Third-party libraries
import numpy as np

def load_data():
    """Return the MNIST data as a tuple containing the training data
```

,

the

validation

data
,

and

the

test

data
.

The

``
training_data
``

is

returned

as

a

tuple

with

two

entries
.

The

```
first
entry
contains
the
actual
training
images
.
This
is
a
numpy
ndarray
with
50,000
entries
.
Each
entry
is
,
in
turn
,
a
```

numpy ndarray with 784 values, representing the 28 * 28 = 784 pixels in a single MNIST image.

The second entry

in the ``training_data`` tuple is a numpy ndarray containing 50,000 entries. Those entries are just the digit values (0 ... 9) for the corresponding images contained in the first entry of the tuple.

The ``validation_data`` and ``test_data`` are similar, except each contains only 10,000 images.

This is a nice data format, but for use in neural networks it's

helpful to modify the

```
format
of
the
``
training_data
``
a
little
.

That
'
s
done
in
the
wrapper
function
``
load_data_wrapper
()
``,
see

below
.

"""
```

```
f = gzip.open('../data/mnist.pkl.gz', 'rb')
training_data, validation_data, test_data = cPickle.load(f)
f
```

```
.
close
()

return
(
training_data
,
validation_data
,
test_data
)

def
load_data_wrapper
()
:

"""
Return
a
tuple
containing
``(
training_data
,
validation_data
,

test_data
)
``.
```

Based

on

``

load_data
``,

but

the

format

is

more

convenient

for

use

in

our

implementation

of

neural

networks
.

In

particular
,

``

training_data
``

is

a

list

containing

50,000

2-
tuples

``(
x
,

y
)
``.

``
x
``

is

a

784-
dimensional

numpy
.
ndarray

containing

the

input

image
.

``
y
``
is
a
10-
dimensional
numpy
.
ndarray
representing
the
unit
vector
corresponding
to
the
correct
digit
for
``
x
``.
``
validation_data
``

and
``
test_data
``
are
lists
containing
10,000
2-
tuples
``(
x
,
y
)
``.
In
each
case
,
``
x
``
is
a
784-
dimensional
numpy
.

ndarry

containing

the

input

image

,

and

``

y

``

is

the

corresponding

classification

,

i

.

e

.,

the

digit

values

(

integers

)

corresponding

to

``

x

``.

Obviously
,
this
means
we
'
re
using
slightly
different
formats
for

the
training
data
and
the
validation
/
test
data
.
These
formats

```
turn out to be the most convenient for use in our neural network code.
"""
tr_d, va_d, te_d = load_data()
```

```
training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
training_results = [vectorized_result(y) for y in tr_d[1]]
```

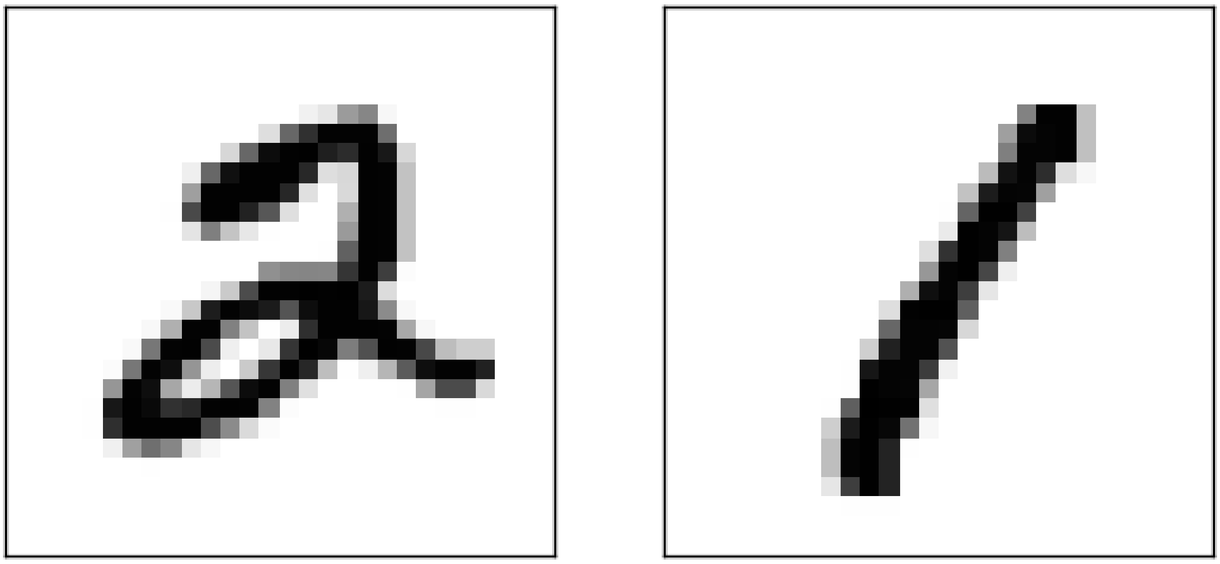
```
training_data = zip(training_inputs, training_results)
validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
validation_data = zip
```

```
(
validation_inputs
,

va_d
[1])




test_inputs

=

[
np
.
reshape
(
x
,

(784,

1)
)

for

x

in

te_d
[0]]




test_data

=

zip
(
test_inputs
,

te_d
[1])
```

```
    return (training_data, validation_data, test_data)

def vectorized_result(j):
    """Return a 10-dimensional unit vector with a 1.0 in the jth
```

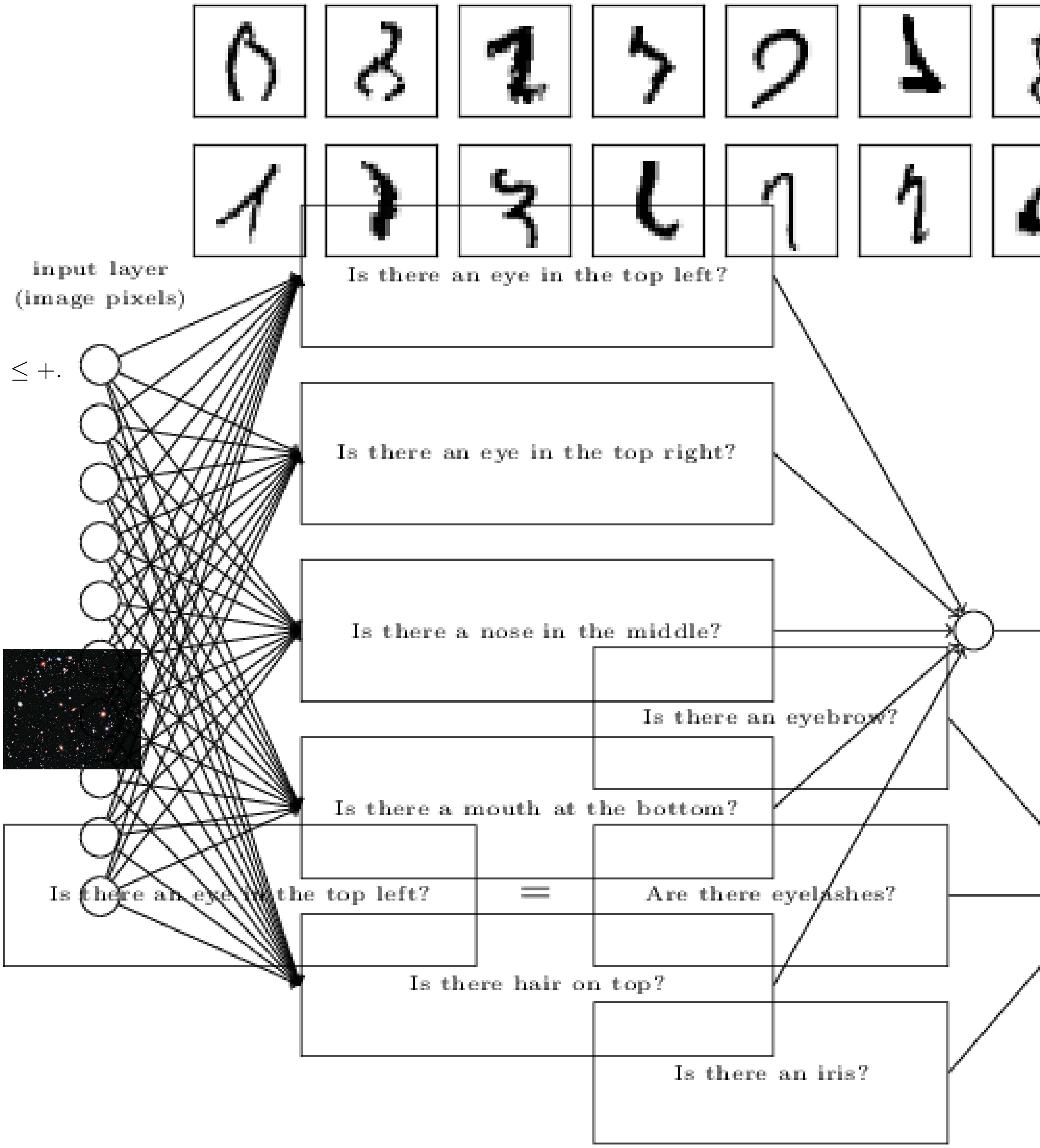
```
position and zeroes elsewhere. This is used to convert a digit (0...9) into a corresponding desired output from the neural network."""
```

```
e = np.zeros((10, 1))
e[j] = 1.0
```

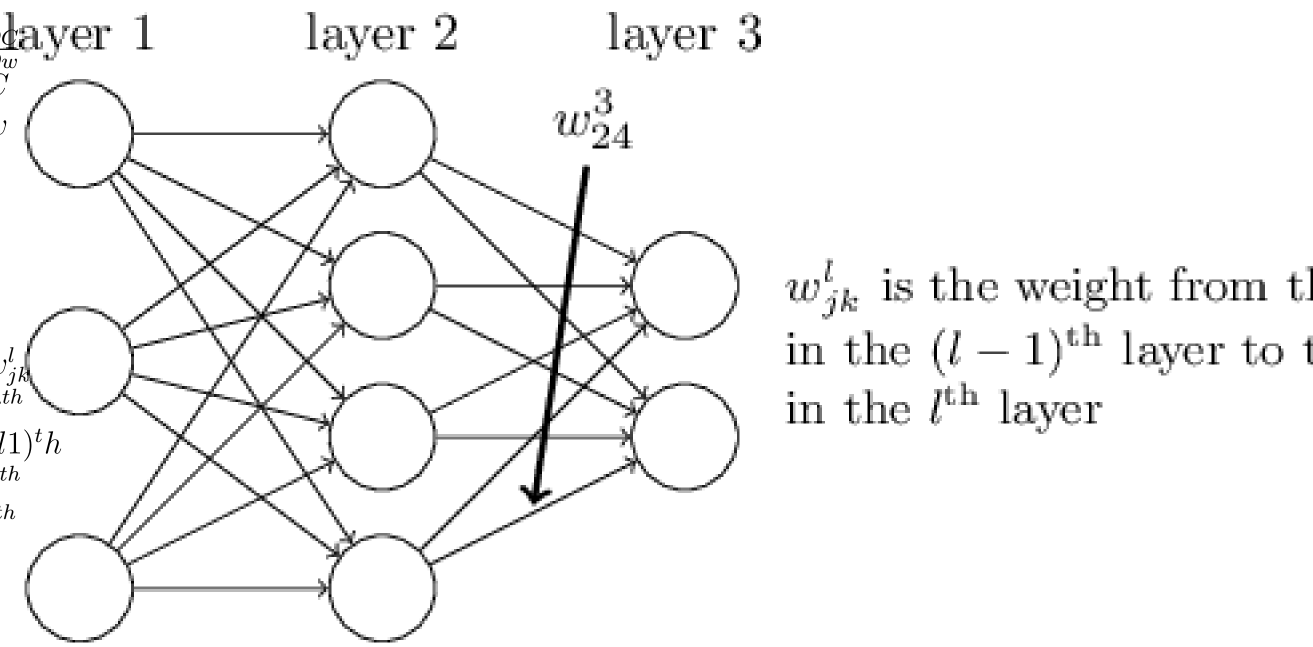


$0, 1, 2, , 9$
$2, 225$
$10, 000$

22.25


input layer
(image pixels)
≤ +.
Is there an eye in the top left?
Is there an eye in the top right?
Is there a nose in the middle?
Is there a mouth at the bottom?
Is there hair on top?
Is there an eyebrow?
Is there an eye in the top left?
=
Are there eyelashes?
Is there an iris?

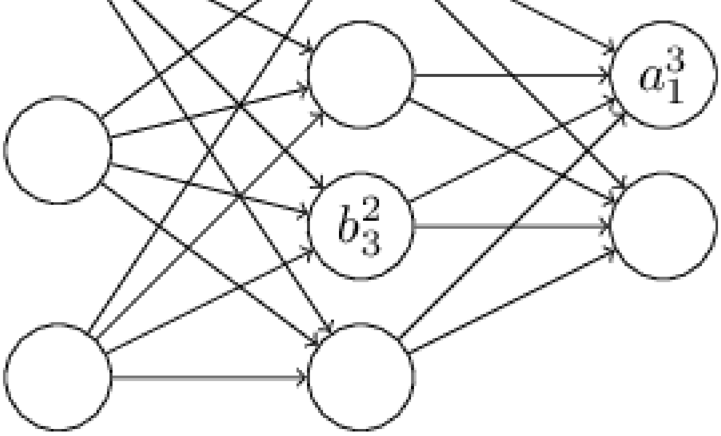
$\frac{\partial C}{\partial w}$
C
w
b



w^l_{jk}
k^{th}
$(l1)^th$
j^{th}
l^{th}

j
k
j
k

b^l_j
l^{th}
a^l_j
j^{th}



l^{th}

j^{th}
l^{th}
$(l1)^{th}$

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

k
$(l1)^{th}$
w^l
l
w^l
l^{th}
j^{th}
w_{jk}^l
l
b^l
b_j^l
l^{th}
a^l
a_j^l

https://www.coursera.org/learn/neural-networks

http://cs231n.github.io/convolutional-networks/

https://www.coursera.org/learn/neural-networks

http://www-cs-faculty.stanford.edu/~uno/abcde.html